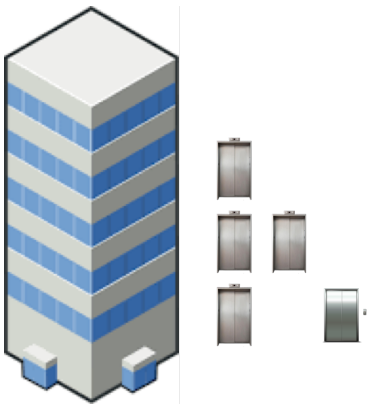
Universidad Complutense de Madrid.

Facultad de Ingeniería Informática

Inteligencia Artificial 1.



Práctica 2B.

Resolución de problemas con búsqueda.

Grupo 5:

Frederick Ernesto Borges Noronha

Víctor Manuel Cavero Gracia

Tabla de contenido

Parte I. Función de coste y heurísticas.

Para la función de coste hemos pensado en implementar una diferente la cual tuviese en cuenta si se tomaba el ascensor lento o el rápido, pero al llevarla a la práctica, ha empeorado el tiempo de búsqueda de una solución.

```
def path_cost(self, c, state1, action, state2):
        newCost = 0
        acc = action.split()[2:]
        if "ASCENSORES" in action:
            for a in acc:
                movimiento, index_asc = a.split(".")
                if "(RAPIDO)" in movimiento:
                    newCost += 1
                else:
                    newCost += self._velAscRapido
        else:
            newCost += 1

        return c + newCost
```

Por otra parte, hemos definido principalmente dos heurísticas en el desarrollo de nuestro problema.

En la solución con **un solo pasajero y un ascensor** definimos una h1 sencilla que simplemente esta basada en la distancia que existe entre el pasajero y su piso de destino, de tal manera que el coste de subir/bajar el ascensor es de una unidad. Esta diferencia esta realizada en valor absoluto para evitar valores negativos.

```
def h1(self, node):
    pisosPasajeros = node.state[0]
    pisosObjetivos = self.goal[0]

    return abs(pisosPasajeros - pisosObjetivos)
```

Durante el desarrollo de la solución con **varios pasajeros y un ascensor** tuvimos que redefinir h1 para sumar cada una de las distancias entre el pasajero y su piso de destino.

```
def h1(self, node):
    sumaDistancias = 0

    pisosPasajeros = node.state[0]
    pisosObjetivos = self.goal[0]

    for x in range(len(pisosPasajeros)):
            sumaDistancias += abs(pisosPasajeros[x] -
pisosObjetivos[x])

    return sumaDistancias
```

Cuando procedimos a realizar la solución con **varios pasajeros y varios ascensores** nos dimos cuenta de que la heurística realizada anteriormente no

era la óptima por lo que tuvimos que mejorarla y así el tiempo de ejecución del problema se reduciría.

```
def h2(self, node):
    '''
    Devuelve la distancia entre el ascensor mas cercano y el pasajero
sumada a
    la distancia entre cada pasajero y su piso final (h1)
    '''

    distanciaPasObj = 0
    distanciaAscPas = 0
    costeSubirBajarPas = 0
    pisosPasajeros = node.state[0]
    pisosObjetivos = self.goal[0]
    pasajerosAsc = node.state[1]
    pisosAsc = node.state[2]

    for indexPas in range(len(pisosPasajeros)):
        # Calculamos la distancia entre cada pasajero y su piso
objetivo.
        distanciaPasObj += abs(pisosPasajeros[indexPas] -
pisosObjetivos[indexPas])
        # Calculamos la distancia entre cada pasajero y su ascensor
mas cercano.
        coste, idxAsc =
self.distanciaAscensorYPasajero(pisosPasajeros[indexPas],pisosAsc)
        distanciaAscPas += coste
        # Calculamos el coste de subir y bajar a los pasajeros
        costeSubirBajarPas +=
self.costeSubirBajar(pisosPasajeros[indexPas],
pisosObjetivos[indexPas], pasajerosAsc, indexPas)

    h = distanciaPasObj + distanciaAscPas + costeSubirBajarPas

    return h
```

A parte de calcular la distancia de cada pasajero a su piso de destino, calculamos su distancia al ascensor más cercano (teniendo además en cuenta para el cálculo el coste del ascensor rápido en el caso de que el pasajero se encuentre en un piso par, haciendo que está heurística sea admisible con la presencia de ascensores rápidos) y asignamos un coste de subir y bajar el pasajero del ascensor. Hemos utilizado para ello las siguientes funciones:

```
def buscar_mas_cercano(self,tupla,valor):
    """
        Se pasa una tupla y el valor que se quiere comparar
    """
    array = np.array(tupla)
    idx = (np.abs(array-valor)).argmin()
    return array[idx], idx

def distanciaAscensorYPasajero(self, pisoPas, pisosAsc):
    pisoAscMasCercano, index = self.buscar_mas_cercano(pisosAsc,
pisoPas)
    return abs(pisoPas - pisoAscMasCercano), index

def buscar_persona_dentro_asc(self,pasajerosAsc, indexPas):
```

```
    aux = np.array(pasajerosAsc)
    result = np.where(aux == 1)
    return indexPas in result[1]

def costeSubirBajar(self, pisoAct, pisoObj, pasajerosAsc, indexPas):
    if (pisoAct == pisoObj):
        if (self.buscar_persona_dentro_asc(pasajerosAsc,indexPas)):
            return 1
        return 0
    else:
        return 2
```

La heurística finalmente toma como resultado la suma de estos tres cálculos realizada para cada uno de los pasajeros existentes.

Parte II. Especificar otras instancias del problema.

Para especificar otras instancias del problema solo debemos realizar pequeños cambios en la llamada a la función que resuelve el problema, veamos el siguiente cuadro con diferentes instancias del problema:

Cod. Caso	# Pasajeros	# Bloques	# Asc. Lentos	# Asc. Rápidos	Tiempo de la solución (s)	Longitud de la solución	Nodos Analizados
A1	1	1	1	0	0,0005	5	6
A2	2	1	1	0	0,0172	15	107
A3	3	1	1	0	0,006	18	28
A4	4	1	1	0	299,3324	33	27411
A5	5	1	1	0	N/R	N/R	N/R
B1	1	1	2	0	0,0026	5	13
B2	2	1	2	0	0,0242	15	51
B3	3	1	2	0	0,1299	19	120
B4	4	1	2	0	0,8839	31	411
B5	5	1	2	0	0,2442	37	148
C1	1	2	2	1	0,0302	5	34
C2	2	2	2	1	2,3461	13	337
C3	3	2	2	1	18,5333	22	920
C4	4	2	2	1	48,6522	31	1359
C5	5	2	2	1	N/R	N/R	N/R

C5*	5	1	2	1	1,9569	33	180
D1	1	3	3	1	0,7194	5	98
D2	2	3	3	1	74,7801	12	1193
D3*	3	3	3	1	40,2471	16	1020
D4	4	3	3	1	N/R	N/R	N/R
D5	5	3	3	1	N/R	N/R	N/R
E1	1	3	4	1	27,8053	5	310
E2	2	3	4	1	N/R	N/R	N/R
E3	3	3	4	1	N/R	N/R	N/R
E4	4	3	4	1	N/R	N/R	N/R
E5	5	3	4	1	N/R	N/R	N/R

En la tabla anterior podemos observar los diferentes casos de prueba que hemos realizado en la práctica junto con sus resultados.

Podemos tomar como conclusión que nuestra implementación del problema funciona perfectamente para situaciones poco complejas donde no se generen muchos nodos, pero en cuanto se aumenta el factor de ramificación los tiempos y los nodos analizados aumentan de forma considerable.

Parte III. Modificación de datos del problema.

- **Cambiar plantas de origen y destino de los pasajeros:**

Podemos ver como gracias a nuestra definición de estado se pueden variar sin ningún tipo de problema el origen y destino de los pasajeros. Estos, obviamente, se tendrán en cuenta en las operaciones.

Representación:

((pisPas1, pisPas2), (pasAsc1, pasAsc2), pisoA1)

Estado inicial -> ((2,4), ((0,0), (0,0)), (0,0))

Estado final-> ((3,11), ((0,0), (0,0)), (0,0))

Donde pasAsc1 y pasAsc2 son túplas que contienen 0 si el pasajero no esta dentro del ascensor y 1 si esta dentro.

Para introducir las plantas de origen simplemente cambiamos el estado inicial y en concreto la tupla *(pisPas1, pisPas2…)* que son las que almacenan el piso de los pasajeros, y para cambiar las plantas de destino se modifica el estado objetivo. También esto queda totalmente probado en los ejemplos presentes en el documento, tales como:

```
pisosAsc = (0,4,8)
pisosPasInicial = (2,4,1,8,1)
pisosPasFinal = (3,11,12,1,9)
ascVacios = tuple(tuple(0 for x in pisosPasInicial) for x in pisosAsc)
estado_inicial = (pisosPasInicial,ascVacios,pisosAsc)
objetivo = (pisosPasFinal,ascVacios,pisosAsc)
# LOS BLOQUES INDICAN LOS PISOS A LOS QUE PUEDEN ACCEDER LOS
ASCENSORES
bloques = ((0,12),(0,12),(0,12))
limAscLento=2
limAscRapido=3
cantAscRapidos=0
problema = ProblemaAscensoresV4(estado_inicial,
                                objetivo,
                                bloques=bloques,
                                limAscLento=limAscLento,
                                limAscRapido=limAscRapido,
                                cantAscRapidos=cantAscRapidos)
resuelve_problema(problema, astar_search, problema.h2)
```

- **Tener en cuenta el tiempo de subir y bajar pasajeros para cambiar de ascensor.**

De hecho, hemos tenido en cuenta el coste de subir y bajar del ascensor en la realización de la heurística *h2*, asignando un coste de 1 para la operación de subir o bajar y un coste de 2 en el caso de que el pasajero no se encuentre en la posición objetivo ya que debería llegar al piso y bajar del ascensor.

```
def costeSubirBajar(self, pisoAct, pisoObj, pasajerosAsc, indexPas):
    if (pisoAct == pisoObj):
        if (self.buscar_persona_dentro_asc(pasajerosAsc,indexPas)):
            return 1
        return 0
    else:
        return 2
```

- **Incluir nuevos pasajeros**

Al igual que en la primera de las modificaciones solo tendrían que introducirse los nuevos pasajeros en la tupla *(pisPas1, pisPas2…)* con su piso inicial (en el estado inicial) y su piso destino (en el estado objetivo).

- **Cambiar la capacidad de los ascensores**

Se podría cambiar modificando las variables *limAscLento* y *limAscRapido* que son introducidas en la declaración del problema. Por defecto reciben los valores del problema presentado en la práctica.

```
def __init__(self, initial,
                goal=None,
                limAscLento=2,
                limAscRapido=3,
                cantAscRapidos=1,
                velAscRapido=2,
                bloques=((0,12),(0,12),(0,12),(0,12),(0,12)) ):
    '''Inicializacion de nuestro problema.'''
    Problem.__init__(self, initial, goal)
```

```
    self._limiteAscLento = limAscLento
    self._limiteAscRapido = limAscRapido
    self._limAsc = bloques
    self._cantAscLentos = len(bloques) - cantAscRapidos
    self._cantAscRapidos = cantAscRapidos
    self._velAscRapido = velAscRapido
    aux = list()
    for bloque in bloques:
        if bloque not in aux:
            aux.append(bloque)
    self._bloques = tuple(aux)
```

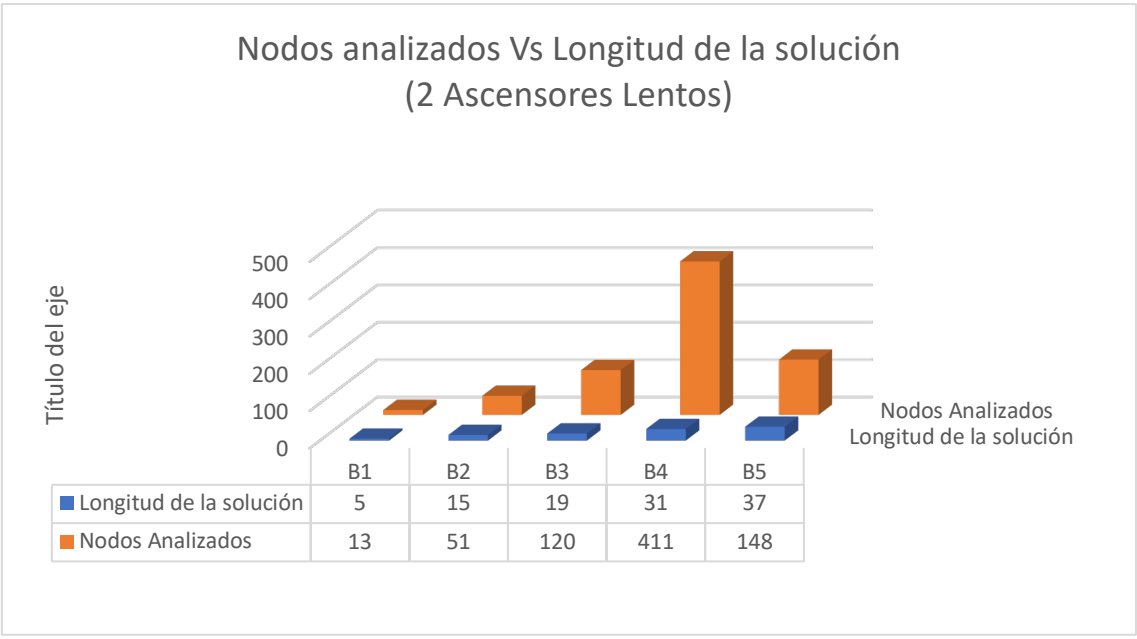
- **Añadir nuevos ascensores, por ejemplo, un ascensor rápido que se mueva entre las plantas impares.**

Podríamos cambiar el estado inicial del ascensor rápido poniéndole que este empiece en el piso número 1 y cambiando el bloque en el que actúa el ascensor, este se le pasa en al inicializar el problema, por lo tanto, no sería necesario modificar el código de la representación.

Parte IV. Análisis de resultados.

Para las heurísticas podemos indicar que debido a la forma de implementar nuestra representación del problema no son heurísticas admisibles y, por lo tanto, tampoco son consistentes. Sin embargo, aunque esto suceda el algoritmo A* garantiza que se obtenga una solución, pero se debe tener en cuenta que no lo hará de forma rápida ya que no dará un coste elevado a los estados incorrectos.

Si tomamos de referencia la tabla que hemos realizado anteriormente, y acotamos algunos problemas para estudiar un poco con graficas tomaremos el único bloque que tiene todos los casos completos (el bloque B).



Como se puede observar en la grafica, al no tener una buena heurística se analizan muchos mas nodos de los necesarios, pero aun así el algoritmo A* concluye sus iteraciones llegando a un resultado.

Por otra parte, si observamos los tiempos en los que se encuentra la solución tenemos lo siguiente:

Podemos observar que para el problema B5 se tarda mucho menos que el problema B4.

Gracias a todo esto podemos concluir que el algoritmo tiene un funcionamiento correcto, pero al realizar problemas con mucha complejidad en el que el estado inicial del problema no este cerca de la solución óptima, el algoritmo generará una gran cantidad de nodos y como consecuencia de esto tardará mucho tiempo.