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

In [1]:
```
cd aima-python
```

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

In [2]:
```
# 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 [3]:
```
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 [4]:
```
# Creamos la clase ProblemaLinterna con los elementos que representarán el proble
ma.
class ProblemaPuzzle(Problem):
    ''' Clase problema (formalizacion de nuestro problema) siguiendo la
        estructura que aima espera que tengan los problemas.'''
    #cada estado se representara como (numeros) siendo numeros una tupla con el o
rden de los numeros en el puzzle

    def __init__(self, initial, goal=66):
        '''Inicializacion de nuestro problema.'''
        self.initial = initial
        self.goal = goal
        self._count = 0
        Problem.__init__(self, initial, goal)


    def actions(self, s):
        '''Devuelve las acciones validas para un estado.'''
        # las acciones validas para un estado son aquellas que al aplicarse nos d
ejan en otro estado valido
        # Hemos pensado como accion válida cambiar dos números de posición

        accs=list()
        for i in range(9):
            for j in range(9):
                if(i != j):
                    accs.append((i,j))
        return accs


    def goal_test(self, s):
        """Return True if the state is a goal."""
        return ((((((((((s[0] + 13) * s[1]) / s[2]) + s[3]) + 12) * s[4]) - s[5
]) - 11) + s[6]) * s[7]) / s[8]) - 10 ) == self.goal

    def result(self, s, a):
        '''Devuelve el estado resultante de aplicar una accion a un estado
           determinado.'''
        #hago un cast al estado para tener el estado como una lista y poder opera
r con cada valor,
        #luego lo vuelvo a dejar como tupla

        self._count+=1

        s_list = list(s)
        aux = s_list[a[0]]
        s_list[a[0]] = s_list[a[1]]
        s_list[a[1]] = aux
        s = tuple(s_list)
        return s
```

In [5]:
```
puzzle = ProblemaPuzzle((4,9,5,8,7,6,2,1,3), 66)
```

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

In [6]:
```
breadth_first_graph_search(puzzle).solution()
```

Out[6]:
```
[(0, 1), (0, 4), (2, 3)]
```

In [ ]:
```
#La salida representa las posiciones de los numeros que se intercambian en el tab
lero
```

In [22]:
```
#Heurística para el puzzle. A cada estado le asigno el valor absoluto de la difer
encia entre el valor del estado objetivo
#y el valor del estado actual. Cuanto más cerca esté del objetivo (66) menor será
el valor que le de la heurística a ese nodo
#por lo que tendrá mas posibilidades de seguir por ese camino

#Muchas veces el método A* tarda mas que el primero en anchura, supongo que será
 porque la heurística no es buena del todo
#o porque al ser un problema no muy complicado los métodos ciegos son mas eficace
s.

import math

def linear(node):
    goal = 66
    s = node.state
    suma_node = ((((((((((s[0] + 13) * s[1]) / s[2]) + s[3]) + 12) * s[4]) - s[
5]) - 11) + s[6]) * s[7]) / s[8]) - 10 )
    return abs(goal - suma_node)
```

In [23]:
```
astar_search(puzzle, linear).solution()
```

Out[23]:
```
[(0, 8), (2, 8), (3, 5), (1, 4), (1, 3), (3, 5)]
```

In [ ]: