Diseño de Agentes Inteligentes

Tarea 2. Agentes Solucionadores de Problemas (PSA) y búsqueda

Descripción del problema

Esta tarea consiste en resolver un problema al formularlo como un problema de estado simple para ser resuelto por un agente de resolución de problemas (PSA), y luego resolverlo utilizando métodos de búsqueda ciega y heurística seleccionados.

Miembros del equipo

1: Iván Alejandro López Valenzuela - A01284875

2: Franco Mendoza Muraira - A01383399

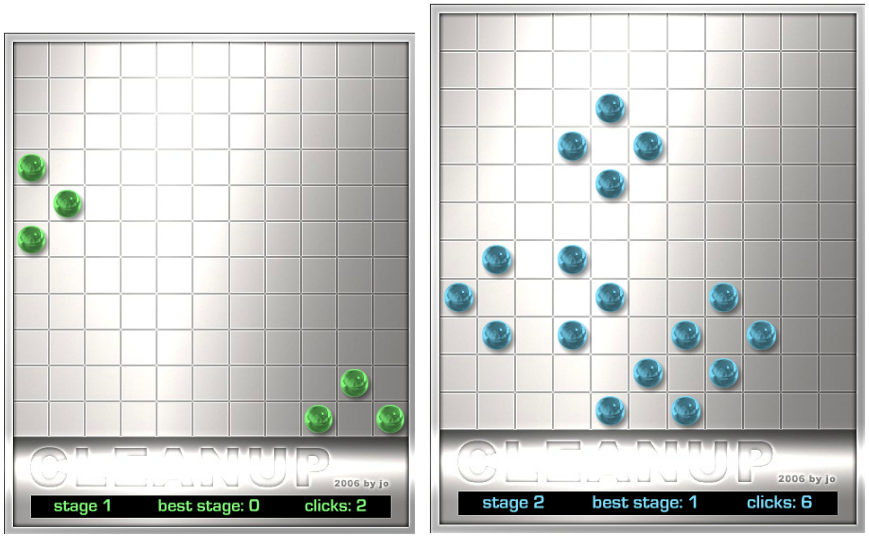
3: Alfonso Elizondo Partida - A01285151

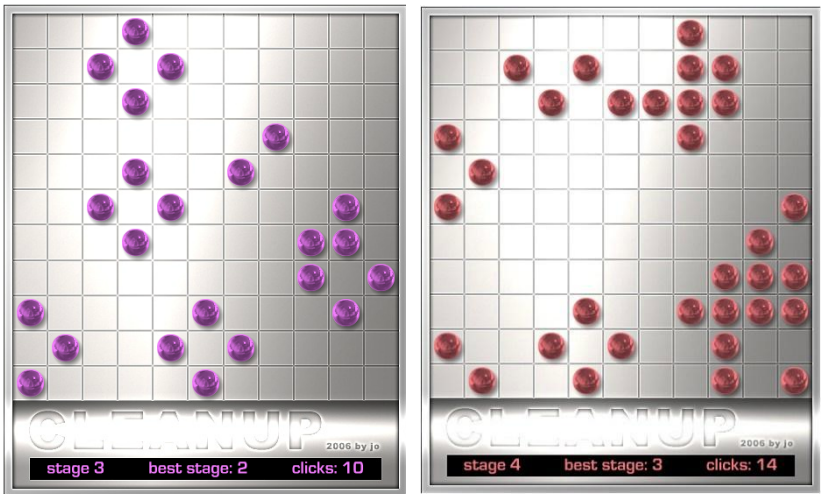
Cleanup Puzzle

Cleanup Puzzle es un juego de mesa que requiere que limpies el tablero. Para ello, deberás hacer clic en las fichas de un tablero tipo cuadrícula. Cada ficha puede estar vacía o contener una bola. El mosaico en el que haga clic no se verá afectado, pero sus vecinos directos (horizontal y verticalmente) se invertirán (si el mosaico contiene una bola, se vaciará y viceversa). El objetivo es determinar una secuencia de clics que elimine todas las bolas del tablero.

El juego, que puedes encontrar y jugar en el enlace https://www.mathsisfun.com/games/cleanup-puzzle.html , consiste en un tablero de 11x11 que se juega en una secuencia de etapas de complejidad incrementada. La complejidad de un escenario está relacionada con la cantidad de bolas en el tablero y sus ubicaciones. En el juego en línea, hay un límite de clics definido para cada etapa y el juego termina cuando no puede encontrar una secuencia de clics lo suficientemente pequeña para limpiar el tablero.

Ejemplos de diferentes etapas del juego en línea:





Para esta actividad debes hacer lo siguiente:

1. Analizar el problema para determinar la información que es relevante para **su formulación como PSA**. Elija una forma de representar estados en el lenguaje de programación (**Python**) donde el tamaño del tablero cuadriculado NxN debe ser un atributo del problema. El número de clics dados en el juego en línea no es relevante para esta tarea.
2. Implemente el código requerido para poder resolver diferentes instancias del problema utilizando los métodos de búsqueda ciega y búsqueda heurística ya programados e incluidos en el código Python de **SimpleAI**. Cuando se ejecutan, deben mostrar correctamente la secuencia de estados y acciones para resolver el problema.
3. Su código **debe funcionar correctamente** al ser utilizado por las funciones que implementan los métodos de búsqueda respectivos.
4. Ejecutar los métodos de búsqueda ciega y heurística para resolver dos instancias del problema, una fácil y otra difícil, y mostrar las soluciones encontradas por cada método.
5. Sería muy bueno y premiado si puede calcular alguna **estadística** sobre los comportamientos de los distintos métodos.
6. Agregue observaciones y conclusiones sobre su experiencia al programar y usar cada algoritmo para encontrar las soluciones: ¿Podrían resolver todos o algunos problemas? ¿Qué tan eficientes fueron? ¿Fue difícil programar el PSA? ¿Qué fue lo más difícil? Además, ¿qué algoritmo parece ser el mejor para los problemas seleccionados? ¿Y qué métodos encontraron las mejores soluciones?

CRITERIOS DE EVALUACIÓN:

Los pesos asignados a las actividades para la evaluación de esta actividad son:

- Formulación PSA: 20%
- Implementación de código python para el problema: 50%
- Solución de varios problemas con los diferentes algoritmos de búsqueda ciega: 20%
- Observaciones y conclusiones: 10%

La calificación será aumentada (otorgada) o reducida (penalizada) dependiendo de la calidad del documento PDF entregado y la documentación interna del código Python con comentarios.

```
#-------------------------------------------------------------------
-----------------------------------------

#    PSA para el problema de Cleanup Puzzle

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

from    simpleai.search    import    SearchProblem,    depth_first,
breadth_first, uniform_cost, greedy, astar

from  simpleai.search.viewers  import  BaseViewer,  ConsoleViewer,
WebViewer

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

#   Definición del problema

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

class CleanupPuzzle(SearchProblem):

    """

          Clase que es usada para definir el problema. Los estados
son representados

        con .

    """

    def __init__(self, puzzle):

          """ Constructor de la clase para el problema del Cleanup
Puzzle.

              Inicializa el problema de acuerdo al puzzle que se
proporciona.

            puzzle: El puzzle que se va a resolver

            tamanio: tamanio del puzzle
```

```
        """
        self.puzzle = puzzle
        self.tamanio = len(puzzle)
            # Llama al constructor de su superclase SearchProblem
(estado inicial = ).
        estado_inicial = self.puzzle
        SearchProblem.__init__(self, estado_inicial)
        self.goal_state =[]
        for i in range(self.tamanio):
            fila = []
            for x in range(self.tamanio):
                fila.append(0)
            self.goal_state.append(tuple(fila))
        for i in range(self.tamanio):
            self.goal_state[i] = tuple(self.goal_state[i])
        self.goal_state = tuple(self.goal_state)
    def actions(self, state):
        #Las acciones se van a hacer en lista.
        actions = []
        for i in range(self.tamanio):
            for j in range(self.tamanio):
                actions.append([i, j])
        return actions
    def result(self, state, action):
        """
            Este método regresa el nuevo estado obtenido despues de
ejecutar la acción.
```

```
            state: ciudad actual.

            action: ciudad a donde voy.

        """

        nuevo = []

        for i in list(state):

            nuevo.append(list(i))

          #Primera parte de la accion es la fila en la que se esta haciendo y la segunda es la columna 3,1 seria la 4ta fila y 2da columna

        fila = action[0]

        columna = action[1]

           # Hacer los cambios en el nuevo estado en los puntos necesarios, dependiendo de los puntos donde se toman las acciones.

           # Se toma en cuenta cuando estan en las orillas de la matriz, para no hacer cambios fuera de la matriz.

        if fila!=0:

            if (state[fila-1][columna]==0):

                nuevo[fila-1][columna] = 1

            else:

                nuevo[fila-1][columna] = 0


        if columna!=0:

            if (state[fila][columna-1]==0):

                nuevo[fila][columna-1] = 1

            else:

                nuevo[fila][columna-1] = 0


        if fila!=self.tamanio-1:

            if (state[fila+1][columna]==0):
```

```
                nuevo[fila+1][columna] = 1
            else:
                nuevo[fila+1][columna] = 0

        if columna!=self.tamanio-1:
            if (state[fila][columna+1]==0):
                nuevo[fila][columna+1] = 1
            else:
                nuevo[fila][columna+1] = 0

        for i in range(len(nuevo)):
            nuevo[i] = tuple(nuevo[i])

        return tuple(nuevo)

    def is_goal(self, state):
        """
              This method evaluates whether the specified state is
the goal state.
            state: La matriz actual.
        """
        #Se busca una matriz llena de ceros.
        return state == self.goal_state
    def cost(self, state, action, state2):
        """
             Este método recibe dos estados y una acción, y regresa
el costo de
            aplicar la acción del primer estado al segundo.
```

```
            Toda accion cuesta 1, mientras menos acciones cuesta
menos.
        """
          #No se pide pero en este caso el costo va a ser = a el
numero de clicks que toma
        return 1


    def heuristic(self, state):
        #Heuristica, mientras mas 1s haya, es mas alto el costo del
puzzle.
        count = 0
        heuristica = list(state)
        for i in heuristica:
            if i==1:
                count+=1
        return count
# Despliega los resultados
def display(result):
    if result is not None:
        for i, (action, state) in enumerate(result.path()):
            if action == None:
                print('Configuración inicial')
            elif i == len(result.path()) - 1:
                    print(i,'- Después de click en:',action,"nuevo
puzzle:")
                            print('¡Meta lograda con costo = ',
result.cost,'!')
            else:
```

```
                    print(i,'- Después de click en:',action,"nuevo
puzzle:")


            for i in state:

                print('  ', i)

    else:

        print('No se puede resolver')

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

#   Programa

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


my_viewer = None

#my_viewer = BaseViewer()        # Solo estadísticas

#my_viewer = ConsoleViewer()     # Texto en la consola

#my_viewer = WebViewer()          # Abrir en un bfilaser en la liga
http://localhost:8000

#Puzzle1

puzzle = ((0, 0, 1, 0),

        (0, 0, 0, 1),

        (1, 0, 0, 1),

        (0, 1, 1, 0))

puzzle = ((0,1,1,0),

          (1,0,1,1),

          (1,1,0,1),

          (0,1,1,0))

puzzle = ((0,1,1,1),

          (1,0,1,1),
```

```
          (1,1,0,1),
          (0,1,1,0))



# Crea un PSA y lo resuelve con la búsqueda primero en anchura
result  =  breadth_first(CleanupPuzzle(puzzle),  graph_search=True,
viewer=my_viewer)
result2=
astar(CleanupPuzzle(puzzle),graph_search=True,viewer=my_viewer)
if my_viewer != None:
    print('Stats:')
    print(my_viewer.stats)


print()
print('>> Búsqueda Primero en Anchura <<')
display(result)


print()
print('>> Búsqueda ahora con A* <<')
display(result2)
```

FÁCIL (ANCHURA)

```
>> Búsqueda Primero en Anchura <<
Configuración inicial
   (0, 0, 1, 0)
   (0, 0, 0, 1)
   (1, 0, 0, 1)
   (0, 1, 1, 0)
1 - Después de click en: [0, 3] nuevo puzzle:
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (1, 0, 0, 1)
   (0, 1, 1, 0)
2 - Después de click en: [3, 0] nuevo puzzle:
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 1)
   (0, 0, 1, 0)
3 - Después de click en: [3, 3] nuevo puzzle:
¡Meta lograda con costo =  3 !
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
```

```
Stats:
{'max_fringe_size': 1027, 'visited_nodes': 432, 'iterations': 432}
```

FÁCIL (HEURÍSTICA A*)

```
>> Búsqueda ahora con A* <<
Configuración inicial
   (0, 0, 1, 0)
   (0, 0, 0, 1)
   (1, 0, 0, 1)
   (0, 1, 1, 0)
1 - Después de click en: [1, 2] nuevo puzzle:
   (0, 0, 0, 0)
   (0, 1, 0, 0)
   (1, 0, 1, 1)
   (0, 1, 1, 0)
2 - Después de click en: [3, 3] nuevo puzzle:
   (0, 0, 0, 0)
   (0, 1, 0, 0)
   (1, 0, 1, 0)
   (0, 1, 0, 0)
3 - Después de click en: [2, 1] nuevo puzzle:
¡Meta lograda con costo =  3 !
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
```

```
Stats:
{'max_fringe_size': 1175, 'visited_nodes': 510, 'iterations': 510}
```

DIFÍCIL (ANCHURA)

```
>> Búsqueda Primero en Anchura <<
Configuración inicial
   (0, 1, 1, 0)
   (1, 0, 1, 1)
   (1, 1, 0, 1)
   (0, 1, 1, 0)
1 - Después de click en: [0, 0] nuevo puzzle:
   (0, 0, 1, 0)
   (0, 0, 1, 1)
   (1, 1, 0, 1)
   (0, 1, 1, 0)
2 - Después de click en: [0, 3] nuevo puzzle:
   (0, 0, 0, 0)
   (0, 0, 1, 0)
   (1, 1, 0, 1)
   (0, 1, 1, 0)
3 - Después de click en: [2, 2] nuevo puzzle:
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (1, 0, 0, 0)
   (0, 1, 0, 0)
4 - Después de click en: [3, 0] nuevo puzzle:
¡Meta lograda con costo =  4 !
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
```

```
Stats:
{'max_fringe_size': 1350, 'visited_nodes': 775, 'iterations': 775}
```

DIFÍCIL (HEURÍSTICA A*)

```
>> Búsqueda ahora con A* <<
Configuración inicial
   (0, 1, 1, 0)
   (1, 0, 1, 1)
   (1, 1, 0, 1)
   (0, 1, 1, 0)
1 - Después de click en: [1, 1] nuevo puzzle:
   (0, 0, 1, 0)
   (0, 0, 0, 1)
   (1, 0, 0, 1)
   (0, 1, 1, 0)
2 - Después de click en: [3, 0] nuevo puzzle:
   (0, 0, 1, 0)
   (0, 0, 0, 1)
   (0, 0, 0, 1)
   (0, 0, 1, 0)
3 - Después de click en: [3, 3] nuevo puzzle:
   (0, 0, 1, 0)
   (0, 0, 0, 1)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
4 - Después de click en: [0, 3] nuevo puzzle:
¡Meta lograda con costo =  4 !
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
   (0, 0, 0, 0)
```

```
Stats:
{'max_fringe_size': 1833, 'visited_nodes': 1265, 'iterations': 1265}
```

EJEMPLO DONDE ES IMPOSIBLE RESOLVER EL PUZZLE:

```
puzzle = ((0,1,1,1),
          (1,0,1,1),
          (1,1,0,1),
          (0,1,1,0))
```

```
Stats:
{'max_fringe_size': 1520, 'visited_nodes': 4096, 'iterations': 4096}

>> Búsqueda Primero en Anchura <<
No se puede resolver
```

```
Stats:
{'max_fringe_size': 1833, 'visited_nodes': 4096, 'iterations': 4096}

>> Búsqueda ahora con A* <<
No se puede resolver
```

Se ve que en el puzzle pasado, que fue modificado agregando un 1 en la esquina superior derecha al puzzle "difícil", es imposible de resolver después de 4096 iteraciones de parte de los 2 métodos, de anchura y A*. Hay muchos más casos donde se ve que no se pueden resolver puzzles, este es solo 1 ejemplo.

OBSERVACIONES Y CONCLUSIONES

Usando el código que fue programado, no es posible resolver el 100% de problemas que se le dan al algoritmo, ya que algunos son imposibles debido a bucles infinitos que se pueden crear en la toma de decisiones. En los puzzles que se intentaron de resolver, se puede ver que se tomaron una alta cantidad de iteraciones para llegar a la solución final de los problemas, y también al intentar problemas más grandes de tableros de 5x5 o 6x6, se vieron altas esperas donde al final el código no aguantaba y no resolvía los problemas hasta después de 2 horas de ejecución, por lo que la eficiencia del algoritmo no es la mejor.

El PSA fue difícil de programar debido a la necesidad de usar tuplas en el uso de esta librería, y hace las cosas más difíciles, ya que estas tuplas se tenían que hacer listas para poder modificarlas, esto fue lo más difícil. También al buscar las posibilidades que se crean al modificar los tableros, cuando las acciones se encontraban en filas y/o columnas en las esquinas.

En nuestro programa PSA usamos los métodos de anchura y de A estrella, los 2 métodos nos dieron los mismos costos finales al encontrar la solución, la diferencia

fue que en los intentos de puzzles fáciles y difíciles, tomó menos iteraciones el de anchura, por lo que se vio más eficiente.