Actividad : Resolución de problema mediante búsqueda heurística

Objetivos de la actividad

Con esta actividad vas a conseguir implementar la estrategia de búsqueda heurística A* para la resolución de un problema real.

Descripción de la actividad

La empresa Amazon desea utilizar un robot para ordenar el inventario de su almacén.

Amazon cuenta con 3 inventarios (mesa con suministros para vender) localizados en unas posiciones específicas del almacén. El robot se debe encargar de mover los 3 inventarios a una posición objetivo.

El robot puede moverse horizontal y verticalmente, y cargar o descargar un inventario.

Un ejemplo del robot, moviendo el inventario, se puede observar en el siguiente vídeo:

In [ ]:

```
#from IPython.display import YouTubeVideo
#YouTubeVideo('UtBa9yVZBJM')
```

En esta actividad has de utilizar la estrategia de búsqueda heurística A* con el fin de generar un plan que permita al robot de Amazon mover el inventario de un estado inicial a un estado objetivo.

Estado inicial

El estado inicial del problema lo vamos a representar en una mátriz 4x4 de carácteres de la siguiente manera:

	0	1	2	3
0	M1	#		M3
1		#		
2	M2		R	
3				

Donde,

- R: representa el robot. Inicialmente está ubicado en la posición [2,2]
- #: representa una pared.
- M1, M2, e M3: representan los tres inventarios que el robot debe mover. Y se encuentran ubicadas en las posiciones [0,0], [2,0] y [0,3] respectivamente.

Estado Objetivo

El robot debe mover los 3 inventarios, M1, M2 y M3, a la siguientes posiciones:

	0	1	2	3
0		#		
1		#		
2				
3		M3	M2	M1

Tareas a realizar

Implementar el algoritmo A* considerando lo siguiente:

1. como función heurística, la Distancia en Manhattan.
2. el coste real (c) de cada acción del robot es 1.
3. el código deberá ejecutarse e indicar la secuencia de acciones a realizar para alcanzar el estado objetivo utilizando una notación sencilla. Por ejemplo: «mover R fila1 columna2» o «mover R fila0 columna2» o «cargar R M1 fila0 columna2».

Documentos a entregar

- Memoria en word o jupyter notebook explicando en detalle el desarrollo de la actividad. Se recomienda un limite máximo de 10 páginas sin contar el código fuente. La memoria como mínimo debe contenter:
  - Portada
  - Desarrollo de la actividad: análisis, pantallazos de ejecución, pruebas realizadas, plan de acción.
  - Dificultades encontradas
  - Referencias bibliografícas con Normas APA
- Código fuente desarrollado correctamente documentado.

Consideraciones finales

Puede seleccionar el lenguaje de programación que usted desee.

Se deberán crear tantas clases o estructuras de datos como sean necesarias para representar el espacio de estados y los nodos de exploración del árbol.

El programa desarrollado debe ser un trabajo original del estudiante. Cualquier evidencia de o trabajos iguales será calificada con una nota de cero (0).

In [ ]:

```
%matplotlib inline

import numpy as np
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import matplotlib.animation as animation
```

Portada

Planificación y Razonamiento Automático

Jorge Augusto Balsells Orellana

Erick Wilfredo Díaz Saborio

---

31 de Enero de 2021

Desarrollo de Actividad

Definición del Tablero

Vamos a definir una matiz de valores numericos con numpy. Ya que los valores de la matriz son numericos se definen 2 diccionarios de python para poder mapear estos valores.

In [ ]:

```
num_object_mapping = {
    2: 'robot',
    3: 'M1',
    4: 'M2',
    5: 'M3',
    1: 'pared',
    0: 'espacio'
```

```
}

object_num_mapping = dict(map(reversed, num_object_mapping.items()))
```

In [ ]:

```
object_num_mapping
```

Out[ ]:

```
{'M1': 3, 'M2': 4, 'M3': 5, 'espacio': 0, 'pared': 1, 'robot': 2}
```

Euclidean Distance

$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

In [ ]:

```
def euclidean_distance(pos1_x, pos1_y, pos2_x, pos2_y):
     return np.sqrt(pow(abs(pos2_x-pos1_x),2)+pow(abs(pos2_y-pos1_y),2))
```

Heuristica f()

Función de estimación de la distancia entre la posición actual y el nodo final. Esta función recibe como parámetros una matriz, la posición actual y la destino.

Calcula los valores de g, f y h de las celdas vecinas a la posición actual, únicamente si es valido moverse a ellas, es decir si la celda vecina esta vacía o es la posición final.

La función retorna una matriz, cada elemento de la matriz es un vector con los valores de g, f, h, $[x,y]$ si podemos movernos a la celda en caso contrario retorna un arreglo vacío.

In [ ]:

```
def fgh_values(matrix, target, position):

    #Guarda la posicion en coordenadas (x,y) del lugar donde esta ubicado
    #en position y de la carga en target
    [pos_y, pos_x] = position
    [tar_y, tar_x] = target

    #se determinan las dimensiones de filas y columnas del territorio o matriz
    mat_x = np.size(initial_state,0)
    mat_y = np.size(initial_state,1)

    #se crea una matriz de 3x3, que contendra los valores de f,g y h de todo el
    #perimetro del valor ingresado.
    f_mat = [[0,0,0],[0,0,0],[0,0,0]]

    #se operan los 9 valores a ingresar en la matriz f_mat
    for x in range(pos_x-1,pos_x+2):
        for y in range(pos_y-1,pos_y+2):

            #se calculan los valores solamente si el punto esta dentro de las
            #dimensiones de la matriz, o es un espacio, o son las coordenadas
            #de la carga a la que se dirige el robot
            if(x>=0 and x<=mat_x-1 and y>=0 and y<=mat_y-1
               and ((matrix[y,x]==object_num_mapping['espacio']) or ((x,y) == (tar_x, tar_y))) ):

                #Calculo de variables
                g = euclidean_distance(pos_x, pos_y, x, y)
                h = abs(tar_y-y) + abs(tar_x-x)
                f = g + h

                #Se crea un solo vector que se guarda en cada posicion de la matriz f_mat
                f_mat[x-pos_x+1][y-pos_y+1] = [round(f,1), round(g,1), round(h,1), [x,y]]
```

```
            else:

                #Si no se cumple la sentencia, guarda un vector vacio en esa posicion
                f_mat[x-pos_x+1][y-pos_y+1] = []
    return f_mat
```

Implementacion de Nodo

La mejor forma de controlar las casillas o nodos, es creando un objeto donde se pueda almacenar cual es el nodo padre y los valores de g, h y f.

In [ ]:

```
class Node():
    def __init__(self, parent=None, position=None, g=0, h=0, f=0):
        self.parent = parent
        self.position = position #tupla (fila,columna)

        self.g = g
        self.h = h
        self.f = f

    def __eq__(self, other):
        return self,
```

Implementación Inventario

En esta clase almacenamos la información basica de los inventarios, el nombre, la posición inicial y destino.

In [ ]:

```
class Inventario():
    def __init__(self, nombre, pos_init, pos_dest, val):
        self.nombre = nombre
        self.pos_init = pos_init
        self.pos_dest = pos_dest
        self.val = val
```

Algoritmo A*

A continuación, se explica de forma detallada la lógica del algoritmo.

En esta parte del código obtenemos el primer nodo de la lista abierta, luego en un ciclo buscamos en la lista abierta el nodo con menor valor f, y se define este nodo como el nodo actual. Y por último lo eliminamos de la lista abierta y lo movemos a la lista cerrada.

```
    # Obteniendo el nodo actual
    current_node = open_list[0]
    current_index = 0

    #Encontrar el valor de f menor en la lista abierta
    for index, item in enumerate(open_list):
        if item.f < current_node.f:
            current_node = item
            current_index = index

    # Sacamos el nodo con menor f de la lista abierta
    # y lo agregamos a la cerrada
    open_list.pop(current_index)
    closed_list.append(current_node)
```

Se valida si la posición actual es la posición final, si esa condición se cumple tomamos el nodo actual y en un ciclo $while$ obtenemos el padre del ciclo hasta que retorne $Null$, indicando que se llego al nodo inicial. Cada uno de los padres se guarda en una lista y se invierte en el $return$ para obtener la lista de nodos ordenada, esta lista indica el camino del nodo inicial a la posición final.

```
    # Validamos si el nodo actual es el final
```

```
    # Validar si es la posicion final
    if current_node.position == end_node.position:
        path = []
        nodo = current_node
        # Recorremos cada nodo padre para obtener la ruta
        while nodo is not None:
            path.append(nodo)
            nodo = nodo.parent

        return path[::-1] # Retornamos la ruta
```

Este es el paso final antes de repetir la iteración, utilizamos la función **fgh_values**, enviándole como parámetros el tablero actual, la posición actual y el destino, la función nos retorna la matriz con las celdas vecinas a las que es posible moverse con los valores (f,g,h).

```
    # Generar los hijos
    values_matrix = fgh_values(board, target=end, position=current_node.position)
```

Con los valores de las celdas cercanas vamos a seleccionar las celdas validas que pueden ser **hijos**, para esta validación primero buscamos que no existan en la lista abierta, si existe en la lista abierta validamos si el valor de g actual es menor al que tiene el nodo en la lista abierta, si mejora entonces actualizamos el nodo en la lista abierta, cambiamos su nodo padre al nodo actual, y actualizamos sus valores de f y g, el valor h se mantiene igual ya que es la distancia al objetivo.

```
    children = []
    for row in values_matrix:
        for cell in row:
            if len(cell) > 0:
                x = cell[3][0] #Columna
                y = cell[3][1] #Fila
                add_flag = True
                new_g = cell[1] + current_node.g
                new_f = new_g + cell[2]

                # Comprobando que no este en lista abierta
                for node in open_list:
                    if node.position == (y,x):
                        add_flag = False
                        if node.g > new_g:
                            node.parent = current_node
                            node.g = new_g
                            node.f = new_f


                # Comprobando que no este en lista cerrada
                for node in closed_list:
                    if node.position == (y,x):
                        add_flag = False
    `
```

Creamos los objetos nodos hijos y se agregan a la lista abierta.

```
                # Creamos un nuevo nodo
                new_node = Node(parent=current_node, position=(y,x), g=new_g, h=cell[2], f=new_f)
                if add_flag:
                    children.append(new_node)

    open_list.extend(children)
```

Implementacion del Algoritmo

In [ ]:

```
def astar(board, start, end):
    # Creando el nodo inicial y final
    start_node = Node(position=start)
    end_node = Node(position=end)

    # Inicializando lista cerrada y lista abierta
    open_list = []
    closed_list = []
```

```
#Agregamos a la lista abierta el nodo inicial
open_list.append(start_node)

while len(open_list) > 0:

    # Obteniendo el nodo actual
    current_node = open_list[0]
    current_index = 0

    #Encontrar el valor de f menor en la lista abierta
    for index, item in enumerate(open_list):
        if item.f < current_node.f:
            current_node = item
            current_index = index

    # Sacamos el nodo con menor f de la lista abierta
    # y lo agregamos a la cerrada
    open_list.pop(current_index)
    closed_list.append(current_node)


    # Validar si es la posicion final
    if current_node.position == end_node.position:
        path = []
        nodo = current_node
        # Recorremos cada nodo padre para obtener la ruta
        while nodo is not None:
            path.append(nodo)
            nodo = nodo.parent

        return path[::-1] # Retornamos la ruta


    # Generar los hijos
    values_matrix = fgh_values(board, target=end, position=current_node.position)

    children = []
    for row in values_matrix:
        for cell in row:
            if len(cell) > 0:
                x = cell[3][0] #Columna
                y = cell[3][1] #Fila
                add_flag = True
                new_g = cell[1] + current_node.g
                new_f = new_g + cell[2]

                # Comprobando que no este en lista abierta
                for node in open_list:
                    if node.position == (y,x):
                        add_flag = False
                        # Si el valor de g actual es menor cambiamos el
                        # nodo padre y sus valores de g,f
                        if node.g > new_g:
                            node.parent = current_node
                            node.g = new_g
                            node.f = new_f


                # Comprobando que no este en lista cerrada
                for node in closed_list:
                    if node.position == (y,x):
                        add_flag = False

                # Creamos un nuevo nodo
                new_node = Node(parent=current_node, position=(y,x), g=new_g, h=cell[2], f=new_f)
                if add_flag:
                    children.append(new_node)

    # Agregamos a la lista abierta los nuevos nodos hijos
    open_list.extend(children)
```

Funciones para graficar

In [ ]:

```
fig = plt.figure()
ims = []

def mapping(option='',matrix=[], nombre=''):
    global ims

    if(option==''):
        pass
    elif(option=='add'):
        im = plt.imshow(matrix, cmap = plt.cm.gray)
        ims.append([im])
    elif(option=='plot'):
        ani = animation.ArtistAnimation(fig, ims, interval=500, blit=True,repeat_delay=100)
        ani.save(nombre, writer='imagemagick', fps=4)


def update_matrix(current_matrix, curr_pos,new_pos, obj_val, begin_load_route=False):
    #Actualizar Tablero
    if (begin_load_route is False):
        current_matrix[curr_pos] = object_num_mapping['espacio']
    current_matrix[new_pos] = obj_val
    return current_matrix
```

```
<Figure size 432x288 with 0 Axes>
```

Ejecutando Algoritmo A*

Inicializando el tablero

In [ ]:

```
#Matriz de estado inicial con
initial_state = np.matrix([[6, 1, 0, 5],
                           [0, 1, 0, 0],
                           [4, 0, 2, 0],
                           [0, 0, 0, 0]])
```

Generacion de Inventarios

Generamos 3 objetos de tipo inventarios con le enviamos como parametro la posicion inicial y la posicion final.

In [ ]:

```
inventarios = []

# Inventario M3
new_invent = Inventario('M3', pos_init=(0,3), pos_dest=(3,1), val=5)
inventarios.append(new_invent)

# Inventario M2
new_invent = Inventario('M2', pos_init=(2,0), pos_dest=(3,2), val=4)
inventarios.append(new_invent)

# Inventario M1
new_invent = Inventario('M1', pos_init=(0,0), pos_dest=(3,3), val=6)
inventarios.append(new_invent)
```

Ejecución

Se realizo una búsqueda por sub-objetivos, tenemos un arreglo de inventarios, utilizamos un ciclo
for
para recorrer el arreglo de inventarios. Con el inventario actual se realizan los siguientes pasos:

1. Ejecutamos el algoritmo A* para que el robot pueda llegar se su posición actual al inventario, se imprime la ruta.
2. Ejecutamos el algoritmo A* de nuevo para que el robot lleve el inventario a su posición destino y se imprime la ruta.

In [ ]:

```
pos_robot = (2,2)
board = initial_state

begin_route_after_load = False
for inventario in inventarios :
    print(f"=================== Inventario {inventario.nombre} ==========================")
```

```
print(f" Robot Posicion actual {pos_robot}")

## Calcular ruta del robot -> inventario
## Algoritmo A*
ruta1 = astar(board, pos_robot, inventario.pos_init)
print(f"=> Ruta del robot al inventario {inventario.nombre}")
last_pos = pos_robot

## Imprimiendo la ruta
for i, nodo in enumerate(ruta1):
    if (nodo.position != last_pos):
        board = update_matrix(board, last_pos, nodo.position, object_num_mapping['robot'], begin_rout
        begin_route_after_load = False
    last_pos = nodo.position
    print(f"Mover R fila: {nodo.position[0]}, columna: {nodo.position[1]}")


## Calcular ruta del estante -> pos. final estante
## Algoritmo A*
ruta2 = astar(board, inventario.pos_init, inventario.pos_dest)
print(f"=> Ruta del inventario {inventario.nombre} a destino {inventario.pos_dest}")
last_pos = inventario.pos_init

## Imprimiendo la ruta
for nodo in ruta2:
    if (nodo.position != last_pos):
        board = update_matrix(board, last_pos, nodo.position, object_num_mapping['robot'])
    last_pos = nodo.position
    print(f"Cargar R[{inventario.nombre}] fila: {nodo.position[0]}, columna: {nodo.position[1]}")


print(board)
begin_route_after_load = True
pos_robot = inventario.pos_dest
```

```
==================== Inventario M3 ============================
 Robot Posicion actual (2, 2)
=> Ruta del robot al inventario M3
Mover R fila: 2, columna: 2
Mover R fila: 1, columna: 3
Mover R fila: 0, columna: 3
=> Ruta del inventario M3 a destino (3, 1)
Cargar R[M3] fila: 0, columna: 3
Cargar R[M3] fila: 1, columna: 2
Cargar R[M3] fila: 2, columna: 1
Cargar R[M3] fila: 3, columna: 1
[[6 1 0 0]
 [0 1 0 0]
 [4 0 0 0]
 [0 2 0 0]]
==================== Inventario M2 ============================
 Robot Posicion actual (3, 1)
=> Ruta del robot al inventario M2
Mover R fila: 3, columna: 1
Mover R fila: 2, columna: 0
=> Ruta del inventario M2 a destino (3, 2)
Cargar R[M2] fila: 2, columna: 0
Cargar R[M2] fila: 2, columna: 1
Cargar R[M2] fila: 3, columna: 2
[[6 1 0 0]
 [0 1 0 0]
 [0 0 0 0]
 [0 2 2 0]]
==================== Inventario M1 ============================
 Robot Posicion actual (3, 2)
=> Ruta del robot al inventario M1
Mover R fila: 3, columna: 2
Mover R fila: 2, columna: 1
Mover R fila: 1, columna: 0
Mover R fila: 0, columna: 0
=> Ruta del inventario M1 a destino (3, 3)
Cargar R[M1] fila: 0, columna: 0
Cargar R[M1] fila: 1, columna: 0
Cargar R[M1] fila: 2, columna: 1
Cargar R[M1] fila: 2, columna: 2
Cargar R[M1] fila: 3, columna: 3
[[0 1 0 0]
 [0 1 0 0]
 [0 0 0 0]
 [0 2 2 2]]
```

Dificultades Encontradas

1. El algoritmo es sencillo de comprender, sin embargo, requiere trabajo trasladarlo a código y que se encuentre bien estructurado.
2. Una dificultad visible en este código, es que se puede llegar a generar una cantidad muy alta de tiempo de ejecución por funciones asintóticas, dado que tiene algunos ciclos anidados en donde el tiempo de ejecución se puede tornar exponencial en casos de matrices muy grandes.
3. El algoritmo A*recorre una cuadrícula posición por posición hasta llegar a su punto final. Si se crea una matriz muy pequeña, el algoritmo puede generar un trazo con una incerteza muy alta en comparación con un terreno cuadriculado. Por otro lado, si se crea una matriz muy grande, se disminuye el error en una posición, pero se aumenta el procesamiento.

Referencias

1. Roy, B. (2020, February 23). A-Star (A*) Search Algorithm - Towards Data Science. Medium. https://towardsdatascience.com/a-star-a-search-algorithm-eb495fb156bb
2. gammafp. (2017, July 27). Pathfinding A* (A estrella) - Tutorial completo en español [Video]. YouTube. https://www.youtube.com/watch?v=X-5JMScsZ14&t=1363s

In [ ]: