Práctica 2B. Problema de los Ascensores

Boris Carballa Corredoira
Juan Carlos Villanueva Quirós
Francisco Javier Blázquez Martínez

1 de diciembre de 2019

Resumen

Para abordar este problema de elevada complejidad decidimos resolverlo en primera instancia para una versión simplificada de este: suponiendo que había un solo ascensor y un solo bloque. Debido a la explosión combinatoria, implementamos una serie de restricciones que nos permiten evitar ramas innecesarias, es decir, ramas para las que podemos encontrar una solución mejor o igual con estas restricciones. Además, añadimos una heurística que nos ayuda a realizar la búsqueda de forma guiada. De esta forma, conseguimos implementar una primera versión que resuelve el problema relajado.

Posteriormente, nos dispusimos a resolver el mismo problema, pero esta vez con varios ascensores, y tras una serie de cambios, obtuvimos una segunda versión que resolvía el problema, pero con un solo bloque. Finalmente, añadiendo el rango de cada ascensor, logramos resolver el problema original en un tiempo disminuido. A lo largo de la memoria, se relatará el proceso seguido desde la primera versión hasta la versión final, terminando con una versión paralelizada opcional, que considera la ejecución simultánea de las acciones de los ascensores.

Índice

1. Logros

⇒ Primera Versión: Conseguimos resolver el problema con un solo ascensor y un solo bloque **para una cantidad de personas y número de plantas arbitrario** en un tiempo muy reducido.

⇒ Segunda Versión: Logramos resolver el mismo problema con varios ascensores y un solo bloque **para una cantidad de personas y número de plantas medianamente grande** en un tiempo muy razonable.

⇒ Tercera Versión: Hallamos **la solución óptima**, en cuanto al número de plantas que se desplazan los ascensores, al problema propuesto con varios ascensores y varios bloques **en un tiempo considerable** y **una solución sub-óptima en un tiempo irrisorio.**

⇒ Versión Paralelizada: **Paralelizamos el problema** para obtener una solución óptima en cuanto a menor tiempo de ejecución.

2. Primera Versión

2.1. Conceptos Clave

Primera aproximación al problema: un solo bloque y un solo ascensor. Para evitar sobrecargar la memoria, **solo añadimos aquí el código de la versión final (la tercera) y la paralelizada**. No obstante, **adjuntamos aparte el código de la primera y la segunda versión por si se desea revisar**.

No comentaremos tampoco la descripción de las constantes, variables locales y clases de las primeras dos versiones, pues eso estará minuciosamente detallado en la versión final. Describimos en esta sección pues, los conceptos y restricciones que a nuestro parecer, han sido claves a la hora de abordar el problema.

Para evitar la explosión combinatoria intrínseca al problema, hemos adoptado una serie de restricciones basadas en invariantes implícitos del problema que nos permiten no dar una acción cuando sea innecesaria. Todas estas restricciones han sido probadas válidas en las hipótesis del problema (**de las cuales cabe destacar la no penalización por parada de los ascensores**) por el método de reducción de diferencias. De esta forma, conseguimos recortar múltiples ramas del árbol de exploración las cuales no nos aportan un camino útil. Enumeramos a continuación, las 4 restricciones que hemos tenido en cuenta:

1. **Un ascensor nunca tiene dentro personas que quieran subir y bajar.** O bien tiene dentro a gente que quiere bajar, o bien a gente que quiere subir. De esta forma, evitamos alejar de su objetivo innecesariamente a las personas que quieren bajar cuando el ascensor sube y viceversa

2. **Si alguien llega a su planta objetivo, forzamos a que esta baje del ascensor**, es decir, devolvemos como única acción posible dejar a dicha persona. Siempre será más óptimo dejar a la persona una vez alcance su objetivo y no alejarla de este

3. **Solo subimos cuando haya una persona esperando en una planta por encima del ascensor o cuando haya alguna persona dentro del ascensor cuya planta objetivo esté por encima.** De esta manera, nos evitamos dar la opción de subir cuando no aporte nada al problema. (Respectivamente con bajar)

4. **No subimos cuando haya un hueco en el ascensor y alguien en la planta que quiera subir.** De esta forma, obligamos a coger a esa persona antes de subir pues siempre será más eficiente acercarla si es posible. (Respectivamente con bajar)

Por otro lado, implementamos una heurística para usarla en la búsqueda *A**. Hemos elegido para esta primera versión que sea la suma para cada persona de las diferencias entre la planta en la que se encuentra y la planta objetivo dividido esto entre la constante CAP_ASCENSOR. La heurística es admisible y consistente (h(s1) <= c(s1,a,s2) + h(s2)). No dividir por esta constante implica usar una heurística no admisible (y por tanto no llegar a solución óptima necesariamente) pero obtener soluciones en un tiempo de ejecución menor.

La adopción de estas restricciones y de una heurística adecuada para la búsqueda, fueron claves para conseguir hallar soluciones al problema relajado pero para una cantidad arbitraria de personas y pisos.

2.2. Solution Manager

Para visualizar correctamente las soluciones que nos calcula el algoritmo, importamos el notebook *SolutionManager* en el cual tenemos almacenados todos los procedimientos para mejorar la visualización de la misma. Describimos pues, una breve descripción de la función que realiza cada procedimiento de dicho módulo. Una vez más, si se quiere revisar el código, puede ser consultado en los archivos adjuntos.

- ***decodificador_acciones***: Hace legible la lista de acciones
- ***quita_dejar_coger_inutiles_acciones***: Quita si hay una cadena dejar-coger o coger-dejar para un mismo ascensor y persona sin que ese ascensor suba o baje en el medio
- ***fusionador_acciones***: Junta dos acciones iguales seguidas:
    1. Si un mismo ascensor coge (o deja persona) a m y n seguidamente: aparecerá que se cogió a las personas m,n.
    2. Si un mismo ascensor sube (o baja) a m y a n seguidamente: aparecerá que subió m+n plantas.
- ***cocina_acciones***: Reduce la lista de acciones lo máximo posible
- ***coste_acciones***: Calcula el coste secuencial, esto es, la suma del número de plantas que recorren en total los ascensores

2.3. Resultados

Describimos aquí, los resultados que obtuvimos en esta primera versión del problema. Si se quisiera probar otro casos distintos, simplemente debemos cambiar el nodo init con las plantas donde se encuentran al comienzo cada persona y la lista goal con las plantas a las que quieren ir. Declaramos el problema con dichos nodos y ejecutamos la búsqueda. Posteriormente, decodificamos la lista de acciones obtenidas (Módulo SolutionManager) para visualizarla cómodamente con una interpretación y no como tuplas codificadas.

Como podemos observar, logramos hallar una solución al problema para una gran cantidad de personas en un tiempo muy disminuido.

[12]:
```
#Caso sencillo
init = NodeState([0,1])
goal = [1,0]
problem = AscensoresSuperSimplificado(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
decodificador_acciones(fusionador_acciones(acciones))
```

Solución

El ascensor 0 coge a la persona 0
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 0
El ascensor 0 coge a la persona 1
El ascensor 0 baja 1 planta
El ascensor 0 deja a la persona 1

```
Nodos analizados: 7
Coste secuencial: 2
Tiempo de búsqueda: 0.000997304916381836 s
```

[14]:
```
#Caso del enunciado
init = NodeState([2,4,1,8,1])
goal = [3,11,12,1,9]
problem = AscensoresSuperSimplificado(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
decodificador_acciones(fusionador_acciones(acciones))
```

Solución

El ascensor 0 sube 1 planta
El ascensor 0 coge a las personas 4, 2
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 4
El ascensor 0 coge a la persona 4
El ascensor 0 sube 6 plantas
El ascensor 0 deja a la persona 2
El ascensor 0 coge a la persona 2
El ascensor 0 sube 1 planta
El ascensor 0 deja a las personas 4, 2
El ascensor 0 baja 1 planta
El ascensor 0 coge a la persona 3
El ascensor 0 baja 7 plantas
El ascensor 0 deja a la persona 3
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 0
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 0
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 1
El ascensor 0 sube 5 plantas
El ascensor 0 coge a la persona 2
El ascensor 0 sube 2 plantas
El ascensor 0 deja a la persona 1
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 2

```
Nodos analizados: 670
Coste secuencial: 28
Tiempo de búsqueda: 0.24212026596069336 s
```

[23]:
```
#Caso 10 personas, 12 plantas:
init = NodeState([10, 8, 1, 12, 7, 9, 5, 11, 6, 2])
goal = [7, 3, 4, 5, 6, 10, 2, 4, 9, 7]
problem = AscensoresSuperSimplificado(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
decodificador_acciones(fusionador_acciones(acciones))
```

Solución

El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 2
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 9
El ascensor 0 deja a la persona 2
El ascensor 0 coge a la persona 2
El ascensor 0 sube 2 plantas
El ascensor 0 deja a la persona 2
El ascensor 0 sube 2 plantas
El ascensor 0 coge a la persona 8
El ascensor 0 deja a la persona 9
El ascensor 0 coge a la persona 9
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 9
El ascensor 0 sube 2 plantas
El ascensor 0 deja a la persona 8
El ascensor 0 coge a la persona 5
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 5
El ascensor 0 sube 2 plantas
El ascensor 0 coge a la persona 3
El ascensor 0 baja 1 planta
El ascensor 0 coge a la persona 7
El ascensor 0 baja 6 plantas
El ascensor 0 deja a la persona 3
El ascensor 0 coge a la persona 6
El ascensor 0 deja a la persona 7
El ascensor 0 coge a la persona 7
El ascensor 0 baja 1 planta
El ascensor 0 deja a las personas 7, 6
El ascensor 0 sube 6 plantas
El ascensor 0 coge a la persona 0
El ascensor 0 baja 2 plantas
El ascensor 0 coge a la persona 1
El ascensor 0 deja a la persona 0
El ascensor 0 coge a la persona 0
El ascensor 0 baja 1 planta
El ascensor 0 deja a la persona 0
El ascensor 0 coge a la persona 4
El ascensor 0 deja a la persona 1
El ascensor 0 coge a la persona 1
El ascensor 0 baja 1 planta
El ascensor 0 deja a la persona 4
El ascensor 0 baja 2 plantas
El ascensor 0 coge a la persona 6
El ascensor 0 baja 1 planta
El ascensor 0 deja a la persona 1
El ascensor 0 baja 1 planta
El ascensor 0 deja a la persona 6

```
Nodos analizados: 5934
Coste secuencial: 34
Tiempo de búsqueda: 6.5752503871917725 s
```

Casos de alta complejidad

Los casos siguientes resultan demasiado lentos para la heurística consistente. Sin embargo si renunciamos a obtener siempre la solución óptima (en términos de minimizar el número de plantas que tiene que desplazarse) obtenemos soluciones muy rápidas. Se debe usar la siguiente heurística:

[18]:
```
def h(node):
    suma  = sum([abs(goal[i]-node.state.personas[i]) for i in range(0,len(node.
 ↪state.personas)) if node.state.personas[i]!=EN_DESTINO and node.state.
 ↪personas[i]!=EN_ASCENSOR])
    suma += sum([abs(goal[persona]-node.state.ascensor_pos) for persona in node.
 ↪state.ascensor_personas])
    return suma
```

Para los casos siguientes, añadimos aquí solo los tiempos y nodos analizados pues la solución es muy extensa. Puede ser consultada la solución detallada en el archivo adjunto.

[19]:
```
#Caso 15 personas, 20 plantas:
init = NodeState([10, 8, 1, 12, 7, 9, 5, 11, 6, 2, 4, 18, 2, 7, 5])
goal = [7, 3, 12, 5, 6, 10, 2, 4, 9, 11, 16, 2, 11, 12, 18]
problem = AscensoresSuperSimplificado(init, goal)
start = time.time()
acciones = astar_search(problem,h).solution()
end = time.time()
decodificador_acciones(fusionador_acciones(acciones))
```

```
Nodos analizados: 93
Coste secuencial: 80
Tiempo de búsqueda: 0.02878546714782715 s
```

[21]:
```
#Caso 30 personas, 100 plantas:
init = NodeState([34, 12, 25, 51, 30, 83, 29, 3, 17, 66, 47, 89, 92, 0, 71, 35,␣
 ↪96, 69, 6, 40, 67, 93, 24, 97, 63, 79, 19, 62, 99, 27])
goal = [21, 92, 18, 28, 35, 68, 74, 81, 89, 27, 24, 70, 79, 61, 8, 91, 7, 63,␣
 ↪37, 99, 72, 58, 59, 73, 25, 66, 5, 86, 34, 83]
problem = AscensoresSuperSimplificado(init, goal)
start = time.time()
acciones = astar_search(problem,h).solution()
end = time.time()
decodificador_acciones(fusionador_acciones(acciones))
```

```
Nodos analizados: 246
Coste secuencial: 771
Tiempo de búsqueda: 0.1365680694580078 s
```

Conclusiones

La definición de las acciones, muy restrictiva gracias a considerar los invariantes implícitos del problema, junto con el control de nodos repetidos del algoritmo, nos ha permitido conseguir una versión del problema que nos lo solucione para una cantidad numerosa de personas en un tiempo razonable. Notemos que no tenemos que indicar al problema el número de plantas a considerar y por lo tanto, podemos poner a cada persona en una planta arbitraria.

3. Segunda Versión

Segunda aproximación al problema: consta de un único bloque, pero esta vez con varios ascensores.

Para conseguir incluir varios ascensores, tuvimos que introducir breves cambios a esta. Principalmente fueron:

- Devolver ahora **para cada ascensor** las acciones factibles en la lista de acciones posibles
- Sustituir algunas de las variables de la primera versión por listas que posibiliten considerar varios ascensores.

De esta manera, conseguimos extender el problema fácilmente a varios ascensores sin perder mucha eficiencia debido a la muy buena primera aproximación del problema.

3.1. Resultados

Describimos aquí, los resultados que obtuvimos en esta segunda versión del problema. Si se quisiera probar otro casos distintos, simplemente debemos cambiar el nodo init con las plantas donde se encuentran al comienzo cada persona y la lista goal con las plantas a las que quieren ir. Además si queremos modificar el número de ascesnores, cambiamos la constante *NUM_ASCENSORES*. Declaramos el problema con dichos nodos y ejecutamos la búsqueda. Posteriormente, decodificamos la lista de acciones obtenidas (Módulo SolutionManager) para visualizarla cómodamente con una interpretación y no como tuplas codificadas.

Vamos a probar los casos de prueba de la versión anterior pero esta vez usando varios ascensores: 2. Como podemos observar, logramos hallar una solución al problema para una gran cantidad de personas y varios ascensores en un tiempo disminuido.

[129]:
```
#Caso sencillo
init = NodeState([3,1])
goal = [4,0]
problem = AscensoresSimplificado(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
decodificador_acciones(cocina_acciones(acciones))
```

Solución

El ascensor 0 sube 3 plantas
El ascensor 0 coge a la persona 0
El ascensor 1 sube 1 planta
El ascensor 1 coge a la persona 1
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 0
El ascensor 1 baja 1 planta
El ascensor 1 deja a la persona 1

```
Nodos analizados: 49
Coste secuencial: 6
Tiempo de búsqueda: 0.0074613094329833984 s
```

[140]:
```
#Caso del enunciado
init = NodeState([2,4,1,8,1])
goal = [3,11,12,1,9]
problem = AscensoresSimplificado(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
decodificador_acciones(cocina_acciones(acciones))
```

Solución

El ascensor 0 sube 1 planta
El ascensor 0 coge a las personas 4, 2
El ascensor 0 sube 8 plantas
El ascensor 0 deja a las personas 4, 2
El ascensor 0 baja 1 planta
El ascensor 0 coge a la persona 3
El ascensor 0 baja 7 plantas
El ascensor 0 deja a la persona 3
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 0
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 0
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 1
El ascensor 0 sube 5 plantas
El ascensor 0 coge a la persona 2
El ascensor 0 sube 2 plantas
El ascensor 0 deja a la persona 1
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 2

```
Nodos analizados: 8483
Coste secuencial: 28
Tiempo de búsqueda: 42.44042134284973 s
```

Casos de alta complejidad

Los casos siguientes resultan demasiado lentos para la heurística consistente. Sin embargo si renunciamos a obtener siempre la solución óptima (en términos de minimizar el número de plantas que tiene que desplazarse) obtenemos soluciones más rápidas. Se debe usar la siguiente heurística:

[144]:
```
def h(node):
    suma = sum([abs(goal[i]-node.state.personas[i]) for i in range(0,len(node.
 ↪state.personas)) if node.state.personas[i]!=EN_DESTINO and node.state.
 ↪personas[i]!=EN_ASCENSOR])
```

```
    for ascensor in range(0, NUM_ASCENSORES):
        suma += sum([abs(goal[persona]-node.state.ascensor_pos[ascensor]) for␣
↪persona in node.state.ascensor_personas[ascensor]])
    return suma
```

[145]:
```
#Caso 10 personas, 12 plantas:
init = NodeState([10, 8, 1, 12, 7, 9, 5, 11, 6, 2])
goal = [7, 3, 4, 5, 6, 10, 2, 4, 9, 7]
problem = AscensoresSimplificado(init, goal)
start = time.time()
acciones = astar_search(problem,h).solution()
end = time.time()
decodificador_acciones(cocina_acciones(acciones))
```

Solución

El ascensor 1 sube 1 planta
El ascensor 1 coge a la persona 2
El ascensor 1 sube 1 planta
El ascensor 1 coge a la persona 9
El ascensor 1 sube 2 plantas
El ascensor 1 deja a la persona 2
El ascensor 1 sube 2 plantas
El ascensor 1 coge a la persona 8
El ascensor 1 sube 1 planta
El ascensor 1 deja a la persona 9
El ascensor 1 sube 2 plantas
El ascensor 1 deja a la persona 8
El ascensor 1 coge a la persona 5
El ascensor 1 sube 1 planta
El ascensor 1 deja a la persona 5
El ascensor 1 sube 2 plantas
El ascensor 1 coge a la persona 3
El ascensor 1 baja 1 planta
El ascensor 1 coge a la persona 7
El ascensor 1 baja 6 plantas
El ascensor 1 deja a la persona 3
El ascensor 1 coge a la persona 6
El ascensor 1 baja 1 planta
El ascensor 1 deja a las personas 7, 6
El ascensor 1 sube 6 plantas
El ascensor 1 coge a la persona 0
El ascensor 1 baja 2 plantas
El ascensor 1 coge a la persona 1
El ascensor 1 baja 1 planta
El ascensor 1 deja a la persona 0
El ascensor 1 coge a la persona 4
El ascensor 1 baja 1 planta
El ascensor 1 deja a la persona 4
El ascensor 1 baja 2 plantas
El ascensor 1 coge a la persona 6
El ascensor 1 baja 1 planta
El ascensor 1 deja a la persona 1
El ascensor 1 baja 1 planta
El ascensor 1 deja a la persona 6

```
Nodos analizados: 4222
Coste secuencial: 34
Tiempo de búsqueda: 27.19186043739319 s
```

Conclusiones

Para poder considerar la actuación de varios ascensores de forma secuencial, únicamente hemos tenido que modificar las variables del estado interno de los nodos que explorábamos haciendo que las variables antes presentes para cada ascensor se conviertan en una lista con estas variables, una para cada ascensor. Asimismo, las acciones se generan de forma idéntica que antes pero para cada uno de los ascensores presentes.

4. Tercera Versión

Solución final del problema sin paralelizar, el edificio consta de varios bloque con varios ascensores. En esta versión, al ser la solución final al problema adjuntamos todo el código. Para añadir varios bloques, vamos a añadir un rango a cada ascensor, que serán las plantas entre las que se podrán mover. Notemos que esto es equivalente a tener varios bloques.

[16]:
```
from search import *
import copy
import time
```

4.1. Constantes

- *NUM_PLANTAS*: Numero de plantas del problema
- *NUM_ASCENSORES*: Numero de ascensores a tener en cuenta
- *NUM_PERSONAS*: Numero de personas a considerar. Se necesita para que el *SolutionManager* muestra correctamente la solucion con varios ascensores
- *CAP_ASCENSOR*: Capacidad de los ascensores
- *EN_ASCENSOR*: Nos indica que una persona se encuentra dentro de un ascensor
- *NINGUNA*: Usada para indicar que una lista es vacia
- *EN_DESTINO*: Denota que una persona ha llegado a su destino

[43]:
```
NUM_PLANTAS    = 12
NUM_ASCENSORES = 4
NUM_PERSONAS = 5
CAP_ASCENSOR   = 2
EN_ASCENSOR = -1
NINGUNA = -1
EN_DESTINO = -2
```

4.2. Solution Manager

Importamos el notebook *SolutionManager* en el cual tenemos almacenados todos los procedimientos que se ejecutan una vez calculada la solución para mejorar la visualización de la misma.

[23]:
```
%run SolutionManager.ipynb
```

4.3. Clase NodeState

Clase que representa un nodo del espacio de exploración. El estado de dicho nodo se compone de:

- *personas*: Lista con la planta en la que se encuentra cada persona, si una persona se encuentra dentro del ascensor toma el valor de la constante EN_ASCENSOR, si se encuentra en su destino toma el valor de la costante EN_DESTINO

- *ascensor_pos*: Lista con las plantas en la que se encuentra cada ascensor
- *ascensor_personas*: Lista de listas con las personas (índices de la lista "personas") que se encuentran dentro de cada ascensor
- *ascensor_rango*: Lista de tuplas (x,y) que índican el rango de cada ascensor, donde x es la planta mínima e y la planta máxima. Si no se introduce ningún rango, suponemos por defecto que pueden ir a todas las plantas

Por otro lado, hemos definido los operadores igual (**eq**), menor que (**lt**) y **hash** para permitir la comparación entre nodos y posibilitar así la correcta inserción de estos en estructuras de datos para que las búsquedas usen control de repetidos.

[24]:
```
class NodeState:

    def __init__(self, init_personas, ascensor_rango=None):
        self.personas = init_personas
        self.ascensor_personas = [[] for _ in range(NUM_ASCENSORES)]
        if ascensor_rango is None:
            self.ascensor_rango = [(0,NUM_PLANTAS) for _ in␣
↪range(NUM_ASCENSORES)] #[min,max], te deja lo mas arriba/abajo que puede
            self.ascensor_pos = [0 for _ in range(NUM_ASCENSORES)]
        else:
            self.ascensor_rango = ascensor_rango
            self.ascensor_pos = [rango[0] for rango in ascensor_rango]

    def __eq__(self, nodo):
        return self.personas == nodo.personas and self.ascensor_pos == nodo.
↪ascensor_pos and self.ascensor_personas == nodo.ascensor_personas

    def __hash__(self):
        return hash((tuple(self.personas), tuple(self.ascensor_pos),␣
↪tuple(tuple(x) for x in self.ascensor_personas)))

    def __lt__(self,nodo):
        return True
```

4.4. Clase Ascensores

Clase que representa el problema en su versión final sin paralelizar. Consta de varias funciones:

- **Init**: Dado un nodo inicial y un nodo objetivo, inicializa la clase
  - *initial*: nodo desde el que empezamos a buscar con los rangos de cada ascensor
  - *goal*: Plantas a las que quieren llegar las respectivas personas
  - *analizados*: número de nodos analizados hasta el momento

- **Actions**: Dado un nodo, devuelve las acciones posibles que podemos ejecutar desde dicho nodo. Las acciones posibles son: subir, bajar,coger_persona y dejar_persona. En esta versión, calculamos las acciones posibles para cada ascensor.
  - *Codificación Acciones*. Hemos decidido codificar las acciones posibles de la siguiente manera:
    - (subir, i, j) $\Rightarrow$ El ascensor i sube j plantas
    - (bajar, i, j) $\Rightarrow$ El ascensor i baja j plantas
    - (coger_persona, i, j) $\Rightarrow$ El ascensor i coge a la persona j
    - (dejar_persona, i, j) $\Rightarrow$ El ascesnro i deja a la persona j
  - *Variables locales*:
    - personas_esta_planta_suben: Lista de las personas que se encuentran en la planta del ascensor y su planta objetivo está por encima
    - personas_esta_planta_bajan: Lista de las personas que se encuentran en la planta del ascensor y su planta objetivo está por debajo
    - persona_menos_arriba: Menor planta de las personas que se encuentran por encima del ascensor y dentro de su rango
    - persona_menos_abajo: Mayor planta de las personas que se encuentran por debajo del ascensor y dentro de su rango
    - destino_menos_arriba: Menor destino de las personas que se encuentran dentro del ascensor o bien el límite máximo del rango si todos los destinos están por encima de este
    - destino_menos_abajo: Mayor destino de las personas que se encuentran dentro del ascensor o bien el límite mínimo del rango si todos los destinos están por debajo de este
  - *Restricciones*. Hemos decidido implementar una serie de restricciones basadas en invariantes implicitos del problema que nos permiten no dar una acción cuando sea innecesario. Todas estas restricciones han sido probadas válidas en las hipótesis del problema (de las cuales cabe destacar la no penalización por parada de los ascensores) por el método de reducción de diferencias:
    1. Un ascensor nunca tiene dentro personas que quieran subir y bajar. O bien tiene dentro a gente que quiere bajar, o bien a gente que quiere subir. De esta forma, evitamos alejar de su objetivo innecesariamente a las personas que quieren bajar cuando el ascensor sube y viceversa

2. Si alguien llega a su planta objetivo, forzamos a que esta baje del ascensor, es decir, devolvemos como única acción posible dejar a dicha persona. Siempre será más óptimo dejar a la persona una vez alcance su objetivo y no alejarla de este

3. Solo subimos cuando haya una persona esperando en una planta por encima del ascensor o cuando haya alguna persona dentro del ascensor cuya planta objetivo esté por encima. De esta manera, nos evitamos dar la opción de subir cuando no aporte nada al problema. (Respectivamente con bajar)

4. No subimos cuando haya un hueco en el ascensor y alguien en la planta que quiera subir. De esta forma, obligamos a coger a esa persona antes de subir pues siempre será más eficiente acercarla si es posible. (Respectivamente con bajar)

- **Result**. Dado un nodo y una acción, devuelve el resultado de aplicar al nodo dicha acción

- **Goal_test**. Dado un nodo, comprueba si hemos alcanzado la solución. Habremos alcanzado el objetivo cuando todas las personas tengan el valor EN_DESTINO

- **path_cost**. Si la acción que aplicamos es *subir* o *bajar*, incrementamos el coste en el número de plantas que subimos o bajamos. El coste total será la suma de las plantas que subimos o bajamos

- **h**. Heurística a usar en la búsqueda *A**. Hemos elegido para esta primera versión que sea la suma para cada persona de las diferencias entre la planta en la que se encuentra y la planta objetivo dividido esto entre la constante CAP_ASCENSOR. La heurística es admisible y consistente (h(s1) <= c(s1,a,s2) + h(s2)). No dividir por esta constante implica usar una heurística no admisible (y por tanto no llegar a solución óptima necesariamente) pero obtener soluciones en un tiempo de ejecución menor.

[44]:
```
class Ascensores(Problem) :

    def __init__(self, initial, goal=None):
        self.initial = initial
        self.goal = goal
        self.analizados = 0

    def actions(self, state):

        accs = list()

        for num_ascensor in range(0,NUM_ASCENSORES):
            # Parámetros relativos al ascensor a evaluar
            ascensor_pos      = state.ascensor_pos[num_ascensor]      # Planta␣
 ↪del ascensor en cuestión
            ascensor_rango    = state.ascensor_rango[num_ascensor]
            ascensor_personas = state.ascensor_personas[num_ascensor] # Personas␣
 ↪dentro del ascensor a tratar
```

```
            personas          = state.personas

            ascensor_vacio    = not ascensor_personas
            ascensor_lleno    = (len(ascensor_personas)==CAP_ASCENSOR)

            personas_esta_planta_suben = [i for i in range(len(personas)) if␣
↪personas[i]==ascensor_pos and self.goal[i]>ascensor_pos]
            personas_esta_planta_bajan = [i for i in range(len(personas)) if␣
↪personas[i]==ascensor_pos and self.goal[i]<ascensor_pos]

            # 1.- Si alguien ha llegado a su destino, o bien, si su destino está␣
↪por encima
            #     o por debajo del rango y si se ha llegado al límite de este,␣
↪entonces
            #     forzamos que baje del ascensor, devolviendo como unica accion␣
↪posible DEJAR PERSONA. (Restriccion 2)
            for persona in ascensor_personas:
                if self.goal[persona]==ascensor_pos or \
                (self.goal[persona]<ascensor_rango[0] and␣
↪ascensor_pos==ascensor_rango[0]) or \
                (self.goal[persona]>ascensor_rango[1] and␣
↪ascensor_pos==ascensor_rango[1]):
                    return [(DEJAR_PERSONA,num_ascensor,persona)]


            # 2.- SUBIR:

            persona_menos_arriba = min([person for person in personas if␣
↪ascensor_pos<person and person<=ascensor_rango[1] and person!=EN_ASCENSOR and␣
↪person!=EN_DESTINO], default=NINGUNA)

            # Si el ascensor está vacío sólo subimos si hay alguna persona por␣
↪arriba
            # y no hay gente en esta planta que quiera subir (Restriccion 3 y 4)
            if ascensor_vacio:
                if persona_menos_arriba!=NINGUNA and␣
↪not(personas_esta_planta_suben):
                    accs.
↪append((SUBIR,num_ascensor,persona_menos_arriba-ascensor_pos))

            # Si el ascensor no está vacío, comprobamos que sus personas suben y␣
↪vamos
            # al mínimo entre a donde suben estas, donde hay una persona por␣
↪encima y el rango superior
            # siempre que no nos quede espacio o no haya gente en esa planta que␣
↪quiere subir
```

```
            # en cuyo caso cogeremos a dichar persona para acercarla lo maximo␣
↪posible (Restriccion 4)
            elif self.goal[ascensor_personas[0]]>ascensor_pos:
                destino_menos_arriba = min(ascensor_rango[1], min([self.goal[i]␣
↪for i in ascensor_personas]))

                if ascensor_lleno or not(personas_esta_planta_suben):
                    if persona_menos_arriba==NINGUNA:
                        accs.
↪append((SUBIR,num_ascensor,destino_menos_arriba-ascensor_pos))
                    else:
                        accs.
↪append((SUBIR,num_ascensor,min(persona_menos_arriba,␣
↪destino_menos_arriba)-ascensor_pos))


            # 3.- BAJAR. La acción bajar es totalmente dual a la acción subir,␣
↪mismas lógica en
            #     las condiciones.

            persona_menos_abajo = max([person for person in personas if␣
↪ascensor_pos>person and person>=ascensor_rango[0] and person!=EN_ASCENSOR and␣
↪person!=EN_DESTINO], default=NINGUNA)

            # Si el ascensor está vacío sólo bajamos si hay alguna persona por␣
↪abajo
            if ascensor_vacio:
                if persona_menos_abajo!=NINGUNA and␣
↪not(personas_esta_planta_bajan):
                    accs.
↪append((BAJAR,num_ascensor,ascensor_pos-persona_menos_abajo))
            elif self.goal[ascensor_personas[0]]<ascensor_pos:
                destino_menos_abajo = max(ascensor_rango[0], max([self.goal[i]␣
↪for i in ascensor_personas]))

                if ascensor_lleno or not(personas_esta_planta_bajan):
                    if persona_menos_abajo==NINGUNA:
                        accs.
↪append((BAJAR,num_ascensor,ascensor_pos-destino_menos_abajo))
                    else:
                        accs.
↪append((BAJAR,num_ascensor,ascensor_pos-max(persona_menos_abajo,␣
↪destino_menos_abajo)))

            # 4.- DEJAR PERSONA: Para cada persona del ascensor, podemos dejarla
            for personaInterior in ascensor_personas:
```

```
                accs.append((DEJAR_PERSONA,num_ascensor,personaInterior))

            # 5. COGER PERSONA. En el interior de ascensor todos suben, o bajan,␣
↪no ambas (Restriccion 1)
            if not ascensor_lleno:
                #Si el ascensor esta vacio, podemos coger a cualquiera, suban o␣
↪bajen
                if ascensor_vacio:
                    if ascensor_pos != ascensor_rango[1]:
                        for persona in personas_esta_planta_suben:
                            accs.append((COGER_PERSONA,num_ascensor,persona))
                    if ascensor_pos != ascensor_rango[0]:
                        for persona in personas_esta_planta_bajan:
                            accs.append((COGER_PERSONA,num_ascensor,persona))
                else:
                    #si tiene a alguien dentro, ese alguien va hacia adentro del␣
↪rango, si no forzábamos su salida en 1.-
                    #Si el ascensor tiene a alguien que baja, solo cogemos a␣
↪gente que baje
                    if self.goal[ascensor_personas[0]]<ascensor_pos:
                        for persona in personas_esta_planta_bajan:
                            accs.append((COGER_PERSONA,num_ascensor,persona))
                    else:
                    #Si el ascensor tiene a alguien que sube, solo cogemos a␣
↪gente que suba
                        for persona in personas_esta_planta_suben:
                            accs.append((COGER_PERSONA,num_ascensor,persona))

        return accs

    def result(self, state, action):
        estado_nuevo = copy.deepcopy(state)

        ascensor = action[1]

        #Si bajamos, la nueva posicion del ascensor será la actual menos las␣
↪plantas a bajar
        if action[0]==BAJAR:
            estado_nuevo.ascensor_pos[ascensor] -= action[2]
        #Si subimos, la nueva posicion del ascensor será la actual más las␣
↪plantas a subir
        elif action[0]==SUBIR:
            estado_nuevo.ascensor_pos[ascensor] += action[2]
        #Si cogemos persona, introducimos en ascensor_personas la persona que␣
↪cogemos. Además indicamos en
        #personas que se encuentra EN_ASCENSOR
```

```
        elif action[0]==COGER_PERSONA:
            estado_nuevo.ascensor_personas[ascensor].append(action[2])
            estado_nuevo.personas[action[2]] = EN_ASCENSOR
        #Si dejamos persona, la quitamos de ascensor_personas. Si ha llegado a␣
↪su destino, ponemos EN_DESTINO
        #y si no, la nueva planta en la que se encuentra (la del ascensor)
        else:
            estado_nuevo.ascensor_personas[ascensor].remove(action[2])

            if self.goal[action[2]]==state.ascensor_pos[ascensor]:
                estado_nuevo.personas[action[2]] = EN_DESTINO
            else:
                estado_nuevo.personas[action[2]] = state.ascensor_pos[ascensor]

        return estado_nuevo

    def goal_test(self, state):
        self.analizados +=1
        return state.personas == [EN_DESTINO for _ in state.personas]

    # El coste es el número de plantas que se desplaza
    def path_cost(self, c, state1, action, state2):
        if action[0]==BAJAR or action[0]==SUBIR:
            return c + action[2]
        else:
            return c

    # Heurística que garantiza solución óptima, eliminando la división por la
    # constante CAP_ASCENSOR obtenemos una heurística que alcanza solución
    # (aunque no necesariamente óptima) en un menor tiempo
    def h(self,node):
        suma = sum([abs(self.goal[i]-node.state.personas[i]) for i in␣
↪range(0,len(node.state.personas)) if node.state.personas[i]!=EN_DESTINO and␣
↪node.state.personas[i]!=EN_ASCENSOR])
        for ascensor in range(0, len(node.state.ascensor_personas)):
            suma += sum([abs(self.goal[persona]-node.state.
↪ascensor_pos[ascensor]) for persona in node.state.ascensor_personas[ascensor]])
        return suma/CAP_ASCENSOR

    def value(self, state):
        raise NotImplementedError
```

4.5. Resultados

Describimos aquí, los resultados que obtuvimos en esta versión final del problema. Si se quisiera probar otro casos distintos, simplemente debemos cambiar el nodo init con las plantas donde se encuentran al comienzo cada persona y los rangos de cada ascensor a considerar, y la lista goal con las plantas a las que quieren ir. Además si queremos modificar el número de ascesnores, cambiamos la constante *NUM_ASCENSORES*. Declaramos el problema con dichos nodos y ejecutamos la búsqueda. Posteriormente, decodificamos la lista de acciones obtenidas (Módulo SolutionManager) para visualizarla cómodamente con una interpretación y no como tuplas codificadas.

Vamos a probar los casos de prueba de la versión anterior pero esta vez usando rangos con los ascensores. Como podemos observar, logramos hallar una solución al problema para una gran cantidad de personas, varios ascensores y varios rangos en un tiempo muy disminuido.

[41]:
```
#Caso sencillo
init = NodeState([2,0],[(0,3),(3,4)])
goal = [4,1]
problem = Ascensores(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
decodificador_acciones(cocina_acciones(acciones))
```

Solución

El ascensor 0 coge a la persona 1
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 1
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 0
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 0
El ascensor 1 coge a la persona 0
El ascensor 1 sube 1 planta
El ascensor 1 deja a la persona 0

```
Nodos analizados: 11
Coste secuencial: 4
Tiempo de búsqueda: 0.0010249614715576172 s
```

[45]:
```
#Caso del enunciado
init = NodeState([2,4,1,8,1],[(0,4),(4,8),(4,8),(8,12)])
goal = [3,11,12,1,9]
problem = Ascensores(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
decodificador_acciones(cocina_acciones(acciones))
```

Solución

El ascensor 0 sube 1 planta
El ascensor 0 coge a las personas 2, 4
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 2
El ascensor 0 coge a la persona 0
El ascensor 0 sube 1 planta
El ascensor 0 deja a las personas 0, 4
El ascensor 0 baja 1 planta
El ascensor 0 coge a la persona 2
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 4
El ascensor 0 sube 1 planta
El ascensor 0 deja a las personas 2, 4
El ascensor 1 coge a la persona 4
El ascensor 2 coge a las personas 1, 2
El ascensor 2 sube 4 plantas
El ascensor 2 deja a las personas 1, 2
El ascensor 2 coge a la persona 3
El ascensor 1 sube 4 plantas
El ascensor 1 deja a la persona 4
El ascensor 2 baja 4 plantas
El ascensor 2 deja a la persona 3
El ascensor 0 coge a la persona 3
El ascensor 0 baja 3 plantas
El ascensor 0 deja a la persona 3
El ascensor 3 coge a las personas 4, 2
El ascensor 3 sube 1 planta
El ascensor 3 deja a las personas 4, 2
El ascensor 3 baja 1 planta
El ascensor 3 coge a la persona 1
El ascensor 3 sube 1 planta
El ascensor 3 coge a la persona 2
El ascensor 3 sube 2 plantas
El ascensor 3 deja a la persona 1
El ascensor 3 sube 1 planta
El ascensor 3 deja a la persona 2

```
Nodos analizados: 32246
Coste secuencial: 27
Tiempo de búsqueda: 486.8533504009247 s
```

¡Resolvemos el caso del enunciado óptimamente en 486 segundos!. Sin embargo si renunciamos a obtener siempre la solución óptima (en términos de minimizar el número de plantas que tiene que desplazarse) obtenemos soluciones más rápidas. Se debe usar la siguiente heurística:

[55]:
```
def h(node):
    suma = sum([abs(goal[i]-node.state.personas[i]) for i in range(0,len(node.
 ↪state.personas)) if node.state.personas[i]!=EN_DESTINO and node.state.
 ↪personas[i]!=EN_ASCENSOR])
    for ascensor in range(0, NUM_ASCENSORES):
        suma += sum([abs(goal[persona]-node.state.ascensor_pos[ascensor]) for␣
 ↪persona in node.state.ascensor_personas[ascensor]])
    return suma
```

[53]:
```
#Caso del enunciado
init = NodeState([2,4,1,8,1],[(0,4),(4,8),(4,8),(8,12)])
goal = [3,11,12,1,9]
problem = Ascensores(init, goal)
start = time.time()
acciones = astar_search(problem, h).solution()
end = time.time()
decodificador_acciones(cocina_acciones(acciones))
```

Solución

El ascensor 1 coge a la persona 1
El ascensor 1 sube 4 plantas
El ascensor 1 deja a la persona 1
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 4
El ascensor 3 coge a la persona 1
El ascensor 3 sube 3 plantas
El ascensor 3 deja a la persona 1
El ascensor 1 coge a la persona 3
El ascensor 1 baja 4 plantas
El ascensor 1 deja a la persona 3
El ascensor 0 coge a la persona 2
El ascensor 0 sube 3 plantas
El ascensor 0 deja a las personas 4, 2
El ascensor 2 coge a las personas 4, 2
El ascensor 2 sube 4 plantas
El ascensor 2 deja a las personas 4, 2
El ascensor 0 coge a la persona 3
El ascensor 0 baja 3 plantas
El ascensor 0 deja a la persona 3
El ascensor 3 baja 3 plantas
El ascensor 3 coge a las personas 4, 2
El ascensor 3 sube 1 planta
El ascensor 3 deja a la persona 4
El ascensor 3 sube 3 plantas
El ascensor 3 deja a la persona 2
El ascensor 0 sube 1 planta
El ascensor 0 coge a la persona 0
El ascensor 0 sube 1 planta
El ascensor 0 deja a la persona 0

```
Nodos analizados: 77
Coste secuencial: 31
Tiempo de búsqueda: 0.021969318389892578 s
```

¡Hallamos una solución sub-óptima en el caso del enunciado en 0.02 segundos!

4.6. Conclusiones

Gracias a nuestra representación interna de los estados de los nodos de exploración (clase NodeState), añadir los rangos de los desplazamiento de los ascensores ha consistido únicamente en añadir esta información a la propia clase NodeState y tener en cuenta en la generación de acciones esta restricción del movimiento de los ascensores.

5. Versión Paralelizada

Solución final del problema paralelizado, el edificio consta de varios bloques con varios ascensores, donde los ascensores son considerados como elementos independientes capaces de ejecutar sus acciones simultáneamente.

Para paralelizar, la idea fundamental será añadir el tiempo de llegada en el estado, tanto para las personas como para los ascensores. Esto nos permitirá ir marcando el momento en el que sucede cada acción y cambiar el punto de vista: buscar no ya la solución en la que los ascensores se desplazan menos plantas, sino aquella en la que es mínimo el tiempo de la persona que más tarda en llegar a su destino.

Añadimos pues, **dos atributos nuevos** a la clase NodeState:

1. personas_tiempo_llegada: Lista con el instante en el que la persona i-ésima llegó a su planta, guardada en personas[i].
2. ascensores_tiempo_llegada: Lista con el instante en el que el ascensor i-ésimo llegó a su planta actual, almacenada en ascensor_pos[i].

Por otro lado, debemos implantar cambios también en las funciones *Result* y *path_cost*.

- **Result**: Ahora esta función también involucra actualizar como es debido las nuevas variables de marcaje temporal de los distintos elementos.
- **path_cost**: Mientras que en las versiones anteriores el coste era el número de plantas que se hacía subir o bajar al ascensor sobre el que se ejecutaba la acción, ahora, al contar con la ejecución paralela, únicamente tenemos en cuenta el ascensor más lento. El método *path_cost* tiene en cuenta los tiempos de todos los ascensores, no únicamente del que ejecuta la acción. Así, acciones como **subir** y **bajar** un determinado ascensor puede tener coste cero siempre y cuando sea plenamente paralelizable.

5.1. Solution Manager Paralelizado

Para visualizar correctamente las soluciones paralelizadas que nos calcula el algoritmo, importamos el notebook *SolutionManagerParalelo* en el cual tenemos almacenados todos los procedimientos para mejorar la visualización de la misma. No describimos cada procedimiento de dicho módulo debido a que es similar a la proporcionada anteriormente en el *Solution Manager*. Aún así, insistimos en que si se quiere consultar el código, lo adjuntamos aparte.

5.2. Resultados

[75]:
```
#Caso sencillo

NUM_PLANTAS    = 4
NUM_ASCENSORES = 2
NUM_PERSONAS = 2
CAP_ASCENSOR   = 2

init = NodeState([2,0],[(0,3),(3,4)])
goal = [4,1]
problem = AscensoresParalelizados(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
acciones_ampliadas = ampliadas_acciones_paralelas(init,acciones)
print("Acciones de los ascensores")
print("Ordenadas por ascensor y momento en el que se realizan")
print()
decodificador_acciones_paralelas(cocina_acciones_paralelas(acciones_ampliadas))
```

Solución Acciones de los ascensores Ordenadas por ascensor y momento en el que se realizan

$[0] \mapsto Elascensor0cogealapersona1$
$[0,1] \mapsto Elascensor0sube1planta$
$[1] \mapsto Elascensor0dejaalapersona1$
$[1,2] \mapsto Elascensor0sube1planta$
$[2] \mapsto Elascensor0cogealapersona0$
$[2,3] \mapsto Elascensor0sube1planta$
$[3] \mapsto Elascensor0dejaalapersona0$
$[0,3] \mapsto Elascensor1espera$
$[3] \mapsto Elascensor1cogealapersona0$
$[3,4] \mapsto Elascensor1sube1planta$
$[4] \mapsto Elascensor1dejaalapersona0$

```
Nodos analizados: 14
Coste secuencial: 4
Coste paralelo: 4
Tiempo de búsqueda: 0.000997304916381836 s
```

[77]:
```
#Caso del enunciado

NUM_PLANTAS    = 12
NUM_ASCENSORES = 4
NUM_PERSONAS = 5
CAP_ASCENSOR   = 2

init = NodeState([2,4,1,8,1],[(0,4),(4,8),(4,8),(8,12)])
goal = [3,11,12,1,9]
problem = AscensoresParalelizados(init, goal)
start = time.time()
acciones = astar_search(problem).solution()
end = time.time()
acciones_ampliadas = ampliadas_acciones_paralelas(init,acciones)
print("Acciones de los ascensores")
print("Ordenadas por ascensor y momento en el que se realizan")
print()
decodificador_acciones_paralelas(cocina_acciones_paralelas(acciones_ampliadas))
```

Solución Acciones de los ascensores Ordenadas por ascensor y momento en el que se realizan

$[0,1] \mapsto Elascensor0sube1planta$
$[1] \mapsto Elascensor0cogealaspersonas4,2$
$[1,2] \mapsto Elascensor0sube1planta$
$[2] \mapsto Elascensor0dejaalapersona4$
$[2] \mapsto Elascensor0cogealapersona0$
$[2,3] \mapsto Elascensor0sube1planta$
$[3] \mapsto Elascensor0dejaalaspersonas0,2$
$[3,4] \mapsto Elascensor0baja1planta$
$[4] \mapsto Elascensor0cogealapersona4$
$[4,5] \mapsto Elascensor0sube1planta$
$[5] \mapsto Elascensor0cogealapersona2$
$[5,6] \mapsto Elascensor0sube1planta$
$[6] \mapsto Elascensor0dejaalaspersonas4,2$
$[6,8] \mapsto Elascensor0espera$
$[8] \mapsto Elascensor0cogealapersona3$
$[8,11] \mapsto Elascensor0baja3plantas$
$[11] \mapsto Elascensor0dejaalapersona3$
$[0] \mapsto Elascensor1cogealapersona1$
$[0,4] \mapsto Elascensor1sube4plantas$
$[4] \mapsto Elascensor1dejaalapersona1$
$[4] \mapsto Elascensor1cogealapersona3$
$[4,8] \mapsto Elascensor1baja4plantas$
$[8] \mapsto Elascensor1dejaalapersona3$
$[8,12] \mapsto Elascensor1sube4plantas$
$[0,6] \mapsto Elascensor2espera$
$[6] \mapsto Elascensor2cogealaspersonas2,4$

[6,10]$\mapsto$ *Elascensor*2*sube*4*plantas*
[10]$\mapsto$ *Elascensor*2*dejaalaspersonas*2, 4
[0,4]$\mapsto$ *Elascensor*3*espera*
[4]$\mapsto$ *Elascensor*3*cogealapersona*1
[4,7]$\mapsto$ *Elascensor*3*sube*3*plantas*
[7]$\mapsto$ *Elascensor*3*dejaalapersona*1
[7,10]$\mapsto$ *Elascensor*3*baja*3*plantas*
[10]$\mapsto$ *Elascensor*3*cogealaspersonas*2, 4
[10,11]$\mapsto$ *Elascensor*3*sube*1*planta*
[11]$\mapsto$ *Elascensor*3*dejaalapersona*4
[11,14]$\mapsto$ *Elascensor*3*sube*3*plantas*
[14]$\mapsto$ *Elascensor*3*dejaalapersona*2

```
Nodos analizados: 1309
Coste secuencial: 35
Coste paralelo: 14
Tiempo de búsqueda: 3.482654571533203 s
```

¡Resolvemos el caso del enunciado óptimamente en 3.26 segundos!.

El coste para este mismo problema (tanto en nodos analizados como en tiempo) es menor que el obtenido en la versión anterior pese a que esta no considera la paralelización de los ascensores. Esto es así gracias a que con nuestra nueva función de coste favorecemos siempre las acciones en las que los ascensores están todos siendo utilizados.

El cambio del paradigma, la pérdida de secuencialidad, hace que se analicen antes aquellas acciones que involucran varios ascensores simultaneamente. Una vez un ascensor esté funcionando, todas las acciones de los restantes ascensores que terminen de ejecutarse antes de que termine la acción de este ascensor tendrán coste cero, lo que gracias al algoritmo A* implica que serán analizadas con preferencia frente a volver a hacer funcionar ese ascensor.

Distinguimos también entre coste secuencial (el número de plantas que recorren los ascensores del problema) y coste paralelo (número de plantas máximo que tiene que recorrer un ascensor). Gracias a que nuestra heurística es admisible obtenemos la solución óptima al problema planteado inicialmente.