Grupo 21

PRÁCTICA 2B ASCENSORES

El problema de los ascensores admite varias aproximaciones debido a que podemos considerar que siempre es soluble, siempre y cuando no haya que resolverlo antes de un tiempo dado. En esta práctica no tenemos en cuenta el tiempo en ese sentido, se busca el menor tiempo, en este caso acciones, pero no se pone un límite, lo que simplifica heurísticas y funciones.

Funciones de coste y heurísticas

El problema de los ascensores se vuelve más complejo cuantos más elementos contiene. Si bien se puede trabajar con versiones simplificadas, todas tienen características similares. Este problema, al tener diversos elementos como son cada una de las personas que van a destinos distintos o los ascensores que las transportan hace que los árboles de búsqueda se expandan muy rápido y los algoritmos tengan más difícil encontrar una solución, en concreto los de búsqueda a ciegas y, si no tienen control de repetidos como el algoritmo de búsqueda en profundidad o en anchura con árbol, será sencillo también que entren en bucles si no cuentan con control de repetidos.
Por otro lado es muy probable que con una heurística simple (aunque certera) algoritmos como A* no tengan problema en encontrar una solución en un tiempo favorable.

La **heurística más simple** que barajamos es el **número de personas que no están en su planta de destino**, es una heurística muy pobre y no consistente, aunque sí admisible:
$\sum_{i=0}^{n} (x_i \neq y_i)$ siendo *x* e *y* la planta actual y la de destino respectivamente y *n* las personas en el problema.
Buscando mejores resultados de los algoritmos planteamos la **distancia Manhattan** de cada usuario a su planta de destino. Esta heurística es consistente ya que utiliza el sumatorio de las distancias para establecer la mejor acción y las distancias no se sobreestiman.
Una **tercera aproximación** utiliza la distancia Manhattan y, además, penaliza que el ascensor esté vacío cuando aún hay gente que no está en su planta de destino. Esto lo conseguimos sumando lo siguiente a la distancia Manhattan del estado actúa por cada bloque de pisos: $(\sum x_n \div y_n)^* z_n$

Donde *x* indica las personas fuera de su planta de destino para el bloque *n*, *y* y *z* simbolizan la capacidad y los huecos libres en ese estado del ascensor del bloque *n* respectivamente.

Por último, una cuarta heurística en la cual tener en cuenta las direcciones de los pasajeros que suben al ascensor con respecto a su distancia de destino como añadido a la última heurística mencionada es prometedora pero no se aplica debido a que el cálculo y cómputo de la heurística en cada estado podría penalizar sobre el tiempo total de ejecución del algoritmo.

Instancias del problema

Para poder plantear el problema desde un principio como un problema genérico e ir añadiendo paso a paso elementos trabajamos en primer lugar con una instancia simplificada del problema.
Esta aproximación nos permite encontrar las partes distintas del problema: **bloque**, **pasajeros** y **ascensor**. Establecemos que las **acciones** las realiza el ascensor y que éstas pueden ser tres: **subir**, **bajar** o **parar**.
Subir: mueve el ascensor hacia la planta superior y con él a los pasajeros que lleva consigo. **Si está subiendo el ascensor llevará el código 1**.
Bajar: mueven el ascensor a la planta inferior junto con los pasajeros que se encuentran dentro. **Si está bajando el ascensor llevará el código -1**.
Tanto subir como bajar actualizan su planta y la de los pasajeros que han subido o bajado en cada ejecución de la acción.
Parar: se encarga de subir y bajar pasajeros del ascensor (siempre que se pueda), y puede hacer ambas en una misma acción. Tras cambiar la situación de un usuario con respecto al ascensor (si está subido o no) ésta se actualiza en los pasajeros que lleva el ascensor en ese momento. **Si está parado el ascensor llevará el código 0**.

Problema simplificado

Simplificamos el problema todo lo posible contando con un único bloque de pisos accesible al completo por el único ascensor que hay y un grupo de personas repartidas entre distintas plantas. Aunque podríamos plantear el problema con una única persona consideramos que el número de personas no afectará a la ejecución (sí al tiempo) mientras las acciones que pueda ejecutar el ascensor estén bien definidas, esto se desarrollará en el análisis con más detenimiento. Aunque los parámetros no son los mismos que el problema inicialmente propuesto, la resolución sí lo es ya que el ascensor debe llevar a cada pasajero a su destino en el menor tiempo posible, para ello no necesitamos modificar la clase Problema, solo sus

funciones. A continuación encontramos la función ‘actions’ para el problema simplificado:

```python
def actions(self,estado):

    #operadores: subir, bajar, parar
    pos_ascensor = estado[1][0]


    #pasajeros = que pasajeros están en el ascensor
    pasajeros = estado[1][1]

    #plantas destino de los pasajeros
    plantas = list()
    for i in range(len(pasajeros)):
        plantas.append(self.goal[i])

    #lista de personas aun por recoger
    recoger = list()

    for i in range(len(estado[0])):
        if i not in pasajeros and estado[0][i] != self.goal[i]:
            recoger.append(i)

    #lista de acciones
    accs = list()

    if pos_ascensor < 4:
        accs.append("Subir")

    if pos_ascensor > 0:
        accs.append("Bajar")

    #no necesitamos comprobar los pasajeros del ascensor porque se actualizan en cada nodo
    #comprobamos ademas si hay alguien que recoger o si alguien ha llegaddo a su destino
    if (pos_ascensor in recoger and len(pasajeros) < 3) or (pos_ascensor in plantas):
        accs.append("Parar")

    return accs
```

Imagen 2.1

Para decidir si sube o baja comprobaremos que la planta del ascensor no está en ninguno de los límites: ni superior ni inferior. Por otra parte, y para no ejecutar acciones en vano, comprobaremos si hay alguien en la planta que nos encontramos y si hay hueco disponible en el ascensor antes de decidir si podemos parar.

Los estados resultado se computan a través de la siguiente función:

```
### EJERCICIO 1.2. COMPLETA LA DEFINICIÓN DE LOS OPERADORES.
 if accion == "Subir":
     ascensor[2] = 1                  #actualizamos el estado del ascensor
     ascensor[0] = ascensor[0] + 1    #actualizamos la planta del ascensor
     for i in range(len(ascensor[1])):            #actualizamos, para cada pasajero del ascensor, su planta
         usuarios[i] = ascensor[0]

 if accion == "Bajar":
     ascensor[2] = -1                 #actualizamos el estado del ascensor
     ascensor[0] = ascensor[0] - 1    #actualizamos la planta del ascensor
     for i in range(len(ascensor[1])):              #actualizamos, para cada pasajero del ascensor, su planta
         usuarios[i] = ascensor[0]

 if accion == "Parar":
     ascensor[1] = [x for x in ascensor[1] if not (self.goal[x] == ascensor[0])]  #bajamos del ascensor a los que
     #han llegado a su planta
     #lo mismo que a continuacion
     #for e in ascensor[1]:
         #if self.goal[0][e] == ascensor[0]:
             #del(ascensor[1][e])
     if len(ascensor[1]) < 3:                                        #subimos gente al ascensor si hay hueco
         for i in range(len(usuarios)):
             if len(ascensor[1]) < 3 and self.goal[i] != estado[0][i]:
                 if (usuarios[i] == ascensor[0]) and (i not in ascensor[1]):
                     ascensor[1].append(i)
```

Imagen 2.2

Como podemos ver las acciones son ejecutadas por el ascensor y se ven reflejadas en la representación del problema. En este caso, si observamos la acción "parar" evaluamos si podemos recoger gente o si podemos bajar gente en la planta en la que nos encontramos.
Es aquí donde nos encargamos de comprobar si podemos realizar una o ambas acciones cuando paramos (subir y/o bajar). Con esto pretendemos diferenciar que un pasajero se encuentre en la planta de destino y en el ascensor de que se encuentre en la planta de destino y fuera del ascensor.

Problema completo: abstracción

Una vez conseguimos evaluar el problema simplificado aplicamos inducción sobre el mismo para aproximarnos al problema original. En este caso contamos con dos bloques de pisos y dos ascensores. Cada ascensor cubre un bloque de pisos y éstos cuentan con una planta en común, que usarán los pasajeros para cambiar de bloque.
El número de personas en este caso es el mismo aunque se encuentran distribuidas entre ambos bloques, esto distribuye las tareas a realizar y cada ascensor tendrá que tener en cuenta la situación de su bloque..

```
    #lista de acciones
    accs = list()

    for i in range(len(ascensores_constante)):   #Recorremos la lista de ascensores para ver que acciones pueden hacer.
        if pos_ascensor[i] < ascensores_constante[i][3]:
            #accs[i].append("Subir ascensor ", i)
            if(i == 0):
                accs.append("Subir 0")
            else:
                accs.append("Subir 1")

        if pos_ascensor[i] > ascensores_constante[i][2]:
            if(i == 0):
                accs.append("Bajar 0")
            else:
                accs.append("Bajar 1")

        #no necesitamos comprobar los pasajeros del ascensor porque se actualizan en cada nodo
        #comprobamos ademas si hay alguien que recoger o si alguien ha llegaddo a su destino

        for j in range(len(recoger[i])):
            if (pos_ascensor[i] == estado[0][recoger[i][j]] and len(ascensores_variable[i][0]) < ascensores_constante[i][1]):
                if(i == 0):
                    accs.append("Parar 0")
                else:
                    accs.append("Parar 1")

    return accs
```

Imagen 2.3

Las acciones son las mismas que en el problema simple y solamente se ejecutan sobre un ascensor cada vez. Para que esto sea posible se indica a qué ascensor pertenece cada acción: ‘Subir 0’, ‘Bajar 1’. Esto implica que los algoritmos, al expandir un nodo, deben tener en cuenta los resultados para cada uno de los ascensores. Los estados resultado se computan de manera similar a como se explicaba en la parte simplificada, pero esta vez tiene en cuenta el doble de operadores (subir, bajar, parar * nº ascensores) y solamente actualiza la parte de la representación que le corresponde.
Elementos como ascensores_constante, ascensores_variable o pos_ascensor se explican con detalle a continuación en la representación del problema.

Representación

Lo primero que tenemos en cuenta al diseñar la representación del problema es que un ascensor, ya que recorre un bloque, puede cubrir la representación del mismo en el contexto de dónde se encuentra y a quién lleva. Con esto establecido solo tenemos que enfocarnos en la representación de los pasajeros en cada estado indicando su planta y no necesitaremos especificar los bloques. Abstraemos el problema y cada elemento único lo representamos en una tupla: pasajeros y ascensores.

Problema simplificado

Como podemos ver a continuación, en la primera tupla se encuentran las plantas actuales de cada pasajero en el problema, mientras que en la segunda representamos el ascensor.

```
estado_inicial = ((0,0,1,0,2), (0, (), 0))
```

Imagen 3.1

El ascensor cuenta con tres elementos diferenciados que son: la planta en la que se encuentra, los pasajeros que tiene dentro y si está subiendo, parado o bajando. Los pasajeros que se encuentran en el ascensor están identificados por un número único que indica su posición en la primera tupla. Así, cuando se encuentren los pasajeros 1 y 4 en el ascensor la tupla se vería así: (_,(1,4),_) y las plantas de los pasajeros serían la 0 y la 2. La siguiente imagen aclara el concepto:

Imagen 3.2

Problema completo

De cara al problema con todos los elementos requeridos y para facilitar la adición o eliminación de nuevas componentes ya sean usuarios o ascensores, de un tipo u otro, nos enfocamos en una representación que diferencie entre todas las partes características de cada elemento que sean invariables y aquellas que, por la ejecución de los algoritmos, deberán variar para representar diferentes estados.

```
asc = Ascensores(((0,5,0,5,1), (((1,2,0,4), (1,2,4,8)), (((0,),0,0),((3,1),5,1)))))
```

Imagen 3.4

La primera tupla es igual a la utilizada en el problema simplificado y representa las plantas actuales de los usuarios. Como podemos observar, independientemente del bloque en que se encuentren los pasajeros, todos se representan en la misma lista. Es en los ascensores donde encontramos un cambio. En primer lugar aclaramos que un **ascensor se representará en dos tuplas**, su bloque será el número de las tuplas que debemos buscar (el primer ascensor se representa en la primera tupla de la izquierda y en la primera tupla de la derecha, ambas empezando desde la izquierda). Debido a que en el problema se habla de ascensor rápido y ascensor normal, y que de algún modo hemos de indicar las plantas que cubre y la capacidad decidimos meter esto en una lista que formará la **parte constante** de los ascensores ya que no variará durante la resolución del problema. Esta parte son las dos primeras tuplas después de la de los usuarios, desglosémosla.

El primer elemento nos indica qué tipo de ascensor es: 1 para normal, 2 para rápido. `(1,2,4,8)`

Imagen 3.5

El segundo elemento nos dice que el ascensor tiene capacidad para 2 personas. Por último el tercer y cuarto elemento, i.e., el 4 y el 8 son las plantas entre las que puede circular el ascensor, en este caso indican que es el ascensor del bloque 2.

¿Y qué es lo demás? Las otras dos tuplas indican la **parte variable** de cada ascensor. Como hemos empezado mirando la parte constante del segundo ascensor, miremos la parte variable del mismo:

```
((3,1),5,1)
```

Imagen 3.6

El primer elemento contendrá a los pasajeros que se encuentran en el ascensor, de igual manera que se representaba en el problema simplificado.
El segundo elemento nos indica en qué planta se encuentra el ascensor en este estado, la 5 actualmente (es el ascensor que cubre el bloque 2, de la 4 a la 8).
El tercer elemento nos dice que la última acción que ha realizado es subir, concretamente 'Subir 1'.

Análisis

Resultados de los algoritmos

Para empezar evaluamos el problema simplificado con uno y varios usuarios para verificar la estimación de la introducción:

```
estado = ((0,), (0, (), 0))
#goal=((3,),)
```

```
%%time
resuelve_ascensores(estado,uniform_cost_search)
```

```
Solución: ['Parar', 'Subir', 'Subir', 'Subir']
Algoritmo: uniform_cost_search
Longitud de la solución: 4. Nodos analizados: 14
```

Imagen 4.1

```
estado_inicial = ((0,0,1,0,2), (0, (), 0))
#goal=((3,3,3,0,2))
```

```
%%time
resuelve_ascensores(estado_inicial,uniform_cost_search)
```

```
Solución: ['Subir', 'Parar', 'Bajar', 'Parar', 'Subir', 'Subir', 'Subir']
Algoritmo: uniform_cost_search
Longitud de la solución: 7. Nodos analizados: 51
CPU times: user 1.77 ms, sys: 84 µs, total: 1.85 ms
```

Imagen 4.2

Como podemos observar la longitud no empeora mucho, pero los nodos son cerca de cuatro veces más al añadir 4 usuarios más, lo que nos da un atisbo de cómo podrían resultar las soluciones al añadir más ascensores. A continuación evaluamos los resultados de los algoritmos para el problema simplificado con los siguientes parámetros: estado inicial = ((0,0,1,0,2), (0, (), 0))
goal=((3,3,3,0,2))

	Longitud	Nodos	Tiempo
Profundidad con grafo	7	27	1.85 ms
Profundidad con árbol	/	/	/
Anchura con grafo	7	48	1.23 ms
Anchura con árbol	/	/	/
Profundidad limitada (10)	9	1497	11.4 ms
Iterativo	7	777	6.68 ms
A* h1	7	46	2.23 ms
A* h2	7	39	1.76 ms
A* h3	7	19	1.73 ms
Primero el mejor h1	7	56	2 ms
Primero el mejor h2	10	47	1 ms
Primero el mejor h3	7	25	1 ms

Los algoritmos de búsqueda a ciegas con árboles no acaban en un tiempo razonable (30 minutos) debido a que entran en bucles ya que no hay control de repetidos, mientras que en los de grafo que tiene control de repetidos podemos ver que el algoritmo en profundidad es uno de los mejores.
Por otra parte, como discutimos en las heurísticas, los algoritmos funcionan mejor cuanto mejor definida está: con la tercera heurística los algoritmos de A* y de primero el mejor dan los mejores resultados.

En cuanto al problema completo hemos diferenciado dos casos para observar con más detalle cómo varían los costes según la complejidad del estado inicial con respecto a la solución:

estado inicial 1 = ((0,7,0,8,1), (((1,2,0,4), (1,2,4,8)), (((0,),0,0),((),5,1))))

estado inicial 2 = ((0,5,0,5,1), (((1,2,0,4), (1,2,4,8)), (((0,),0,0),((3,1),5,1))))
goal=((3,5,3,5,1))

Mostramos una imagen para que sea más comprensible:

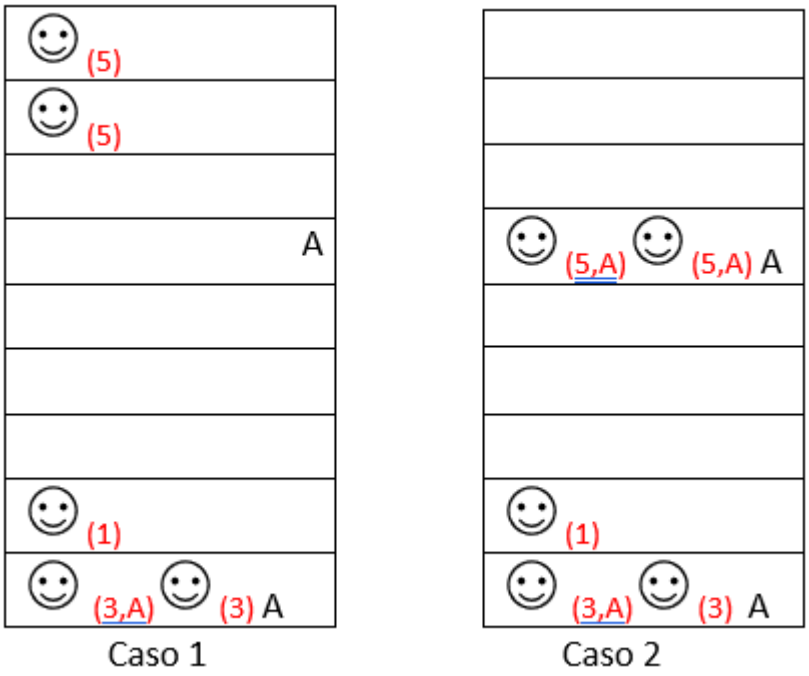

Imagen 4.3

Cada usuario está representado por una cara y entre paréntesis indicamos su planta de destino correspondiente. Aquellos que al lado de la planta de destino tienen una 'A' roja es representan que están dentro del ascensor. Asímismo se indica la planta en la que se encuentra cada ascensor con una 'A' negra.
Observemos a continuación los distintos resultados:

	Longitud	Nodos	Tiempo
Uniforme 1	12	597	93.7 ms
Uniforme 2	4	51	4 ms
Profundidad con grafo 1	32	48	3 ms
Profundidad con grafo 2	20	34	2 ms
Profundidad con árbol	/	/	/
Anchura con grafo	4	45	2.08 ms

Anchura con árbol	/	/	/
Profundidad limitada (10)	10	5272	70.4 ms
Iterativo	4	173	2.8 ms
A* h1-1	12	473	87 ms
A* h1-2	4	30	2 ms
A* h2-1	12	129	25 ms
A* h2-2	6	14	2.01 ms
A* h3-1	12	502	99.7 ms
A* h3-2	4	30	3.01 ms
Primero el mejor h1	16	297	
Primero el mejor h2	28	65	
Primero el mejor h3	20	68	

De igual manera que al aplicarlos a los problemas simplificados, aquellos algoritmos de búsqueda ciega que utilizan árboles no acaban en un tiempo razonable debido a que no diferencian estados repetidos y situaciones como que un ascensor suba y baje de manera repetida son factibles. Destacamos cómo los resultados de los algoritmos en el segundo caso mejoran notablemente con respecto al primero debido a la simplicidad de la solución. Por otra parte en el algoritmo de primero el mejor vemos una diferencia destacable entre los resultados de la primera y la última heurística. Facilitándole a través de la heurística valores más exactos de cada nodo en relación a su estado el algoritmo necesita analizar menos nodos hasta encontrar la solución.
Por el contrario el algoritmo de A* se ve perjudicado con una heurística así en cuanto a la evaluación de nodos ya que busca el camino más corto y no tienen en cuenta el algoritmo ni la heurística variables como si los usuarios están en su planta de destino pero no en el ascensor o si los pasajeros que suben llevan todos la misma dirección. Al favorecer llenar el ascensor, éste no siempre hace las rutas óptimas.