Practica2

27 de octubre de 2019

1. Práctica 2.A Toma de contacto con AIMA

La práctica está organizada en 3 partes. En la primera se muestra a través de ejemplos cómo se implementan algunos problemas clásicos como el de las jarras o el problema del ocho puzzle. En la segunda parte se muestra el uso de los algoritmos de búsqueda. En la tercera parte aprenderemos a medir las propiedades de los algoritmos. En el notebook encontraras claramente identificados los lugares en los que debes incluir código o comentarios.

1.1. Parte I: Representación de problemas de espacios de estados.

1.1.1. El primer paso es importar el código que necesitamos de search.py de AIMA y usar la clase Problem. En esta parte en vez de importarla la hemos copiado aquí para la explicación.

Como hemos visto en clase la representación de un problema de espacio de estados consiste en: * Representar estados y acciones mediante una estructura de datos. * Definir: estado_inicial, es_estado_final(*), acciones()*, aplica(,) y coste_de_aplicar_accion, si el problema tiene coste.

La siguiente clase Problem representa este esquema general de cualquier problema de espacio de estados. Un problema concreto será una subclase de Problema, y requerirá implementar acciones, aplica y eventualmente **init**, actions, goal_test. La función coste_de_aplicar_accion la hemos incluido nosotros.

```
In [1]: from search import *
```

```
In [2]:    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 esta 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
```

Ahora vamos a ver un ejemplo de cómo definir un problema como subclase de problema. En concreto, el problema de las jarras, visto en clase que es muy sencillo.

```
In [3]: class Jarras(Problem):
            """Problema de las jarras:
            Representaremos los estados como tuplas (x,y) de dos números enteros,
            donde x es el número de litros de la jarra de 4 e y es el número de litros
            de la jarra de 3"""


            def __init__(self):
                self.initial = (0,0)

            def actions(self,estado):
                jarra_de_4=estado[0]
                jarra_de_3=estado[1]
                accs=list()
                if jarra_de_4 > 0:
                    accs.append("vaciar jarra de 4")
                    if jarra_de_3 < 3:
                        accs.append("trasvasar de jarra de 4 a jarra de 3")
                if jarra_de_4 < 4:
                    accs.append("llenar jarra de 4")
                    if jarra_de_3 > 0:
                        accs.append("trasvasar de jarra de 3 a jarra de 4")
                if jarra_de_3 > 0:
                    accs.append("vaciar jarra de 3")
                if jarra_de_3 < 3:
                    accs.append("llenar jarra de 3")
                return accs

            def result(self,estado,accion):
                j4=estado[0]
                j3=estado[1]
                if accion=="llenar jarra de 4":
                    return (4,j3)
                elif accion=="llenar jarra de 3":
                    return (j4,3)
                elif accion=="vaciar jarra de 4":
                    return (0,j3)
                elif accion=="vaciar jarra de 3":
                    return (j4,0)
                elif accion=="trasvasar de jarra de 4 a jarra de 3":
                    return (j4-3+j3,3) if j3+j4 >= 3 else (0,j3+j4)
                else: #  "trasvasar de jarra de 3 a jarra de 4"
                    return (j3+j4,0) if j3+j4 <= 4 else (4,j3-4+j4)

            def goal_test(self,estado):
                return estado[0]==2
```

Vamos a probar algunos ejemplos.

```
In [4]: p =Jarras()
        p.initial

Out[4]: (0, 0)

In [5]: p.actions(p.initial)

Out[5]: ['llenar jarra de 4', 'llenar jarra de 3']

In [6]: p.result(p.initial,"llenar jarra de 4")

Out[6]: (4, 0)

In [7]: p.coste_de_aplicar_accion(p.initial,"llenar jarra de 4")

Out[7]: 1

In [8]: p.goal_test(p.initial)

Out[8]: False
```

2. Representa el problema del puzzle de 8

Ahora vamos a definir la clase Ocho_Puzzle, que implementa la representación del problema del 8-puzzle visto en clase. Se os proporciona una versión incompleta y tendréis que completar el código que se presenta a continuación, en los lugares marcados con interrogantes.

2.0.1. 8 Puzzle

Tablero 3x3 cuyo objetivo es mover la configuración de las piezas desde un estado inicial dado a un estado objetivo moviendo las fichas al espacio en blanco.

ejemplo:-

```
    Inicial                 Goal
| 7 | 2 | 4 |         | 1 | 2 | 3 |
| 5 | 0 | 6 |         | 4 | 5 | 6 |
| 8 | 3 | 1 |         | 7 | 8 | 0 |
```

Hay 9! configuraciones iniciales pero ojo! porque no todas tienen solución. Tenlo en cuenta al hacer las pruebas.

2.0.2. EJERCICIO 1. COMPLETA LA DEFINICIÓN DE LOS OPERADORES.

```
In [9]: class Ocho_Puzzle(Problem):
            """Problema a del 8-puzzle.  Los estados serán tuplas de nueve elementos,
            permutaciones de los números del 0 al 8 (el 0 es el hueco). Representan la
            disposición de las fichas en el tablero, leídas por filas de arriba a
            abajo, y dentro de cada fila, de izquierda a derecha. Las cuatro
            acciones del problema las representaremos mediante las cadenas:
            "Mover hueco arriba", "Mover hueco abajo", "Mover hueco izquierda" y
            "Mover hueco derecha", respectivamente."""

            def __init__(self, initial, goal=(1, 2, 3, 4, 5, 6, 7, 8, 0)):
                """ Define goal state and initialize a problem """
                self.goal = goal
                Problem.__init__(self, initial, goal)

            def actions(self,estado):
                pos_hueco=estado.index(0) # busco la posicion del 0
                accs=list()
                if pos_hueco not in (0,1,2):
                    accs.append("Mover hueco arriba")

                ### EJERCICIO 1.1. COMPLETA LA DEFINICIÓN DE LOS OPERADORES.

                if pos_hueco not in (0, 3, 6):
                    accs.append("Mover hueco izquierda")

                if pos_hueco not in (6, 7, 8):
                    accs.append("Mover hueco abajo")

                if pos_hueco not in (2, 5, 8):
                    accs.append("Mover hueco derecha")
                return accs

            def result(self,estado,accion):
                pos_hueco = estado.index(0)
                l = list(estado)
                if accion == "Mover hueco arriba":
                    l[pos_hueco] = l[pos_hueco-3]
                    l[pos_hueco-3] = 0

               ### EJERCICIO 1.2. COMPLETA LA DEFINICIÓN DE LOS OPERADORES.

                if accion == "Mover hueco izquierda":
                    l[pos_hueco] = l[pos_hueco-1]
                    l[pos_hueco-1] = 0
                if accion == "Mover hueco abajo":
                    l[pos_hueco] = l[pos_hueco+3]
```

```
            l[pos_hueco+3] = 0
        if accion == "Mover hueco derecha":
            l[pos_hueco] = l[pos_hueco+1]
            l[pos_hueco+1] = 0

        return tuple(l)

    def h(self, node):
        #"Return the heuristic value for a given state".
        return 1
```

2.0.3. Ejercicio 1.3. Probar los siguientes ejemplos que se pueden ejecutar una vez se ha definido la clase:

```
In [10]: p8 = Ocho_Puzzle((2, 8, 3, 1, 6, 4, 7, 0, 5))
         p8.initial

Out[10]: (2, 8, 3, 1, 6, 4, 7, 0, 5)

In [11]: p8.actions(p8.initial)
         #Respuesta: ['Mover hueco arriba', 'Mover hueco izquierda', 'Mover hueco derecha']

Out[11]: ['Mover hueco arriba', 'Mover hueco izquierda', 'Mover hueco derecha']

In [12]: p8.result(p8.initial,"Mover hueco arriba")

Out[12]: (2, 8, 3, 1, 0, 4, 7, 6, 5)

In [13]: # Crash debido a que no puede moverse hacia abajo
         # p8.result(p8.initial,"Mover hueco abajo")

In [14]: p8.result(p8.initial,"Mover hueco derecha")

Out[14]: (2, 8, 3, 1, 6, 4, 7, 5, 0)

In [15]: p8.coste_de_aplicar_accion(p8.initial,"Mover hueco abajo")

Out[15]: 1
```

2.1. Parte II: Experimentación con los algoritmos implementados. Ejecución de los algoritmos de búsqueda de soluciones para una instancia del Problema.

2.1.1. Usaremos búsqueda en anchura y en profundidad para encontrar soluciones tanto al problema de las jarras como al problema del ocho puzzle con distintos estados iniciales.

```
In [16]: # Cargamos el módulo con los algoritmos de búsqueda.
         from search import *
```

```
from search import breadth_first_tree_search, depth_first_tree_search,
depth_first_graph_search, breadth_first_graph_search
```

```
In [17]: breadth_first_tree_search(Jarras()).solution()
```

```
Out[17]: ['llenar jarra de 4',
          'trasvasar de jarra de 4 a jarra de 3',
          'vaciar jarra de 3',
          'trasvasar de jarra de 4 a jarra de 3',
          'llenar jarra de 4',
          'trasvasar de jarra de 4 a jarra de 3']
```

```
In [18]: depth_first_graph_search(Jarras()).solution()
```

```
Out[18]: ['llenar jarra de 3',
          'trasvasar de jarra de 3 a jarra de 4',
          'llenar jarra de 3',
          'trasvasar de jarra de 3 a jarra de 4',
          'vaciar jarra de 4',
          'trasvasar de jarra de 3 a jarra de 4']
```

2.1.2. Ejercicio 2. Prueba los algoritmos de búsqueda ciega con el puzzle de 8

```
In [19]: p8 = Ocho_Puzzle((2, 8, 3, 1, 6, 4, 7, 0, 5))
         p8.initial
```

```
Out[19]: (2, 8, 3, 1, 6, 4, 7, 0, 5)
```

```
In [20]: p8.goal
```

```
Out[20]: (1, 2, 3, 4, 5, 6, 7, 8, 0)
```

```
In [21]: # ------------- NO FINALIZA --------------
         # Busqueda en anchura sin control de repetidos.
         # Busqueda en anchura es completo.. debería terminar...
         # ¿qué crees que está pasando?
         #
         # breadth_first_tree_search(Ocho_Puzzle((2, 8, 3, 1, 6, 4, 7, 0, 5))).solution()
```

Que la búsqueda en anchura sea completa significa que, de haber solución, la encuentra. Sin embargo, dado ese estado inicial, el problema no tiene solución (comprobado en la función check_solvability). Por lo tanto, no para de generar nodos y no acabará (hasta quedarse sin memoria).

```
In [22]: breadth_first_tree_search(Ocho_Puzzle((1,2,3,4,5,6,0,7,8))).solution()
```

```
Out[22]: ['Mover hueco derecha', 'Mover hueco derecha']

In [23]: estado = Ocho_Puzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))

In [24]: breadth_first_tree_search(estado).solution()
         # Respuesta: ['UP', 'LEFT', 'UP', 'LEFT', 'DOWN', 'RIGHT', 'RIGHT', 'DOWN']

Out[24]: ['Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco abajo',
          'Mover hueco derecha',
          'Mover hueco derecha',
          'Mover hueco abajo']

In [25]: # ------------- NO FINALIZA --------------
         # Ver conclusiones expuestas más abajo para más información
         #
         # depth_first_graph_search(estado).solution()

In [26]: breadth_first_graph_search(estado).solution()

Out[26]: ['Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco abajo',
          'Mover hueco derecha',
          'Mover hueco derecha',
          'Mover hueco abajo']
```

En este ejercicio se han podido observar los resultados y tiempo de la ejecución de los algoritmos de búsqueda ciega. Escribe aquí tus conclusiones:

La llamada a breadth_first_tree_search(estado).solution() nos muestra la solución más corta. La llamada a breadth_first_graph_search(estado).solution() tiene control de repetidos, por lo que para casos grandes será mas rápida que la anterior (en casos pequeños puede no compensar por el coste de controlar los nodos repetidos). Sin embargo, ambos algoritmos muestran la misma solución.

Analicemos un poco más a fondo depth_first_graph_search(estado).solution(): Como la función actions devuelve las posibles acciones en este orden: arriba,izquierda,abajo y derecha, hará (arriba,arriba,...) y tiene que calcular todas las hojas posibles antes de pasar a la rama (arriba, izquierda,...) por lo que tiene que calcular una gran parte del árbol total antes de encontrar la solución. Sin embargo, tiene que terminar en algún momento ya que tiene control de repetidos y tiene solución.

2.1.3. Ejercicio 3: definir al menos las siguientes funciones heurísticas para el 8 puzzle:

- linear(node): cuenta el número de casillas mal colocadas respecto al estado final.
- manhattan(node): suma la distancia Manhattan desde cada casilla a la posición en la que debería estar en el estado final.
- max_heuristic(node: maximo de las dos anteriores
- sqrt_manhattan(node): raíz cuadrada de la distancia Manhattan

In [27]:
```
# Heuristicas para el 8 Puzzle

        def linear(node):
            #goal = node.state.goal
            bad = 0
            goal = (1,2,3,4,5,6,7,8,0)
            state = node.state
            for i in range(0,8):
                if state[i]!=goal[i]:
                    bad += 1

            return bad

        def manhattan(node):
            state = node.state
            goal = (1,2,3,4,5,6,7,8,0)
            mhd = 0
            for i in range(0,8): # columnas y filas empiezan en 0
                row_i = i // 3
                col_i = i % 3
                num = state[i]
                if num != 0:
                    row_goal_num = (num-1) // 3
                    #division entera, se resta 1 porque goal no empieza en 0
                    col_goal_num = (num-1) % 3
                else:
                # el 0 es un caso distinto porque no está colocado en goal
                # en su sitio 'natural'
                    row_goal_num = 2
                    col_goal_num = 2
                mhd += abs(row_goal_num - row_i) + abs(col_goal_num - col_i)

            return mhd

        def sqrt_manhattan(node):
            #state = node.state

            mhd = manhattan(node)

            return math.sqrt(mhd)
```

```
def max_heuristic(node):
    score1 = manhattan(node)
    score2 = linear(node)
    return max(score1, score2)
```

2.1.4. Ejercicio 4.

Usar las implementaciones de los algoritmos que correspondan a búsqueda_coste_uniforme, busqueda_primero_el_mejor y búsqueda_a_estrella (con las heurísticas anteriores) para resolver el problema del 8 puzzle para el siguiente estado inicial y comparar los costes temporales usando %timeit y comentar los resultados.

```
+---+---+---+
| 2 | 4 | 3 |
+---+---+---+
| 1 | 5 | 6 |
+---+---+---+
| 7 | 8 | H |
+---+---+---+
```

```
In [28]: puzle = Ocho_Puzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
         astar_search(puzle).solution()
```

```
Out[28]: ['Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco abajo',
          'Mover hueco derecha',
          'Mover hueco derecha',
          'Mover hueco abajo']
```

```
In [29]: astar_search(puzle,manhattan).solution()
```

```
Out[29]: ['Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco abajo',
          'Mover hueco derecha',
          'Mover hueco derecha',
          'Mover hueco abajo']
```

```
In [30]: puzle.initial
```

```
Out[30]: (2, 4, 3, 1, 5, 6, 7, 8, 0)

In [31]: puzle.goal

Out[31]: (1, 2, 3, 4, 5, 6, 7, 8, 0)

In [32]: astar_search(puzle,linear).solution()

Out[32]: ['Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco abajo',
          'Mover hueco derecha',
          'Mover hueco derecha',
          'Mover hueco abajo']

In [33]: astar_search(puzle,max_heuristic).solution()

Out[33]: ['Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco abajo',
          'Mover hueco derecha',
          'Mover hueco derecha',
          'Mover hueco abajo']
```

¿Has notado diferencias en los tiempos de ejecución? Vamos a medirlo.Aunque las heurísticas no afectan a la solución obtenida sí hay diferencias importantes en el tiempo de cálculo

```
In [34]: puzzle_1 = Ocho_Puzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
         puzzle_2 = Ocho_Puzzle((1, 2, 3, 4, 5, 6, 0, 7, 8))
         puzzle_3 = Ocho_Puzzle((1, 2, 3, 4, 5, 7, 8, 6, 0))

In [35]: %%timeit
         astar_search(puzzle_1, linear)
         astar_search(puzzle_2, linear)
         astar_search(puzzle_3, linear)

100 loops, best of 3: 2.91 ms per loop


In [36]: %%timeit
         astar_search(puzzle_1, manhattan)
         astar_search(puzzle_2, manhattan)
         astar_search(puzzle_3, manhattan)
```

```
100 loops, best of 3: 2.6 ms per loop
```

```
In [37]: %%timeit
         astar_search(puzzle_1, sqrt_manhattan)
         astar_search(puzzle_2, sqrt_manhattan)
         astar_search(puzzle_3, sqrt_manhattan)

10 loops, best of 3: 30.2 ms per loop
```

```
In [38]: %%timeit
         astar_search(puzzle_1, max_heuristic)
         astar_search(puzzle_2, max_heuristic)
         astar_search(puzzle_3, max_heuristic)

100 loops, best of 3: 2.91 ms per loop
```

```
In [39]: %%timeit
         best_first_graph_search(puzzle_1, linear)
         best_first_graph_search(puzzle_2, linear)
         best_first_graph_search(puzzle_3, linear)

10 loops, best of 3: 69.9 ms per loop
```

```
In [40]: %%timeit
         best_first_graph_search(puzzle_1, manhattan)
         best_first_graph_search(puzzle_2, manhattan)
         best_first_graph_search(puzzle_3, manhattan)

100 loops, best of 3: 2.37 ms per loop
```

```
In [41]: %%timeit
         best_first_graph_search(puzzle_1, sqrt_manhattan)
         best_first_graph_search(puzzle_2, sqrt_manhattan)
         best_first_graph_search(puzzle_3, sqrt_manhattan)

100 loops, best of 3: 2.4 ms per loop
```

```
In [42]: %%timeit
         best_first_graph_search(puzzle_1, max_heuristic)
         best_first_graph_search(puzzle_2, max_heuristic)
         best_first_graph_search(puzzle_3, max_heuristic)
```

```
100 loops, best of 3: 2.57 ms per loop
```

```
In [43]: %%timeit
         uniform_cost_search(puzzle_1)
         uniform_cost_search(puzzle_2)
         uniform_cost_search(puzzle_3)
```

```
1 loop, best of 3: 380 ms per loop
```

Escribe aquí tus conclusiones sobre qué heurística es mejor y por qué.

La heurística con el mejor tiempo siempre es la manhattan, que resulta de sumar las distancias Manhattan de cada elemento a su posicion final. La manhattan es más informada que la lineal, ya que para las celdas bien colocadas ambas asignan 0 y para las mal colocadas lineal asigna 1 y manhattan >=1 (y luego suman estos valores), luego lineal <= manhattan. También la manhattan es mejor que su raíz pues también es más informada (para valores >=1 sqrt(alpha)<=alpha) y además ahorra el cálculo de la raíz. Tiene sentido que la heurística max_heuristic no compense: tiene que calcular siempre dos heurísticas, linear y manhattan, para acabar cogiendo la manhattan.

2.2. Parte III: Calcular estadísticas sobre la ejecución de los algoritmos para resolución de problemas de ocho puzzle.

2.2.1. El objetivo es comprobar experimentalmente las propiedades teóricas de los algoritmos vistas en clase.

Usaremos la función %timeit para medir los tiempos y para el espacio una version modificada de Problema que almacena el número de nodos.

En la clase modificada también vamos a incluir el calculo que nos diga si un cierto tablero tiene solución o no. Esto es muy útil.. como hemos comentado al principio solo algunos tableros tienen solucion. En el caso del puzle de 8 un tablero se ha demostrado que se puede comprobar calculando el número de inversiones. Si es impar el tablero no tiene solución.

The solvability of a configuration can be checked by calculating the Inversion Permutation. If the total Inversion Permutation is even then the initial configuration is solvable else the initial configuration is not solvable which means that only 9!/2 initial states lead to a solution.

```
In [44]: # Hacemos una definición ampliada de la clase Problem de AIMA que nos va a
         # permitir experimentar con distintos estados iniciales, algoritmos y heurísticas,
         # para resolver el 8-puzzle. Añadimos en la clase ampliada la capacidad para
         # contar el número de nodos analizados durante la búsqueda:
```

```
class Problema_con_Analizados(Problem):

    """Es un problema que se comporta exactamente igual que el que recibe al
       inicializarse, y además incorpora unos atributos nuevos para almacenar el
       número de nodos analizados durante la búsqueda. De esta manera, no
       tenemos que modificar el código del algoritmo de búsqueda."""

    def __init__(self, problem):
        self.initial = problem.initial
        self.problem = problem
        self.analizados  = 0
        self.goal = problem.goal

    def actions(self, estado):
        return self.problem.actions(estado)

    def result(self, estado, accion):
        return self.problem.result(estado, accion)

    def goal_test(self, estado):
        self.analizados += 1
        return self.problem.goal_test(estado)

    def coste_de_aplicar_accion(self, estado, accion):
        return self.problem.coste_de_aplicar_accion(estado,accion)

    def check_solvability(self, state):
        """ Checks if the given state is solvable """

        inversion = 0
        for i in range(len(state)):
            for j in range(i+1, len(state)):
                if (state[i] > state[j]) and state[i] != 0 and state[j]!= 0:
                    inversion += 1

        return inversion % 2 == 0
```

```
In [45]: estado_inicial = (1,2,3,4,5,6,7,0,8)
         p8p=Problema_con_Analizados(Ocho_Puzzle(estado_inicial))
         p8 = Ocho_Puzzle(estado_inicial)

In [46]: p8p.initial

Out[46]: (1, 2, 3, 4, 5, 6, 7, 0, 8)

In [47]: p8p.goal

Out[47]: (1, 2, 3, 4, 5, 6, 7, 8, 0)
```

```
In [48]: puzzle_1 = Ocho_Puzzle((2, 4, 3, 1, 5, 6, 7, 8, 0))
         astar_search(puzzle_1, manhattan).solution()

Out[48]: ['Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco arriba',
          'Mover hueco izquierda',
          'Mover hueco abajo',
          'Mover hueco derecha',
          'Mover hueco derecha',
          'Mover hueco abajo']

In [49]: astar_search(p8, manhattan).solution()

Out[49]: ['Mover hueco derecha']

In [50]: astar_search(p8p, manhattan).solution()

Out[50]: ['Mover hueco derecha']

In [51]: def resuelve_ocho_puzzle(estado_inicial, algoritmo, h=None):
             """Función para aplicar un algoritmo de búsqueda dado al problema del ocho
                puzzle, con un estado inicial dado y (cuando el algoritmo lo necesite)
                una heurística dada.
                Ejemplo de uso:
                    puzzle_1 = (2, 4, 3, 1, 5, 6, 7, 8, 0)
                    resuelve_ocho_puzzle(puzzle_1,astar_search,h2_ocho_puzzle)
                 Solución: ['Mover hueco arriba', 'Mover hueco izquierda',
                 'Mover hueco arriba', 'Mover hueco izquierda', 'Mover hueco abajo',
                 'Mover hueco derecha', 'Mover hueco derecha', 'Mover hueco abajo']
                 Algoritmo: astar_search
                 Heurística: h2_ocho_puzzle
                 Longitud de la solución: 8. Nodos analizados: 11
                """

             p8p=Problema_con_Analizados(Ocho_Puzzle(estado_inicial))
             if p8p.check_solvability(estado_inicial):
                 if h:
                     sol= algoritmo(p8p,h).solution()
                 else:
                     sol= algoritmo(p8p).solution()
                 print("Solución: {0}".format(sol))
                 print("Algoritmo: {0}".format(algoritmo.__name__))
                 if h:
                     print("Heurística: {0}".format(h.__name__))
                 else:
                     pass
```

```
                print("Longitud de la solución: {0}. Nodos analizados: {1}"
                        .format(len(sol),p8p.analizados))
            else:
                print("Este problema no tiene solucion. ")
```

```
In [52]: resuelve_ocho_puzzle(estado_inicial,astar_search,manhattan)
```

```
Solución: ['Mover hueco derecha']
Algoritmo: astar_search
Heurística: manhattan
Longitud de la solución: 1. Nodos analizados: 2
```

```
In [62]: %%time
         estado_inicial = (4,5,6,1,0,3,7,8,2)
         resuelve_ocho_puzzle(estado_inicial, astar_search, manhattan)
```

```
Solución: ['Mover hueco derecha', 'Mover hueco arriba', 'Mover hueco izquierda',
'Mover hueco abajo', 'Mover hueco izquierda', 'Mover hueco abajo', 'Mover hueco derecha',
'Mover hueco derecha', 'Mover hueco arriba', 'Mover hueco arriba', 'Mover hueco izquierda',
'Mover hueco abajo', 'Mover hueco abajo', 'Mover hueco izquierda', 'Mover hueco arriba',
'Mover hueco arriba', 'Mover hueco derecha', 'Mover hueco abajo', 'Mover hueco derecha',
'Mover hueco abajo']
Algoritmo: astar_search
Heurística: manhattan
Longitud de la solución: 20. Nodos analizados: 1636
CPU times: user 702 ms, sys: 9 µs, total: 702 ms
Wall time: 701 ms
```

Ejercicio 5

Intentar resolver usando las distintas búsquedas y en su caso, las distintas heurísticas, el problema del 8 puzzle para los siguientes estados

E1 = (2,1,3,4,8,6,7,0,5), E2 = (1,0,3,4,8,6,7,2,5), E3 = (4,5,6,1,0,3,7,8,2), E4 = (1,2,3,0,5,6,4,7,8)

Se pide, en cada caso, obtener detalles del tiempo y espacio necesario para la resolución de estos estados. Hacerlo con la función resuelve_ocho_puzzle, para obtener, además de la solución, la longitud (el coste) de la solución obtenida y el número de nodos analizados. Anotar los resultados en la siguiente tabla (L, longitud de la solución, NA, nodos analizados, T, tiempo, y justificarlos con las distintas propiedades teóricas estudiadas.

	E1	E2	E3	E4
Anchura	L= 17 T= 6.39s NA= 14017	L= 11 T= 57.7ms NA= 1234	L= 20 T= 1m26s NA= 52506	L= 3 T= 184mics NA= 13
Profundidad	L= 56963 T= 19m16s NA= 125976	L= 42141 T= 4m42s NA= 45939	L= 65020 T= 14m 37s NA= 85654	L= 27 T= 1mms NA= 28
Coste Uniforme	L= 17 T= 44.1s NA= 14092	L= 11 T= 170ms NA= 870	L= 20 T= 9m2s NA= 48428	L= 3 T= 319mics NA= 13
Primero el mejor (linear)	L= 39 T= 8.99ms NA= 171	L= 33 T= 2.45ms NA= 78	L= 76 T= 131ms NA= 747	L= 3 T= 226mics NA= 4
Primero el mejor (manhattan)	L= 25 T= 9.04ms NA= 175	L= 15 T= 1.64ms NA= 53	L= 38 T= 13.9ms NA= 220	L= 3 T= 247mics NA= 4
A* (linear)	L= 17 T= 164ms NA= 835	L= 11 T= 2.63ms NA= 74	L= 20 T= 2.21s NA= 3230	L= 3 T= 239mics NA= 4
A* (manhattan)	L= 17 T= 50.6ms NA= 433	L= 11 T= 1.59ms NA= 49	L= 20 T= 633ms NA= 1636	L= 3 T= 249mics NA= 4

Tras ejecutar los distintos algoritmos de búsqueda y anotar los respectivos resultados, observamos que la diferencia entre las búsquedas ciegas y las búsquedas con heurísticas es abismal. Mientras que el algoritmo de búsqueda en profundidad ha llegado a tardar 19 minutos y 16 segundos teniendo que analizar 125.976 nodos, las búsquedas heurísticas han llegado a encontrar una solución en tan solo 8.99 microsegundos. Se confirman pues, las distintas propiedades teóricas estudiadas: los algoritmos de búsqueda no informados son muy ineficientes en la mayoría de los casos y ante la explosión combinatoria, la fuerza bruta es impracticable. Pero las búsquedas heurísticas, mejorando sustancialmente la eficiencia, nos arrojan un rayo de luz que nos ilumina entre tanta oscuridad.

2.2.2. Ejercicio 6: [OPCIONAL] Definir una heurística más informada y completa la tabla anterior para ver cómo afecta al número de nodos generados por los algoritmos

La heurística manhattan se puede ver como una simplificación del problema en el que todas las casillas están vacias salvo una por lo que podemos llevar esta a su posición de destino con únicamente k movimientos, siendo k la distancia manhattan pues no permitimos movimientos en diagonal.

Nosotros hemos planteado, para idear una heurística más informada, calcular cuándo una pieza en su desplazamiento hasta su casilla de destino va a obligar a moverse a algunas casillas ya colocadas. Consideremos el siguiente ejemplo:

```
Disposición tablero   Numeración diagonales
   +---+---+---+        +---+---+---+
   | - | 2 | 3 |        | D | 1 | 2 |
   +---+---+---+        +---+---+---+
   | 4 | 5 | 6 |        | 1 | 2 | 3 |
   +---+---+---+        +---+---+---+
   | 7 | 8 | 1 |        | 2 | 3 | 0 |
   +---+---+---+        +---+---+---+
```

Todas las piezas se encuentran en su posición de destino salvo la pieza 1, por tanto, con esta disposición del tablero la heurística manhattan nos daría un valor de 4. Sin embargo, es fácil ver que, para llegar del punto O al punto D (figura de numeración de diagonales), necesariamente tenemos que atravesar las diagonales 1, 2 y 3 de la figura y, dado que en cada movimiento únicamente podemos colocar en su casilla de destino una pieza, esto nos implica que vamos a tener que descolocar como mínimo una pieza de estas diagonales y luego volver a colocarla.

En definitiva, si identificamos una pieza que debe atravesar k diagonales con todas sus piezas correctamente colocadas podemos sumar a la distancia manhattan 2*k unidades. Es claro por tanto con el razonamiento anterior que es consistente, nuestra heurística sigue acotando en todo caso inferiormente el número de unidades y, además, por su propio cálculo es claro que es más informada que la heurística manhattan.

Nuestra heurística por tanto va a tratar de dejarse engañar por estados cercanos al estado final pero con piezas muy alejadas de su posición de destino pues, en estos casos, para llegar a la solución, pese a que nuestro tablero muestre un estado de casi resolución, es necesario descolocar piezas ya situadas. Implementamos ahora esta heurística y analizamos sus propiedades:

In [54]:
```
# Por simplicidad esta heurística únicamente contempla las diagonales 1 2 y 3 de
# la figura anteriormente mostrada. Se penaliza alejar mucho los elementos de sus
# casillas pese a tener un gran número de piezas colocadas.

def manhattan_improved(node):
    state = node.state
    goal = (1,2,3,4,5,6,7,8,0)

    # 1.- Cálculo de la distancia manhattan
    mhd = 0
    for i in range(0,8): # columnas y filas empiezan en 0
        row_i = i // 3
        col_i = i % 3
        num = state[i]
        if num != 0:
            row_goal_num = (num-1) // 3
            col_goal_num = (num-1) % 3
        else:
            row_goal_num = 2
            col_goal_num = 2
        mhd += abs(row_goal_num - row_i) + abs(col_goal_num - col_i)
```

```
            # 2.- Análisis por diagonales

            # 2.1.- Primera diagonal colocada y el uno descolocado:
            if state[1]==2 and state[3]==4 and state[0]!=1:
                mhd +=2
            # 2.2.- Segunda diagonal colocada y las piezas 1 2 o 4 teniendo que
            #       atravesar esta diagonal:
            if state[2]==3 and state[5]==5 and state[6]==7 and
            bool({1,2,4} & {state[5], state[7],state[8]}) :
                   mhd += 2
            # 2.3.- Tercera diagonal colocada (y suponemos no solución):
            if state[5]==6 and state[7]==8:
                mhd +=2

            return mhd
```

Análisis de las propiedades:

```
In [63]: %%time

         # E1 = (2,1,3,4,8,6,7,0,5)
         estado_inicial = (2,1,3,4,8,6,7,0,5)
         resuelve_ocho_puzzle(estado_inicial, astar_search, manhattan_improved)
```

```
Solución: ['Mover hueco izquierda', 'Mover hueco arriba', 'Mover hueco arriba',
'Mover hueco derecha', 'Mover hueco derecha', 'Mover hueco abajo', 'Mover hueco izquierda',
'Mover hueco izquierda', 'Mover hueco abajo', 'Mover hueco derecha', 'Mover hueco derecha',
'Mover hueco arriba', 'Mover hueco arriba', 'Mover hueco izquierda', 'Mover hueco abajo',
'Mover hueco abajo', 'Mover hueco derecha']
Algoritmo: astar_search
Heurística: manhattan_improved
Longitud de la solución: 17. Nodos analizados: 410
CPU times: user 65.5 ms, sys: 0 ns, total: 65.5 ms
Wall time: 64.3 ms
```

```
In [64]: %%time

         # E2 = (1,0,3,4,8,6,7,2,5)
         estado_inicial = (1,0,3,4,8,6,7,2,5)
         resuelve_ocho_puzzle(estado_inicial, astar_search, manhattan_improved)
```

```
Solución: ['Mover hueco derecha', 'Mover hueco abajo', 'Mover hueco izquierda',
'Mover hueco abajo', 'Mover hueco derecha', 'Mover hueco arriba', 'Mover hueco arriba',
```

```
'Mover hueco izquierda', 'Mover hueco abajo', 'Mover hueco abajo', 'Mover hueco derecha']
Algoritmo: astar_search
Heurística: manhattan_improved
Longitud de la solución: 11. Nodos analizados: 48
CPU times: user 3.47 ms, sys: 0 ns, total: 3.47 ms
Wall time: 3.34 ms
```

```
In [60]: %%time

         #  E3 = (4,5,6,1,0,3,7,8,2)
         estado_inicial = (4,5,6,1,0,3,7,8,2)
         resuelve_ocho_puzzle(estado_inicial, astar_search, manhattan_improved)
```

```
Solución: ['Mover hueco derecha', 'Mover hueco arriba', 'Mover hueco izquierda',
'Mover hueco abajo', 'Mover hueco izquierda', 'Mover hueco abajo', 'Mover hueco derecha',
'Mover hueco derecha', 'Mover hueco arriba', 'Mover hueco arriba', 'Mover hueco izquierda',
'Mover hueco abajo', 'Mover hueco abajo', 'Mover hueco izquierda', 'Mover hueco arriba',
'Mover hueco arriba', 'Mover hueco derecha', 'Mover hueco abajo', 'Mover hueco derecha',
'Mover hueco abajo']
Algoritmo: astar_search
Heurística: manhattan_improved
Longitud de la solución: 20. Nodos analizados: 1528
CPU times: user 602 ms, sys: 46 µs, total: 602 ms
Wall time: 601 ms
```

```
In [61]: %%time

         # E4 = (1,2,3,0,5,6,4,7,8)
         estado_inicial = (1,2,3,0,5,6,4,7,8)
         resuelve_ocho_puzzle(estado_inicial, astar_search, manhattan_improved)
```

```
Solución: ['Mover hueco abajo', 'Mover hueco derecha', 'Mover hueco derecha']
Algoritmo: astar_search
Heurística: manhattan_improved
Longitud de la solución: 3. Nodos analizados: 4
CPU times: user 512 µs, sys: 0 ns, total: 512 µs
Wall time: 453 µs
```

Conclusiones:

A* (manhattan_improved)	L= 17	L= 11	L= 20	L= 3
	T= 65.5 ms	T= 3.1 ms	T= 602 ms	T= 512 µs
	NA= 410	NA= 48	NA= 1528	NA= 4

En definitiva podemos apreciar que para casos de escasa complejidad, con estados iniciales muy cercanos a una solución, el incremento de coste de cómputo de la heurística no se ve reflejado en una mejora del tiempo de ejecución. Sin embargo, en casos de distribución aleatoria de las piezas sí que se aprecia una mejora significativa del tiempo de ejecución. Además se aprecia que en todos los ejemplos se analiza un número menor de nodos.

La heurística definida mejora a la heurística manhattan en tableros totalmente desordenados.

Francisco Javier Blázquez Martínez, Boris Carballa Corredoira, Juan Carlos Villanueva Quirós

El problema de los misioneros

1. EL PROBLEMA DE LOS MISIONEROS --- PRÁCTICA 2 IA GRUPO 5

1.1. Definición del problema siguiendo el esquema proporcionado por AIMA

In [1]:
```
# Importamos la librería search de AIMA
from search import *
```

In [2]:
```
# Definición del problema de los misioneros según el esquema de AIMA
class ProblemaMisioneros(Problem):
    ''' Clase problema (formalizacion de nuestro problema) siguiendo la
        estructura que aima espera que tengan los problemas.'''
    def __init__(self, initial, goal=None):
        '''Inicializacion de nuestro problema.'''
        Problem.__init__(self, initial, goal)
        # cada accion tiene un texto para identificar al operador y despues una
        # tupla con la cantidad de misioneros y canibales que se mueven en la canoa
        self._actions = [('1c', (0,1)), ('1m', (1, 0)), ('2c', (0, 2)),
        ('2m', (2, 0)), ('1m1c', (1, 1))]

    def actions(self, s):
        '''Devuelve las acciones validas para un estado.'''
        # las acciones validas para un estado son aquellas que al aplicarse
        # nos dejan en otro estado valido
        return [a for a in self._actions if self._is_valid(self.result(s, a))]

    def _is_valid(self, s):
        '''Determina si un estado es valido o no.'''
        # un estado es valido si no hay mas canibales que misioneros en ninguna
        # orilla, y si las cantidades estan entre 0 y 3
        return (s[0] >= s[1] or s[0] == 0) and ((3 - s[0]) >= (3 - s[1]) or
        s[0] == 3) and (0 <= s[0] <= 3) and (0 <= s[1] <= 3)

    def result(self, s, a):
        '''Devuelve el estado resultante de aplicar una accion a un estado
           determinado.'''
        # el estado resultante tiene la canoa en el lado opuesto, y con las
        # cantidades de misioneros y canibales actualizadas segun la cantidad
        # que viajaron en la canoa
```

```
            if s[2] == 0:
                return (s[0] - a[1][0], s[1] - a[1][1], 1)
            else:
                return (s[0] + a[1][0], s[1] + a[1][1], 0)
```

In [3]:
```
# Creación del problema dado un estado inicial y un estado objetivo
estado = ProblemaMisioneros((3, 3, 0), (0, 0, 1))
```

1.2. a) Resolved el problema por los distintos métodos de búsqueda vistos:

In [4]:
```
# Solución por búsqueda en anchura sin control de estados repetidos
breadth_first_tree_search(estado).solution()
```

Out[4]:
```
[('2c', (0, 2)),
 ('1c', (0, 1)),
 ('2c', (0, 2)),
 ('1c', (0, 1)),
 ('2m', (2, 0)),
 ('1m1c', (1, 1)),
 ('2m', (2, 0)),
 ('1c', (0, 1)),
 ('2c', (0, 2)),
 ('1c', (0, 1)),
 ('2c', (0, 2))]
```

In [5]:
```
# Solución por búsqueda en anchura con control de estados repetidos
breadth_first_graph_search(estado).solution()
```

Out[5]:
```
[('2c', (0, 2)),
 ('1c', (0, 1)),
 ('2c', (0, 2)),
 ('1c', (0, 1)),
 ('2m', (2, 0)),
 ('1m1c', (1, 1)),
 ('2m', (2, 0)),
 ('1c', (0, 1)),
 ('2c', (0, 2)),
 ('1c', (0, 1)),
 ('2c', (0, 2))]
```

In [6]:
```
# Solución por búsqueda en profundidad sin control de estados repetidos
depth_first_graph_search(estado).solution()
```

Out[6]:
```
[('1m1c', (1, 1)),
 ('1m', (1, 0)),
 ('2c', (0, 2)),
 ('1c', (0, 1)),
 ('2m', (2, 0)),
 ('1m1c', (1, 1)),
```

```
         ('2m', (2, 0)),
         ('1c', (0, 1)),
         ('2c', (0, 2)),
         ('1m', (1, 0)),
         ('1m1c', (1, 1))]
```

```
In [7]: # ------------- NO FINALIZA --------------
        # Solución por búsqueda en profundidad sin control de estados repetidos
        # depth_first_tree_search(estado).solution()
```

1.3. b) Analizad el coste de las soluciones encontradas:

Búsqueda en anchura sin control de repetidos: Encuentra una solución que requiere de once operaciones. Además, suponiendo que cuando hablamos del coste nos referimos al número de operaciones ejecutadas (con cada operación por tanto de coste uno), el algoritmo de búsqueda en anchura alcanza siempre una solución óptima. La solución dada requiere el mínimo número de operaciones para llegar al estado objetivo.

Búsqueda en anchura con control de repetidos: Encuentra una solución que requiere de once operaciones, de hecho, encuentra la misma que el algoritmo que no controla nodos repetidos. Esto es lógico porque la naturaleza del algoritmo es la misma, completo, por lo que siempre que exista solución encuentra una solución y óptimo (gracias a que suponemos el coste igual al número de operaciones), por lo que encuentra una solución que requiere el mínimo número de operaciones.

¿Qué diferencia hallamos entonces entre estos dos algoritmos hermanos? El primero al no requerir de estructuras de datos adicionales para el control de nodos repetidos ni requerir de operaciones adicionales de comprobación es un algoritmo válido e incluso más rápido cuando existen soluciones con un bajo número de operaciones. Cuando este no es el caso, el algoritmo analiza árboles de exploración enteros repetidas veces (árboles que crecen exponencialmente) por lo que el coste en tiempo es mayor.

Búsqueda en profundidad sin control de repetidos: En este caso el algoritmo no es capaz de hallar una solución al problema ni de terminar su ejecución. Esto es debido a que al no existir un control de nodos explorados y, más especialmente, a la existencia de operaciones inversas el algoritmo puede caer en bucles infinitos explorando ciclos.

Por ejemplo, la siguiente secuencia no atraviesa estados prohibidos y genera un bucle:
S0 -> Mover dos caníbales derecha -> Mover dos caníbales izquierda -> S0

Búsqueda en profundidad con control de repetidos: Al añadir el control de nodos repetidos el algoritmo nunca examinará un nodo ya explorado lo que impide que se creen bucles infinitos debidos a la exploración de ciclos en el grafo de estados y operaciones de nuestro problema. Con este algoritmo se halla una solución de coste once operaciones por lo que sabemos que es óptima.

1.4. c) Analizad el coste en memoria de los distintos algoritmos y la causa de sus diferencias:

```
In [8]: # Ampliamos la clase con atributos para realizar las métricas de memoria:
        class ProblemaConMetricas(Problem):

            """Clase extendida para incorporar atributos para la medida del coste
             de ejecución"""

            def __init__(self, problem):
                self.initial = problem.initial
                self.problem = problem
                self.analizados  = 0
                self.goal = problem.goal

            def actions(self, estado):
                return self.problem.actions(estado)

            def _is_valid(self, estado):
                return self.problem._is_valid(estado)

            def result(self, estado, accion):
                return self.problem.result(estado, accion)

            def goal_test(self, estado):
                self.analizados += 1
                return self.problem.goal_test(estado)

            def coste_de_aplicar_accion(self, estado, accion):
                return self.problem.coste_de_aplicar_accion(estado,accion)

In [40]: def resuelve_y_muestra_metricas(problema, algoritmo, h=None):

             if h: sol= algoritmo(problema, h).solution()
             else: sol= algoritmo(problema).solution()

             print("Longitud de la solución: {0}. Nodos analizados: {1}"
             .format(len(sol),problema.analizados))

In [41]: problema_misioneros = ProblemaMisioneros((3, 3, 0), (0, 0, 1))
```

Búsqueda en anchura sin control de repetidos:

```
In [11]: %%time
         breadth_first_tree_search(estado).solution()

CPU times: user 113 ms, sys: 772 µs, total: 114 ms
Wall time: 113 ms
```

```
Out[11]: [('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2m', (2, 0)),
          ('1m1c', (1, 1)),
          ('2m', (2, 0)),
          ('1c', (0, 1)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2c', (0, 2))]
```

```
In [12]: %%timeit
         breadth_first_tree_search(estado).solution()
```

```
10 loops, best of 3: 67.3 ms per loop
```

```
In [42]: problema_metricas   = ProblemaConMetricas(problema_misioneros)
         resuelve_y_muestra_metricas(problema_metricas, breadth_first_tree_search)
```

```
Longitud de la solución: 11. Nodos analizados: 11878
```

Búsqueda en anchura con control de repetidos:

```
In [50]: %%time
         breadth_first_graph_search(estado).solution()
```

```
CPU times: user 154 µs, sys: 6 µs, total: 160 µs
Wall time: 164 µs
```

```
Out[50]: [('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2m', (2, 0)),
          ('1m1c', (1, 1)),
          ('2m', (2, 0)),
          ('1c', (0, 1)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2c', (0, 2))]
```

```
In [18]: %%timeit
         breadth_first_graph_search(estado).solution()
```

```
10000 loops, best of 3: 79.6 µs per loop
```

```
In [43]: problema_metricas   = ProblemaConMetricas(problema_misioneros)
         resuelve_y_muestra_metricas(problema_metricas, breadth_first_graph_search)
```

```
Longitud de la solución: 11. Nodos analizados: 15
```

Búsqueda en profundidad con control de repetidos:

```
In [51]: %%time
         depth_first_graph_search(estado).solution()
```

```
CPU times: user 151 µs, sys: 6 µs, total: 157 µs
Wall time: 160 µs
```

```
Out[51]: [('1m1c', (1, 1)),
          ('1m', (1, 0)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2m', (2, 0)),
          ('1m1c', (1, 1)),
          ('2m', (2, 0)),
          ('1c', (0, 1)),
          ('2c', (0, 2)),
          ('1m', (1, 0)),
          ('1m1c', (1, 1))]
```

```
In [20]: %%timeit
         depth_first_graph_search(estado).solution()
```

```
10000 loops, best of 3: 78.4 µs per loop
```

```
In [48]: problema_metricas   = ProblemaConMetricas(problema_misioneros)
         resuelve_y_muestra_metricas(problema_metricas, depth_first_graph_search)
```

```
Longitud de la solución: 11. Nodos analizados: 12
```

Búsqueda en profundidad sin control de repetidos:

```
In [ ]: # ------------- NO FINALIZA --------------
        # %%time
        # depth_first_tree_search(estado).solution()
```

```
In [ ]: # ------------- NO FINALIZA --------------
        # %%timeit
        # depth_first_tree_search(estado).solution()
```

```
In [ ]: # ------------- NO FINALIZA --------------
        # problema_metricas   = ProblemaConMetricas(problema_misioneros)
        # resuelve_y_muestra_metricas(problema_metricas, depth_first_tree_search)
```

1.5. d) ¿Qué algoritmo consideras mejor? Razonad la respuesta:

Está claro en esta situación que el algoritmo de búsqueda en profundidad sin control de repetidos no puede ser el mejor por no ser ni siquiera válido. De los restantes observamos que la búsqueda en anchura sin control de repetidos genera también una enorme cantidad de nodos explorados (12000 frente a los 12 de la búsqueda en profundidad con control de repetidos) y que a su vez su tiempo de ejecución es también del orden de mil veces mayor a su versión con control de ciclos. Esto es debido a que gracias a la representación tan sencilla del problema (con a penas tres valores enteros) las comprobaciones son de bajísimo coste algorítmico.

Por tanto, para este problema queda patente que el control de ciclos incrementa el coste de las iteraciones ridículamente y reduce en órdenes exponenciales (en función de la profundidad de las soluciones encontradas) el número de nodos analizados. De los dos alogritmos con control de repeticiones el de búsqueda en profundidad parece ser el mejor debido a la existencia de múltiples soluciones distribuidas por el árbol de exploración. La existencia de soluciones a baja profundidad hace la búsqueda en anchura también una opción válida y eficaz.

1.6. Ejercicio opcional. Define alguna heurística y estudia las propiedades del algoritmo A*

La lógica parece indicarnos que cuantos más misioneros y caníbales haya en la orilla derecha más cerca de la solución estamos. Como bien se explica en el temario de la asignatura, dentro de las propias condiciones del problema se especifica que no se puede llevar a más de dos personas en la lancha. Es por esto que para llevar k personas de la orilla izquierda a la derecha necesitaremos como mínimo k/2 viajes en la lancha (de hecho generalmente podríamos crear una heurística más informada gracias a que únicamente tenemos una balsa y, por tanto, los viajes de vuelta serán imprescindibles en ciertos casos). Necesitar como mínimo k/2 viajes (con k el número de personas en la orilla izquierda) nos hace ver que la heurística definida es en efecto consistente.

```
In [29]: ## Heurística elegida: (Número de personas en la orilla izquierda) /
         ## (2 = Capacidad de la barca)
         def heuristica(node):
             state = node.state
             return (state[0] + state[1])/2
```

```
In [30]: astar_search(problema_misioneros, heuristica).solution()
```

```
Out[30]: [('1m1c', (1, 1)),
          ('1m', (1, 0)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2m', (2, 0)),
          ('1m1c', (1, 1)),
          ('2m', (2, 0)),
          ('1c', (0, 1)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2c', (0, 2))]
```

Estudiamos ahora las propiedades de la heurística dada

```
In [46]: %%time
         astar_search(estado, heuristica).solution()

CPU times: user 470 µs, sys: 24 µs, total: 494 µs
Wall time: 501 µs


Out[46]: [('1m1c', (1, 1)),
          ('1m', (1, 0)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2m', (2, 0)),
          ('1m1c', (1, 1)),
          ('2m', (2, 0)),
          ('1c', (0, 1)),
          ('2c', (0, 2)),
          ('1c', (0, 1)),
          ('2c', (0, 2))]

In [49]: %%timeit
         astar_search(estado, heuristica).solution()

10000 loops, best of 3: 143 µs per loop


In [47]: problema_metricas   = ProblemaConMetricas(problema_misioneros)
         resuelve_y_muestra_metricas(problema_metricas, astar_search, heuristica)

Longitud de la solución: 11. Nodos analizados: 14
```

Los resultados muestran que la búsqueda ciega en profundidad analiza menos nodos que la búsqueda guiada por nuestra heurística. Esto, junto con el mayor coste algorítmico causado por el añadido de complejidad de la heurística hace que también en los tiempos de ejecución se refleje un empeoramiento con respecto a los métodos de búsqueda no informados.

La conclusión que podemos obtener es que, dada la escasa complejidad (en cuanto a tamaño del árbol de exploración) del problema y dada la existencia de soluciones a baja profundidad (con un bajo número de operaciones), el uso de una heurística si bien no supone un empeoramiento sustancial del tiempo ni memoria empleada en la resolución del problema (gracias a que la heurística es de cómputo en coste constante y bajo), si que es innecesaria para este problema en concreto.

Si en vez de tratar el problema de los misioneros (3 misioneros y 3 caníbales) tratásemos el problema de las Jornadas Mundiales de la Juventud (JMJ, con 3000 misioneros y 3000 jóvenes caníbales) el uso de una heurística sería absolutamente imprescindible y los métodos de búsqueda ciega resultarían inaplicables.