PRACTICA_2B

December 1, 2019

GRUPO 7

Beatriz Herguedas Pinedo

Pablo Hernández Aguado

1 Apartado 1. Ascensor único. Implementación.

El problema representado consiste en:

- Un ascensor que se desplaza entre las plantas de un edificio, cuyas características son:
  - Una carga máxima de pasajeros.
  - Una planta límite inferior.
  - Una planta límite superior.
- Un número **N** de pasajeros, cada cual quiere desplazarse con el ascensor de una planta a otra.

1.0.1 Funciones auxiliares.

Implementamos previamente una serie de funciones auxiliares que nos ayudarán con la implementación del problema.

[27]:
```
def getUpDowns(passNum, passState, passGoal):
    """ Devuelve una tupla con las distancias (con signo)
        entre el estado inicial de los pasajeros y sus
        destinos. """
    uD = list()
    for i in range(passNum):
        ## > Positivo si sube.
        ## > Negativo si baja.
        ## > Cero si ya está en su destino.
        uD.append( passGoal[i] - passState[i] )

    return tuple( uD )
```

[28]:
```
def getOps(step, floor, minF, maxF):
    """ Obtiene los posibles movimientos de un ascensor (abajo y/o arriba),
    teniendo en cuanta sus plantas límite. """
```

```
    ops = list()
    if floor != maxF:
        ops.append(step)
    if floor != minF:
        ops.append(-step)

    return ops
```

[29]:
```
def getPassActions(actionsSet, elevSet, passSet):
    """ Carga en 'actionsSet' el conjunto de todas las
        acciones posibles de la acción de un ascensor en
        los pasajeros del estado.

        Estos deben estar en la misma planta del ascensor,
        no haber llegado ya a su destino, y querer moverse
        en la dirección del ascensor. """
    # Valores del ascensor:
    (elevOp, elevFloor, elevLoad) = elevSet

    # Valores de los pasajeros:
    (passNum, passState, passGoal, passUpDown) = passSet

    # Lista de los pasajeros que pueden subirse al ascensor:
    passIndex = list()
    for i in range(passNum):
        if (    passState[i] != passGoal[i] and  ### No ha llegado a su destino.
                passState[i] == elevFloor and    ### Está en la planta del␣
↪ascensor.
                passUpDown[i] * elevOp > 0 ):    ### Va en la dirección del␣
↪ascensor.
            passIndex.append(i)

    limit = min(len(passIndex), elevLoad) + 1

    for k in range(1, limit):
        ## Combinaciones posibles de pasajeros para
        ## para cada posible carga del ascensor.
        comb = list( combinations( passIndex, k) )
        for c in comb:
            action = [ tuple(c), elevOp ]
            actionsSet.append( action.copy() )
```

1.0.2 Implementación del problema.

Espacio de estados

Es obvio que cada estado del problema debe representar la planta en que se encuentran los **N**

pasajeros en ese momento, lo que representamos con una tupla de tamaño **N**: *(p1, . . . , pN)*.

No obstante, por la forma de implementar el desplazamiento del ascensor, se hace necesario además saber la planta en la que se encuentra el ascensor en ese momento: *eF*.

Por tanto, finalmente nos queda que los estados son de la forma:

S = ((p1, . . . , pN), eF)

Representación de las acciones

En primer lugar, hemos de decidir cómo se mueve el ascensor. Una opción sería tener en cuenta todas las posibles plantas que puede subir o bajar el ascensor *de una vez* desde la planta en que se encuentra. Sin embargo, esto aumentaría considerablemente el número de acciones posibles a partir de un estado y, para representar el coste de una acción, habría que tener en cuenta su desplazamiento, lo que haría la implementación más compleja.

Es por ello que decidimos que las acciones sólo puedan desplazar al ascensor a una planta consecutiva a la que se encuentra. Además, puesto que en un funcionamiento óptimo el ascensor no se detendría hasta que los pasajeros hayan llegado a su destino, descartamos la opción de que no se desplace. Nos quedan entonces 2 movimientos posibles: arriba y abajo, es decir, +1 planta y -1 planta.

Una vez hecha esta decisión, la acción deberá contener también información sobre qué pasajeros se van a desplazar en con el ascensor (se van a subir en él). Representamos esto con una tupla de tamaño **0 <= k <= min{N, L}** que contiene los índices (de la tupla del estado) de los pasajeros que se van a desplazar, donde **L = "carga máxima del ascensor"**.

Así una acción nos quedaría de la forma:

A = ((i1, . . . , ik), op)

Efecto de una acción sobre un estado

Así, dado un determinado estado *S* y una acción *A*, para cada *i* de la tupla de índices de la acción, hacemos:

(p1, . . . , pN) [i] += op

Y actualizamos también la planta del ascensor:

eF += op

Reducción de las acciones posibles

Resulta trivial que si un pasajero quiere subir de una planta inferior (inicial) a una superior, este no va subirse en el ascensor si este va a bajar. Si lo hiciese, el ascensor tendría que volver a subirle a esa planta para llegar a su objetivo, lo que aumentaría innecesariamente el coste en tiempo de cualquier solución.

Por tanto, conviene diferenciar a los pasajeros que suben de los que bajan para no generar acciones de más. Para ello, utilizamos la tupla **passUpDown** que indica, para cada pasajero, su distancia (con signo) entre su planta inicial y la de destino. Si esta distancia es positiva, entonces el pasajero sube; si es negativa, el pasajero baja; y si es nula, el pasajero ya está en su destino (inicialmente).

Por ello, cuando un ascensor está en una planta, dada una acción del mismo, arriba o abajo, tomamos sólo a los pasajeros que suben o bajan, respectivamente. Esto reduce considerablemente

el número de acciones, al reducir el número de combinaciones de pasajeros que pueden subirse al ascensor.

Código

[30]:
```
class AscensorSimple(Problem):
    def __init__(self, initial, goal, elevSettings):
        # Tupla con el estado inicial, que contiene:
        # > Tupla con las posiciones de los pasajeros.
        # > Número con la planta inicial del ascensor.
        self.initial = initial

        # Tupla con las posiciones finales de los pasajeros.
        self.goal = goal

        # Carga máxima de pasajeros en el ascensor.
        self.elevLoad = elevSettings[0]

        # Planta más baja a la que llega el ascensor.
        self.minFloor = elevSettings[1]

        # Planta más alta a la que llega el ascensor.
        self.maxFloor = elevSettings[2]

        Problem.__init__(self, initial, goal)

        #-------------------------------------

        # Número de pasajeros del problema.
        self.passNum = len( initial[0] )

        # Lista que marca si un pasajero sube o baja (o ninguna de las dos)
        self.passUpDown = getUpDowns(self.passNum, initial[0], goal)
    ##-------------------------------------------------------------------------

    def actions(self, state):
        """ Las acciones consisten de una tupla formada por:
            > Tupla con los pasajeros que desplaza el ascensor.
            > Movimiento del ascensor (arriba o abajo) """
        # Estado de los pasajeros.
        passState = state[0]

        # Planta del ascensor.
        elevFloor = state[1]

        # Lista de acciones.
        actionSet = list()
```

```
        # 1. Tomamos las acciones posibles del ascensor:
        elevOps = getOps(1, elevFloor, self.minFloor, self.maxFloor)

        # 2. Buscamos las acciones posibles de los pasajeros:
        if elevFloor in state[0]:
            ## De los que suben y los que bajan:
            for elevOp in elevOps:
                getPassActions(
                    actionSet,
                    (elevOp, elevFloor, self.elevLoad),
                    (self.passNum, passState, self.goal, self.passUpDown)
                )

        ## 3. Movimientos del ascensor sin pasajeros:
        for elevOp in elevOps:
            action = [ tuple(), elevOp ]
            actionSet.append( action.copy() )

        return actionSet
    ##_------------------------------------------------------------------------


    def result(self, state, action):
        """ Dada una acción, se recorre la tupla de pasajeros
            desplazados por el ascensor (action[0]) y se suma
            a su posición en el estado (state[0]) (la previa
            al desplazamiento) el movimiento del ascensor arriba
            o abajo (action[1]).

            Se actualiza también la posición del ascensor (state[1]),
            sumándole el movimiento del mismo (action[1]). """
        # Lista del estado de los pasajeros previo a la acción.
        passResult = list( state[0] )

        # Lista de los pasajeros afectados por la acción:
        elevPassengers = action[0]

        # Movimiento del ascensor:
        elevMove = action[1]

        # 1. Actualización de los pasajeros.
        for passIndex in elevPassengers:
            ## Se actualiza cada posición respecto a la acción.
            passResult[passIndex] += elevMove

        # 2. Actualización del ascensor.
        elevFloor = state[1]
```

```
        elevNewFloor = elevFloor + elevMove

        # Devolvemos una tupla con un nuevo estado, que contiene:
        # > Tupla con los pasajeros actualizados.
        # > Número con la nueva posición del ascensor.
        return ( tuple(passResult), elevNewFloor )

    def goal_test(self, state):
        # Estado de los pasajeros
        passState = state[0]

        # Se comprueba si es igual al objetivo.
        return passState == self.goal

    def path_cost(self, c, state1, action, state2):
        # En toda acción pasa siempre una unidad de tiempo.
        # El coste aumenta en 1:
        return c + 1
    #______________________________________________________________________________
```

Vemos un ejemplo al que aplicamos las funciones implementadas.

[29]:
```
e1_inicial = ((0,1,5,5,5,5,5,5,8,10), 5)
e1_final =    (1,2,3,4,5,6,7,8,9,10)
e1_ascensor = (2,0,12)
```

[30]:
```
ejemplo1 = AscensorSimple( e1_inicial, e1_final, e1_ascensor )
```

Vemos las acciones resultantes del estado inicial.

[31]:
```
ejemplo1.actions(e1_inicial)
```

[31]:
```
[[(5,), 1],
 [(6,), 1],
 [(7,), 1],
 [(5, 6), 1],
 [(5, 7), 1],
 [(6, 7), 1],
 [(2,), -1],
 [(3,), -1],
 [(2, 3), -1],
 [(), 1],
 [(), -1]]
```

Vemos el resultado de aplicar la 5ª acción de la lista al estado inicial.

[33]:
```
ejemplo1.result(e1_inicial, [(5,7), 1])
```

[33]:
```
((0, 1, 5, 5, 5, 6, 5, 6, 8, 10), 6)
```

A continuación, vemos otro ejemplo con un estado más reducido, al que aplicamos métodos de búsqueda que no requieren de heurísticas.

[36]:
```
e2_inicial = ((0,2,4), 3)
e2_final =    (3,0,1)
e2_ascensor = (2,0,4)
```

[37]:
```
ejemplo2 = AscensorSimple( e2_inicial, e2_final, e2_ascensor )
```

Búsqueda en anchura

[57]:
```
breadth_first_tree_search(ejemplo2).solution()
```

[57]:
```
[[(), 1],
 [(2,), -1],
 [(2,), -1],
 [(1, 2), -1],
 [(1,), -1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1]]
```

[58]:
```
%%timeit
breadth_first_tree_search(ejemplo2).solution()
```

```
18.4 ms ± 261 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[63]:
```
breadth_first_graph_search(ejemplo2).solution()
```

[63]:
```
[[(), 1],
 [(2,), -1],
 [(2,), -1],
 [(1, 2), -1],
 [(1,), -1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1]]
```

[64]:
```
%%timeit
breadth_first_graph_search(ejemplo2).solution()
```

```
3.96 ms ± 81.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Búsqueda en profundidad

[66]:
```
## NO ACABA
#depth_first_tree_search(ejemplo2).solution()
```

[67]:
```
depth_first_graph_search(ejemplo2).solution()
```

[67]:
```
[[(), -1],
 [(), -1],
 [(), -1],
 [(0,), 1],
 [(), 1],
 [(), 1],
 [(), 1],
 [(2,), -1],
 [(), -1],
 [(), -1],
 [(0,), 1],
 [(), 1],
 [(2,), -1],
 [(1, 2), -1],
 [(), 1],
 [(0,), 1],
 [(), -1],
 [(), -1],
 [(1,), -1]]
```

[68]:
```
%%timeit
depth_first_graph_search(ejemplo2).solution()
```

```
866 µs ± 18.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Coste uniforme

[73]:
```
uniform_cost_search(ejemplo2).solution()
```

[73]:
```
[[(), 1],
 [(2,), -1],
 [(2,), -1],
 [(1, 2), -1],
 [(1,), -1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1]]
```

[72]:
```
%%timeit
uniform_cost_search(ejemplo2).solution()
```

```
14 ms ± 520 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

2 Apartado 2. Ascensor único. Heurísticas.

Ya que todas las heurísticas que se van a mostrar utilizan información que contiene el objeto del problema, se han de implementar como funciones de la clase. Por tanto, mostramos el código de una nueva clase **AscensorSimpleHeuristicas**, con las heurísticas como funciones, y explicamos las heurísticas implementadas a continuación de este.

Antes de explicar las heurísticas, creamos también una serie de problemas con tamaños distintos para poner a prueba las heurísticas.

[280]:
```
class AscensorSimpleHeuristicas(Problem):
    def __init__(self, initial, goal, elevSettings):
        # Tupla con el estado inicial, que contiene:
        # > Tupla con las posiciones de los pasajeros.
        # > Número con la planta inicial del ascensor.
        self.initial = initial

        # Tupla con las posiciones finales de los pasajeros.
        self.goal = goal

        # Carga máxima de pasajeros en el ascensor.
        self.elevLoad = elevSettings[0]

        # Planta más baja a la que llega el ascensor.
        self.minFloor = elevSettings[1]

        # Planta más alta a la que llega el ascensor.
        self.maxFloor = elevSettings[2]

        Problem.__init__(self, initial, goal)

        #-------------------------------------

        # Número de pasajeros del problema.
        self.passNum = len( initial[0] )

        # Lista que marca si un pasajero sube o baja (o ninguna de las dos)
        self.passUpDown = getUpDowns(self.passNum, initial[0], goal)

        # Nodos analizados.
        self.analizados = 0
    ##-------------------------------------------------------------------------

    def actions(self, state):
        """ Las acciones consisten de una tupla formada por:
            > Tupla con los pasajeros que desplaza el ascensor.
            > Movimiento del ascensor (arriba o abajo) """
        # Estado de los pasajeros.
```

```
        passState = state[0]

        # Planta del ascensor.
        elevFloor = state[1]

        # Lista de acciones.
        actionSet = list()

        # 1. Tomamos las acciones posibles del ascensor:
        elevOps = getOps(1, elevFloor, self.minFloor, self.maxFloor)

        # 2. Buscamos las acciones posibles de los pasajeros:
        if elevFloor in state[0]:
            ## De los que suben y los que bajan:
            for elevOp in elevOps:
                getPassActions(
                    actionSet,
                    (elevOp, elevFloor, self.elevLoad),
                    (self.passNum, passState, self.goal, self.passUpDown)
                )

        ## 3. Movimientos del ascensor sin pasajeros:
        for elevOp in elevOps:
            action = [ tuple(), elevOp ]
            actionSet.append( action.copy() )

        return actionSet
    ##--------------------------------------------------------------------------


    def result(self, state, action):
        """ Dada una acción, se recorre la tupla de pasajeros
            desplazados por el ascensor (action[0]) y se suma
            a su posición en el estado (state[0]) (la previa
            al desplazamiento) el movimiento del ascensor arriba
            o abajo (action[1]).

            Se actualiza también la posición del ascensor (state[1]),
            sumándole el movimiento del mismo (action[1]). """
        # Lista del estado de los pasajeros previo a la acción.
        passResult = list( state[0] )

        # Lista de los pasajeros afectados por la acción:
        elevPassengers = action[0]

        # Movimiento del ascensor:
        elevMove = action[1]
```

```
        # 1. Actualización de los pasajeros.
        for passIndex in elevPassengers:
            ## Se actualiza cada posición respecto a la acción.
            passResult[passIndex] += elevMove

        # 2. Actualización del ascensor.
        elevFloor = state[1]
        elevNewFloor = elevFloor + elevMove

        # Devolvemos una tupla con un nuevo estado, que contiene:
        # > Tupla con los pasajeros actualizados.
        # > Número con la nueva posición del ascensor.
        return ( tuple(passResult), elevNewFloor )

    def goal_test(self, state):
        # Nodo analizado.
        self.analizados += 1

        # Estado de los pasajeros
        passState = state[0]

        # Se comprueba si es igual al objetivo.
        return passState == self.goal

    def path_cost(self, c, state1, action, state2):
        # En toda acción pasa siempre una unidad de tiempo.
        # El coste aumenta en 1:
        return c + 1
    #--------------------------------------------------------------------------

    def maxState(self, node):
        # Estado de los pasajeros
        passState = node.state[0]

        # Planta máxima del estado.
        maxPass = self.minFloor

        # Planta mínima del estado.
        minPass = self.maxFloor

        # 1. Cálculo del mínimo y máximo
        for i in range(self.passNum):
            if passState[i] != self.goal[i]:
                if passState[i] > maxPass:
                    maxPass = passState[i]
                if passState[i] < minPass:
```

```
                    minPass = passState[i]

        distance = maxPass - minPass

        if distance < 0: # Todos los pasajeros están en su destino.
            return 0
        else:
            return distance
#-------------------------------------------------------------------------

    ### HEURÍSTICA LINEAL PONDERADA.

    def linear(self, node):
        # Estado de los pasajeros:
        passState = node.state[0]

        # Coste acumulado:
        cost = 0

        # Cálculo de las distancias:
        for i in range(self.passNum):
            cost += abs( passState[i] - self.goal[i] )

        return cost

    def weightedLinear(self, node):
        return self.linear(node) / self.elevLoad
    #---------------------------------------------------------------------

    ### HEURÍSTICA LINEAL 'COMPROBADA'

    def checkedLinear(self, node):
        # Estado de los pasajeros:
        passState = node.state[0]

        # Carga de cada transición entre plantas (inicialmente 0):
        transitionLoads = [ 0 for i in range(self.minFloor, self.maxFloor) ]

        # 1. Se calcula la carga de cada transición:
        for i in range(self.passNum):
            if self.passUpDown[i] > 0: # El pasajero sube.
                for j in range( passState[i], self.goal[i] ):
                    transitionLoads[j] += 1
            if self.passUpDown[i] < 0: # El pasajero baja.
                for j in range( self.goal[i], passState[i]):
                    transitionLoads[j] += 1
```

```
        # Coste de la heurística.
        cost = 0

        # 2. Se cálcula el número mínimo de pases por cada transición:
        for load in transitionLoads:
            if load != 0: # Tiene algo de carga.
                if load % self.elevLoad == 0: # Múltiplo de la carga máxima
                    cost += load // self.elevLoad
                else: # No múltiplo (se suma un viaje)
                    cost += load // self.elevLoad + 1

        return cost
    #-------------------------------------------------------------------------

    ### HEURÍSTICA LINEAL 'DIRIGIDA' Y 'COMPROBADA'

    def directedCheckedLinear(self, node):
        # Estado de los pasajeros:
        passState = node.state[0]

        # Carga de cada transición hacia arriba entre plantas (inicialmente 0):
        upLoads = [ 0 for i in range(self.minFloor, self.maxFloor) ]

        # 1. Se calculan las cargas para los pasajeros que suben:
        for i in range(self.passNum):
            if self.passUpDown[i] > 0: # El pasajero sube.
                for j in range( passState[i], self.goal[i] ):
                    upLoads[j - self.minFloor] += 1

        # Transiciones mínimas hacia arriba:
        upCost = 0

        # 2. Se cálcula el número mínimo de transiciones hacia arriba:
        for load in upLoads:
            if load != 0: # Tiene algo de carga.
                if load % self.elevLoad == 0: # Múltiplo de la carga máxima
                    upCost += load // self.elevLoad
                else: # No múltiplo (se suma un viaje)
                    upCost += load // self.elevLoad + 1

        # Carga de cada transición hacia abajo entre plantas (inicialmente 0):
        downLoads = [ 0 for i in range(self.minFloor, self.maxFloor) ]

        # 3. Análogo para los pasajeros que bajan:
        for i in range(self.passNum):
            if self.passUpDown[i] < 0: # El pasajero sube.
                for j in range( self.goal[i], passState[i] ):
```

```
                    downLoads[j - self.minFloor] += 1

        # Transiciones mínimas hacia abajo:
        downCost = 0

        # 4. Análogo para las transiciones hacia abajo:
        for load in downLoads:
            if load != 0: # Tiene algo de carga.
                if load % self.elevLoad == 0: # Múltiplo de la carga máxima
                    downCost += load // self.elevLoad
                else: # No múltiplo (se suma un viaje)
                    downCost += load // self.elevLoad + 1

        return upCost + downCost
    #-------------------------------------------------------------------------
```

Ejemplos

[283]:
```
inicial_H1 = ((0,3,3,1,4), 3)
final_H1 =    (4,0,1,0,2)
ascensor_H1 = (2,0,4)
ejemplo_H1 = AscensorSimpleHeuristicas( inicial_H1, final_H1, ascensor_H1 )
```

[284]:
```
inicial_H2 = ((4,5,1,0,4), 2)
final_H2 =    (0,3,5,3,5)
ascensor_H2 = (2,0,5)
ejemplo_H2 = AscensorSimpleHeuristicas( inicial_H2, final_H2, ascensor_H2 )
```

[285]:
```
inicial_H3 = ((5,3,1,6,6), 3)
final_H3 =    (3,0,5,1,3)
ascensor_H3 = (2,0,6)
ejemplo_H3 = AscensorSimpleHeuristicas( inicial_H3, final_H3, ascensor_H3 )
```

[286]:
```
inicial_H4 = ((0,2,7,3,4), 4)
final_H4 =    (6,5,2,6,2)
ascensor_H4 = (2,0,7)
ejemplo_H4 = AscensorSimpleHeuristicas( inicial_H4, final_H4, ascensor_H4 )
```

[287]:
```
inicial_H5 = ((8,7,8,0,4), 5)
final_H5 =    (4,3,2,5,8)
ascensor_H5 = (2,0,8)
ejemplo_H5 = AscensorSimpleHeuristicas( inicial_H5, final_H5, ascensor_H5 )
```

[288]:
```
inicial_H6 = ((7,0,3,7,2), 7)
final_H6 =    (0,6,9,1,8)
ascensor_H6 = (2,0,9)
ejemplo_H6 = AscensorSimpleHeuristicas( inicial_H6, final_H6, ascensor_H6 )
```

[289]:
```
inicial_H7 = ((8,8,4,8,0), 8)
final_H7 =    (2,3,10,2,10)
ascensor_H7 = (2,0,10)
ejemplo_H7 = AscensorSimpleHeuristicas( inicial_H7, final_H7, ascensor_H7 )
```

Método de información de búsqueda.

[278]:
```
def getSolutionInfo(sol, problem):
    print("Longitud de la solución: {0}. Nodos analizados: {1}".
 ↪format(len(sol),problem.analizados))
```

Heurística máxima

Esta es una de las heurísticas más simples. Dado un estado, se toma la distancia máxima entre los pasajeros que no están en su destino. Puesto que el ascensor tendrá que visitar ambas plantas para llevar a cada pasajero a su destino, este recorrerá necesariamente esa distancia. Por tanto, la heurística es siempre menor que el coste y, por tanto, válida.

[81]:
```
def maxState(self, node):
    """ Heurística que toma la distancia máxima entre
        los pasajeros del estado que no están en su destino.
        Es trivial que el ascensor deberá visitar ambas plantas
        para llevarlos a su destino, por lo que deberá hacer el
        recorrido entre las mismas. Por ello, esta heurística es
        correcta. """
    # Estado de los pasajeros
    passState = node.state[0]

    # Planta máxima del estado.
    maxPass = self.minFloor

    # Planta mínima del estado.
    minPass = self.maxFloor

    # 1. Cálculo del mínimo y máximo
    for i in range(self.passNum):
        if passState[i] != self.goal[i]:
            if passState[i] > maxPass:
                maxPass = passState[i]
            if passState[i] < minPass:
                minPass = passState[i]

    distance = maxPass - minPass

    if distance < 0: # Todos los pasajeros están en su destino.
        return 0
    else:
        return distance
```

[351]:
```
ejemplo_H1 = AscensorSimpleHeuristicas( inicial_H1, final_H1, ascensor_H1 )
solMaxH1 = astar_search(ejemplo_H1, ejemplo_H1.maxState).solution()
solMaxH1
```

[351]:
```
[[(1, 2), -1],
 [(1, 2), -1],
 [(1, 3), -1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1],
 [(4,), -1],
 [(4,), -1]]
```

[352]:
```
getSolutionInfo( solMaxH1, ejemplo_H1 )
```

```
Longitud de la solución: 9. Nodos analizados: 691
```

[353]:
```
%%timeit
astar_search(ejemplo_H1, ejemplo_H1.maxState).solution()
```

```
411 ms ± 9.24 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[354]:
```
ejemplo_H2 = AscensorSimpleHeuristicas( inicial_H2, final_H2, ascensor_H2 )
solMaxH2 = astar_search(ejemplo_H2, ejemplo_H2.maxState).solution()
solMaxH2
```

[354]:
```
[[(), -1],
 [(), -1],
 [(3,), 1],
 [(2, 3), 1],
 [(2, 3), 1],
 [(2,), 1],
 [(2, 4), 1],
 [(1,), -1],
 [(0, 1), -1],
 [(0,), -1],
 [(0,), -1],
 [(0,), -1]]
```

[355]:
```
getSolutionInfo( solMaxH2, ejemplo_H2 )
```

```
Longitud de la solución: 12. Nodos analizados: 612
```

[137]:
```
%%timeit
astar_search(ejemplo_H2, ejemplo_H2.maxState).solution()
```

```
424 ms ± 4.84 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[356]:
```
ejemplo_H3 = AscensorSimpleHeuristicas( inicial_H3, final_H3, ascensor_H3 )
solMaxH3 = astar_search(ejemplo_H3, ejemplo_H3.maxState).solution()
solMaxH3
```

[356]:
```
[[(), 1],
 [(), 1],
 [(), 1],
 [(3, 4), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(1, 3), -1],
 [(1, 3), -1],
 [(1,), -1],
 [(), 1],
 [(2,), 1],
 [(2,), 1],
 [(2,), 1],
 [(2,), 1],
 [(4,), -1],
 [(4,), -1]]
```

[357]:
```
getSolutionInfo( solMaxH3, ejemplo_H3 )
```

```
Longitud de la solución: 16. Nodos analizados: 6081
```

[139]:
```
%%timeit
astar_search(ejemplo_H3, ejemplo_H3.maxState).solution()
```

```
23.5 s ± 297 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[366]:
```
ejemplo_H4 = AscensorSimpleHeuristicas( inicial_H4, final_H4, ascensor_H4 )
```

[367]:
```
%%time
solMaxH4 = astar_search(ejemplo_H4, ejemplo_H4.maxState).solution()
```

```
Wall time: 1min 7s
```

[368]:
```
solMaxH4
```

[368]:
```
[[(4,), -1],
 [(3,), 1],
 [(3,), 1],
 [(), 1],
 [(), 1],
 [(2,), -1],
 [(2,), -1],
 [(2,), -1],
 [(2,), -1],
```

```
 [(2, 4), -1],
 [(), -1],
 [(), -1],
 [(0,), 1],
 [(0,), 1],
 [(0, 1), 1],
 [(0, 1), 1],
 [(0, 1), 1],
 [(0, 3), 1]]
```

[369]:
```
getSolutionInfo( solMaxH4, ejemplo_H4 )
```

```
Longitud de la solución: 18. Nodos analizados: 10472
```

Heurística lineal ponderada. Esta heurística es el resultado de intentar aplicar la *heurística lineal* al problema, siendo esta la que resulta de sumar las distancias de cada pasajero a su planta destino. Es cierto que el ascensor debe desplazarse a cada planta en que hay un pasajero y llevarlo a su destino, por lo que recorre la distancia entre ese pasajero y su objetivo. Sin embargo, debido a que el ascensor podría llevar a más de una persona, es fácil hallar un ejemplo que descarta esta función como una heurística válida:

```
Sea el estado (2,2) y el objetivo (6,6), la *heurística lineal* nos daría como valor 8 = (6-2) +
```

Nótese que este es uno de los peores casos que se puede dar, es decir, el ascensor puede realizar llevar a todos los pasajeros a su destino desplazándose siempre con su carga máxima (en el ejemplo esta sería 2). Por tanto, dada la suma **distSum** de las distancias y la carga máxima **elevLoad** del ascensor, este haría exactamente **distSum / elevLoad**. En cualquier otro caso, el coste sería superior a esta cantidad, lo que nos da una heurística válida, que llamamos **Lineal Ponderada**.

[ ]:
```
def linear(self, node):
    # Estado de los pasajeros:
    passState = node.state[0]

    # Coste acumulado:
    cost = 0

    # Cálculo de las distancias:
    for i in range(self.passNum):
        cost += abs( passState[i] - self.goal[i] )

    return cost

def weightedLinear(self, node):
    return self.linear(node) / self.elevLoad
```

[402]:
```
ejemplo_H1 = AscensorSimpleHeuristicas( inicial_H1, final_H1, ascensor_H1 )
solWLH1 = astar_search(ejemplo_H1, ejemplo_H1.weightedLinear).solution()
solWLH1
```

[402]:
```
[[(1, 2), -1],
 [(1, 2), -1],
 [(1, 3), -1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1],
 [(4,), -1],
 [(4,), -1]]
```

[403]:
```
getSolutionInfo( solWLH1, ejemplo_H1 )
```

```
Longitud de la solución: 9. Nodos analizados: 290
```

[304]:
```
%%timeit
astar_search(ejemplo_H1, ejemplo_H1.weightedLinear).solution()
```

```
118 ms ± 6.87 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[404]:
```
ejemplo_H2 = AscensorSimpleHeuristicas( inicial_H2, final_H2, ascensor_H2 )
solWLH2 = astar_search(ejemplo_H2, ejemplo_H2.weightedLinear).solution()
solWLH2
```

[404]:
```
[[(), -1],
 [(), -1],
 [(3,), 1],
 [(2, 3), 1],
 [(2, 3), 1],
 [(2,), 1],
 [(2, 4), 1],
 [(1,), -1],
 [(0, 1), -1],
 [(0,), -1],
 [(0,), -1],
 [(0,), -1]]
```

[405]:
```
getSolutionInfo( solWLH2, ejemplo_H2 )
```

```
Longitud de la solución: 12. Nodos analizados: 275
```

[148]:
```
%%timeit
astar_search(ejemplo_H2, ejemplo_H2.weightedLinear).solution()
```

```
133 ms ± 4.45 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[406]:
```
ejemplo_H3 = AscensorSimpleHeuristicas( inicial_H3, final_H3, ascensor_H3 )
solWLH3 = astar_search(ejemplo_H3, ejemplo_H3.weightedLinear).solution()
solWLH3
```

[406]:
```
[[(), 1],
 [(), 1],
 [(), 1],
 [(3, 4), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(1, 3), -1],
 [(1, 3), -1],
 [(1,), -1],
 [(), 1],
 [(2,), 1],
 [(2,), 1],
 [(2,), 1],
 [(2,), 1],
 [(4,), -1],
 [(4,), -1]]
```

[407]:
```
getSolutionInfo( solWLH3, ejemplo_H3 )
```

```
Longitud de la solución: 16. Nodos analizados: 4399
```

[150]:
```
%%timeit
astar_search(ejemplo_H3, ejemplo_H3.weightedLinear).solution()
```

```
19.9 s ± 497 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[408]:
```
ejemplo_H4 = AscensorSimpleHeuristicas( inicial_H4, final_H4, ascensor_H4 )
solWLH4 = astar_search(ejemplo_H4, ejemplo_H4.weightedLinear).solution()
solWLH4
```

[408]:
```
[[(4,), -1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1],
 [(), 1],
 [(2,), -1],
 [(2,), -1],
 [(2,), -1],
 [(2,), -1],
 [(2, 4), -1],
 [(), -1],
 [(), -1],
 [(0,), 1],
 [(0,), 1],
 [(0, 1), 1],
 [(0, 1), 1],
 [(0, 1), 1],
```

```
 [(0,), 1]]
```

[409]:
```
getSolutionInfo( solWLH4, ejemplo_H4 )
```

```
Longitud de la solución: 18. Nodos analizados: 4504
```

[226]:
```
%%timeit
astar_search(ejemplo_H4, ejemplo_H4.weightedLinear).solution()
```

```
13.5 s ± 540 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[362]:
```
ejemplo_H5 = AscensorSimpleHeuristicas( inicial_H5, final_H5, ascensor_H5 )
```

[363]:
```
%%time
solWLH5 = astar_search(ejemplo_H5, ejemplo_H5.weightedLinear).solution()
```

```
Wall time: 1min 30s
```

[364]:
```
solWLH5
```

[364]:
```
[[(), 1],
 [(), 1],
 [(1,), -1],
 [(1,), -1],
 [(1,), -1],
 [(4,), 1],
 [(4,), 1],
 [(4,), 1],
 [(4,), 1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(1, 2), -1],
 [(2,), -1],
 [(), -1],
 [(), -1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1]]
```

[365]:
```
getSolutionInfo( solWLH5, ejemplo_H5 )
```

```
Longitud de la solución: 22. Nodos analizados: 12426
```

Heurística lineal 'comprobada'. Siguiendo el razonamiento de la heurística anterior, podemos llegar a otras más informadas. Para explicar esta heurística, vamos a introducir algo de vocabulario específico.

Llamaremos **transición** a cada estado intermedio entre dos plantas consecutivas; por ejemplo, sea la transición entre la planta 2 y 3, decimos que el ascensor pasa por esa transición cuando va de la planta 2 a la 3 o viceversa. Cuando un ascensor pasa por una transición, ya sea subiendo o bajando, decimos que ha hecho un **pase**. Así, parte de esta heurística implica calcular el número de pases que se dan sobre cada transición entre plantas.

De esta forma, dado un pasajero con una posición inicial y su destino, decimos que este pasará por cada transición entre esas dos plantas, por lo que sumamos un pase a cada una de esas transiciones; hemos de hacer esto con cada pasajero.

Veamos este cálculo, tomando, por ejemplo, los estados inicial y final del enunciado de la práctica:

- Estado inicial: (2,4,1,8,1)
- Estado final: (3,11,12,1,9)

Calculamos la acumulación de pases en cada transición:

Pasajero	0 ~ 1	1 ~ 2	2 ~ 3	3 ~ 4	4 ~ 5	5 ~ 6	6 ~ 7	7 ~ 8	8 ~ 9	9 ~ 10	10 ~ 11	11 ~ 12
0	0	0	1	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	1	1	1	1	0
2	0	1	1	1	1	1	1	1	1	1	1	1
3	0	1	1	1	1	1	1	1	0	0	0	0
4	0	1	1	1	1	1	1	1	1	0	0	0
Pases	0	3	4	3	4	4	4	4	3	2	2	1

Lo que el número de pases de una transición nos dice es que un ascensor que sólo pudiese llevar a una persona, tendría que pasar por cada transición, cómo mínimo, tantos pases como hay acumulados en cada una.

No obstante, nuestro ascensor podría ser capaz de llevar a más de una persona, por lo que hemos de adaptarnos a esa característica. Ya que cada pase del ascensor con su carga máxima quitaría tantos pases individuales como su carga máxima, basta calcular el número de pases mínimos necesarios con carga máxima para suplir los pases individuales.

- Si el número de pases es 0, el ascensor no tiene por qué pasar.
- Si el número de pases **Np** es múltiplo de **elevLoad**, entonces tendrá que pasar **Np / elevLoad** veces con carga máxima.
- Si el número de pases **Np** no es múltiplo de **elevLoas**, entonces tendrá que pasar **Np // elevLoad + 1** veces con carga máxima.

Seguimos con el ejemplo, con carga máxima 2:

Pases	0	3	4	3	4	4	4	4	3	2	2	1	+
LOAD = 2	0	2	2	2	2	2	2	2	2	1	1	1	= 19

Esto nos deja un coste mínimo de 19 movimientos.

Nos queda por tanto una heurística válida, que denominamos **Lineal Comprobada**.

[182]:
```
def checkedLinear(self, node):
    # Estado de los pasajeros:
    passState = node.state[0]

    # Carga de cada transición entre plantas (inicialmente 0):
    transitionLoads = [ 0 for i in range(self.minFloor, self.maxFloor) ]

    # 1. Se calcula la carga de cada transición:
    for i in range(self.passNum):
        if self.passUpDown[i] > 0: # El pasajero sube.
            for j in range( passState[i], self.goal[i] ):
                transitionLoads[j] += 1
        if self.passUpDown[i] < 0: # El pasajero baja.
            for j in range( self.goal[i], passState[i]):
                transitionLoads[j] += 1

    # Coste de la heurística.
    cost = 0

    # 2. Se cálcula el número mínimo de pases por cada transición:
    for load in transitionLoads:
        if load != 0: # Tiene algo de carga.
            if load % self.elevLoad == 0: # Múltiplo de la carga máxima
                cost += load // self.elevLoad
            else: # No múltiplo (se suma un viaje)
                cost += load // self.elevLoad + 1

    return cost
```

[370]:
```
ejemplo_H1 = AscensorSimpleHeuristicas( inicial_H1, final_H1, ascensor_H1 )
solCLH1 = astar_search(ejemplo_H1, ejemplo_H1.checkedLinear).solution()
solCLH1
```

[370]:
```
[[(1, 2), -1],
 [(1, 2), -1],
 [(1, 3), -1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1],
 [(4,), -1],
 [(4,), -1]]
```

[371]:
```
getSolutionInfo( solCLH1, ejemplo_H1 )
```

```
Longitud de la solución: 9. Nodos analizados: 169
```

[184]:
```
%%timeit
astar_search(ejemplo_H1, ejemplo_H1.checkedLinear).solution()
```

```
41.5 ms ± 3.83 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[372]:
```
ejemplo_H2 = AscensorSimpleHeuristicas( inicial_H2, final_H2, ascensor_H2 )
solCLH2 = astar_search(ejemplo_H2, ejemplo_H2.checkedLinear).solution()
solCLH2
```

[372]:
```
[[(), -1],
 (), -1],
 [(3,), 1],
 [(2, 3), 1],
 [(2, 3), 1],
 [(2,), 1],
 [(2, 4), 1],
 [(1,), -1],
 [(0, 1), -1],
 [(0,), -1],
 [(0,), -1],
 [(0,), -1]]
```

[373]:
```
getSolutionInfo( solCLH2, ejemplo_H2 )
```

```
Longitud de la solución: 12. Nodos analizados: 171
```

[186]:
```
%%timeit
astar_search(ejemplo_H2, ejemplo_H2.checkedLinear).solution()
```

```
57.2 ms ± 4.39 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[374]:
```
ejemplo_H3 = AscensorSimpleHeuristicas( inicial_H3, final_H3, ascensor_H3 )
solCLH3 = astar_search(ejemplo_H3, ejemplo_H3.checkedLinear).solution()
solCLH3
```

[374]:
```
[[(), 1],
 (), 1],
 [(), 1],
 [(3, 4), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(1, 3), -1],
 [(1, 3), -1],
 [(1,), -1],
 [(), 1],
 [(2,), 1],
```

```
 [(2,), 1],
 [(2,), 1],
 [(2,), 1],
 [(4,), -1],
 [(4,), -1]]
```

[375]:
```
getSolutionInfo( solCLH3, ejemplo_H3 )
```

```
Longitud de la solución: 16. Nodos analizados: 1074
```

[177]:
```
%%timeit
astar_search(ejemplo_H3, ejemplo_H3.checkedLinear).solution()
```

```
1.42 s ± 86.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[376]:
```
ejemplo_H4 = AscensorSimpleHeuristicas( inicial_H4, final_H4, ascensor_H4 )
solCLH4 = astar_search(ejemplo_H4, ejemplo_H4.checkedLinear).solution()
solCLH4
```

[376]:
```
[[(4,), -1],
 [(3,), 1],
 [(3,), 1],
 [(), 1],
 [(), 1],
 [(2,), -1],
 [(2,), -1],
 [(2,), -1],
 [(2,), -1],
 [(2, 4), -1],
 [(), -1],
 [(), -1],
 [(0,), 1],
 [(0,), 1],
 [(0, 1), 1],
 [(0, 1), 1],
 [(0, 1), 1],
 [(0, 3), 1]]
```

[377]:
```
getSolutionInfo( solCLH4, ejemplo_H4 )
```

```
Longitud de la solución: 18. Nodos analizados: 2323
```

[181]:
```
%%timeit
astar_search(ejemplo_H4, ejemplo_H4.checkedLinear).solution()
```

```
3.97 s ± 140 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[378]:
```
ejemplo_H5 = AscensorSimpleHeuristicas( inicial_H5, final_H5, ascensor_H5 )
solCLH5 = astar_search(ejemplo_H5, ejemplo_H5.checkedLinear).solution()
solCLH5
```

[378]:
```
[[(), 1],
 (), 1],
 [(1,), -1],
 [(1,), -1],
 [(1,), -1],
 [(4,), 1],
 [(4,), 1],
 [(4,), 1],
 [(4,), 1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(1, 2), -1],
 [(2,), -1],
 (), -1],
 (), -1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1]]
```

[379]:
```
getSolutionInfo( solCLH5, ejemplo_H5 )
```

```
Longitud de la solución: 22. Nodos analizados: 5205
```

[235]:
```
%%timeit
astar_search(ejemplo_H5, ejemplo_H5.checkedLinear).solution()
```

```
15.7 s ± 355 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[382]:
```
ejemplo_H6 = AscensorSimpleHeuristicas( inicial_H6, final_H6, ascensor_H6 )
```

[383]:
```
%%time
solCLH6 = astar_search(ejemplo_H6, ejemplo_H6.checkedLinear).solution()
```

```
Wall time: 53.8 s
```

[384]:
```
solCLH6
```

[384]:
```
[[(0, 3), -1],
 (0, 3), -1],
 (0, 3), -1],
```

```
[(0, 3), -1],
[(0, 3), -1],
[(0, 3), -1],
[(0,), -1],
[(1,), 1],
[(1,), 1],
[(1, 4), 1],
[(4,), 1],
[(), -1],
[(1, 2), 1],
[(4,), 1],
[(), -1],
[(1, 2), 1],
[(4,), 1],
[(), -1],
[(1, 2), 1],
[(2, 4), 1],
[(2, 4), 1],
[(2,), 1]]
```

[385]:
```
getSolutionInfo( solCLH6, ejemplo_H6 )
```

```
Longitud de la solución: 22. Nodos analizados: 5803
```

Heurística lineal 'dirigida' y 'comprobada'. Esta heurística es una descendiente directa de la anterior, pues está basada en el mismo método de cálculo de pases por las transiciones. No obstante, esta tiene en cuenta la dirección (arriba o abajo) que debe tomar el pasajero para llegar a su destino.

Vemos un ejemplo simple para observar cómo mejora a la heurística anterior tener en cuenta este detalle.

- Estado inicial: (2,10)
- Estado final: (9,1)

Calculamos la acumulación de pases en cada transición:

	0 ~	1 ~	2 ~	3 ~	4 ~	5 ~	6 ~	7 ~	8 ~	9 ~	10 ~	11 ~	
Pasajero	1	2	3	4	5	6	7	8	9	10	11	12	+
0	0	(-) 1	(-) 1	(-) 1	(-) 1	(-) 1	(-) 1	(-) 1	(-) 1	(-) 1	0	0	
1	0	0	(+) 1	(+) 1	(+) 1	(+) 1	(+) 1	(+) 1	(+) 1	0	0	0	
Pases	0	1	2	2	2	2	2	2	2	1	0	0	
LOAD = 2	0	1	1	1	1	1	1	1	1	1	0	0	= 9

Con la heurística anterior, el coste mínimo de este problema sería de 9 movimientos. Sin embargo,

se ve claramente que el número es mucho mayor pues, en tanto que un pasajero baja y el otro sube, no podrían desplazarse a la vez; es decir, el ascensor tendría que llevar primero a uno y luego subir a por el otro. Esto nos muestra que, a la hora de calcular los pases de las transiciones, podemos separar por un lado los pasajeros que suben de los que bajan, hacer el mismo cálculo de la heurística anterior con cada caso, y sumar los costes.

Lo hacemos con el ejemplo de la heurística anterior.

SUBEN:

Pasajero	0 ~ 1	1 ~ 2	2 ~ 3	3 ~ 4	4 ~ 5	5 ~ 6	6 ~ 7	7 ~ 8	8 ~ 9	9 ~ 10	10 ~ 11	11 ~ 12	+
0	0	0	1	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	1	1	1	1	1	1	1	0	
2	0	1	1	1	1	1	1	1	1	1	1	1	
4	0	1	1	1	1	1	1	1	1	0	0	0	
Pases	0	2	3	2	3	3	3	3	3	2	2	1	
LOAD = 2	0	1	2	1	2	2	2	2	2	1	1	1	= 17

BAJAN:

Pasajero	0 ~ 1	1 ~ 2	2 ~ 3	3 ~ 4	4 ~ 5	5 ~ 6	6 ~ 7	7 ~ 8	8 ~ 9	9 ~ 10	10 ~ 11	11 ~ 12	+
3	0	1	1	1	1	1	1	1	0	0	0	0	
Pases	0	1	1	1	1	1	1	1	0	0	0	0	
LOAD = 2	0	1	1	1	1	1	1	1	0	0	0	0	= 7

Nos queda un coste mínimo de 17 + 7 = 24 movimientos

Este procedimiento nos da una heurística válida más informada. Al tener en cuenta la dirección de viaje de los pasajeros, la hemos denominado **Lineal Dirigida y Comprobada**.

[187]:
```
def directedCheckedLinear(self, node):
    # Estado de los pasajeros:
    passState = node.state[0]

    # Carga de cada transición hacia arriba entre plantas (inicialmente 0):
    upLoads = [ 0 for i in range(self.minFloor, self.maxFloor) ]

    # 1. Se calculan las cargas para los pasajeros que suben:
    for i in range(self.passNum):
        if self.passUpDown[i] > 0: # El pasajero sube.
            for j in range( passState[i], self.goal[i] ):
                upLoads[j - self.minFloor] += 1
```

```
    # Transiciones mínimas hacia arriba:
    upCost = 0

    # 2. Se cálcula el número mínimo de transiciones hacia arriba:
    for load in upLoads:
        if load != 0: # Tiene algo de carga.
            if load % self.elevLoad == 0: # Múltiplo de la carga máxima
                upCost += load // self.elevLoad
            else: # No múltiplo (se suma un viaje)
                upCost += load // self.elevLoad + 1

    # Carga de cada transición hacia abajo entre plantas (inicialmente 0):
    downLoads = [ 0 for i in range(self.minFloor, self.maxFloor) ]

    # 3. Análogo para los pasajeros que bajan:
    for i in range(self.passNum):
        if self.passUpDown[i] < 0: # El pasajero sube.
            for j in range( self.goal[i], passState[i] ):
                downLoads[j - self.minFloor] += 1

    # Transiciones mínimas hacia abajo:
    downCost = 0

    # 4. Análogo para las transiciones hacia abajo:
    for load in downLoads:
        if load != 0: # Tiene algo de carga.
            if load % self.elevLoad == 0: # Múltiplo de la carga máxima
                downCost += load // self.elevLoad
            else: # No múltiplo (se suma un viaje)
                downCost += load // self.elevLoad + 1

    return upCost + downCost
```

[386]:
```
ejemplo_H1 = AscensorSimpleHeuristicas( inicial_H1, final_H1, ascensor_H1 )
solDCLH1 = astar_search(ejemplo_H1, ejemplo_H1.directedCheckedLinear).solution()
solDCLH1
```

[386]:
```
[[(1, 2), -1],
 [(1, 2), -1],
 [(1, 3), -1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1],
 [(0,), 1],
 [(4,), -1],
 [(4,), -1]]
```

[387]:
```
getSolutionInfo( solDCLH1, ejemplo_H1 )
```

```
Longitud de la solución: 9. Nodos analizados: 12
```

[189]:
```
%%timeit
astar_search(ejemplo_H1, ejemplo_H1.directedCheckedLinear).solution()
```

```
1.58 ms ± 50.8 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

[388]:
```
ejemplo_H2 = AscensorSimpleHeuristicas( inicial_H2, final_H2, ascensor_H2 )
solDCLH2 = astar_search(ejemplo_H2, ejemplo_H2.directedCheckedLinear).solution()
solDCLH2
```

[388]:
```
[[(), -1],
 [(), -1],
 [(3,), 1],
 [(2, 3), 1],
 [(2, 3), 1],
 [(2,), 1],
 [(2, 4), 1],
 [(1,), -1],
 [(0, 1), -1],
 [(0,), -1],
 [(0,), -1],
 [(0,), -1]]
```

[389]:
```
getSolutionInfo( solDCLH2, ejemplo_H2 )
```

```
Longitud de la solución: 12. Nodos analizados: 16
```

[192]:
```
%%timeit
astar_search(ejemplo_H2, ejemplo_H2.directedCheckedLinear).solution()
```

```
2.1 ms ± 51.4 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

[390]:
```
ejemplo_H3 = AscensorSimpleHeuristicas( inicial_H3, final_H3, ascensor_H3 )
solDCLH3 = astar_search(ejemplo_H3, ejemplo_H3.directedCheckedLinear).solution()
solDCLH3
```

[390]:
```
[[(), 1],
 [(), 1],
 [(), 1],
 [(3, 4), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(1, 3), -1],
 [(1, 3), -1],
 [(1,), -1],
```

```
 [(), 1],
 [(2,), 1],
 [(2,), 1],
 [(2,), 1],
 [(2,), 1],
 [(4,), -1],
 [(4,), -1]]
```

[391]:
```
getSolutionInfo( solDCLH3, ejemplo_H3 )
```

```
Longitud de la solución: 16. Nodos analizados: 234
```

[194]:
```
%%timeit
astar_search(ejemplo_H3, ejemplo_H3.directedCheckedLinear).solution()
```

```
116 ms ± 2.14 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[392]:
```
ejemplo_H4 = AscensorSimpleHeuristicas( inicial_H4, final_H4, ascensor_H4 )
solDCLH4 = astar_search(ejemplo_H4, ejemplo_H4.directedCheckedLinear).solution()
solDCLH4
```

[392]:
```
[[(4,), -1],
 [(3,), 1],
 [(3,), 1],
 [(), 1],
 [(), 1],
 [(2,), -1],
 [(2,), -1],
 [(2,), -1],
 [(2,), -1],
 [(2, 4), -1],
 [(), -1],
 [(), -1],
 [(0,), 1],
 [(0,), 1],
 [(0, 1), 1],
 [(0, 1), 1],
 [(0, 1), 1],
 [(0, 3), 1]]
```

[393]:
```
getSolutionInfo( solDCLH4, ejemplo_H4 )
```

```
Longitud de la solución: 18. Nodos analizados: 957
```

[196]:
```
%%timeit
astar_search(ejemplo_H4, ejemplo_H4.directedCheckedLinear).solution()
```

```
1.06 s ± 65.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[394]:
```
ejemplo_H5 = AscensorSimpleHeuristicas( inicial_H5, final_H5, ascensor_H5 )
solDCLH5 = astar_search(ejemplo_H5, ejemplo_H5.directedCheckedLinear).solution()
solDCLH5
```

[394]:
```
[[(), 1],
 (), 1],
 [(1,), -1],
 [(1,), -1],
 [(1,), -1],
 [(4,), 1],
 [(4,), 1],
 [(4,), 1],
 [(4,), 1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(0, 2), -1],
 [(1, 2), -1],
 [(2,), -1],
 (), -1],
 (), -1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1],
 [(3,), 1]]
```

[395]:
```
getSolutionInfo( solDCLH5, ejemplo_H5 )
```

```
Longitud de la solución: 22. Nodos analizados: 1107
```

[237]:
```
%%timeit
astar_search(ejemplo_H5, ejemplo_H5.directedCheckedLinear).solution()
```

```
1.5 s ± 117 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[396]:
```
ejemplo_H6 = AscensorSimpleHeuristicas( inicial_H6, final_H6, ascensor_H6 )
solDCLH6 = astar_search(ejemplo_H6, ejemplo_H6.directedCheckedLinear).solution()
solDCLH6
```

[396]:
```
[[(0, 3), -1],
 (0, 3), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(0,), -1],
```

```
 [(1,), 1],
 [(1,), 1],
 [(1, 4), 1],
 [(4,), 1],
 [(), -1],
 [(1, 2), 1],
 [(4,), 1],
 [(), -1],
 [(1, 2), 1],
 [(4,), 1],
 [(), -1],
 [(1, 2), 1],
 [(2, 4), 1],
 [(2, 4), 1],
 [(2,), 1]]
```

[397]:
```
getSolutionInfo( solDCLH6, ejemplo_H6 )
```

```
Longitud de la solución: 22. Nodos analizados: 2380
```

[260]:
```
%%timeit
astar_search(ejemplo_H6, ejemplo_H6.directedCheckedLinear).solution()
```

```
7.58 s ± 296 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[398]:
```
ejemplo_H7 = AscensorSimpleHeuristicas( inicial_H7, final_H7, ascensor_H7 )
```

[399]:
```
%%time
solDCLH7 = astar_search(ejemplo_H7, ejemplo_H7.directedCheckedLinear).solution()
```

```
Wall time: 22.2 s
```

[400]:
```
solDCLH7
```

[400]:
```
[[(0, 3), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(0, 3), -1],
 [(), -1],
 [(), -1],
 [(4,), 1],
 [(4,), 1],
 [(4,), 1],
 [(4,), 1],
 [(2, 4), 1],
```

```
 [(2, 4), 1],
 [(2, 4), 1],
 [(2, 4), 1],
 [(2, 4), 1],
 [(2, 4), 1],
 [(), -1],
 [(), -1],
 [(1,), -1],
 [(1,), -1],
 [(1,), -1],
 [(1,), -1],
 [(1,), -1]]
```

[401]:
```
getSolutionInfo( solDCLH7, ejemplo_H7 )
```

```
Longitud de la solución: 25. Nodos analizados: 3455
```

2.0.1 Tabla de comparación.

Mostramos ahora una tabla para comparar la eficiencia de las distintas heurísticas en los ejemplos analizados.

Estado inicial		H1	H2	H3	H4	H5	H6	H7
A (máxima)*	**L**	9	12	16	18	NA	NA	NA
	T	411	424	23500	67000	NA	NA	NA
	NA	691	612	6081	10472	NA	NA	NA
A (lineal ponderada)*	**L**	9	12	16	18	22	NA	NA
	T	118	133	19900	13500	90000	NA	NA
	NA	290	275	4399	4504	12426	NA	NA
A (lineal comprobada)*	**L**	9	12	16	18	22	22	NA
	T	41.5	57.2	1420	3970	15700	53800	NA
	NA	169	171	1074	2323	5205	5803	NA
A (comprobada, dirigida)*	**L**	9	12	16	18	22	22	25
	T	1.58	2.1	116	1060	1500	7580	22200
	NA	12	16	234	957	1107	2380	3455

Se observa con claridad cómo el número de nodos analizados y el tiempo de ejecución mejoran con cada heurística analizada.

Ejemplo del enunciado. Resolvemos el ejemplo del enunciado, pero utilizando un ascensor con carga máxima 3 para hacerlo. Con carga máxima 2, la búsqueda se hace muy costosa.

[281]:
```
inicial_HE = ((2,4,1,8,1), 1)
final_HE =    (3,11,12,1,9)
ascensor_HE = (3,0,12)
ejemplo_HE = AscensorSimpleHeuristicas( inicial_HE, final_HE, ascensor_HE)
```

[282]:
```
%%time
sol_HE = astar_search(ejemplo_HE, ejemplo_HE.directedCheckedLinear).solution()
```

```
Wall time: 1min
```

[283]:
```
print(len(sol_HE))
sol_HE
```

```
22
```

[283]:
```
[[(2, 4), 1],
 [(0, 2, 4), 1],
 [(2, 4), 1],
 [(1, 2, 4), 1],
 [(1, 2, 4), 1],
 [(1, 2, 4), 1],
 [(1, 2, 4), 1],
 [(1, 2, 4), 1],
 [(1, 2), 1],
 [(1, 2), 1],
 [(2,), 1],
 [(), -1],
 [(), -1],
 [(), -1],
 [(), -1],
 [(3,), -1],
 [(3,), -1],
 [(3,), -1],
 [(3,), -1],
 [(3,), -1],
 [(3,), -1],
 [(3,), -1]]
```

3 Apartado 3. Ascensores y bloques.

En este apartado, presentamos una solución al problema consistente en:

- Un edificio de **M** plantas dividido en **K** bloques consecutivos, *{B1,...,BK}*, de forma que, dado 1 <= j < K, el bloque B_j está por debajo del bloque B_(j+1) y estos comparten una única planta (llamamos a estas plantas de **intercambio**).
- Por cada bloque, un único ascensor que opera en sus plantas, con carga máxima propia.
- Un número **N** de pasajeros, cada cual quiere desplazarse de una planta a otra del edificio, desplazándose entre bloques si es necesario.

3.0.1 Procedimiento.

Buscamos evitar tratar el problema como tal, de forma que no sea necesario buscar heurísticas que serían demasiado complejas. Sin embargo, queremos utilizar las heurísticas que hemos desarrol-

lado para un único ascensor. Así es que la mejor manera de resolver este problema es dividirlo en subproblemas que podemos resolver con búsqueda A*.

Delimitación de los subproblemas

En primer lugar, hemos de decidir cómo van a ser nuestros subproblemas. Está claro que para utilizar el método del apartado anterior, cada subproblema ha tener un ascensor: dados **K** bloques, construimos **K** subproblemas, cada uno tratando el ascensor que opera en cada bloque.

Es trivial que cada subproblema no debe operar sobre los **N** pasajeros del edificio, sino solo sobre los que se encuentran en su bloque. Es más, tampoco esto es completamente cierto: hemos de tener en cuenta a los pasajeros que esperan en las plantas de intercambio, que no pueden pertenecer a 2 subproblemas distintos. Para ello, basta con saber, dado un pasajero en una de esas plantas conflictivas, si este sube o baja.

Si sube, pertenecerá al subproblema del bloque superior, pues el ascensor del bloque inferior lo único que podría hacer con él sería bajarlo de planta. Si baja, pertenecerá al subproblema del bloque inferior, por el mismo razonamiento opuesto.

De esta forma, la división en subproblemas supone una partición de los **N** pasajeros del problema. Un conjunto de esta partición es el de los pasajeros que han llegado a su destino, que no pertenece a ningún subproblema.

Objetivo de los subproblemas

Ahora, dado el estado inicial de los pasajeros de un subproblema **Pj**, relativo al bloque **Bj**, hemos de decidir cuáles son sus destinos. Estos no podrán excederse del rango de plantas del ascensor. En concreto, distinguimos 3 casos:

- El destino del pasajero en el problema general está en el mismo bloque Bj. Por tanto, su destino en el subproblema no varía.
- El destino del pasajero en el problema general está en un bloque superior. Por tanto, su destino en el subproblema debe ser la planta de intercambio superior.
- El destino del pasajero en el problema general está en un bloque inferior. Por tanto, su destino en el subproblema debe ser la planta de intercambio inferior.

Una vez hecho esto, tenemos un estado inicial y un objetivo válido para cada subproblema, y la resolución estos nos proporcionará una serie de acciones para cada subproblema. Seguidamente, hemos de decidir cómo aplicarlas.

Método de resolución

Una aproximación a la resolución del problema, sería aplicar todas las acciones derivadas de los subproblemas. De esta forma, si algún subproblema se queda sin acciones porque se ha resuelto antes que el resto, el ascensor de este bloque se quedaría parado para cada una de las acciones restantes. Una vez agotadas las acciones, se volvería a dividir en subproblemas y repetir el proceso; así hasta que se llegue al objetivo general.

Este método llevaría, para cada bloque, a los pasajeros a sus destinos dentro de ese bloque; sin embargo, este provocaría cierta complicación. Tras agotar todas las acciones, nos quedaría un estado general del problema en el que:

1. Los pasajeros que tenían su destino en el mismo bloque en que se encontraban llegarán a él, por lo que podemos descartarlos, para la siguiente iteración.

2. Los pasajeros que tenían su destino fuera del bloque en que estaban acabarán aglomerándose en las plantas de intercambio.

Por ello, sería más costoso para un ascensor con poca carga máxima desplazar a tantas personas en la siguiente iteración. Además, no se correspondería fielmente con la realidad el hecho de que cuando un pasajero llega a una planta de intercambio, este haya de esperar a que el resto llegue también a las demás para usar el ascensor.

Aprovechando que la resolución de los subproblemas mediante heurística es lo suficientemente rápida para un número bajo de plantas, podemos entonces salvar esta complicación, aunque eso nos suponga resolver un mayor número de subproblemas.

En concreto, el método consistiría en, una vez resueltos los subproblemas, aplicar sus acciones paso a paso y comprobar en cada paso si alguno de los pasajeros que tiene que cambiar de bloque ha llegado a su planta de intercambio correspondiente. Cuando esto ocurra, desechamos las acciones restantes y recalculamos los subproblemas en otra iteración. Este método evita que algunos ascensores queden inactivos, esperando a que terminen el resto, a pesar de tener ya pasajeros propios en sus plantas de intercambio inferior y superior.

Observación sobre ámbito de los ascensores

Puesto que los ascensores sólo comparten una planta, el método de resolución recibirá una tupla con **K + 1** plantas, donde el primer y el último número son las plantas límites del edificio (por tanto, la planta inferior del primer bloque y la superior del último), y los números intermedios son las correspondientes plantas de intercambio.

3.0.2 Implementación del método.

En tanto que este problema no utilizará heurísticas propias, lo implementaremos como una función que devuelve el conjunto de acciones del problema general. Cada acción concreta contendrá la acción particular de cada ascensor.

Funciones auxiliares.

[32]:
```
def getBlockProblem(blockSet, blockRange, problemSettings):
    """ Función que, dado un bloque, comprueba para cada pasajero si debe␣
↪pertenecer al subproblema
        de ese bloque. """
    (bInitial, bGoal, bIndex) = blockSet
    (minF, maxF) = blockRange
    (passState, goal, interPassengers) = problemSettings

    passNum = len(passState)
    for j in range(passNum):
        (p, pGoal) = ( passState[j], goal[j] )
        if pGoal < minF: # Destino bajo el bloque.
            if minF < p and p <= maxF: # Y pasajero en el bloque (exc.␣
↪intercambiador inferior)
                bInitial.append(p)
                bGoal.append(minF)
                bIndex.append(j)
```

```
                interPassengers.append( (j, minF) )
        elif maxF < pGoal: # Destino sobre el bloque
            if minF <= p and p < maxF: # Y pasajero en el bloque (exc. intercambiador superior)
                bInitial.append(p)
                bGoal.append(maxF)
                bIndex.append(j)

                interPassengers.append( (j, maxF) )
        else: # Goal in block.
            if minF <= p and p <= maxF: # And initial in block.
                bInitial.append(p)
                bGoal.append(pGoal)
                bIndex.append(j)
```

[33]:
```
def parseBlockAction(blockAction, blockIndex):
    """ Aplica un formato correcto a la acción de un subproblema para
        introducirla en la acción del problema general. """
    mainIndex = list()
    for i in blockAction[0]:
        idx = blockIndex[i]
        mainIndex.append(idx)

    return ( tuple(mainIndex), blockAction[1] )
```

[34]:
```
def doBlockAction(currentState, blockNum, blockAction, blockIndex):
    """ Interpreta correctamente una acción de un subproblema para
        aplicarla al estado del problema general. """
    op = blockAction[1]
    for i in blockAction[0]:
        idx = blockIndex[i]
        currentState[0][idx] += op

    currentState[1][blockNum] += op
```

Código del método.

[37]:
```
def AscensoresBloques(initial, goal, blockSettings):
    # Número de pasajeros del problema general.
    passNum = len( initial[0] )

    # Número de bloques del problema general.
    blockNum = len( initial[1] )

    # Lista mutable que lleva el estado actual del problema, y contiene:
    # > Lista con el estado de los pasajeros.
```

```
    # > Lista con el estado de los ascensores.
    currentState = [ list(initial[0]), list(initial[1]) ]

    # Tupla con la carga máxima del ascensor de cada bloque.
    blockLoads = blockSettings[0]

    # Tupla con las plantas de intercambio entre bloques.
    limitFloors = blockSettings[1]

    # Lista que almacena las acciones de la solución general.
    problemSolution = list()

    # Método de resolución:
    while tuple( currentState[0] ) != goal:
        ## Lista de pasajeros que han de hacer intercambio, y contiene:
        ## > Tuplas de la forma (idx, dest), donde:
        ##   > idx es el índice del pasajero en el problema general.
        ##   > dest es su destino en esta iteración.
        interPassengers = list()

        ## 1. Simplificación a problemas individuales:

        # Lista de tuplas con la información de los subproblemas, cada cual␣
↪contiene:
        # > Lista con los estados 'iniciales' de pasajeros para ese bloque.
        # > Lista con los estados 'objetivo' de pasajeros para ese bloque.
        # > Lista que asocia los pasajeros (índices) del subproblema con el␣
↪problema general.
        blockProblems = list()

        # Para cada bloque, se consigue la información que nos permite construir␣
↪los subproblemas.
        for i in range(blockNum):
            blockSet = [bInitial, bGoal, bIndex] = (list(), list(), list())
            getBlockProblem(
                blockSet,
                ( limitFloors[i] , limitFloors[i + 1] ),
                ( currentState[0], goal, interPassengers )
            )

            blockProblems.append( tuple(blockSet) )

        ## 2. Para cada subproblema se calcula su solución.

        # Lista de soluciones de cada subproblema.
        blockSolutions = list()
```

```
        # Cálculo de soluciones:
        for i in range(blockNum):
            blockInitial = ( tuple( blockProblems[i][0]), currentState[1][i])
            blockGoal = tuple( blockProblems[i][1] )
            blockElev = ( blockLoads[i], limitFloors[i], limitFloors[i + 1] )
            problem = AscensorSimpleHeuristicas(blockInitial, blockGoal,␣
↪blockElev)

            sol = astar_search(problem, problem.directedCheckedLinear).solution()
            blockSolutions.append( tuple(sol) )

        # 3. Se ejecutan acciones hasta que:
        #    > Algún pasajero que debía hacer transbordo llega a su␣
↪intercambiador.
        #    > Ya no quedan acciones en ningún subproblema.

        # Booleano que indica el primer caso de salida.
        inter = False

        # Índice de acción.
        actIdx = 0
        while not inter:
            # Acción del problema general.
            wAction = list()

            # Booleano que indica el segundo caso de salida.
            isAction = False

            # Se ejecuta, para cada subproblema, la acción correspondiente a␣
↪'actIdx'.
            for i in range(blockNum):
                sol = blockSolutions[i]
                if len(sol) > actIdx: # Quedan acciones en el subproblema.
                    isAction = True
                    blockAction = sol[actIdx]
                    blockIndex = blockProblems[i][2]

                    # Se añade (correctamente) la subacción a la acción del␣
↪problema general.
                    wAction.append( parseBlockAction(blockAction, blockIndex) )

                    # Se modifica el estado actual del problema general (en el␣
↪ámbito del bloque i)
                    doBlockAction( currentState, i, blockAction, blockIndex )
                else:
                    # Si no quedan acciones, el ascensor no se mueve.
                    wAction.append( (tuple(), 0) )
```

```
            # Si no quedan acciones en ningún subproblema, salimos.
            if not isAction:
                break

            # Se añade la acción 'actIdx' a la solución general.
            problemSolution.append( wAction.copy() )

            # Se comprueba si algún pasajero de 'transbordo' ha llegado al␣
↪intercambiador.
            for (idx, iGoal) in interPassengers:
                if currentState[0][idx] == iGoal:
                    inter = True
                    break # Deja de comprobar.

            ## Actualización de contador.
            actIdx += 1

    return problemSolution
```

Función de impresión de la solución.

[53]:
```
def printBlockSolution(initialState, solution, blockNum):
    state = [ list(initialState[0]), list(initialState[1]) ]

    print( len(solution), ' movimientos:')
    print('-----------')
    print(state)
    print('___________')

    for action in solution:

        for i in range(blockNum):
            subAction = action[i]
            for j in subAction[0]:
                state[0][j] += subAction[1]

            state[1][i] += subAction[1]

            print(action[i])
        print('-----------')
        print(state)
        print('___________')
```

Ejemplo del enunciado. Resolvemos el ejemplo del enunciado de la práctica, prescindiendo del ascensor rápido y el segundo ascensor del segundo bloque.

[59]:
```
initialEB1 = ( (2,4,1,8,1), (2,6,10) )
goalEB1 = (3,11,12,1,9)
settingsEB1 = ( (2,2,2), (0,4,8,12) )

solEB1 = AscensoresBloques(initialEB1, goalEB1, settingsEB1)
printBlockSolution(initialEB1, solEB1, 3)
```

```
24  movimientos:
-----------
[[2, 4, 1, 8, 1], [2, 6, 10]]
-----------
((), -1)
((), 1)
((), 0)
-----------
[[2, 4, 1, 8, 1], [1, 7, 10]]
-----------
((2, 4), 1)
((), 1)
((), 0)
-----------
[[2, 4, 2, 8, 2], [2, 8, 10]]
-----------
((4,), 1)
((3,), -1)
((), 0)
-----------
[[2, 4, 2, 7, 3], [3, 7, 10]]
-----------
((), -1)
((3,), -1)
((), 0)
-----------
[[2, 4, 2, 6, 3], [2, 6, 10]]
-----------
((0, 2), 1)
((3,), -1)
((), 0)
-----------
[[3, 4, 3, 5, 3], [3, 5, 10]]
-----------
((2, 4), 1)
((3,), -1)
((), 0)
-----------
[[3, 4, 4, 4, 4], [4, 4, 10]]
-----------
```

```
((3,), -1)
((4,), 1)
((), 0)
-----------
[[3, 4, 4, 3, 5], [3, 5, 10]]
-----------
((3,), -1)
((), -1)
((), 0)
-----------
[[3, 4, 4, 2, 5], [2, 4, 10]]
-----------
((3,), -1)
((1, 2), 1)
((), 0)
-----------
[[3, 5, 5, 1, 5], [1, 5, 10]]
-----------
((), 0)
((4,), 1)
((), 0)
-----------
[[3, 5, 5, 1, 6], [1, 6, 10]]
-----------
((), 0)
((), -1)
((), 0)
-----------
[[3, 5, 5, 1, 6], [1, 5, 10]]
-----------
((), 0)
((1, 2), 1)
((), 0)
-----------
[[3, 6, 6, 1, 6], [1, 6, 10]]
-----------
((), 0)
((4,), 1)
((), 0)
-----------
[[3, 6, 6, 1, 7], [1, 7, 10]]
-----------
((), 0)
((), -1)
((), 0)
-----------
[[3, 6, 6, 1, 7], [1, 6, 10]]
-----------
```

```
((), 0)
((1, 2), 1)
((), 0)
-----------
[[3, 7, 7, 1, 7], [1, 7, 10]]
-----------
((), 0)
((4,), 1)
((), 0)
-----------
[[3, 7, 7, 1, 8], [1, 8, 10]]
-----------
((), 0)
((), -1)
((), -1)
-----------
[[3, 7, 7, 1, 8], [1, 7, 9]]
-----------
((), 0)
((1, 2), 1)
((), -1)
-----------
[[3, 8, 8, 1, 8], [1, 8, 8]]
-----------
((), 0)
((), 0)
((4,), 1)
-----------
[[3, 8, 8, 1, 9], [1, 8, 9]]
-----------
((), 0)
((), 0)
((), -1)
-----------
[[3, 8, 8, 1, 9], [1, 8, 8]]
-----------
((), 0)
((), 0)
((1, 2), 1)
-----------
[[3, 9, 9, 1, 9], [1, 8, 9]]
-----------
((), 0)
((), 0)
((1, 2), 1)
-----------
[[3, 10, 10, 1, 9], [1, 8, 10]]
-----------
```

```
((), 0)
((), 0)
((1, 2), 1)
-----------
[[3, 11, 11, 1, 9], [1, 8, 11]]
-----------
((), 0)
((), 0)
((2,), 1)
-----------
[[3, 11, 12, 1, 9], [1, 8, 12]]
-----------
```

[57]:
```
%%timeit
AscensoresBloques(initialEB1, goalEB1, settingsEB1)
```

```
120 ms ± 1.46 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Como en el apartado anterior, podemos lograr una solución más eficiente aumentando la carga máxima de los ascensores en 1.

[61]:
```
initialEB2 = ( (2,4,1,8,1), (2,6,10) )
goalEB2 = (3,11,12,1,9)
settingsEB2 = ( (3,3,3), (0,4,8,12) )

solEB2 = AscensoresBloques(initialEB2, goalEB2, settingsEB2)
printBlockSolution(initialEB2, solEB2, 3)
```

```
16  movimientos:
-----------
[[2, 4, 1, 8, 1], [2, 6, 10]]
-----------
((), -1)
((), 1)
((), 0)
-----------
[[2, 4, 1, 8, 1], [1, 7, 10]]
-----------
((2, 4), 1)
((), 1)
((), 0)
-----------
[[2, 4, 2, 8, 2], [2, 8, 10]]
-----------
((0, 2, 4), 1)
((3,), -1)
((), 0)
-----------
[[3, 4, 3, 7, 3], [3, 7, 10]]
```

```
-----------
((2, 4), 1)
((3,), -1)
((), 0)
-----------
[[3, 4, 4, 6, 4], [4, 6, 10]]
-----------
((), 0)
((3,), -1)
((), 0)
-----------
[[3, 4, 4, 5, 4], [4, 5, 10]]
-----------
((), 0)
((3,), -1)
((), 0)
-----------
[[3, 4, 4, 4, 4], [4, 4, 10]]
-----------
((3,), -1)
((1, 2, 4), 1)
((), 0)
-----------
[[3, 5, 5, 3, 5], [3, 5, 10]]
-----------
((3,), -1)
((1, 2, 4), 1)
((), 0)
-----------
[[3, 6, 6, 2, 6], [2, 6, 10]]
-----------
((3,), -1)
((1, 2, 4), 1)
((), 0)
-----------
[[3, 7, 7, 1, 7], [1, 7, 10]]
-----------
((), 0)
((1, 2, 4), 1)
((), 0)
-----------
[[3, 8, 8, 1, 8], [1, 8, 10]]
-----------
((), 0)
((), 0)
((), -1)
-----------
[[3, 8, 8, 1, 8], [1, 8, 9]]
```

```
-----------
((), 0)
((), 0)
((), -1)
-----------
[[3, 8, 8, 1, 8], [1, 8, 8]]
-----------
((), 0)
((), 0)
((1, 2, 4), 1)
-----------
[[3, 9, 9, 1, 9], [1, 8, 9]]
-----------
((), 0)
((), 0)
((1, 2), 1)
-----------
[[3, 10, 10, 1, 9], [1, 8, 10]]
-----------
((), 0)
((), 0)
((1, 2), 1)
-----------
[[3, 11, 11, 1, 9], [1, 8, 11]]
-----------
((), 0)
((), 0)
((2,), 1)
-----------
[[3, 11, 12, 1, 9], [1, 8, 12]]
-----------
```

Se genera una solución con 16 movimientos, que mejora la solución del apartado anterior (con carga máxima 3) en 6 movimientos.

[63]:
```
%%timeit
AscensoresBloques(initialEB2, goalEB2, settingsEB2)
```

```
6.51 ms ± 349 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

También podemos probar a reducir el número de bloques disponibles.

[65]:
```
initialEB3 = ( (2,4,1,8,1), (3,9) )
goalEB3 = (3,11,12,1,9)
settingsEB3 = ( (3,3), (0,6,12) )

solEB3 = AscensoresBloques(initialEB3, goalEB3, settingsEB3)
printBlockSolution(initialEB3, solEB3, 2)
```

```
13  movimientos:
-----------
[[2, 4, 1, 8, 1], [3, 9]]
-----------
((), -1)
((), -1)
-----------
[[2, 4, 1, 8, 1], [2, 8]]
-----------
((), -1)
((3,), -1)
-----------
[[2, 4, 1, 7, 1], [1, 7]]
-----------
((2, 4), 1)
((3,), -1)
-----------
[[2, 4, 2, 6, 2], [2, 6]]
-----------
((0, 2, 4), 1)
((), 0)
-----------
[[3, 4, 3, 6, 3], [3, 6]]
-----------
((2, 4), 1)
((), 0)
-----------
[[3, 4, 4, 6, 4], [4, 6]]
-----------
((1, 2, 4), 1)
((), 0)
-----------
[[3, 5, 5, 6, 5], [5, 6]]
-----------
((1, 2, 4), 1)
((), 0)
-----------
[[3, 6, 6, 6, 6], [6, 6]]
-----------
((3,), -1)
((1, 2, 4), 1)
-----------
[[3, 7, 7, 5, 7], [5, 7]]
-----------
((3,), -1)
((1, 2, 4), 1)
-----------
[[3, 8, 8, 4, 8], [4, 8]]
```

```
-----------
((3,), -1)
((1, 2, 4), 1)
-----------
[[3, 9, 9, 3, 9], [3, 9]]
-----------
((3,), -1)
((1, 2), 1)
-----------
[[3, 10, 10, 2, 9], [2, 10]]
-----------
((3,), -1)
((1, 2), 1)
-----------
[[3, 11, 11, 1, 9], [1, 11]]
-----------
((), 0)
((2,), 1)
-----------
[[3, 11, 12, 1, 9], [1, 12]]
-----------
```

Observación. Cabe resaltar lo que ocurre cuando aumentamos la carga máxima del ascensor, tanto en estos ejemplos como en los del apartado anterior.

Por un lado, con carga máxima 2, el ascensor, al encontrarse con 3 personas que suben, procede de la siguiente forma:

Nos centramos en el ascensor del segundo bloque: este tiene que subir a los pasajeros de las posiciones 1, 2 y 4 a la planta 8. Sin embargo, en vez de subir primero a 2 pasajeros a la planta 8, bajar a la 4, y subir al que queda a la planta 8 (12 movimientos); su acción es: subir a 1, bajar y subir a los otros 2, repitiendo hasta que están todos en la planta 8 (12 movimientos).

En efecto, este no es el comportamiento real de un ascensor. Para intentar arreglarlo, podríamos añadir un coste adiccional cuando el ascensor cambie de sentido o cuando tenga que pararse para dejar pasajeros en una planta. No obstante, este cálculo no se puede implementar debido a la clase 'Problem', pues la función 'path_cost' no tiene información sobre la acción anterior.

4 Apartado 4. Ascensores dobles. Implementación

El problema representado consiste en:

- Dos ascensores que se desplazan entre las plantas de un edificio, cuyas características son:
  - Una carga máxima de pasajeros.
  - Una planta límite inferior.
  - Una planta límite superior.
- Un número N de pasajeros, cada cual quiere desplazarse con los ascensores de una planta a otra.

4.0.1 Implementación del problema.

Representación de estados y acciones

Para representar este problema nos basamos en los estados y bases que hemos utilizado al resolver el problema de bloques. Es decir, dados **N** pasajeros, los estados son de la forma:

donde la primera tupla representa el estado de los pasajeros, y la segunda las plantas en que se encuentran ambos ascensores.

Por otro lado, para las acciones, supongamos que el primer ascensor actúa sobre **K** pasajeros, mientras el segundo lo hace sobre **L**. Así, las acciones son de la forma:

donde las tuplas son los índices de los pasajeros que desplaza cada ascensor y **op1, op2** son el movimiento del ascensor.

Cálculo de las acciones

Para calcular las acciones posibles de un estado, procedemos para cada ascensor de forma análoga a como hicimos con el ascensor simple. Tomando para cada acción del primer ascensor el resto de acciones del segundo obtenemos el conjunto de acciones del estado. No obstante, hemos de comprobar antes de hacer esto si las plantas de los ascensores coinciden; si es así, debemos descartar las acciones en las que ambos ascensores mueven a un mismo pasajero.

Por otro lado, añadimos también las acciones en que un ascensor puede quedarse parado.

4.0.2 Código.

[70]:
```
def intersection(lst1, lst2):
    lst3 = [value for value in lst1 if value in lst2]
    return lst3
```

[75]:
```
class AscensoresDobles(Problem):
    def __init__(self, initial, goal, elevSettings):
        # Tupla con el estado inicial, que contiene:
        # > Tupla con las posiciones de los pasajeros.
        # > Tupla con las platas iniciales de los ascensores.
        self.initial = initial

        # Tupla con las posiciones finales de los pasajeros.
        self.goal = goal

        # Tupla con carga máxima de pasajeros en cada ascensor.
        self.elevLoad = elevSettings[0]

        # Planta más baja a la que llegan los ascensores.
        self.minFloor = elevSettings[1]

        # Planta más alta a la que llegan los ascensores.
        self.maxFloor = elevSettings[2]
```

```
        Problem.__init__(self, initial, goal)

        #-------------------------------------

        # Número de pasajeros del problema.
        self.passNum = len( initial[0] )

        # Número de ascensores del problema.
        self.elevNum = 2

        # Lista que marca si un pasajero sube o baja (o ninguna de las dos)
        self.passUpDown = getUpDowns(self.passNum, initial[0], goal)
    ##-------------------------------------------------------------------------

    def actions(self, state):
        """ Las acciones consisten de una tupla formada por:
            > Tupla con los pasajeros que desplaza el ascensor.
            > Movimiento del ascensor (arriba o abajo) """
        # Estado de los pasajeros.
        passState = state[0]

        # Planta del ascensor.
        (eF1, eF2) = state[1]

        # Cargas de los ascensores.
        (eL1, eL2) = self.elevLoad

        # Lista de acciones (1).
        actionSet1 = list()

        # 1. Tomamos las acciones posibles del ascensor (1):
        elevOps1 = getOps(1, eF1, self.minFloor, self.maxFloor)

        # 2. Buscamos las acciones posibles de los pasajeros (1):
        if eF1 in passState:
            ## De los que suben y los que bajan:
            for elevOp in elevOps1:
                getPassActions(
                    actionSet1,
                    (elevOp, eF1, eL1),
                    (self.passNum, passState, self.goal, self.passUpDown)
                )

        ## 3. Movimientos del ascensor sin pasajeros (1):
        for elevOp in elevOps1:
            action = [ tuple(), elevOp ]
            actionSet1.append( action.copy() )
```

```
        actionSet1.append( [ tuple(), 0 ] )

        # Lista de acciones (2).
        actionSet2 = list()

        # 4. Tomamos las acciones posibles del ascensor (2):
        elevOps2 = getOps(1, eF2, self.minFloor, self.maxFloor)

        # 5. Buscamos las acciones posibles de los pasajeros (2):
        if eF2 in passState:
            ## De los que suben y los que bajan:
            for elevOp in elevOps2:
                getPassActions(
                    actionSet2,
                    (elevOp, eF2, eL2),
                    (self.passNum, passState, self.goal, self.passUpDown)
                )

        # 6. Movimientos del ascensor sin pasajeros (2):
        for elevop in elevOps2:
            action = [ tuple(), elevOp ]
            actionSet2.append( action. copy() )

        actionSet2.append( [ tuple(), 0 ] )

        # 7. Agrupamos las acciones:

        actionSet = list()
        if eF1 != eF2:
            for act1 in actionSet1:
                for act2 in actionSet2:
                    actionSet.append( [act1, act2] )
        else:
            for act1 in actionSet1:
                for act2 in actionSet2:
                    if len ( intersection(act1[0], act2[0]) ) == 0:
                        actionSet.append( [act1, act2] )

        return actionSet
    ##------------------------------------------------------------------------


    def result(self, state, action):
        """ Dada una acción, se recorre la tupla de pasajeros
            desplazados por el ascensor (action[0]) y se suma
            a su posición en el estado (state[0]) (la previa
```

```
            al desplazamiento) el movimiento del ascensor arriba
            o abajo (action[1]).

            Se actualiza también la posición del ascensor (state[1]),
            sumándole el movimiento del mismo (action[1]). """
        # Lista del estado de los pasajeros previo a la acción.
        passResult = list( state[0] )

        for i in action[0][0]:
            passResult[i] += action[0][1]

        newEF1 = state[1][0] + action[0][1]

        for i in action[1][0]:
            passResult[i] += action[1][1]

        newEF2 = state[1][1] + action[1][1]

        # Devolvemos una tupla con un nuevo estado, que contiene:
        # > Tupla con los pasajeros actualizados.
        # > Tupla con las nuevas posiciones de los ascensores.
        return ( tuple(passResult), (newEF1, newEF2) )

    def goal_test(self, state):
        # Estado de los pasajeros
        passState = state[0]

        # Se comprueba si es igual al objetivo.
        return passState == self.goal

    def path_cost(self, c, state1, action, state2):
        # En toda acción pasa siempre una unidad de tiempo.
        # El coste aumenta en 1:
        return c + 1
    #______________________________________________________________________________
```

Veamos el funcionamiento de estas funciones con un ejemplo:

[76]:
```
d1_inicial = ((0,1,5,5,5,5,5,5,8,10), (5,5))
d1_final =    (1,2,3,4,5,6,7,8,9,10)
d1_ascensores = ((3,2),0,12)
```

[77]:
```
edoble1 = AscensoresDobles(d1_inicial, d1_final, d1_ascensores)
```

[78]:
```
edoble1.actions(d1_inicial)
```

```
[78]: [[[(5,), 1], [(6,), 1]],
       [[(5,), 1], [(7,), 1]],
       [[(5,), 1], [(6, 7), 1]],
       [[(5,), 1], [(2,), -1]],
       [[(5,), 1], [(3,), -1]],
       [[(5,), 1], [(2, 3), -1]],
       [[(5,), 1], [(), -1]],
       [[(5,), 1], [(), -1]],
       [[(5,), 1], [(), 0]],
       [[(6,), 1], [(5,), 1]],
       [[(6,), 1], [(7,), 1]],
       [[(6,), 1], [(5, 7), 1]],
       [[(6,), 1], [(2,), -1]],
       [[(6,), 1], [(3,), -1]],
       [[(6,), 1], [(2, 3), -1]],
       [[(6,), 1], [(), -1]],
       [[(6,), 1], [(), -1]],
       [[(6,), 1], [(), 0]],
       [[(7,), 1], [(5,), 1]],
       [[(7,), 1], [(6,), 1]],
       [[(7,), 1], [(5, 6), 1]],
       [[(7,), 1], [(2,), -1]],
       [[(7,), 1], [(3,), -1]],
       [[(7,), 1], [(2, 3), -1]],
       [[(7,), 1], [(), -1]],
       [[(7,), 1], [(), -1]],
       [[(7,), 1], [(), 0]],
       [[(5, 6), 1], [(7,), 1]],
       [[(5, 6), 1], [(2,), -1]],
       [[(5, 6), 1], [(3,), -1]],
       [[(5, 6), 1], [(2, 3), -1]],
       [[(5, 6), 1], [(), -1]],
       [[(5, 6), 1], [(), -1]],
       [[(5, 6), 1], [(), 0]],
       [[(5, 7), 1], [(6,), 1]],
       [[(5, 7), 1], [(2,), -1]],
       [[(5, 7), 1], [(3,), -1]],
       [[(5, 7), 1], [(2, 3), -1]],
       [[(5, 7), 1], [(), -1]],
       [[(5, 7), 1], [(), -1]],
       [[(5, 7), 1], [(), 0]],
       [[(6, 7), 1], [(5,), 1]],
       [[(6, 7), 1], [(2,), -1]],
       [[(6, 7), 1], [(3,), -1]],
       [[(6, 7), 1], [(2, 3), -1]],
       [[(6, 7), 1], [(), -1]],
       [[(6, 7), 1], [(), -1]],
```

```
[[(6, 7), 1], [(), 0]],
[[(5, 6, 7), 1], [(2,), -1]],
[[(5, 6, 7), 1], [(3,), -1]],
[[(5, 6, 7), 1], [(2, 3), -1]],
[[(5, 6, 7), 1], [(), -1]],
[[(5, 6, 7), 1], [(), -1]],
[[(5, 6, 7), 1], [(), 0]],
[[(2,), -1], [(5,), 1]],
[[(2,), -1], [(6,), 1]],
[[(2,), -1], [(7,), 1]],
[[(2,), -1], [(5, 6), 1]],
[[(2,), -1], [(5, 7), 1]],
[[(2,), -1], [(6, 7), 1]],
[[(2,), -1], [(3,), -1]],
[[(2,), -1], [(), -1]],
[[(2,), -1], [(), -1]],
[[(2,), -1], [(), 0]],
[[(3,), -1], [(5,), 1]],
[[(3,), -1], [(6,), 1]],
[[(3,), -1], [(7,), 1]],
[[(3,), -1], [(5, 6), 1]],
[[(3,), -1], [(5, 7), 1]],
[[(3,), -1], [(6, 7), 1]],
[[(3,), -1], [(2,), -1]],
[[(3,), -1], [(), -1]],
[[(3,), -1], [(), -1]],
[[(3,), -1], [(), 0]],
[[(2, 3), -1], [(5,), 1]],
[[(2, 3), -1], [(6,), 1]],
[[(2, 3), -1], [(7,), 1]],
[[(2, 3), -1], [(5, 6), 1]],
[[(2, 3), -1], [(5, 7), 1]],
[[(2, 3), -1], [(6, 7), 1]],
[[(2, 3), -1], [(), -1]],
[[(2, 3), -1], [(), -1]],
[[(2, 3), -1], [(), 0]],
[[(), 1], [(5,), 1]],
[[(), 1], [(6,), 1]],
[[(), 1], [(7,), 1]],
[[(), 1], [(5, 6), 1]],
[[(), 1], [(5, 7), 1]],
[[(), 1], [(6, 7), 1]],
[[(), 1], [(2,), -1]],
[[(), 1], [(3,), -1]],
[[(), 1], [(2, 3), -1]],
[[(), 1], [(), -1]],
[[(), 1], [(), -1]],
```

```
 [[(), 1], [(), 0]],
 [[(), -1], [(5,), 1]],
 [[(), -1], [(6,), 1]],
 [[(), -1], [(7,), 1]],
 [[(), -1], [(5, 6), 1]],
 [[(), -1], [(5, 7), 1]],
 [[(), -1], [(6, 7), 1]],
 [[(), -1], [(2,), -1]],
 [[(), -1], [(3,), -1]],
 [[(), -1], [(2, 3), -1]],
 [[(), -1], [(), -1]],
 [[(), -1], [(), -1]],
 [[(), -1], [(), 0]],
 [[(), 0], [(5,), 1]],
 [[(), 0], [(6,), 1]],
 [[(), 0], [(7,), 1]],
 [[(), 0], [(5, 6), 1]],
 [[(), 0], [(5, 7), 1]],
 [[(), 0], [(6, 7), 1]],
 [[(), 0], [(2,), -1]],
 [[(), 0], [(3,), -1]],
 [[(), 0], [(2, 3), -1]],
 [[(), 0], [(), -1]],
 [[(), 0], [(), -1]],
 [[(), 0], [(), 0]]]
```

[79]:
```
edoble1.result(d1_inicial, [[(5, 6), 1], [(2, 3), -1]])
```

[79]:
```
((0, 1, 4, 4, 5, 6, 6, 5, 8, 10), (6, 4))
```

Vemos otro ejemplo simple para resolverlo con los métodos de búsqueda no informados.

[81]:
```
d2_inicial = ((0,2,4), (1,3))
d2_final =    (3,0,1)
d2_ascensores = ((2,2), 0,4)
```

[82]:
```
edoble2 = AscensoresDobles(d2_inicial, d2_final, d2_ascensores)
```

[ ]:
```

```

Búsqueda en anchura (control de repetidos).

[84]:
```
breadth_first_graph_search(edoble2).solution()
```

[84]:
```
[[[(), 1], [(), -1]],
 [[(), 1], [(1,), -1]],
 [[(), 1], [(1,), -1]],
```

```
 [[(2,), -1], [(0,), 1]],
 [[(2,), -1], [(0,), 1]],
 [[(0,), 1], [(2,), -1]]]
```

[88]:
```
%%timeit
breadth_first_graph_search(edoble2).solution()
```

```
160 ms ± 7.58 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Coste uniforme.

[85]:
```
uniform_cost_search(edoble2).solution()
```

[85]:
```
[[[(), -1], [(), 0]],
 [[(), 0], [(), 1]],
 [[(0,), 1], [(2,), -1]],
 [[(0,), 1], [(2,), -1]],
 [[(1, 2), -1], [(0,), 1]],
 [[(1,), -1], [(), -1]]]
```

[86]:
```
%%timeit
uniform_cost_search(edoble2).solution()
```

```
3.01 s ± 46.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

5 Apartado 5. Ascensores Dobles. Heurística.

Como en con las anteriores heurísticas, la que vamos a explicar necesita información que tiene el objeto del Problema, por lo que ahora ponemos el código de la nueva clase con la heurística implementada, y explicamos esta a continuación.

[93]:
```
class AscensoresDoblesHeuristicas(Problem):
    def __init__(self, initial, goal, elevSettings):
        # Tupla con el estado inicial, que contiene:
        # > Tupla con las posiciones de los pasajeros.
        # > Tupla con las platas iniciales de los ascensores.
        self.initial = initial

        # Tupla con las posiciones finales de los pasajeros.
        self.goal = goal

        # Tupla con carga máxima de pasajeros en cada ascensor.
        self.elevLoad = elevSettings[0]

        # Planta más baja a la que llegan los ascensores.
        self.minFloor = elevSettings[1]

        # Planta más alta a la que llegan los ascensores.
```

```
        self.maxFloor = elevSettings[2]

        Problem.__init__(self, initial, goal)

        #-----------------------------------

        # Número de pasajeros del problema.
        self.passNum = len( initial[0] )

        # Número de ascensores del problema.
        self.elevNum = 2

        # Lista que marca si un pasajero sube o baja (o ninguna de las dos)
        self.passUpDown = getUpDowns(self.passNum, initial[0], goal)
    ##-------------------------------------------------------------------------

    def actions(self, state):
        """ Las acciones consisten de una tupla formada por:
            > Tupla con los pasajeros que desplaza el ascensor.
            > Movimiento del ascensor (arriba o abajo) """
        # Estado de los pasajeros.
        passState = state[0]

        # Planta del ascensor.
        (eF1, eF2) = state[1]

        # Cargas de los ascensores.
        (eL1, eL2) = self.elevLoad

        # Lista de acciones (1).
        actionSet1 = list()

        # 1. Tomamos las acciones posibles del ascensor (1):
        elevOps1 = getOps(1, eF1, self.minFloor, self.maxFloor)

        # 2. Buscamos las acciones posibles de los pasajeros (1):
        if eF1 in passState:
            ## De los que suben y los que bajan:
            for elevOp in elevOps1:
                getPassActions(
                    actionSet1,
                    (elevOp, eF1, eL1),
                    (self.passNum, passState, self.goal, self.passUpDown)
                )

        ## 3. Movimientos del ascensor sin pasajeros (1):
        for elevOp in elevOps1:
```

```
            action = [ tuple(), elevOp ]
            actionSet1.append( action.copy() )

        actionSet1.append( [ tuple(), 0 ] )

        # Lista de acciones (2).
        actionSet2 = list()

        # 4. Tomamos las acciones posibles del ascensor (2):
        elevOps2 = getOps(1, eF2, self.minFloor, self.maxFloor)

        # 5. Buscamos las acciones posibles de los pasajeros (2):
        if eF2 in passState:
            ## De los que suben y los que bajan:
            for elevOp in elevOps2:
                getPassActions(
                    actionSet2,
                    (elevOp, eF2, eL2),
                    (self.passNum, passState, self.goal, self.passUpDown)
                )

        # 6. Movimientos del ascensor sin pasajeros (2):
        for elevop in elevOps2:
            action = [ tuple(), elevOp ]
            actionSet2.append( action. copy() )

        actionSet2.append( [ tuple(), 0 ] )

        # 7. Agrupamos las acciones:

        actionSet = list()
        if eF1 != eF2:
            for act1 in actionSet1:
                for act2 in actionSet2:
                    actionSet.append( [act1, act2] )
        else:
            for act1 in actionSet1:
                for act2 in actionSet2:
                    if len ( intersection(act1[0], act2[0]) ) == 0:
                        actionSet.append( [act1, act2] )

        return actionSet
    ##-------------------------------------------------------------------------


    def result(self, state, action):
        """ Dada una acción, se recorre la tupla de pasajeros
```

```
            desplazados por el ascensor (action[0]) y se suma
            a su posición en el estado (state[0]) (la previa
            al desplazamiento) el movimiento del ascensor arriba
            o abajo (action[1]).

            Se actualiza también la posición del ascensor (state[1]),
            sumándole el movimiento del mismo (action[1]). """
        # Lista del estado de los pasajeros previo a la acción.
        passResult = list( state[0] )

        for i in action[0][0]:
            passResult[i] += action[0][1]

        newEF1 = state[1][0] + action[0][1]

        for i in action[1][0]:
            passResult[i] += action[1][1]

        newEF2 = state[1][1] + action[1][1]

        # Devolvemos una tupla con un nuevo estado, que contiene:
        # > Tupla con los pasajeros actualizados.
        # > Tupla con las nuevas posiciones de los ascensores.
        return ( tuple(passResult), (newEF1, newEF2) )

    def goal_test(self, state):
        # Estado de los pasajeros
        passState = state[0]

        # Se comprueba si es igual al objetivo.
        return passState == self.goal

    def path_cost(self, c, state1, action, state2):
        # En toda acción pasa siempre una unidad de tiempo.
        # El coste aumenta en 1:
        return c + 1
    #______________________________________________________________________________


    ### HEURÍSTICA LINEAL 'DIRIGIDA' Y 'COMPROBADA'

    def DDCLinear(self, node):
        # Estado de los pasajeros:
        passState = node.state[0]

        # Carga máxima agregada.
        maxLoad = max(self.elevLoad)
```

```
        # Carga de cada transición hacia arriba entre plantas (inicialmente 0):
        upLoads = [ 0 for i in range(self.minFloor, self.maxFloor) ]

        # 1. Se calculan las cargas para los pasajeros que suben:
        for i in range(self.passNum):
            if self.passUpDown[i] > 0: # El pasajero sube.
                for j in range( passState[i], self.goal[i] ):
                    upLoads[j - self.minFloor] += 1

        for i in range(self.maxFloor - self.minFloor):
            load = upLoads[i]
            if load != 0:
                if load % maxLoad == 0:
                    upLoads[i] = load // maxLoad
                else:
                    upLoads[i] = load // maxLoad + 1

        upDistsNum = max(upLoads)

        upDists = list()
        for n in range(upDistsNum):
            newDist = True
            dist = 0
            for i in range(self.maxFloor - self.minFloor):
                load = upLoads[i]
                if load > 0:
                    upLoads[i] -= 1
                    dist += 1
                    if newDist:
                        newDist = False
                elif load == 0:
                    if not newDist:
                        upDists.append(dist)
                        dist = 0
                        newDist = True
            if not newDist:
                upDists.append(dist)

        # Carga de cada transición hacia abajo entre plantas (inicialmente 0):
        downLoads = [ 0 for i in range(self.minFloor, self.maxFloor) ]

        # 3. Análogo para los pasajeros que bajan:
        for i in range(self.passNum):
            if self.passUpDown[i] < 0: # El pasajero sube.
                for j in range( self.goal[i], passState[i] ):
                    downLoads[j - self.minFloor] += 1
```

```
        for i in range(self.maxFloor - self.minFloor):
            load = downLoads[i]
            if load != 0:
                if load % maxLoad == 0:
                    downLoads[i] = load // maxLoad
                else:
                    downLoads[i] = load // maxLoad + 1

        downDistsNum = max(downLoads)

        downDists = list()
        for n in range(downDistsNum):
            newDist = True
            dist = 0
            for i in range(self.maxFloor - self.minFloor):
                load = downLoads[i]
                if load > 0:
                    downLoads[i] -= 1
                    dist += 1
                    if newDist:
                        newDist = False
                elif load == 0:
                    if not newDist:
                        downDists.append(dist)
                        dist = 0
                        newDist = True
            if not newDist:
                downDists.append(dist)

        ## Cola de prioridad.
        pQueue = PriorityQueue(order = 'max')
        for dist in upDists:
            pQueue.append(dist)
        for dist in downDists:
            pQueue.append(dist)

        cost = 0
        while len( pQueue ) > 1: # Al menos una dupla
            d1 = pQueue.pop() # Máximo
            d2 = pQueue.pop()
            diff = d1 - d2
            cost += d1

            saving = True
            while saving and len( pQueue ) > 0:
                d3 = pQueue.pop()
```

```
                if d3 >= diff:
                    d3 -= diff
                    pQueue.append(d3)
                    saving = False
                else:
                    diff -= d3

        if len( pQueue ) > 0:
            d = pQueue.pop()
            cost += d

        return cost
    #--------------------------------------------------------------------------
```

5.0.1 Heurística para ascensores dobles.

Introducción

El hecho de que haya 2 ascensores disponibles tiene como consecuencia que se pueden bajar y subir pasajeros a la vez, lo que reduce el número de movimientos necesarios y, a su vez, invalida las heurísticas construidas para un ascensor único.

Por ejemplo, supongamos un problema simple en el que el estado inicial de los pasajeros es (0,12) y su destino (8,4). Es obvio que un ascensor tendría que realizar los 2 viajes, de la planta 0 a la 8 y luego de la planta 12 a la 4. No obstante, con 2 ascensores, estos podrían hacer ambos viajes a la vez, siendo el caso más simple que ambos se encuentren en las plantas iniciales 0 y 12 (el coste final sería el del viaje más largo: en este caso es 8 en ambos).

Así, para tratar este problema, comenzamos relajándolo de forma que los 2 ascensores tengan carga máxima 1. Para obtener un coste mínimo de este problema, podemos suponer que inicialmente, los 2 ascensores hacen un viaje cada uno. Para el primero que acabe, suponemos que comienza a hacer de forma inmediate otro viaje disponible; y lo mismo hace el segundo. Se procede así hasta acabar los viajes, y el *tiempo* que hayamos medido con este método es claramente un coste mínimo del problema relajado

Una manera de calcular este simple de forma simple es ordenar los viajes disponibles por su longitud, de mayor a menor. Siempre que en la lista ordenada haya al menos 2 elementos, los extraemos y procedemos así:

1. Sumamos al coste la longitud máxima de entre ambos viajes (que será la del que hemos extraído primero) y calculamos la diferencia de longitud entre ambos (nunca negativa).
2. Mientras queden elementos en la lista, extraemos uno y:
    - Si ese elemento es mayor o igual que la diferencia, le restamos la diferencia e insertamos el resultado en la lista ordenada. Volvemos al paso 1.
    - Si el elemento es menor que la diferencia, lo descartamos y restamos la longitud a la diferencia. Vuelta al paso 2.

Cuando en la lista queden menos de 2 elementos, comprobamos si está vacía o queda uno:

- Si está vacía no hacemos nada.

- Si queda uno, le sumamos la longitud del mismo al coste.

Nos queda finalmente el coste mínimo que buscábamos.

Heurística

Ahora, volvemos al problema original, en el que la carga máxima puede ser mayor que 1. Utilizaremos el ejemplo del enunciado para ilustrar el método de esta heurística. En primer lugar, retomamos el proceso de la heurística *dirigida y comprobada* del ascensor simple, y calculamos la acumulación de pases en las transiciones.

SUBEN:

Pasajero	0 ~ 1	1 ~ 2	2 ~ 3	3 ~ 4	4 ~ 5	5 ~ 6	6 ~ 7	7 ~ 8	8 ~ 9	9 ~ 10	10 ~ 11	11 ~ 12	+
0	0	0	1	0	0	0	0	0	0	0	0	0	
1	0	0	0	0	1	1	1	1	1	1	1	0	
2	0	1	1	1	1	1	1	1	1	1	1	1	
4	0	1	1	1	1	1	1	1	1	0	0	0	
Pases	0	2	3	2	3	3	3	3	3	2	2	1	
LOAD = 2	0	1	2	1	2	2	2	2	2	1	1	1	= 17

BAJAN:

Pasajero	0 ~ 1	1 ~ 2	2 ~ 3	3 ~ 4	4 ~ 5	5 ~ 6	6 ~ 7	7 ~ 8	8 ~ 9	9 ~ 10	10 ~ 11	11 ~ 12	+
3	0	1	1	1	1	1	1	1	0	0	0	0	
Pases	0	1	1	1	1	1	1	1	0	0	0	0	
LOAD = 2	0	1	1	1	1	1	1	1	0	0	0	0	= 7

Recordemos que la fila final de cada tabla indica el número de movimientos que han de hacerse necesariamente para cada transición. Nótese que este cálculo ya no tiene en cuenta la carga de los ascensores, sino que simplemente constatan que el ascensor ha de pasar (subiendo o bajando) por esas transiciones tantas veces como se indica.

Es decir, podemos traducir estas filas a un problema equivalente de ascensores dobles con carga máxima 1. Vemos como:

Para la fila de subida, podemos hacer:

=	0	1	2	1	2	2	2	2	2	1	1	1
+	0	1	1	1	1	1	1	1	1	1	1	1
+	0	0	0	0	1	1	1	1	1	0	0	0
+	0	0	1	0	0	0	0	0	0	0	0	0

Hemos extraído 3 viajes distintos de subida, dados por el estado inicial (1,4,2), y el estado final (12,9,3).

De forma análoga con la fila de bajada, obtenemos un sólo viaje de 8 a 1.

=	0	1	1	1	1	1	1	1	0	0	0	0
+	0	1	1	1	1	1	1	1	0	0	0	0

Ahora, calculamos el coste mínimo utilizando la lista ordenada, que nos queda (11,7,5,1).

1. Extremos el 11 y el 7; sumamos 11 al coste inicial (0) y tomamos la diferencia *4 = 11 - 7*.
2. Tomamos 5 y le restamos la diferencia; queda 1, que insertamos en la lista, que ahora es (1,1)
3. Extremos los dos últimos 1; sumamos 1 al coste inicial.

Así, nos queda un coste mínimo de 12 movimientos.

Observaciones

En el ejemplo anterior hemos tomado como carga máxima 2. Esto quiere decir que ninguno de los ascensores supera esa carga máxima por separado. Así, si los ascensores tuviesen carga máxima distinta, tendríamos que elegir la mayor para calcular esta heurística.

Por otro lado, cuando descomponemos la fila de acumulación de pases por transición, lo hacemos de forma que nos queden los viajes más largos posibles.

Ejemplos. Para probar esta heurística, reutilizaremos los ejemplos que utilizamos para las heurísticas del ascensor simple, adaptándolos correctamente al problema de los ascensores dobles.

[94]:
```
iniDoble_H1 = ((0,3,3,1,4), (3,0))
final_H1 =    (4,0,1,0,2)
ascDoble_H1 = ((2,2),0,4)
ejDoble_H1 = AscensoresDoblesHeuristicas( iniDoble_H1, final_H1, ascDoble_H1 )
```

[95]:
```
iniDoble_H2 = ((4,5,1,0,4), (2,3))
final_H2 =    (0,3,5,3,5)
ascDoble_H2 = ((2,2),0,5)
ejDoble_H2 = AscensoresDoblesHeuristicas( iniDoble_H2, final_H2, ascDoble_H2 )
```

[96]:
```
iniDoble_H3 = ((5,3,1,6,6), (3,6))
final_H3 =    (3,0,5,1,3)
ascDoble_H3 = ((2,2),0,6)
ejDoble_H3 = AscensoresDoblesHeuristicas( iniDoble_H3, final_H3, ascDoble_H3 )
```

[126]:
```
iniDoble_H4 = ((0,2,7,3,4), (7,0))
final_H4 =    (6,5,2,6,2)
ascDoble_H4 = ((3,3),0,7)
ejDoble_H4 = AscensoresDoblesHeuristicas( iniDoble_H4, final_H4, ascDoble_H4 )
```

[127]:
```
iniDoble_H5 = ((8,7,8,0,4), (5,5))
final_H5 =    (4,3,2,5,8)
ascDoble_H5 = ((3,3),0,8)
ejDoble_H5 = AscensoresDoblesHeuristicas( iniDoble_H5, final_H5, ascDoble_H5 )
```

[139]:
```
iniDoble_H6 = ((7,0,3,7,2), (7,0))
final_H6 =    (0,6,9,1,8)
ascDoble_H6 = ((3,3),0,9)
ejDoble_H6 = AscensoresDoblesHeuristicas( iniDoble_H6, final_H6, ascDoble_H6 )
```

[142]:
```
iniDoble_H7 = ((8,8,4,8,0), (8,0))
final_H7 =    (2,3,10,2,10)
ascDoble_H7 = ((3,3),0,10)
ejDoble_H7 = AscensoresDoblesHeuristicas( iniDoble_H7, final_H7, ascDoble_H7 )
```

[111]:
```
solDobleH1 = astar_search(ejDoble_H1, ejDoble_H1.DDCLinear).solution()
print( len(solDobleH1) )
solDobleH1
```

```
6
```

[111]:
```
[[[(), 1], [(0,), 1]],
 [[(4,), -1], [(0,), 1]],
 [[(1, 2), -1], [(0,), 1]],
 [[(), 0], [(4,), -1]],
 [[(), 1], [(1, 2), -1]],
 [[(0,), 1], [(1, 3), -1]]]
```

[112]:
```
%%timeit
astar_search(ejDoble_H1, ejDoble_H1.DDCLinear)
```

```
3.93 s ± 171 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[113]:
```
solDobleH2 = astar_search(ejDoble_H2, ejDoble_H2.DDCLinear).solution()
print( len(solDobleH2) )
solDobleH2
```

```
8
```

[113]:
```
[[[(), 1], [(), -1]],
 [[(), 1], [(), -1]],
 [[(), 1], [(), -1]],
 [[(1,), -1], [(3,), 1]],
 [[(0, 1), -1], [(2, 3), 1]],
 [[(0,), -1], [(2, 3), 1]],
 [[(0,), -1], [(2,), 1]],
 [[(0,), -1], [(2, 4), 1]]]
```

[114]:
```
%%timeit
astar_search(ejDoble_H2, ejDoble_H2.DDCLinear)
```

```
65.8 ms ± 1.76 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[115]:
```
solDobleH3 = astar_search(ejDoble_H3, ejDoble_H3.DDCLinear).solution()
print( len(solDobleH3) )
solDobleH3
```

```
8
```

[115]:
```
[[[(1,), -1], [(3, 4), -1]],
 [[(1,), -1], [(0, 3), -1]],
 [[(2,), 1], [(0, 3), -1]],
 [[(2,), 1], [(3,), -1]],
 [[(2,), 1], [(3,), -1]],
 [[(2,), 1], [(1,), -1]],
 [[(4,), -1], [(), 0]],
 [[(4,), -1], [(), 0]]]
```

[116]:
```
%%timeit
astar_search(ejDoble_H3, ejDoble_H3.DDCLinear)
```

```
1.71 s ± 42.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

A partir de las 8 plantas, la búsqueda A* tarda mucho más (más de 3 minutos). Sin embargo, la podemos disminuir el tiempo si damos más capacidad a los ascensores, consiguiendo mejores tiempos.

[128]:
```
solDobleH4 = astar_search(ejDoble_H4, ejDoble_H4.DDCLinear).solution()
print( len(solDobleH4) )
solDobleH4
```

```
6
```

[128]:
```
[[[(2,), -1], [(0,), 1]],
 [[(2,), -1], [(0,), 1]],
 [[(2,), -1], [(0, 1), 1]],
 [[(2, 4), -1], [(0, 1, 3), 1]],
 [[(2, 4), -1], [(0, 1, 3), 1]],
 [[(), -1], [(0, 3), 1]]]
```

[130]:
```
%%timeit
astar_search(ejDoble_H4, ejDoble_H4.DDCLinear)
```

```
274 ms ± 23.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[129]:
```
solDobleH5 = astar_search(ejDoble_H5, ejDoble_H5.DDCLinear).solution()
print( len(solDobleH5) )
solDobleH5
```

```
11
```

[129]:
```
[[[(), -1], [(), -1]],
 [[(), -1], [(4,), 1]],
 [[(), -1], [(4,), 1]],
 [[(), -1], [(4,), 1]],
 [[(), -1], [(4,), 1]],
 [[(), 0], [(0, 2), -1]],
 [[(3,), 1], [(0, 1, 2), -1]],
 [[(3,), 1], [(0, 1, 2), -1]],
 [[(3,), 1], [(0, 1, 2), -1]],
 [[(3,), 1], [(1, 2), -1]],
 [[(3,), 1], [(2,), -1]]]
```

[131]:
```
%%timeit
astar_search(ejDoble_H5, ejDoble_H5.DDCLinear)
```

```
12.7 s ± 275 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[140]:
```
solDobleH6 = astar_search(ejDoble_H6, ejDoble_H6.DDCLinear).solution()
print( len(solDobleH6) )
solDobleH6
```

```
9
```

[140]:
```
[[[(0, 3), -1], [(1,), 1]],
 [[(0, 3), -1], [(1,), 1]],
 [[(0, 3), -1], [(1, 4), 1]],
 [[(0, 3), -1], [(1, 2, 4), 1]],
 [[(0, 3), -1], [(1, 2, 4), 1]],
 [[(0, 3), -1], [(1, 2, 4), 1]],
 [[(0,), -1], [(2, 4), 1]],
 [[(), 0], [(2, 4), 1]],
 [[(), 0], [(2,), 1]]]
```

[135]:
```
%%timeit
astar_search(ejDoble_H6, ejDoble_H6.DDCLinear)
```

```
54.2 ms ± 3.67 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[143]:
```
solDobleH7 = astar_search(ejDoble_H7, ejDoble_H7.DDCLinear).solution()
print( len(solDobleH7) )
solDobleH7
```

```
10
```

[143]:
```
[[[(0, 1, 3), -1], [(4,), 1]],
 [[(0, 1, 3), -1], [(4,), 1]],
 [[(0, 1, 3), -1], [(4,), 1]],
 [[(0, 1, 3), -1], [(4,), 1]],
 [[(0, 1, 3), -1], [(2, 4), 1]],
 [[(0, 3), -1], [(2, 4), 1]],
 [[(), -1], [(2, 4), 1]],
 [[(), -1], [(2, 4), 1]],
 [[(), 0], [(2, 4), 1]],
 [[(), 0], [(2, 4), 1]]]
```

[144]:
```
%%timeit
astar_search(ejDoble_H7, ejDoble_H7.DDCLinear)
```

```
321 ms ± 35.4 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Ejemplo del enunciado. Para carga máxima 2, la búsqueda se hace muy costosa. Lo probamos para carga máxima 3.

[152]:
```
inicial_DE = ( (2,4,1,8,1), (3,9))
final_DE = (3,11,12,1,9)
ascensores_DE = ((3,3), 0, 12)
ejemplo_DE = AscensoresDoblesHeuristicas( inicial_DE, final_DE, ascensores_DE )
```

[154]:
```
%%time
solDE = astar_search(ejemplo_DE, ejemplo_DE.DDCLinear).solution()
```

```
Wall time: 32.6 s
```

[155]:
```
print( len(solDE) )
solDE
```

```
13
```

[155]:
```
[[[(), -1], [(), -1]],
 [[(), -1], [(3,), -1]],
 [[(2, 4), 1], [(3,), -1]],
 [[(0, 2, 4), 1], [(3,), -1]],
 [[(2, 4), 1], [(3,), -1]],
 [[(3,), -1], [(1, 2, 4), 1]],
 [[(3,), -1], [(1, 2, 4), 1]],
 [[(3,), -1], [(1, 2, 4), 1]],
 [[(), -1], [(1, 2, 4), 1]],
 [[(), 0], [(1, 2, 4), 1]],
 [[(), 0], [(1, 2), 1]],
 [[(), 0], [(1, 2), 1]],
```

```
[[(), 0], [(2,), 1]]]
```

6 Apartado 6. Ascensores dobles y bloques.

En este apartado, actualizamos el método de resolución de los problemas con bloques, añadiendo la posibilidad de que un bloque tenga 2 ascensores. El método de resolución es análogo, cambiando sólo algunos detalles de la implementación para introducir los ascensores dobles.

[156]:
```
def parseDoubleBlockAction(blockAction, blockIndex):
    """ Aplica un formato correcto a la acción de un subproblema de
        ascensores dobles para introducirla en la acción del
        problema general. """
    mainIndex1 = list()
    for i in blockAction[0][0]:
        idx = blockIndex[i]
        mainIndex1.append(idx)

    mainIndex2 = list()
    for i in blockAction[1][0]:
        idx = blockIndex[i]
        mainIndex2.append(idx)

    return ( ( tuple(mainIndex1), blockAction[0][1] ), ( tuple(mainIndex2),␣
 ↪blockAction[1][1]) )
```

[158]:
```
def doDoubleBlockAction(currentState, blockNum, blockAction, blockIndex):
    """ Interpreta correctamente una acción de un subproblema
        de ascensores dobles para aplicarla al estado del
        problema general. """
    op1 = blockAction[0][1]
    for i in blockAction[0][0]:
        idx = blockIndex[i]
        currentState[0][idx] += op1

    currentState[1][blockNum][0] += op1

    op2 = blockAction[1][1]
    for i in blockAction[1][0]:
        idx = blockIndex[i]
        currentState[0][idx] += op2

    currentState[1][blockNum][1] += op2
```

[208]:
```
def AscensoresDoblesBloques(initial, goal, blockSettings, doubles, arrivals =␣
 ↪list()):
    # Número de pasajeros del problema general.
```

```
    passNum = len( initial[0] )

    # Número de bloques del problema general.
    blockNum = len( initial[1] )

    # Lista mutable que lleva el estado actual del problema, y contiene:
    # > Lista con el estado de los pasajeros.
    # > Lista con el estado de los ascensores.
    currentState = [
        list(initial[0]),
        [ list(initial[1][i]) if doubles[i] else initial[1][i] for i in␣
↪range(blockNum) ]
    ]

    # Tupla con la carga máxima del ascensor de cada bloque.
    blockLoads = blockSettings[0]

    # Tupla con las plantas de intercambio entre bloques.
    limitFloors = blockSettings[1]

    # Lista que almacena las acciones de la solución general.
    problemSolution = list()

    # Método de resolución:
    while tuple( currentState[0] ) != goal:
        ## Lista de pasajeros que han de hacer intercambio, y contiene:
        ## > Tuplas de la forma (idx, dest), donde:
        ##   > idx es el índice del pasajero en el problema general.
        ##   > dest es su destino en esta iteración.
        interPassengers = list()

        ## 1. Simplificación a problemas individuales:

        # Lista de tuplas con la información de los subproblemas, cada cual␣
↪contiene:
        # > Lista con los estados 'iniciales' de pasajeros para ese bloque.
        # > Lista con los estados 'objetivo' de pasajeros para ese bloque.
        # > Lista que asocia los pasajeros (índices) del subproblema con el␣
↪problema general.
        blockProblems = list()

        # Para cada bloque, se consigue la información que nos permite construir␣
↪los subproblemas.
        for i in range(blockNum):
            blockSet = [bInitial, bGoal, bIndex] = (list(), list(), list())
            getBlockProblem(
                blockSet,
```

```
                ( limitFloors[i] , limitFloors[i + 1] ),
                ( currentState[0], goal, interPassengers )
            )

            blockProblems.append( tuple(blockSet) )

        ## 2. Para cada subproblema se calcula su solución.

        # Lista de soluciones de cada subproblema.
        blockSolutions = list()

        # Cálculo de soluciones:
        for i in range(blockNum):
            blockGoal = tuple( blockProblems[i][1] )
            blockElev = ( blockLoads[i], limitFloors[i], limitFloors[i + 1] )
            sol = None

            if doubles[i]:
                blockInitial = ( tuple( blockProblems[i][0] ), tuple(␣
↪currentState[1][i] ) )
                problem = AscensoresDoblesHeuristicas(blockInitial, blockGoal,␣
↪blockElev)
                sol = astar_search(problem, problem.DDCLinear).solution()
            else:
                blockInitial = ( tuple( blockProblems[i][0]), currentState[1][i])
                problem = AscensorSimpleHeuristicas(blockInitial, blockGoal,␣
↪blockElev)
                sol = astar_search(problem, problem.directedCheckedLinear).
↪solution()

            blockSolutions.append( tuple(sol) )

        # 3. Se ejecutan acciones hasta que:
        #    > Algún pasajero que debía hacer transbordo llega a su␣
↪intercambiador.
        #    > Ya no quedan acciones en ningún subproblema.

        # Booleano que indica el primer caso de salida.
        inter = False

        # Índice de acción.
        actIdx = 0
        while not inter:
            # Acción del problema general.
            wAction = list()

            # Booleano que indica el segundo caso de salida.
```

```
            isAction = False

            # Se ejecuta, para cada subproblema, la acción correspondiente a␣
↪'actIdx'.
            for i in range(blockNum):
                sol = blockSolutions[i]
                if doubles[i]:
                    if len(sol) > actIdx:
                        isAction = True
                        blockAction = sol[actIdx]
                        blockIndex = blockProblems[i][2]

                        wAction.append( parseDoubleBlockAction(blockAction,␣
↪blockIndex) )
                        doDoubleBlockAction( currentState, i, blockAction,␣
↪blockIndex )
                    else:
                        wAction.append( ( (tuple(), 0), (tuple(), 0) ) )
                else:
                    if len(sol) > actIdx: # Quedan acciones en el subproblema.
                        isAction = True
                        blockAction = sol[actIdx]
                        blockIndex = blockProblems[i][2]

                        wAction.append( parseBlockAction(blockAction,␣
↪blockIndex) )
                        doBlockAction( currentState, i, blockAction, blockIndex )
                    else:
                        wAction.append( (tuple(), 0) )

            # Si no quedan acciones en ningún subproblema, salimos.
            if not isAction:
                break

            # Se añade la acción 'actIdx' a la solución general.
            problemSolution.append( wAction.copy() )

            # Se comprueba si algún pasajero de 'transbordo' ha llegado al␣
↪intercambiador.
            for (idx, iGoal) in interPassengers:
                if currentState[0][idx] == iGoal:
                    inter = True
                    break # Deja de comprobar.

            ## Actualización de contador.
            actIdx += 1
```

```
        ## 0. Comprobación de llegada a destino.
        for i in range(passNum):
            if currentState[0][i] == goal[i] and i not in arrivals:
                arrivals.append( i )

    return problemSolution
```

[174]:
```
def printDoubleBlockSolution(initialState, solution, blockNum, doubles):
    state = [
        list(initialState[0]),
        [ list(initialState[1][i]) if doubles[i] else initialState[1][i] for i␣
 ↪in range(blockNum) ]
    ]

    print( len(solution), ' movimientos:')
    print('-----------')
    print(state)
    print('___________')

    for action in solution:
        for i in range(blockNum):
            subAction = action[i]
            if doubles[i]:
                for j in subAction[0][0]:
                    state[0][j] += subAction[0][1]
                for j in subAction[1][0]:
                    state[0][j] += subAction[1][1]

                state[1][i][0] += subAction[0][1]
                state[1][i][1] += subAction[1][1]
            else:
                for j in subAction[0]:
                    state[0][j] += subAction[1]

                state[1][i] += subAction[1]

            print(action[i])
        print('-----------')
        print(state)
        print('___________')
```

Ejemplo del enunciado.

[175]:
```
initialDBE1 = ( (2,4,1,8,1), (2,(4,8),10) )
goalDBE1 = (3,11,12,1,9)
settingsDBE1 = ( (2,(2,2),2), (0,4,8,12) )
doublesDBE1 = (False, True, False)
```

```
solDBE1 = AscensoresDoblesBloques(initialDBE1, goalDBE1, settingsDBE1,␣
 ↪doublesDBE1)
printDoubleBlockSolution(initialDBE1, solDBE1, 3, doublesDBE1)
```

```
17  movimientos:
-----------
[[2, 4, 1, 8, 1], [2, [4, 8], 10]]
-----------
((), -1)
(((1,), 1), ((3,), -1))
((), 0)
-----------
[[2, 5, 1, 7, 1], [1, [5, 7], 10]]
-----------
((2, 4), 1)
(((1,), 1), ((3,), -1))
((), 0)
-----------
[[2, 6, 2, 6, 2], [2, [6, 6], 10]]
-----------
((4,), 1)
(((3,), -1), ((1,), 1))
((), 0)
-----------
[[2, 7, 2, 5, 3], [3, [5, 7], 10]]
-----------
((), -1)
(((3,), -1), ((1,), 1))
((), 0)
-----------
[[2, 8, 2, 4, 3], [2, [4, 8], 10]]
-----------
((0, 2), 1)
((((), 0), ((), 0))
((), -1)
-----------
[[3, 8, 3, 4, 3], [3, [4, 8], 9]]
-----------
((2, 4), 1)
((((), 0), ((), 0))
((), -1)
-----------
[[3, 8, 4, 4, 4], [4, [4, 8], 8]]
-----------
((3,), -1)
(((2, 4), 1), ((), 0))
```

```
((1,), 1)
-----------
[[3, 9, 5, 3, 5], [3, [5, 8], 9]]
-----------
((3,), -1)
(((2, 4), 1), ((), -1))
((1,), 1)
-----------
[[3, 10, 6, 2, 6], [2, [6, 7], 10]]
-----------
((3,), -1)
(((2, 4), 1), ((), 0))
((1,), 1)
-----------
[[3, 11, 7, 1, 7], [1, [7, 7], 11]]
-----------
((), 0)
(((), -1), ((2, 4), 1))
((), 0)
-----------
[[3, 11, 8, 1, 8], [1, [6, 8], 11]]
-----------
((), 0)
(((), 0), ((), 0))
((), -1)
-----------
[[3, 11, 8, 1, 8], [1, [6, 8], 10]]
-----------
((), 0)
(((), 0), ((), 0))
((), -1)
-----------
[[3, 11, 8, 1, 8], [1, [6, 8], 9]]
-----------
((), 0)
(((), 0), ((), 0))
((), -1)
-----------
[[3, 11, 8, 1, 8], [1, [6, 8], 8]]
-----------
((), 0)
(((), 0), ((), 0))
((2, 4), 1)
-----------
[[3, 11, 9, 1, 9], [1, [6, 8], 9]]
-----------
((), 0)
(((), 0), ((), 0))
```

```
((2,), 1)
-----------
[[3, 11, 10, 1, 9], [1, [6, 8], 10]]
-----------
((), 0)
(((), 0), ((), 0))
((2,), 1)
-----------
[[3, 11, 11, 1, 9], [1, [6, 8], 11]]
-----------
((), 0)
(((), 0), ((), 0))
((2,), 1)
-----------
[[3, 11, 12, 1, 9], [1, [6, 8], 12]]
-----------
```

[176]:
```
%%timeit
AscensoresDoblesBloques(initialDBE1, goalDBE1, settingsDBE1, doublesDBE1)
```

```
28.1 ms ± 1.62 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Le añadimos 2 ascensores al primer bloque también.

[180]:
```
initialDBE2 = ( (2,4,1,8,1), ((0,4),(4,8),10) )
goalDBE2 = (3,11,12,1,9)
settingsDBE2 = ( ((2,2),(2,2),2), (0,4,8,12) )
doublesDBE2 = (True, True, False)

solDBE2 = AscensoresDoblesBloques(initialDBE2, goalDBE2, settingsDBE2,␣
 ↪doublesDBE2)
printDoubleBlockSolution(initialDBE2, solDBE2, 3, doublesDBE2)
```

```
14  movimientos:
-----------
[[2, 4, 1, 8, 1], [[0, 4], [4, 8], 10]]
-----------
(((), 1), ((), 0))
(((1,), 1), ((3,), -1))
((), 0)
-----------
[[2, 5, 1, 7, 1], [[1, 4], [5, 7], 10]]
-----------
(((2, 4), 1), ((), -1))
(((1,), 1), ((3,), -1))
((), 0)
-----------
[[2, 6, 2, 6, 2], [[2, 3], [6, 6], 10]]
```

```
-----------
(((2, 4), 1), ((), -1))
(((3,), -1), ((1,), 1))
((), 0)
-----------
[[2, 7, 3, 5, 3], [[3, 2], [5, 7], 10]]
-----------
(((2, 4), 1), ((0,), 1))
(((3,), -1), ((1,), 1))
((), 0)
-----------
[[3, 8, 4, 4, 4], [[4, 3], [4, 8], 10]]
-----------
(((3,), -1), ((), -1))
(((2, 4), 1), ((), 0))
((), -1)
-----------
[[3, 8, 5, 3, 5], [[3, 2], [5, 8], 9]]
-----------
(((3,), -1), ((), -1))
(((2, 4), 1), ((), -1))
((), -1)
-----------
[[3, 8, 6, 2, 6], [[2, 1], [6, 7], 8]]
-----------
(((3,), -1), ((), -1))
(((2, 4), 1), ((), 0))
((1,), 1)
-----------
[[3, 9, 7, 1, 7], [[1, 0], [7, 7], 9]]
-----------
(((), 0), ((), 0))
(((), -1), ((2, 4), 1))
((1,), 1)
-----------
[[3, 10, 8, 1, 8], [[1, 0], [6, 8], 10]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), -1)
-----------
[[3, 10, 8, 1, 8], [[1, 0], [6, 8], 9]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), -1)
-----------
[[3, 10, 8, 1, 8], [[1, 0], [6, 8], 8]]
```

```
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((2, 4), 1)
-----------
[[3, 10, 9, 1, 9], [[1, 0], [6, 8], 9]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((2,), 1)
-----------
[[3, 10, 10, 1, 9], [[1, 0], [6, 8], 10]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((1, 2), 1)
-----------
[[3, 11, 11, 1, 9], [[1, 0], [6, 8], 11]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((2,), 1)
-----------
[[3, 11, 12, 1, 9], [[1, 0], [6, 8], 12]]
-----------
```

[181]:
```
%%timeit
AscensoresDoblesBloques(initialDBE2, goalDBE2, settingsDBE2, doublesDBE2)
```

```
114 ms ± 4.46 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

7 Apartado 7. Problema del enunciado.

En este apartado, aportamos un algoritmo de resolución del problema del enunciado, que consiste en:

- Un edificio de 3 bloques, tal que:
  - El primer bloque tiene las plantas 0 a 4 y un ascensor de carga máxima 2.
  - El segundo bloque tiene las plantas 4 a 8 y 2 ascensores, cada uno con carga máxima 3.
  - El tercer bloque tiene las plantas 8 a 12 y un ascensor de carga máxima 2.
- Un ascensor rápido que se desplaza por todo el edificio entre las plantas pares, y es el doble de rápido que los ascensores de los bloques.
- Un número N de pasajeros, cada cual quiere desplazarse con los ascensores de una planta a otra.

7.0.1 Algoritmo de resolución.

Para resolver este problema, empleamos el algoritmo del apartado anterior, que es capaz de resolver el problema sin tener en cuenta al ascensor rápido.

En efecto, el algoritmo comienza resolviendo el problema sin tener en cuenta el ascensor rápido, obteniendo una solución con un determinado coste. Hemos actualizado el problema anterior para incluir una pila (como lista de Python) de *llegadas* para comprobar cuáles son los pasajeros que tardan más en llegar a su destino.

A continuación, con el objetivo de buscar una solución mejor, extraemos a un pasajero de la pila de *llegadas* y lo introducimos en la lista *evenPassengers*, que contiene los índices de los pasajeros que viajarán en el ascensor rápido; y se procede así:

1. Se formula un problema por bloques, en el que todos los pasajeros mantienen el estado inicial del problema general y, por un lado los pasajeros que no viajan en el ascensor rápido mantienen su destino, mientras que aquellos que sí van a viajar con el ascensor rápido tienen como destino llegar a una planta par (si es que no están ya en una). Se ejecutan las acciones de este problema hasta que los pasajeros de *evenPassengers* han llegado a sus plantas pares correspondientes.
2. A continuación, se formula el problema del ascensor rápido con los pasajeros de *evenPassengers*, que ya están en plantas pares. Para este problema, utilizamos la clase del ascesor simple, de forma que, puesto que el *paso* del ascensor es 2, dividimos todos los estados iniciales del problema entre 2: las plantas de los pasajeros, las plantas de destino (teniendo en cuenta si son impares) y la planta del ascensor. Guardamos la solución que nos da el problema.
3. Seguimos resolviendo un problema por bloques con el resto de pasajeros que no están en *evenPassengers*, y guardamos también la solución. Aquí, también actualizamos la pila de *llegadas*, cambiándola enteramente por lo que nos dé este problema.
4. Con estas 2 soluciones, vamos ejecutando acciones y actualizando el estado actual del problema. Se hace esto hasta que no quedan acciones del ascensor rápido (si se agotan antes la del problema de bloques, estas se llenan con acciones estáticas).
5. Con las posiciones del estado actual, formulamos otra vez un problema por bloques, en el que el estado inicial es el estado actual del problema y el objetivo es el del problema general. De forma que las acciones de este problema acabarán llevando a los pasajeros a sus destinos definitivamente.
6. Ahora que hemos calculado esta solución, comprobamos si mejora a la anterior, si lo hace la tomamos como solución general; si no, extraemos otro pasajero de la pila de *llegadas* y volvemos a 1 (y así mientras que la pila de *llegadas* no se vacía).

7.0.2 Código.

[252]:
```
def getPartialSolution(initial, goal, blockSettings, doubles, fastSettings,␣
↪arrivals, evenPassengers):
    problemPartialSolution = list()

    passNum = len( initial[0] )
```

```
    blockNum = len( initial[1] )

    limitFloors = blockSettings[1]

    evenLoad = fastSettings[0][0]
    evenStep = fastSettings[1][0]

    currentState = [
        list(initial[0]),
        [ list(initial[1][i]) if doubles[i] else initial[1][i] for i in␣
↪range(blockNum) ],
        list(initial[2])
    ]

    # 1.1. Initial block solution, in which 'evenPassengers' move to even floors.
    preBlockState = tuple( initial[0] )
    preBlockGoal = list()
    for i in range(passNum):
        if i in evenPassengers:
            if goal[i] > initial[0][i] and initial[0][i] % 2 != 0:
                preBlockGoal.append( initial[0][i] + 1 )
            elif goal[i] < initial[0][i] and initial[0][i] % 2 != 0:
                preBlockGoal.append( initial[0][i] - 1 )
            else:
                preBlockGoal.append( initial[0][i] )
        else:
            preBlockGoal.append( goal[i] )

    preSolution = AscensoresDoblesBloques(
        ( preBlockState, tuple(currentState[1]) ), tuple(preBlockGoal),
        blockSettings, doubles
    )

    # 1.2. Initial block solution application, in which currentState is updated
    # until evenPassengers arrived their even floor.
    preEvenArrived = list()
    for preAction in preSolution:
        for i in range(blockNum):
            blockAction = preAction[i]
            if doubles[i]:
                for j in blockAction[0][0]:
                    currentState[0][j] += blockAction[0][1]
                for j in blockAction[1][0]:
                    currentState[0][j] += blockAction[1][1]

                currentState[1][i][0] += blockAction[0][1]
                currentState[1][i][1] += blockAction[1][1]
```

```
            else:
                for j in blockAction[0]:
                    currentState[0][j] += blockAction[1]

                currentState[1][i] += blockAction[1]

        # Indexes are fine here.
        problemPartialSolution.append( ( preAction, ( tuple(), 0) ) )

        for i in range(passNum):
            if currentState[0][i] == preBlockGoal[i] and i not in preEvenArrived:
                preEvenArrived.append(i)

        if len(preEvenArrived) == len(evenPassengers):
            break

    # 2.1. Solution of fast elevator.
    fastState = list()
    fastGoal = list()
    for i in evenPassengers:
        fastState.append( currentState[0][i] // 2 )
        if goal[i] < currentState[0][i]:
            fastGoal.append( ceil( goal[i] / 2 ) )
        else:
            fastGoal.append( floor( goal[i] / 2 ) )

    minF = ceil( limitFloors[0] / 2 )
    maxF = floor( limitFloors[blockNum] / 2 )
    fastProblem = AscensorSimpleHeuristicas(
        ( tuple(fastState), currentState[2][0] // 2 ), tuple( fastGoal ),
        (evenLoad, minF, maxF)
    )
    fastSolution = astar_search(fastProblem, fastProblem.directedCheckedLinear).
↪solution()

    # 2.2. Solution of blocks.
    blockState = list()
    blockGoal = list()
    blockIndex = list()
    for i in range(passNum):
        if i not in evenPassengers:
            blockState.append( currentState[0][i] )
            blockGoal.append( goal[i] )
            blockIndex.append(i)

    arrivals = list()
    partialBlockSolution = AscensoresDoblesBloques(
```

```
        ( tuple(blockState), currentState[1] ), tuple(blockGoal),
        blockSettings, doubles, arrivals
    )

    for i in range( len(arrivals) ):
        arrivals[i] = blockIndex[ arrivals[i] ]

    # 2.3. Solution application.
    actIdx = 0
    while len(fastSolution) > actIdx:
        ## Fast Action
        fAction = fastSolution[actIdx]

        ## Index correction.
        fAction[0] = list(fAction[0])
        for i in range( len(fAction[0]) ):
            fAction[0][i] = evenPassengers[ fAction[0][i] ]

        fAction[1] *= evenStep

        for i in fAction[0]:
            currentState[0][i] += fAction[1]

        currentState[2][0] += fAction[1]

        ## Block Action
        wAction = list()
        if len( partialBlockSolution ) > actIdx:
            wAction = partialBlockSolution[actIdx]
            for i in range(blockNum):
                blockAction = wAction[i]
                if doubles[i]:
                    doDoubleBlockAction( currentState, i, blockAction,␣
↪blockIndex )
                else:
                    doBlockAction( currentState, i, blockAction, blockIndex )

            ## Index correction.
            for i in range(blockNum):
                wAction[i] = list( wAction[i] )
                if doubles[i]:
                    wAction[i][0] = list(wAction[i][0])
                    wAction[i][0][0] = list(wAction[i][0][0])
                    wAction[i][1] = list(wAction[i][1])
                    wAction[i][1][0] = list(wAction[i][1][0])
                    for j in range( len(wAction[i][0][0]) ):
                        wAction[i][0][0][j] = blockIndex[ wAction[i][0][0][j] ]
```

```
                    for j in range( len(wAction[i][1][0]) ):
                        wAction[i][1][0][j] = blockIndex[ wAction[i][1][0][j] ]
                else:
                    wAction[i][0] = list(wAction[i][0])
                    for j in range( len(wAction[i][0]) ):
                        wAction[i][0][j] = blockIndex[ wAction[i][0][j] ]
        else:
            for i in range(blockNum):
                if doubles[i]:
                    wAction.append( ( (tuple(), 0), (tuple(), 0) ) )
                else:
                    wAction.append( ( tuple(), 0 ) )

        problemPartialSolution.append( ( wAction.copy(), fAction.copy() ) )
        actIdx += 1

    # 3.1. Last problem, in which all passengers are moved to their destinies.
    postSolution = AscensoresDoblesBloques(
        ( tuple( currentState[0] ), tuple( currentState[1] ) ), goal,
        blockSettings, doubles
    )

    for postAction in postSolution:
        problemPartialSolution.append( ( postAction.copy(), ( tuple(), 0 ) ) )

    return problemPartialSolution
```

[253]:
```
def AscensoresEnunciado(initial, goal, blockSettings, doubles, fastSettings,␣
↪limit = 5):
    # Lista que almacena las acciones de la solución general.
    problemSolution = list()

    # Ascensores rápidos.
    (evenFast, unevenFast) = initial[2]

    if evenFast < 0 and unevenFast < 0:
        return AscensoresDoblesBloques(initial, goal, blockSettings, doubles)
    elif evenFast >= 0 and unevenFast < 0:
        # Solución sin ascensor rápido.
        arrivals = list()
        blockSolution = AscensoresDoblesBloques(initial, goal, blockSettings,␣
↪doubles, arrivals)

        ## Solución general.
        problemSolution = list()
        for wAction in blockSolution:
            problemSolution.append( (wAction.copy(), [ tuple(), 0] ) )
```

```
        bestCost = len( problemSolution )

        # Bucle de mejora.
        evenPassengers = list()
        while len( arrivals ) > 0 and len( evenPassengers ) < limit:
            evenPassengers.append( arrivals.pop() )

            problemPartialSolution = getPartialSolution(
                initial, goal,
                blockSettings, doubles, fastSettings,
                arrivals, evenPassengers
            )

            if len(problemPartialSolution) < bestCost:
                bestCost = len(problemPartialSolution)
                problemSolution = problemPartialSolution.copy()

        return problemSolution
```

Función de impresión de la solución.

[256]:
```
def printEnunciadoSolution(initialState, solution, blockNum, doubles):
    state = [
        list(initialState[0]),
        [ list(initialState[1][i]) if doubles[i] else initialState[1][i] for i␣
 ↪in range(blockNum) ],
        list(initialState[2])
    ]

    print( len(solution), ' movimientos:')
    print('-----------')
    print(state)
    print('-----------')

    for action in solution:
        for i in range(blockNum):
            subAction = action[0][i]
            if doubles[i]:
                for j in subAction[0][0]:
                    state[0][j] += subAction[0][1]
                for j in subAction[1][0]:
                    state[0][j] += subAction[1][1]

                state[1][i][0] += subAction[0][1]
                state[1][i][1] += subAction[1][1]
            else:
```

```
                for j in subAction[0]:
                    state[0][j] += subAction[1]

                state[1][i] += subAction[1]

            print(subAction)

        for j in action[1][0]:
            state[0][j] += action[1][1]

        state[2][0] += action[1][1]

        print('Fast -> ', action[1])


        print('-----------')
        print(state)
        print('__________')
```

7.0.3 Ejemplo del enunciado.

[260]:
```
initialDef = ( (2,4,1,8,1), (2,(4,8),10), (2, -1) )
goalDef = (3,11,12,1,9)
settingsDef = ( (2,(2,2),2), (0,4,8,12) )
doublesDef = (False, True, False)
fastDef = ( (3,0), (2,0) )

solDef = AscensoresEnunciado(initialDef, goalDef, settingsDef, doublesDef,␣
 ↪fastDef)
print( len(solDef) )
printEnunciadoSolution(initialDef, solDef, 3, doublesDef)
```

```
11
11  movimientos:
-----------
[[2, 4, 1, 8, 1], [2, [4, 8], 10], [2, -1]]
-----------
((), -1)
(((1,), 1), ((3,), -1))
((), 0)
Fast ->  ((), 0)
-----------
[[2, 5, 1, 7, 1], [1, [5, 7], 10], [2, -1]]
-----------
((2, 4), 1)
(((1,), 1), ((3,), -1))
((), 0)
```

```
Fast ->  ((), 0)
-----------
[[2, 6, 2, 6, 2], [2, [6, 6], 10], [2, -1]]
-----------
[[0], 1]
[[[3], -1], [[1], 1]]
[[], 0]
Fast ->  [[4, 2], 2]
-----------
[[3, 7, 4, 5, 4], [3, [5, 7], 10], [4, -1]]
-----------
[[], 0]
[[[3], -1], [[1], 1]]
[[], 0]
Fast ->  [[4, 2], 2]
-----------
[[3, 8, 6, 4, 6], [3, [4, 8], 10], [6, -1]]
-----------
[[], 1]
[[[], 0], [[], 0]]
[[], -1]
Fast ->  [[4, 2], 2]
-----------
[[3, 8, 8, 4, 8], [4, [4, 8], 9], [8, -1]]
-----------
[[3], -1]
[[[], 0], [[], 0]]
[[], -1]
Fast ->  [[2], 2]
-----------
[[3, 8, 10, 3, 8], [3, [4, 8], 8], [10, -1]]
-----------
[[3], -1]
[[[], 0], [[], 0]]
[[1], 1]
Fast ->  [[2], 2]
-----------
[[3, 9, 12, 2, 8], [2, [4, 8], 9], [12, -1]]
-----------
((3,), -1)
(((), 0), ((), 0))
((), -1)
Fast ->  ((), 0)
-----------
[[3, 9, 12, 1, 8], [1, [4, 8], 8], [12, -1]]
-----------
((), 0)
(((), 0), ((), 0))
```

```
((4,), 1)
Fast ->  ((), 0)
-----------
[[3, 9, 12, 1, 9], [1, [4, 8], 9], [12, -1]]
-----------
((), 0)
(((), 0), ((), 0))
((1,), 1)
Fast ->  ((), 0)
-----------
[[3, 10, 12, 1, 9], [1, [4, 8], 10], [12, -1]]
-----------
((), 0)
(((), 0), ((), 0))
((1,), 1)
Fast ->  ((), 0)
-----------
[[3, 11, 12, 1, 9], [1, [4, 8], 11], [12, -1]]
-----------
```

Otras configuraciones. Gracias a la implementación generalizada, podemos probar el problema con otras configuraciones.

[410]:
```
initialDef2 = ( (10,11,5,7,6), (2,(4,8),10), (2, -1) )
goalDef2 = (0,3,0,1,10)
settingsDef2 = ( (2,(2,2),2), (0,4,8,12) )
doublesDef2 = (False, True, False)
fastDef2 = ( (3,0), (2,0) )

solDef2 = AscensoresEnunciado(initialDef2, goalDef2, settingsDef2, doublesDef2,␣
 ↪fastDef2)
print( len(solDef2) )
printEnunciadoSolution(initialDef2, solDef2, 3, doublesDef2)
```

```
16
16  movimientos:
-----------
[[10, 11, 5, 7, 6], [2, [4, 8], 10], [2, -1]]
-----------
((), 0)
(((), 1), ((), 0))
((), 1)
Fast ->  ((), 0)
-----------
[[10, 11, 5, 7, 6], [2, [5, 8], 11], [2, -1]]
-----------
((), 0)
(((2,), -1), ((), -1))
```

```
((1,), -1)
Fast ->  ((), 0)
-----------
[[10, 10, 4, 7, 6], [2, [4, 7], 10], [2, -1]]
-----------
((), 1)
(((), 0), ((3,), -1))
((0, 1), -1)
Fast ->  ((), 0)
-----------
[[9, 9, 4, 6, 6], [3, [4, 6], 9], [2, -1]]
-----------
[[], 1]
[[[], 0], [[4], 1]]
[[0, 1], -1]
Fast ->  [[], 2]
-----------
[[8, 8, 4, 6, 7], [4, [4, 7], 8], [4, -1]]
-----------
[[2], -1]
[[[], 0], [[4], 1]]
[[], 0]
Fast ->  [[], 2]
-----------
[[8, 8, 3, 6, 8], [3, [4, 8], 8], [6, -1]]
-----------
[[2], -1]
[[[], 0], [[0, 1], -1]]
[[4], 1]
Fast ->  [[3], -2]
-----------
[[7, 7, 2, 4, 9], [2, [4, 7], 9], [4, -1]]
-----------
[[2], -1]
[[[], 0], [[0, 1], -1]]
[[4], 1]
Fast ->  [[3], -2]
-----------
[[6, 6, 1, 2, 10], [1, [4, 6], 10], [2, -1]]
-----------
((), 1)
(((), 0), ((0, 1), -1))
((), 0)
Fast ->  ((), 0)
-----------
[[5, 5, 1, 2, 10], [2, [4, 5], 10], [2, -1]]
-----------
((3,), -1)
```

```
((), 0), ((0, 1), -1))
((), 0)
Fast ->  ((), 0)
-----------
[[4, 4, 1, 1, 10], [1, [4, 4], 10], [2, -1]]
-----------
((), 1)
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[4, 4, 1, 1, 10], [2, [4, 4], 10], [2, -1]]
-----------
((), 1)
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[4, 4, 1, 1, 10], [3, [4, 4], 10], [2, -1]]
-----------
((), 1)
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[4, 4, 1, 1, 10], [4, [4, 4], 10], [2, -1]]
-----------
((0, 1), -1)
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[3, 3, 1, 1, 10], [3, [4, 4], 10], [2, -1]]
-----------
((0,), -1)
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[2, 3, 1, 1, 10], [2, [4, 4], 10], [2, -1]]
-----------
((0,), -1)
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[1, 3, 1, 1, 10], [1, [4, 4], 10], [2, -1]]
-----------
```

```
((0, 2), -1)
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[0, 3, 0, 1, 10], [0, [4, 4], 10], [2, -1]]
-----------
```

E incluso añadir, por ejemplo, dos ascensores en el bloque inferior.

[411]:
```
initialDef3 = ( (10,11,5,7,6), ((3,3),(4,8),10), (2, -1) )
goalDef3 = (0,3,0,1,10)
settingsDef3 = ( ((2,2),(2,2),2), (0,4,8,12) )
doublesDef3 = (True, True, False)
fastDef3 = ( (3,0), (2,0) )

solDef3 = AscensoresEnunciado(initialDef3, goalDef3, settingsDef3, doublesDef3,␣
 ↪fastDef3)
print( len(solDef3) )
printEnunciadoSolution(initialDef3, solDef3, 3, doublesDef3)
```

```
14
14  movimientos:
-----------
[[10, 11, 5, 7, 6], [[3, 3], [4, 8], 10], [2, -1]]
-----------
(((), 0), ((), 0))
(((), 1), ((), 0))
((), 1)
Fast ->  ((), 0)
-----------
[[10, 11, 5, 7, 6], [[3, 3], [5, 8], 11], [2, -1]]
-----------
(((), 0), ((), 0))
(((2,), -1), ((), -1))
((1,), -1)
Fast ->  ((), 0)
-----------
[[10, 10, 4, 7, 6], [[3, 3], [4, 7], 10], [2, -1]]
-----------
(((), 0), ((), 0))
(((), 0), ((3,), -1))
((), 0)
Fast ->  ((), 0)
-----------
[[10, 10, 4, 6, 6], [[3, 3], [4, 6], 10], [2, -1]]
-----------
(((), 0), ((), 0))
```

```
(((), 0), ((), 0))
((), 0)
Fast ->  [[], 2]
-----------
[[10, 10, 4, 6, 6], [[3, 3], [4, 6], 10], [4, -1]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), 0)
Fast ->  [[], 2]
-----------
[[10, 10, 4, 6, 6], [[3, 3], [4, 6], 10], [6, -1]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), 0)
Fast ->  [[4], 2]
-----------
[[10, 10, 4, 6, 8], [[3, 3], [4, 6], 10], [8, -1]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), 0)
Fast ->  [[4], 2]
-----------
[[10, 10, 4, 6, 10], [[3, 3], [4, 6], 10], [10, -1]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), 0)
Fast ->  [[1, 0], -2]
-----------
[[8, 8, 4, 6, 10], [[3, 3], [4, 6], 10], [8, -1]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), 0)
Fast ->  [[1, 0], -2]
-----------
[[6, 6, 4, 6, 10], [[3, 3], [4, 6], 10], [6, -1]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), 0)
Fast ->  [[1, 0, 3], -2]
-----------
[[4, 4, 4, 4, 10], [[3, 3], [4, 6], 10], [4, -1]]
-----------
```

```
(((), 0), ((), 0))
(((), 0), ((), 0))
((), 0)
Fast ->  [[2, 0, 3], -2]
-----------
[[2, 4, 2, 2, 10], [[3, 3], [4, 6], 10], [2, -1]]
-----------
(((), 0), ((), 0))
(((), 0), ((), 0))
((), 0)
Fast ->  [[2, 0], -2]
-----------
[[0, 4, 0, 2, 10], [[3, 3], [4, 6], 10], [0, -1]]
-----------
(((), 1), ((), -1))
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[0, 4, 0, 2, 10], [[4, 2], [4, 6], 10], [0, -1]]
-----------
(((1,), -1), ((3,), -1))
(((), 0), ((), 0))
((), 0)
Fast ->  ((), 0)
-----------
[[0, 3, 0, 1, 10], [[3, 1], [4, 6], 10], [0, -1]]
-----------
```

8 Apartado 8. Conclusiones.

8.1 Consecuencias de los estados inicial y final sobre el problema.

Finalmente, hemos conseguido implementar un algoritmo capaz de manejarse en los términos del enunciado, con un número de 5 pasajeros. No obstante, un incremento de pasajeros podría suponer un incremento en el coste de calcular la solución al problema. En efecto, en los ejemplos más sencillos del Ascensor Simple y los Ascensores Dobles, el incremento de plantas del edificio suponía un coste adicional en el cálculo. Además, al calcular el problema del enunciado para estos casos, se hacía necesario utilizar ascensores de mayor carga para conseguir una solución más rápida.

Por tanto, como conclusión, el un aumento en el número de pasajeros supondría un aumento en el coste de calcular la solución, pero este podría ser mitigado aumentando la carga de los ascensores utilizados.

8.2 Subida y bajada de pasajeros.

Como hemos visto en apartados anteriores, no podemos calcular el coste de subida y bajada de pasajeros, en tanto que la función de *path_cost* no conoce cuál ha sido la acción anterior. Además, esto genera desplazamientos *extraños* en los ascensores, que puede subir y bajar repetidamente para desplazar a más pasajeros de los que soporta en una misma dirección.

Una solución sería modifica el fichero *search.py* para arreglar esta deficiencia, pero prescindimos de implentarlo.

8.3 Ascensor rápido impar.

La implementación de un ascensor rápido que se desplazase por plantas impares sería análoga a la del ascensor que lo hace por pares. Surgirían algunas complicaciones al tener los 2 ascensores funcionando al mismo tiempo, que se resolverían fácilmente dividiendo mediante combinaciones a los pasajeros que van a viajar en los ascensores rápidos en cada iteración, y seguir el mismo proceder hasta hallar la solución más rápida.