In [1]:
```
#Jesús Martín Moraleda(Amarillo) y Jorge Arevalo Echeverria(Azul) G13
```

In [2]:
```
cd aima-python
```

```
C:\Users\jorge\Desktop\IA\Practicas\Practica1\Parte B\aima-python
```

In [3]:
```
# Cargamos el módulo con los algoritmos de búsqueda.
from search import *
from search import breadth_first_tree_search, depth_first_tree_search, depth_firs
t_graph_search, breadth_first_graph_search
```

In [4]:
```
class Problem(object):

    """The abstract class for a formal problem. You should subclass
    this and implement the methods actions and result, and possibly
    __init__, goal_test, and path_cost. Then you will create instances
    of your subclass and solve them with the various search functions."""


    def __init__(self, initial, goal=None):
        """The constructor specifies the initial state, and possibly a goal
        state, if there is a unique goal. Your subclass's constructor can add
        other arguments."""
        self.initial = initial
        self.goal = goal

    def actions(self, state):
        """Return the actions that can be executed in the given
        state. The result would typically be a list, but if there are
        many actions, consider yielding them one at a time in an
        iterator, rather than building them all at once."""
        raise NotImplementedError

    def result(self, state, action):
        """Return the state that results from executing the given
        action in the given state. The action must be one of
        self.actions(state)."""
        raise NotImplementedError

    def goal_test(self, state):
        """Return True if the state is a goal. The default method compares the
        state to self.goal or checks for state in self.goal if it is a
        list, as specified in the constructor. Override this method if
        checking against a single self.goal is not enough."""
        if isinstance(self.goal, list):
            return is_in(state, self.goal)
        else:
            return state == self.goal

    def path_cost(self, c, state1, action, state2):
        """Return the cost of a solution path that arrives at state2 from
        state1 via action, assuming cost c to get up to state1. If the problem
        is such that the path doesn't matter, this function will only look at
        state2.  If the path does matter, it will consider c and maybe state1
        and action. The default method costs 1 for every step in the path."""
        return c + 1

    def value(self, state):
        """For optimization problems, each state has a value.  Hill-climbing
        and related algorithms try to maximize this value."""
        raise NotImplementedError

    def coste_de_aplicar_accion(self, estado, accion):
        """Hemos incluido está función que devuelve el coste de un único operador
(aplicar accion a estado). Por defecto, este
        coste es 1. Reimplementar si el problema define otro coste """
        return 1
```

In [5]:
```
# Creamos la clase ProblemaLinterna con los elementos que representarán el proble
ma.
class ProblemaLinterna(Problem):
    ''' Clase problema (formalizacion de nuestro problema) siguiendo la
        estructura que aima espera que tengan los problemas.'''
    #el estado se representara como (linterna,tiempoAcumulado, ladoPersonas) sien
do lado, el lado en el que se encuentra la linterna
    # y tiempoAcumlado el tiempo que llevamos hasta el momento

    def __init__(self, initial, goal = 300):
        '''Inicializacion de nuestro problema.'''
        self.goal = goal
        Problem.__init__(self, initial, goal)
        #tiempo de cada persona
        self._tiempoPersona = [10,30,60,80,120]

    def actions(self, s):
        '''Devuelve las acciones validas para un estado.'''
        #Las acciones es coger a una persona y llevarla con otra hacia el lado fi
nal, o que vuelva una al estado original

        accs=list()
        for i in range(5):
            if s[2][i] == s[0]: #si la persona i está en el mismo lado que la lin
terna
                for j in range(i,5):
                    if s[2][j] == s[0]:#si la persona j esta en el mismo lado que
la linterna
                        if s[0] == 0: # si la linterna esta en el origen solo tie
ne sentido que crucen dos personas(distintas)
                            if i != j:
                                accs.append((i,j))
                        else: #si esta en el lado final solo tiene sentido que vu
elva una persona
                            accs.append((j,j))



        return accs
        #la idea es generar las acciones como una tupla, por ejemplo. si cruzan 1
y 2: (1,2)
        #puedes hacer todas las combinaciones con un bucle dependiendo del número
de personas

    def goal_test(self, state):
        """Return True if the state is a goal."""

        for i in range(5):
            if(state[2][i] == 0):
                return False

        return (state[1] <= self.goal)

    def result(self, s, a):
        '''Devuelve el estado resultante de aplicar una accion a un estado
           determinado.'''
        # el estado resultante tiene la linterna en el lado opuesto, y con las ca
ntidades tiempo actualizadas,
        # segun el tiempo maximo de las dos personas que cruzaran el puente
        #hago un casteo de la tupla de en que lado estan las personas para poder
 editarlo, no se hacerlo de otra forma
        #aunque tiene que haber alguna

        s_list = list(s[2])
```

```
        if s[0] == 0:
            s_list[a[0]] = 1
            s_list[a[1]] = 1
            return (1, s[1] + max(self._tiempoPersona[a[0]], self._tiempoPersona[
a[1]]), tuple(s_list))
        else:#si la linterna esta en el lado contrario solo vuelve una persona, p
or lo que a[0] = a[1]
            s_list[a[0]] = 0
            return (0, s[1] + self._tiempoPersona[a[0]], tuple(s_list))
```

In [6]:
```
linterna = ProblemaLinterna((0,0,(0,0,0,0,0)), 300)
```

## primero en anchura y en profundidad para ver heuristica y despues resolver el problema para A*

In [8]:
```
breadth_first_graph_search(linterna).solution()
```

Out[8]:
```
[(0, 1), (0, 0), (0, 2), (0, 0), (3, 4), (1, 1), (0, 1)]
```

In [12]:
```
#La solución es la tupla de personas que van, y la persona que vuelve. Cuando la
 tupla tiene dos numeros distintos representa
#las dos personas que van del origen al destino. Cuando la tupla tiene el mismo e
lemento, ej: (0,0), representa la persona
#que vuelve del destino al origen para traer la linterna
```

In [13]:
```
# Heuristicas para el problema de la linterna. Le asigno el numero de personas qu
e no están en el lado objetivo.
#También había pensado en incluir el tiempo total que lleva acumulado ese estado,
pero creo que eso no sería muy bueno para
#la heurística
import math

def linear(node):
    goal = (1,1,1,1,1)
    state = node.state
    #nos devuelve el numero de personas que no están en el destino
    return sum([1 for i in range(5) if state[2][i] != goal[i]])
```

In [14]:
```
astar_search(linterna,linear).solution()
```

Out[14]:
```
[(0, 1), (0, 0), (3, 4), (1, 1), (0, 1), (0, 0), (0, 2)]
```

In [ ]:
```

```