

PROYECTO 1
INTELIGENCIA ARTIFICIAL

IMPLEMENTACIÓN DE COSTO UNIFORME RECURSIVO

INTEGRANTES:
VALENTINA COBO BASTIDAS - 202060174
DAVID STEVEN ARCE FRANCO - 202067533
JUAN FELIPE JARAMILLO - 20260257

PROFESOR:
JOSHUA TRIANA

INSTITUCIÓN:
UNIVERSIDAD DEL VALLE SEDE TULUÁ

SEMESTRE:
SÉPTIMO

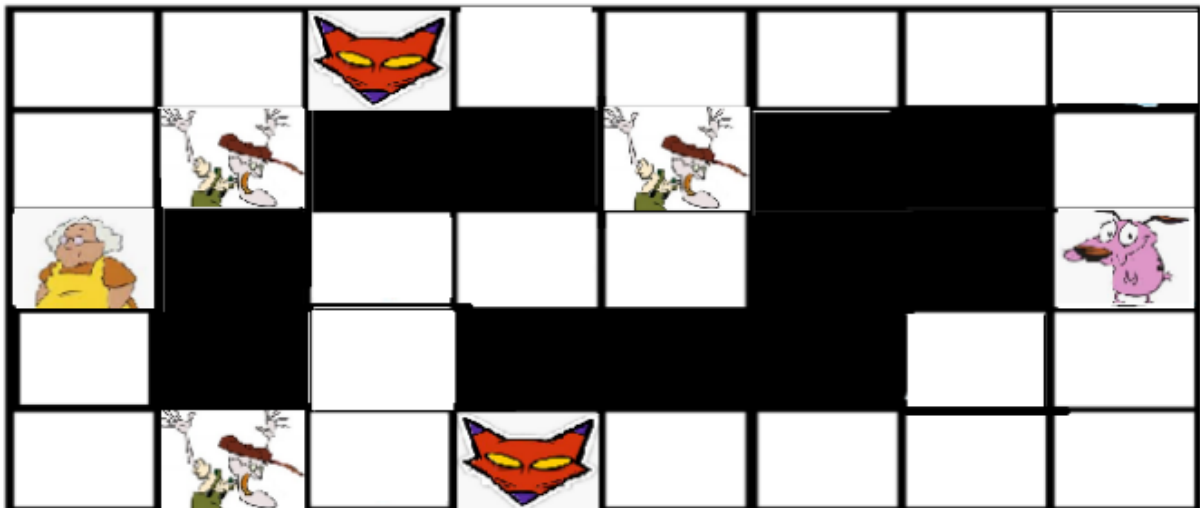
ANÁLISIS DEL PROBLEMA

La idea del problema es la siguiente:

- Se aplica el coste uniforme normal, van expandiendo el de menor costo
- Cuando usted está expandiendo una rama y esta ya tiene un costo mayor al otro nodo de otra rama, olvida esa rama.
- Olvidar la rama significa que los nodos hijos se borran de memoria, pero el mejor hijo (si hay 3 hijos, conserva el valor menor). Entonces el costo del nodo padre que queda será ese coste .
- La rama podrá ser re-expandida si su coste vuelve a ser el mejor

Los pasos anteriores son suficientes para que una persona entienda cómo desarrollar el procedimiento, pero al codificar el computador no entiende este tipo de lenguaje, ¿como sabe cuándo detenerse?, ¿como sabe como cambiar de nodo al menor?, ¿como sabe cual es el nodo menor?.

Además que este algoritmo se debe aplicar a un juego, en este caso vamos a utilizar el propuesto por el profesor:



- Coraje el perro cobarde quiere ir donde Muriel.
- Pasar por un espacio vacío o llegar donde Muriel cuesta 1.
- Pasar por su abusador amo hace que corra más rápido (irónicamente) y disminuye su costo acumulado en 2 unidades.
- Pasar por un gato malvado hace que se desmaye e incrementa el costo de pasar por esa casilla en 3 unidades.

Teniendo en cuenta esto, podemos ver el panel del juego como una matriz de costos como la siguiente:

```

1  matriz = [
2      [1, 1, 3, 1, 1, 1, 1, 1],
3      [1, -2, 0, 0, -2, 0, 0, 1],
4      [1, 0, 1, 1, 1, 0, 0, 1],
5      [1, 0, 1, 0, 0, 0, 1, 1],
6      [1, -2, 1, 3, 1, 1, 1, 1],
7  ]

```

- Espacio vacío o meta = 1
- Gato = 3
- Viejito = -2
- Pared = 0

ahora debemos de definir una estructura de datos que va a representar cada nodo de nuestro árbol:

```

class TreeNode:
    def __init__(self, cost, position, padre):
        self.cost = cost
        self.hijos = []
        self.position = position
        self.nodoPadre = padre
        self.raiz = False
        self.cambiado = False
        self.posicion_anterior_de_mi_padre = None
        self.completo = False
        self.matriz = matriz

```

- self.cost = hace referencia al costo acumulado
- self.hijos = al expandir un nodo crea sus hijos y los almacena en una lista.
- self.position = hace referencia a la posición donde se encuentra el nodo en la matriz ej: [i=2, j=7].
- self.nodoPadre = hace referencia al padre del nodo actual, esto nos sirve para recorrer el árbol.
- self.raiz = si el nodo en el que estamos es la raíz, esto nos servirá para hacer algunas validaciones.
- self.cambiado = es True cuando el nodo fue reemplazado por su mejor hijo.

- self.completo = es True cuando ya no se puede continuar por un nodo porque no tiene hijos o cuando el nodo de menor costo es mi hijo
- self.posicion_anterior_de_mi_padre = como su nombre lo dice almacena el indice donde antes estaba mi padre en los hijos de mi abuelo

MÉTODOS Y FUNCIONES:

1.

```
#comparamos la posicion de un nodo para saber si ya llegamos a la meta
def is_goal(self, meta):
    if self.position[0] == meta[0] and self.position[1] == meta[1]:
        return True
    return False
```

comparamos la posición de la meta y la del nodo, si son iguales es meta, de lo contrario no

2.

```
#verificamos si al expandir tengo hijos, si no tengo hijos ya estoy completo
def i_have_childrens(self):
    if not self.hijos:
        self.completo = True
        return False
    return True
```

como nodo.hijos es una lista podemos ver si está vacía para verificar si tengo hijos o no

3.

```
#obtienen los nodos y coordenadas de mi ruta hasta la raiz
def get_recorrido(self):
    if self is None:
        return nodos_recorridos, coordenadas
    nodos_recorridos = [self]
    coordenadas = [self.position]
    while not self.raiz:
        nodos_recorridos.append(self.nodoPadre)
        coordenadas.append(self.nodoPadre.position)
        self = self.nodoPadre
    coordenadas.reverse()
    return nodos_recorridos, coordenadas
```

obtengo mi recorrido accediendo a mi padre y al padre de mi padre, así sucesivamente hasta que me topo con la raíz, obtenemos los nodos visitados y las coordenadas de la matriz

4.

```
#pregunta si mi hijo es el menor para ver si seguir por ahi o cambiar de nodo
def mi_hijo_es_el_mejor(self, nodo):
    for hijo in self.hijos:
        if nodo == hijo:
            return True
    return False
```

verifico que mi hijo es el mejor comparando el nodo menor resultante y mis hijos

5.

```
#me retorna el numero de hijo que soy de mi padre
def indice_en_padre(self):
    if self.nodoPadre is not None:
        for i, hijo in enumerate(self.nodoPadre.hijos):
            if hijo is self:
                return i
    return None
```

obtenemos el número de hijo que soy de mi padre comparando los hijos de mi padre conmigo

6.

```
#cada nodo guarda su matriz para saber por donde no devolverse,
# en este caso donde haya un cero
def poner_en_cero_matriz(matriz, coordenadas_a_cero):
    nueva_matriz = [fila[:] for fila in matriz]

    for fila, columna in coordenadas_a_cero:
        if 0 <= fila < len(nueva_matriz) and 0 <= columna < len(nueva_matriz[0]):
            nueva_matriz[fila][columna] = 0

    return nueva_matriz
```

matriz hace referencia a la matriz original y coordenadas_a_cero hace referencia a mi recorrido, así procedemos a poner en cero los lugares por donde ya pase para no devolverme

7.

```
#cambio el padre por su mejor hijo
def acomodar_arbol(nodo):
    i = nodo.indice_en_padre()
    if i == None:
        return

    mejor_hijo = min(nodo.hijos, key=lambda hijo: hijo.cost)
    indice_mejor_hijo = nodo.hijos.index(mejor_hijo)

    nodo.nodoPadre.hijos[i] = nodo.hijos[indice_mejor_hijo]
    nodo.hijos[indice_mejor_hijo].cambiado = True
    nodo.hijos[indice_mejor_hijo].posicion_anterior_de_mi_padre = i
```

acomodar el árbol pone a mi mejor hijo en mi posición, para esto obtenemos la el index de mi mejor hijo y lo pongo en el índice donde iba yo entro de mi padre, además pongo el cambiado de mi hijo en True y guardo en mi hijo mi posición anterior dentro de mi padre para usarlo después al revertir

8.

```
#si ahora un nodo cambiado es el mejor lo revierto para continuar
def revertir(nodo):
    i = nodo.posicion_anterior_de_mi_padre
    nodo.nodoPadre.nodoPadre.hijos[i] = nodo.nodoPadre
    nodo.nodoPadre.completo = True
```

revertir usa la posicion_anterior_de_mi_padre para colocar a mi padre como hijo de mi abuelo y mi padre se convierte en un nodo completo porque ya su hijo es el mejor

9.

```
# Realiza un recorrido DFS para recoger todos los nodos en una lista
# y despues los acomoda por menor costo
def organizar_nodos_desde_nodo(nodo):
    def recorrido_dfs(nodo, nodos):
        if nodo is None:
            return
        nodos.append(nodo)
        for hijo in nodo.hijos:
            recorrido_dfs(hijo, nodos)

    nodos = []
    recorrido_dfs(nodo, nodos)
    nodos.sort(key=lambda nodo: nodo.cost) # Organiza la lista de nodos de menor a mayor
    return nodos
```

Realiza un recorrido DFS para recoger todos los nodos en una lista y después los acomoda por menor costo

10.

```
#retorna una lista con los valores de nodos_creados_ordenados
# sin los valores de mi_recorrido, ademas elimina los nodos que esten completos
def get_best_node(mi_recorrido, nodos_creados_ordenados):
    nueva_lista = [valor
                    for valor in nodos_creados_ordenados
                    if valor not in mi_recorrido and not valor.completo]
    return nueva_lista
```

retorna una lista ordenada de menor a mayor sin mi recorrido y los nodos completos para tomar del índice cero de esa lista y tomarlo como el siguiente nodo

11.

```
#verifica que la meta se encuentre dentro de la matriz
def coordenada_en_matriz(matriz, fila, columna):
    num_filas = len(matriz)
    num_columnas = len(matriz[0])

    if 0 <= fila < num_filas and 0 <= columna < num_columnas:
        return True
    else:
        return False
```

pasamos la matriz original y las coordenadas de la meta y verificamos que la meta este dentro del rango de la matriz

12.

```
1  #retorna los hijos de un nodo
2  def expandir(matriz, padre):
3      children = []
4      i = padre.position[0]
5      j = padre.position[1]
6      costo_acumulado = padre.cost
7      if i-1 >= 0:
8          if matriz[i-1][j] != 0:
9              nodo = TreeNode(matriz[i-1][j] + costo_acumulado, #costo acumulado
10                             [i-1,j], #posicion en la matriz
11                             padre) # nodo padre
12              children.append(nodo)
13      if j+1 < len(matriz[0]):
14          if matriz[i][j+1] != 0:
15              nodo = TreeNode(matriz[i][j+1] + costo_acumulado,
16                             [i,j+1],
17                             padre)
18              children.append(nodo)
19      if i+1 < len(matriz):
20          if matriz[i+1][j] != 0:
21              nodo = TreeNode(matriz[i+1][j] + costo_acumulado,
22                             [i+1,j],
23                             padre)
24              children.append(nodo)
25      if j-1 >= 0:
26          if matriz[i][j-1] != 0:
27              nodo = TreeNode(matriz[i][j-1] + costo_acumulado,
28                             [i,j-1],
29                             padre)
30              children.append(nodo)
31      matriz[i][j] = 0
32      return children
```

expandir crea los hijos de un nodo, se crean en orden a los movimientos que se puede realizar en la matriz, omitiendo los ceros que es donde no puede pasar

13.

```
1  # construir es una función recursiva que va creando el árbol y retorna las
2  # coordenadas de la matriz con menor costo
3  def construir(matriz, nodo, meta):
4      if nodo == None:
5          return "nodo vacío"
6
7      if not coordenada_en_matriz(matriz, meta[0], meta[1]):
8          return "la meta no existe dentro de la matriz"
9
10     if matriz[meta[0]][meta[1]] == 0:
11         return "nunca llegarás a la meta porque no puedes atravesar la pared"
12
13     if nodo.is_goal(meta):
14         #si llegamos a la meta retorno las coordenadas para recorrer la matriz
15         _, coordenadas = nodo.get_recorrido()
16         return nodo.cost, coordenadas
17
18     #si no es meta expando el nodo
19     _, coordenadas = nodo.get_recorrido()
20     nodo.matriz = poner_en_cero_matriz(matriz, coordenadas)
21     nodo.hijos = expandir(nodo.matriz, nodo)
22
23     #obtengo la información del nodo siguiente (el de menor costo)
24     nodos_ordenados = organizar_nodos_desde_nodo(raiz)
25     mi_recorrido, _ = nodo.get_recorrido()
26     nodo_siguiente = get_best_node(mi_recorrido, nodos_ordenados)
27
28     #si no tengo hijos es porque ya no hay recorrido entonces cambio de nodo
29     if not nodo.i_have_childrens():
30         if not nodo_siguiente:
31             return "es imposible llegar a la meta"
32         return construir(matriz, nodo_siguiente[0], meta)
33
34     #verifico si mi hijo es el mejor para así no hacer cambios al árbol
35     if nodo.mi_hijo_es_el_mejor(nodo_siguiente[0]):
36         nodo.completo = True
37         if nodo_siguiente[0].cambiado:
38             revertir(nodo_siguiente[0])
39         return construir(matriz, nodo_siguiente[0], meta)
40
41     #cambia la posición del padre por su mejor hijo
42     acomodar_arbol(nodo)
43
44     #si el nodo siguiente fue un nodo cambiado (osea reemplaze al
45     # padre por su mejor hijo) lo revierto
46     if nodo_siguiente[0].cambiado:
47         revertir(nodo_siguiente[0])
48
49     return construir(matriz, nodo_siguiente[0], meta)
```

esta es la función principal y sigue los siguientes pasos:

1. pregunta si el nodo es vacío

2. pregunta si la coordenada de la meta existe, si existe sigue
3. verifica si la meta es una pared, si no es sigue
4. pregunta si el nodo actual es meta, si es meta retorna su recorrido sino sigue
5. si no es meta, expande para esto debe:
 1. obtener su recorrido
 2. obtener su matriz con su recorrido en ceros
 3. expandir el nodo
6. buscar cual es el nodo con menor costo:
 1. obtener los nodos organizados desde la raíz
 2. obtener su recorrido
 3. eliminar su recorrido de los nodos organizados desde la raíz
 4. acceder al primer elemento de la resta de estas listas
7. pregunta si tuvo hijos, sino es un nodo terminado, además pregunta si no hay nodo siguiente, osea que ya no hay caminos por recorrer y se acaba el juego, si si hay caminos que recorrer recorro el siguiente
8. verifico si mi nodo es el mejor comparándolo con el nodo siguiente, si es asi mi nodo es completo, además verificar si el siguiente nodo es cambiado para revertirlo y poder continuar por ese camino
9. si mi hijo no es el mejor significa que me tengo que cambiar de nodo y acomodar mi mejor hijo en mi posición
10. verificar si el siguiente nodo es cambiado para revertirlo y continuó por ese camino hasta que llegó a la meta o si existe alguna complicación

```

1 # Tamaño de la matriz
2 filas = 5
3 columnas = 8
4
5 # Número a colocar en posiciones aleatorias
6 #muriel = 4, coraje = 5 (esto solo para pintar la interfaz)
7 numero = [3, -2, 0, 4, 5]
8
9 # Cantidad de veces que se debe colocar el número
10 cantGatos = 2
11 cantAmos = 3
12 cantObstaculos = 11
13 muriel = 1
14 coraje = 1
15
16 # Inicializar una matriz con unos que son los espacios en blanco
17 matriz = [[1] * columnas for _ in range(filas)]
18
19 # Llena la matriz con el número en posiciones aleatorias
20 matriz_aleatoria = matriz_random.llenar_matriz(matriz, numero, cantGatos, cantAmos, cantObstaculos, muriel, coraje)
21
22 #cambiar los numeros 4 y 5 por 1
23 for row in range(len(matriz_aleatoria)):
24     for col in range(len(matriz_aleatoria[0])):
25         if matriz_aleatoria[row][col] == 4:
26             finish = [row,col]
27             matriz_aleatoria[row][col] = 1
28         if matriz_aleatoria[row][col] == 5:
29             start = [row,col]
30             matriz_aleatoria[row][col] = 1

```

En esta parte del código se ingresan los datos para generar la matriz aleatoria

```

1 #cambiar los numeros 4 y 5 por 1
2 for row in range(len(matriz_aleatoria)):
3     for col in range(len(matriz_aleatoria[0])):
4         if matriz_aleatoria[row][col] == 4:
5             finish = [row,col]
6             matriz_aleatoria[row][col] = 1
7         if matriz_aleatoria[row][col] == 5:
8             start = [row,col]
9             matriz_aleatoria[row][col] = 1

```

Como el método para generar la matriz aleatoria genera datos con cuatro(4) y cinco(5) se necesita volver a cambiarlos por 1, esto para empezar a realizar la búsqueda por coste, solo se ponen esos números de 4 y 5 para la interfaz.

```

1  import random
2
3  def llenar_matriz(matriz, numero, cantGatos, cantAmos, cantObstaculos, muriel, coraje):
4      rows, cols = len(matriz), len(matriz[0])
5      posiciones = set() # Utilizamos un conjunto para evitar posiciones repetidas
6
7      while cantGatos > 0:
8          fila = random.randint(0, rows - 1)
9          columna = random.randint(0, cols - 1)
10
11         # Verifica si la posición ya ha sido ocupada
12         if (fila, columna) not in posiciones:
13             matriz[fila][columna] = numero[0]
14             posiciones.add((fila, columna))
15             cantGatos -= 1
16
17     while cantAmos > 0:
18         fila = random.randint(0, rows - 1)
19         columna = random.randint(0, cols - 1)
20
21         # Verifica si la posición ya ha sido ocupada
22         if (fila, columna) not in posiciones:
23             matriz[fila][columna] = numero[1]
24             posiciones.add((fila, columna))
25             cantAmos -= 1
26
27     while cantObstaculos > 0:
28         fila = random.randint(0, rows - 1)
29         columna = random.randint(0, cols - 1)
30
31         # Verifica si la posición ya ha sido ocupada
32         if (fila, columna) not in posiciones:
33             matriz[fila][columna] = numero[2]
34             posiciones.add((fila, columna))
35             cantObstaculos -= 1
36
37     while muriel > 0:
38         fila = random.randint(0, rows - 1)
39         columna = random.randint(0, cols - 1)
40
41         # Verifica si la posición ya ha sido ocupada
42         if (fila, columna) not in posiciones:
43             matriz[fila][columna] = numero[3]
44             posiciones.add((fila, columna))
45             muriel -= 1
46
47     while coraje > 0:
48         fila = random.randint(0, rows - 1)
49         columna = random.randint(0, cols - 1)
50
51         # Verifica si la posición ya ha sido ocupada
52         if (fila, columna) not in posiciones:
53             matriz[fila][columna] = numero[4]
54             posiciones.add((fila, columna))
55             coraje -= 1
56
57     return matriz

```

Este es el método para generar la matriz aleatoria donde recibe una matriz inicial, los números que se necesitan en la matriz y la cantidad de estos.

Los métodos para pintar la interfaz gráfica son los siguientes

16

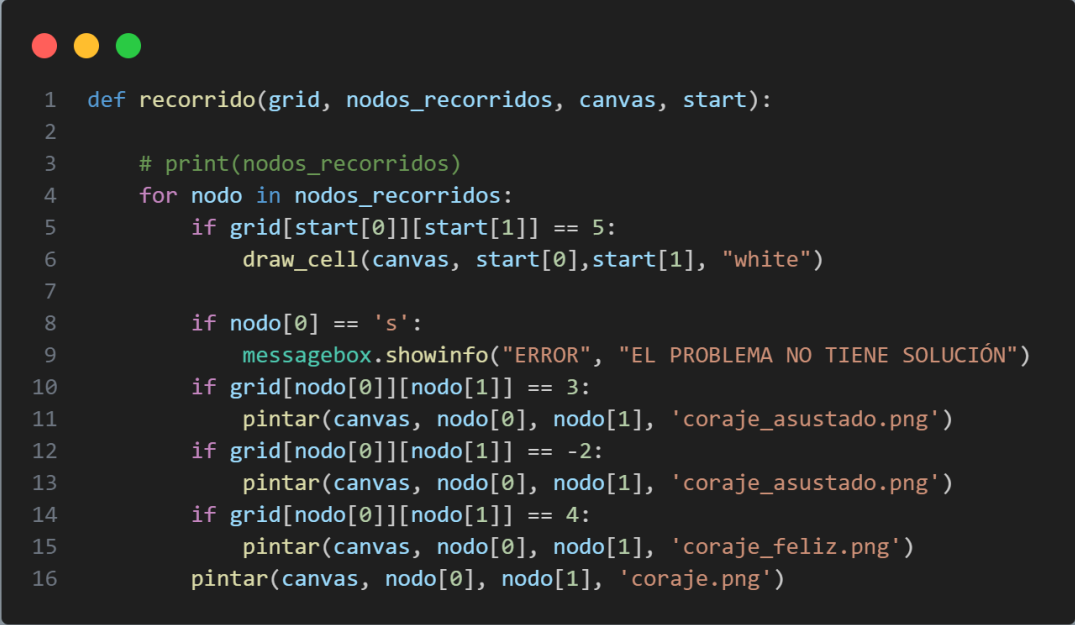
```
1 def pintar(canvas, x, y, image_path):
2     img = Image.open(image_path)
3     img = img.resize((cell_size, cell_size), Image.LANCZOS) # Redimensiona la imagen al tamaño de la celda
4     photo = ImageTk.PhotoImage(img)
5     canvas.create_image(y * cell_size, x * cell_size, anchor=tk.NW, image=photo)
6     canvas.photo = photo
7     time.sleep(0.1) # Añadir un pequeño retraso para visualización
8     canvas.update()
9
10 def draw_cell(canvas, x, y, color):
11     canvas.create_rectangle(y * cell_size, x * cell_size, (y + 1) * cell_size, (x + 1) * cell_size, fill=color)
```

Este método pintar dibuja las imágenes y el método draw_cell pinta las celdas

17

```
1 def start_visualization(grid, start, finish, nodos_recorridos):
2     root = tk.Tk()
3     root.title("Recorrido de nodos")
4
5     canvas = tk.Canvas(root, width=len(grid[0]) * cell_size, height=len(grid) * cell_size)
6     canvas.pack()
7
8     grid[finish[0]][finish[1]] = 4
9     grid[start[0]][start[1]] = 5
10    image_paths = {
11        3: 'gato.png',
12        -2: 'amo.png',
13        4: 'muriel.png',
14        5: 'coraje.png'
15    }
16    image_references = {}
17    for row in range(len(grid)):
18        for col in range(len(grid[0])):
19            if grid[row][col] == 0:
20                draw_cell(canvas, row, col, "black")
21            elif grid[row][col] == 1:
22                draw_cell(canvas, row, col, "white")
23            # elif grid[row][col] == 3:
24            #     draw_cell(canvas, row, col, "green")
25            elif grid[row][col] in image_paths:
26                img = Image.open(image_paths[grid[row][col]])
27                img = img.resize((cell_size, cell_size), Image.LANCZOS)
28                photo = ImageTk.PhotoImage(img)
29                image_references[(row, col)] = photo # Guarda la referencia de la imagen
30                canvas.create_image(col * cell_size, row * cell_size, anchor=tk.NW, image=photo)
31
32    # Crear un botón de inicio
33    start_button = tk.Button(root, text="Iniciar", command=partial(recorrido, grid, nodos_recorridos, canvas, start))
34    start_button.pack()
35
36    root.mainloop()
```

El método start_visualization es el que inicia la interfaz



```
1 def recorrido(grid, nodos_recorridos, canvas, start):
2
3     # print(nodos_recorridos)
4     for nodo in nodos_recorridos:
5         if grid[start[0]][start[1]] == 5:
6             draw_cell(canvas, start[0], start[1], "white")
7
8         if nodo[0] == 's':
9             messagebox.showinfo("ERROR", "EL PROBLEMA NO TIENE SOLUCIÓN")
10        if grid[nodo[0]][nodo[1]] == 3:
11            pintar(canvas, nodo[0], nodo[1], 'coraje_asustado.png')
12        if grid[nodo[0]][nodo[1]] == -2:
13            pintar(canvas, nodo[0], nodo[1], 'coraje_asustado.png')
14        if grid[nodo[0]][nodo[1]] == 4:
15            pintar(canvas, nodo[0], nodo[1], 'coraje_feliz.png')
16        pintar(canvas, nodo[0], nodo[1], 'coraje.png')
```

El método recorrido es el que pinta al agente recorriendo el laberinto