Universidad de San Carlos de Guatemala
Escuela de Ingeniería en Ciencias y Sistemas, Facultad de Ingeniería
Introducción a la programación y computación 2, 1er. Semestre 2022.

PROYECTO2: MISIONES DE RESCAT

202012039 – Angel Francisco Sique Santos

Resumen

La empresa Chapín Warriors, S. A. ha desarrollado equipos automatizados para rescatar civiles y extraer recursos de las ciudades que se encuentran inmersas en conflictos bélicos.

Con el fin de realizar las misiones de rescate y extracción, Chapín Warriors, S. A. ha construido drones autónomos e invisibles para los radares llamados ChapinEyes. Los ChapinEyes sobrevuelan las ciudades y construyen un mapa bidimensional de la misma, este mapa bidimensional consiste en una malla de celdas, donde cada celda es identificada como un camino, un punto de entrada, una unidad de defensa, una unidad civil, un recurso o una celda intransitable.

Palabras clave

Listas

Graphviz

Matriz

Programación orientada a objetos

Python

Abstract

The company Chapín Warriors, S.A. has developed automated equipment to rescue civilians and extract resources from cities that are immersed in armed conflicts.

In order to carry out rescue and extraction missions, Chapín Warriors, S.A. has built autonomous and radar-invisible drones called ChapinEyes. The ChapinEyes fly over the cities and build a two-dimensional map of it, this two-dimensional map consists of a grid of cells, where each cell is identified as a road, an entry point, a defense unit, a civil unit, a resource or an impassable cell.

Keywords

Lists

Graphviz

Matrix

Object-oriented programming

Python

Introducción

El tema principal de este proyecto es las estructuras de datos. La manera de implementación en este caso es de matrices dispersas, el cómo recorrerlas, buscar entre ellas y como ordenarlas fue lo que se necesito para cubrir las necesidades que se nos plantean en el problema de este proyecto.

Para mostrar gráficamente las ciudades y caminos recorridos se utilizó la herramienta Graphviz.

El tipo de archivos permitido para la lectura es XML, para leerlo se implementó ElementTree.

Desarrollo del tema

Las clases usadas fueron:

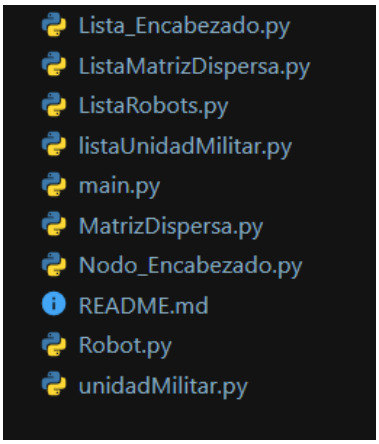


Figura 1. Clases.

Fuente: Elaboración propia

a. Lista_Encabezado.py:

Es una lista enlazada que es llenada con Nodos Encabezados que son objetos definidos en Nodo_Encabezados.py. Este se compone de anterior y siguiente además de nodos internos hacia los cuales están conectados, dependiendo del caso puede ser arriba o abajo.

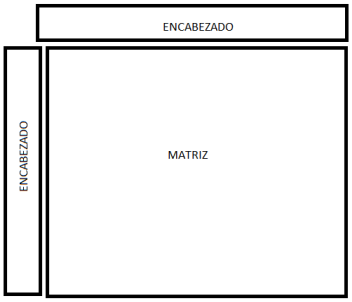


Figura 2. Ejemplo de encabezado.

Fuente: Elaboración propia

b. ListaMatrizDispersa.py:

Es una lista simple de matrices dispersas que representan ciudades. Las matrices dispersas se componen de varios Lista_Encabezado, que representan filas y columnas, las posiciones son Nodo_Interno que están dentro de las Lista_Encabezado, cada uno con su apuntador hacia arriba, abajo y derecha. Además de un atributo tipo que define su valor en la ciudad, el atributo tipo puede tomar el valor de “Entrada”, “Civil”, “UnidadMilitar’, “Recurso”, “Camino”, “Visitado”, “CaminandoCivil”, “VisitadoCivil”, “Vencido” y entre otras. Esto lo necesitamos definir para que sea más fácil moverse en la ciudad. Este método está implementado en la clase main.py.

En esta clase también está el método para graficar las ciudades. Para graficar como se nos solicita debemos definirlo en la parte donde se grafican los nodos internos.

Figura 3. Graficar.

Fuente: Elaboración propia

c. listaUnidadMilitar.py:

Es una lista simple donde se guardan las unidades militares. A las unidades militares se les define su posición y capacidad de pelea. Estas son representadas en la ciudad con un color rojo.

```
class unidadMilitar():
    def __init__(self,x,y,pelea,ciudad):
        self.ciudad = ciudad
        self.pos_x = x
        self.pos_y = y
        self.fuerza = pelea
        self.siguiente = None
```

Figura 3. unidadMilitar.py.

Fuente: Elaboración propia

d. ListaRobots.py:

Es una lista simple donde se guardan los robots. A los robots se les define su nombre, tipo y capacidad de pelea. Estos son usados para hacer las misiones de rescate y extracción de recursos.

```
class Robot():
    def __init__(self,nombre,pelea,tipo):
        self.tipo = tipo
        self.nombre = nombre
        self.fuerza = pelea
        self.siguiente = None
```

Figura 3. Robot.py.

Fuente: Elaboración propia

e. main.py:

Aquí es donde se crea la interfaz grafica y donde se encuentran los métodos para encontrar una ruta segura hacia el recurso o unidad civil seleccionada. Primero se crea una ventana que solicita el archivo. Luego de obtener el archivo se lee y analiza haciendo uso de elemntTree. Aquí mismo se realizan las validaciones para remplazar las ciudades y robots repetidos.

```
tree = ET.parse(ruta)
raiz = tree.getroot()
for r in raiz:
    for subchild in r:
        if subchild.tag == 'ciudad':
            unidad_militar = ListaUnidadMilitar()
            for subsubchild in subchild:
                if subsubchild.tag == 'nombre':
                    nombre = subsubchild.text
                    if lista_matriz.search_item(nombre) != False:
                        lista_matriz.eliminarCiudad(nombre)
                    f = open('Ciudades/'+subsubchild.text.replace(" ","") +'.txt','w')
                    print(subsubchild.text)
                if subsubchild.tag == 'fila':
                    f.write(subsubchild.text + "\n")
                if subsubchild.tag == 'unidadMilitar':
                    unidad_militar.insertLastunidadMilitar(subsubchild.attrib['fila'],subsubchild.attrib['columna'],subsubchild.text,nombre)
            f.close()
            lista_matriz.insertLastMatrizDispersa(0,unidad_militar,nombre)
            unidad_militar.showUnidadesMilitares()
        if subchild.tag == 'robot':
            for subsubchild in subchild:
                if subsubchild.tag == 'nombre':
                    if lista_robots.search_item(subsubchild.text,subsubchild.attrib['tipo']) != False:
                        lista_robots.eliminarRobot(subsubchild.text)
                    if 'capacidad' in subsubchild.attrib:
                        lista_robots.insertLastRobot(subsubchild.text,subsubchild.attrib['capacidad'],subsubchild.attrib['tipo'])
                    else:
                        lista_robots.insertLastRobot(subsubchild.text,'0',subsubchild.attrib['tipo'])
```

Figura 4. ElementTree

Fuente: Elaboración propia

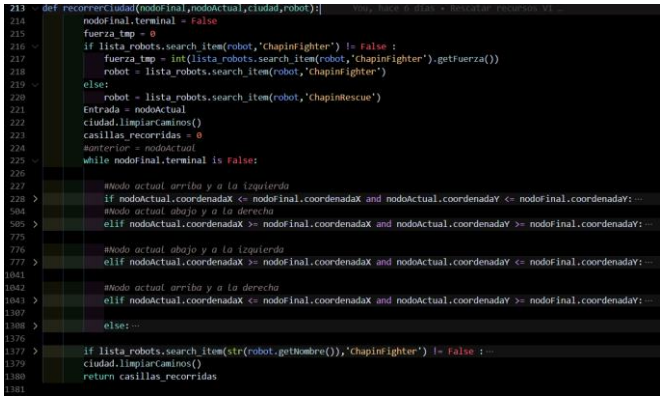
Luego de analizar el archivo se le pregunta al usuario que tipo de misión quiere hacer, luego se pide la ciudad, el robot y dependiendo del caso el recurso o civil que se quiere rescatar. Luego entra a un ciclo donde se revisan todas las entradas en la ciudad y se muestra la ciudad con la ruta que el programa decide que es la más optima.

```
def caminoCortoParaCivil(ciudad,robot):
    civiles = ciudad.showNodoCivil()
    if civiles == '':
        sg.popup_ok('La unidad civil se escogió automaticamente porque solo existe una \n ',title='Civil')
        nodoFinal = ciudad.getNodoPorTipo('Civil')
        print(civiles)
    else:
        nodoFinal = ciudad.getNodo(sg.popup_get_text('Fila del Civil \n' + civiles,'Fila'),sg.popup_get_text('Columna del Civil \n' + civiles,'Columna'))
        print(civiles)
    if nodoFinal != None and nodoFinal != False and nodoFinal.tipo == 'Civil':
        tmp = ciudad.filas.primero
        while(tmp is not None):
            nodoTmp = tmp.acceso
            while(nodoTmp is not None):
                if nodoTmp.tipo == 'Entrada':
                    distancia = recorrerCiudad(nodoFinal,nodoTmp,ciudad,robot)
                    if distancia != 0 and int(nodoFinal.getDistancia()) >  int(distancia):
                        nodoFinal.setDistancia(distancia)
                        nodoFinal.setEntrada(nodoTmp)
                        nodoFinal.terminal = False
                nodoTmp = nodoTmp.derecha
            tmp = tmp.siguiente
        if nodoFinal != False and nodoFinal.getEntrada() != None:
            recorrerCiudadFinal(nodoFinal,ciudad,robot)
            print('listo')
            webbrowser.open('matriz_'+str(ciudad.getCiudad()).replace(" ","")+'.pdf')
        else:
            sg.popup_error('No hay forma de llegar al civil :(')
    else:
        sg.popup_error('La posición indicada no tiene ningún civil')
```

Figura 5. Obtener nodoFinal.

Fuente: Elaboración propia

Para obtener la ruta más optima cada Entrada entra a una prueba donde se revisa qué tanto se le complica llegar al nodoFinal. En esta prueba se analiza la posición del nodoInicial y el final, para tratar de acercarse de la manera más corta posible y también se analiza si se puede llegar al nodoFinal, de no ser así nos imprimirá un error donde nos diga que no es posible llegar y si es posible procederá a graficar la ruta que considero más optima.



```
def recorrerCiudad(nodoFinal,nodoActual,ciudad,robot):
    nodoFinal.terminal = False
    fuerza_tmp = 0
    if lista_robots.search_item(robot,'ChapinFighter') != False :
        fuerza_tmp = int(lista_robots.search_item(robot,'ChapinFighter').getFuerza())
        robot = lista_robots.search_item(robot,'ChapinFighter')
    else:
        robot = lista_robots.search_item(robot,'ChapinRescue')
    Entrada = nodoActual
    ciudad.limpiarCaminos()
    casillas_recorridas = 0
    #anterior = nodoActual
    while nodoFinal.terminal is False:

        #Nodo actual arriba y a la izquierda
        if nodoActual.coordenadaX <= nodoFinal.coordenadaX and nodoActual.coordenadaY <= nodoFinal.coordenadaY:
        #Nodo actual abajo y a la derecha
        elif nodoActual.coordenadaX >= nodoFinal.coordenadaX and nodoActual.coordenadaY >= nodoFinal.coordenadaY:

        #Nodo actual abajo y a la izquierda
        elif nodoActual.coordenadaX >= nodoFinal.coordenadaX and nodoActual.coordenadaY <= nodoFinal.coordenadaY:

        #Nodo actual arriba y a la derecha
        elif nodoActual.coordenadaX <= nodoFinal.coordenadaX and nodoActual.coordenadaY >= nodoFinal.coordenadaY:

        else:

    if lista_robots.search_item(str(robot.getNombre()),'ChapinFighter') != False :
    ciudad.limpiarCaminos()
    return casillas_recorridas
```

Figura 6. Recorrer ciudad.

Fuente: Elaboración propia

Conclusiones

Las listas enlazadas simples y las matrices dispersas fueron lo más indispensable en este proyecto ya que fueron la base de todo, no se pudiera implementar de otra manera tan eficiente todo lo requerido y al usar listas simples se optimiza también el uso de espacio en memoria que si se usara otro tipo de lista que también podría funcionar, pero malgasta más memoria.

El uso de Graphviz refuerza conocimientos sobre enlazar nodos unos con otros, ya que es para esto que se utiliza esta herramienta y además que esto es indispensable en la programación orientada a objetos donde siempre se está enlazando objetos con otros.

El uso de Python para resolver este problema fue agradable ya que su modalidad de uso es un poco más simplificada que lenguajes como Java, aunque se podría resolver el mismo problema con otro lenguaje, se siente más simplificado trabajar con Python. Es buena forma de reforzar habilidades con este lenguaje y mejorar la lógica al resolver este tipo de problemas.

Referencias bibliográficas

Anónimo. (2019). *Listas simples enlazadas*. Obtenido de uc3m: http://www.it.uc3m.es/java/2011-12/units/pilas-colas/guides/2/guide_es_solution.html

graphviz.org. (10 de Agosto de 2021). *¿Qué es Graphviz?* Obtenido de Graphviz: https://graphviz.org/

programmerclick. (s.f.). *programmerclick*. Obtenido de https://programmerclick.com/article/62121131643/

python.org. (6 de Marzo de 2022). *ElementTree*. Obtenido de python.org: https://docs.python.org/es/3/library/xml.etree.elementtree.html

Apéndices

a. Diagrama de clases: