Universidad del Caribe
CONOCIMIENTO Y CULTURA PARA EL DESARROLLO HUMANO

Méndez Pool Joan de Jesús | 160300102 Domingo 08/09/2019
Docente: Hector Fernando Gómez García

Agentes Reactivos

Introducción

Un agente es un perceptor del ambiente a través de sensores y actúa sobre el ambiente a través de actuadores. Los agentes funcionan con ciclos de percepción. pensamientos, y actos, por lo que un agente está conformado de la siguiente forma **Agente = Arquitectura + Programa**.

Un agente racional interpreta cada posible secuencia, un agente racional debería seleccionar una acción que maximice la medida de rendimiento dada la evidencia proveniente por la secuencia de percepción que el agente tiene. La racionalidad es una medida de rendimiento.
Cuando se define a un agente racional se agrupan un conjunto de propiedades denominado por Rendimiento, Ambiente, Actuadores y Sensores.
El agente sobre el cual trabajamos en esta ocasión es el conocido Vacuum Cleaner o Aspiradora, trataremos de simular el comportamiento que presenta el agente, la aspiradora presenta las siguiente características:

- Funcionamiento: Limpieza, Eficiencia: Distancia recorrida para limpiar, Duración de Batería, Seguridad.
- Ambiente: Apartamentos, Casas, Habitaciones, Piso de Madera, Alfombra, Otros obstáculos.
- Actuadores: Ruedas, Diferentes Cepillos, Extractor de polvo.
- Sensores: Cámara, Sensor detector de suciedad, Sensores Infrarrojo de paredes.

Estas son todas las características que al agente debe cumplir para poder trabajar de forma eficiente,
La implementación realizada sobre el Vacuum Cleaner se realiza a nivel de simulación, es decir buscamos imitar el comportamiento de la aspiradora en un entorno creado de forma virtual, para esto nos apoyamos sobre la implementación en el lenguaje de programación Python, con soporte de librerías del mismo lenguaje tales como OpenCV y Matplotlib. Nuestra implementación toma como base una retícula con varios obstáculos en el, y nuestro agente se posiciona de forma aleatoria evitando quedar sobre los obstáculos.

Implementación

Antes de implementar el sistema de producción debemos realizar unas tareas para pasar al comportamiento que tendrá el agente, el primer paso es definir para la posición actual un vector de estados de los ocho vecinos más cercanos de la siguiente forma:

$$S = \begin{matrix} S1 & S2 & S3 \\ S8 & x & S4 \\ S7 & S6 & S5 \end{matrix}$$

Debemos seguir el orden que les asignamos ya que nos permitirán definir una serie de variables para definir su disponibilidad en la retícula, el estado solo se le pueden asignar valores de 0 y 1. Además definimos otro estado de forma paralela para definir si esa posición se sale del rango permitido por el tamaño del ambiente.

$$B = \begin{matrix} S1 & S2 & S3 \\ S8 & x & S4 \\ S7 & S6 & S5 \end{matrix}$$

De esta forma podemos anticipar de forma un poco más eficiente si se ha llegado a una esquina cóncava, además de definir una serie de reglas para seguir manteniendo la lógica del sistema de producción, por ejemplo si se llega a un límite por el lado derecho, ya no se puede mover a esta dirección por lo que debe mantener la lógica en el estado S prohibiendo que se puede mover más hacia la derecha, de esta manera se evita que el agente caiga por el borde de la retícula.

Por lo siguiente debemos definir los patrones de comportamiento para realizar los movimientos permitidos por la lógica de la solución, es decir separamos los estados en cuatro patrones posibles lo cuales representamos en variables de la siguiente forma:

$$x_1 = S2 + S3$$
$$x_2 = S4 + S5$$
$$x_3 = S6 + S7$$
$$x_4 = S8 + S1$$

Ahora debemos pasar al sistema de producción el cual se define como el conjunto de reglas que realizan el comportamiento del agente. En un sistema de producción es importante definir el orden del comportamiento del agente, por lo que en un sistema de producción es usual colocar una serie de reglas adicionales al principio y al final del listado de reglas. La primera regla debe ser para detener al agente cuando cumpla su objetivo y la última define el comportamiento favorito del agente, esta se ejecuta con baja prioridad sobre las demás reglas.

Nuestro sistema de producción esta definido por las siguientes reglas:

1. Si $c(S)$ entonces nil
2. Si $x_1=1$ y $x_2=0$ entonces **este**
3. Si $x_2=1$ y $x_3=0$ entonces **sur**
4. Si $x_3=1$ y $x_4=0$ entonces **oeste**
5. Si $x_4=1$ y $x_1=0$ entonces **norte**
6. Si 1 entonces **norte**

c(S) indica si el agente ha encontrado una esquina cóncava.

Para verificar si se cumplen las condiciones de comportamiento por medio de la señales binarias es de ayuda utilizar Unidades Lógicas Umbralizadas como una serie de filtros para obtener el comportamiento correcto de acuerdo a la situación que se presente. Para ello tomamos el primer caso con x_1 y x_2 *:*

$$x_1\bar{x}_2=(S2+S3)\overline{(S4+S5)}$$

Partiendo del patrón que se forma con el primer término positivo y el segundo termino negado, se puede formar un patrón para cada caso de comportamiento de la siguiente forma:

$$x_2\bar{x}_3=(S4+S5)\overline{(S6+S7)}$$

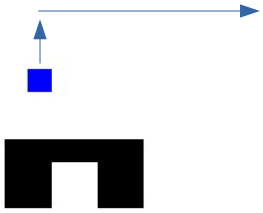
$$x_3\bar{x}_4=(S6+S7)\overline{(S8+S1)}$$

$$x_4\bar{x}_1=(S8+S1)\overline{(S2+S3)}$$

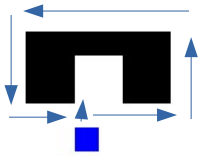
Con lo cual obtenemos cada filtro de señal binaria para elegir el comportamiento adecuado de acuerdo a la posición y estado actual del agente.

Resultados:

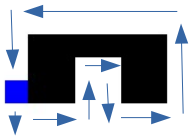
Prueba 1:



Prueba 2:



Prueba 3:



Se implementó el código en Jupyter Notebook, donde se crearon animaciones con Matplotlib para visualizar el comportamiento del agente, el documento se anexará en un documento aparte del reporte.

Código

```
 1 import numpy as np
 2 import matplotlib.pyplot as plt
 3 import matplotlib.animation as animation
 4 import cv2
 5 import random
 6 from IPython.display import HTML
 7 %matplotlib notebook
 8 # Constantes de colores
 9 black, white =((0,0,0),(255,255,255))
10       # Posición del Agente
11blue = (0,0,255)
12       # posiciones vecinas definidas por el siguiente orden de terminos (y, x)
13       neighbors = [(-1,-1), (-1,0), (-1,1), (0,1), (1,1), (1,0),(1,-1), (0,-1) ]
14       # Conjunto de Movimientos Permitidos
15       # definido por orden: (y,x)
16       moveset = {"norte":(-1,0), "este":(0,1), "oeste":(0,-1), "sur":(1,0) }
17
18       def Grid():
19           image = np.zeros((12,12,3), np.uint8)
20           # Solidos
21           image[4][2] = white
22           image[5][2] = white
23           image[6][2] = white
24
25           image[4][3] = white
```

```
26          image[5][3] = white
27          image[6][3] = white
28
29          image[4][4] = white
30          image[4][5] = white
31
32          image[4][6] = white
33          image[5][6] = white
34          image[6][6] = white
35
36          image[4][7] = white
37          image[5][7] = white
38          image[6][7] = white
39          # Se invierte la imagen binaria
40          etval, imin = cv2.threshold(image, 1, 255, cv2.THRESH_BINARY_INV)
41
42          Set_Rand_Position(imin)
43
44          return imin
45
46      def Copy_Grid(image):
47          cim = image.copy()
48          (y,x)=Detect_Position(image)
49          cim[y][x] = white
50          return cim
51
52      # Colocar el Agente en una posición Aleatoria evitando los sólidos
53      def Set_Rand_Position(image):
54          F = True
```

```
55          while F:
56              (y, x) = (random.randint(0,image.shape[0]-1), random.randint(0,image.shape[1]-1))
57              if not (tuple(image[y][x])==black):
58                  image[y][x] = blue
59                  F = False
60
61      # Detecta la posición del agente
62      def Detect_Position(image):
63          for (yi,column) in enumerate(image):
64              for (xi,value) in enumerate(column):
65                  if tuple(value) == blue:
66                      (y,x) = (yi,xi)
67          return (y,x)
68
69      # Verifica los estados de los vecinos en la posición dada
70      def Neighborhood(position, image):
71          y, x = position
72          S = []
73          B = []
74          for (yi, xi) in neighbors:
75              if not Image_Range((yi+y, xi+x), image.shape):
76                  B.append(1)
77                  S.append(1)
78              else:
79                  B.append(0)
80                  if (tuple(image[yi+y][xi+x])==black):
81                      S.append(1)
82
83                  else:
```

```
84                  S.append(0)
85          return (S,B)
86
87      # Verifica si la posición se encuentra dentro de los limites de la imagen
88      def Image_Range(position, shape):
89          cy, cx = position
90          flag = np.logical_and([cx >= 0, cy >= 0], [cx < shape[1], cy < shape[0]])
91          return np.logical_and(flag[0], flag[1])
92
93      # Verifica si se llego a una esquina Cóncava
94      def Concava(B):
95          val = len(list(filter(lambda x: x > 0, B)))
96          return True if (val >= 5) else False
97
98      # Cambia la posición del agente
99      def Change_Position(position, move, image):
100         im = image.copy()
101         y, x = position
102         yi, xi = move
103         im[y][x] = white
104         im[yi+y][xi+x] = blue
105         return (im,(yi+y,xi+x))
106
107     # Compuertas Lógicas
108     def And(A,B):
109         return 1 if np.logical_and(A, B) else 0
110     def Or(A,B):
111         return 1 if np.logical_or(A, B) else 0
112     def Not(A):
```

```
113         return 1 if np.logical_not(A) else 0
114
115     # Unidad Lógica Umbralizada
116     def TLU(Xi, Yi):
117         # Aplicación de la regla de decisión de movimiento
118         val = And(Or(Xi[0],Xi[1]), Not(Or(Yi[0], Yi[1])))
119         # Umbral de 0.5
120         return True if val > 0.5 else False
121
122     def Animation(gif):
123         # Código para la creación de la animación en JS # Funciona
124         fig = plt.figure()
125         plt.axis('off')
126         # ims is a list of lists, each row is a list of artists to draw in the current frame; here we are
just
127         # animating one artist, the image, in each frame
128         ims = []
129         for i in range(len(gif)):
130             im = plt.imshow(gif[i], animated=True)
131             ims.append([im])
132         ani = animation.ArtistAnimation(fig, ims, interval=200, blit=True, repeat_delay=1500)
133         return HTML(ani.to_jshtml())
134
135     # Simulación
136     def Running():
137         im = Grid()
138         cim = Copy_Grid(im)
139         Run = True
140         (y,x)= Detect_Position(im)
```

```
141         visited = set()
142         gif = [im]
143         while Run:
144             # S = Estados de los vecinos S1...Sn
145             # B = Limites de la reticula
146             S, B = Neighborhood((y,x), cim)
147             # x1 = S2 + S3, x2 = S4 + S5, x3 = S6 + S7, x4 = S8 + S1
148             X = { "x1": S[1:3], "x2": S[3:5], "x3": S[5:7], "x4": [S[-1], S[0]]}
149             if Concava(B) or ((y,x) in visited):
150                 Run = False
151             elif TLU(X["x1"], X["x2"]):
152                 (nim, (cy,cx)) = Change_Position((y,x), moveset["este"], cim)
153             elif TLU(X["x2"], X["x3"]):
154                 (nim, (cy,cx)) = Change_Position((y,x), moveset["sur"], cim)
155             elif TLU(X["x3"], X["x4"]):
156                 (nim, (cy,cx)) = Change_Position((y,x), moveset["oeste"], cim)
157             elif TLU(X["x4"], X["x1"]):
158                 (nim, (cy,cx)) = Change_Position((y,x), moveset["norte"], cim)
159             else:
160                 (nim, (cy,cx)) = Change_Position((y,x), moveset["norte"], cim)
161             if Run:
162                 visited.add((y,x))
163                 (y,x) = (cy,cx)
164                 gif.append(nim)
165         return Animation(gif)
166
167     Running()
```

**Se anexará código en Jupyer para visualizar de mejor manera el código de implementación*