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.

AUTOMATIZACION DE EXTRACCION DE RECURSOS Y UNIDADES CIVILES CHAPIN WARRIORS S.A.

202006353 – Juan Josue Zuleta Beb

Resumen

Se realizo una minuciosa preparación en modelado de estructuras de datos para solucionar el problema planteado por la empresa Chapin Warriors S.A. con lo cual se logró el manejo de archivos con extensión XML en el lenguaje de programación Python con el uso de la librería MiniDom. Implementa programación orientada a objetos para el manejo de la memoria estática y se utilizan clases Nodo y Lista enlazada para el manejo de la memoria dinámica. Para el desarrollo del proyecto se utilizó estructuras tradicionales de ordenamientos de datos para generar matrices con los datos, se generó gráficos por medio de la herramienta Graphviz, presentando las distintas partes de la ciudad en forma de matriz, para que el usuario tenga una mejor idea de lo que son los patrones extraídos del archivo XML. El dominio total del proyecto se basó en el paradigma de programación orientado a objetos y la manipulación de estructuras de datos por medio de la implementación de tipo de datos abstractos TDAs. En conclusión, se puede decir que este proyecto es una introducción a las estructuras de datos y a la implementación de tipos de datos abstractos dentro del paradigma de programación orientada a objetos.

Palabras clave

- XML
- MiniDom
- Nodo
- Lista Enlazada
- TDAs (Tipo de Datos Abstractos)
- Graphviz

Abstract

A meticulous preparation in data structure modeling was carried out to solve the problema posed by th company Chapin Warriors S.A. with which the handling of files with XML extension in the Python programming language with the use of the MiniDom library is presented. It implements object-oriented programming for static memory management and Node and LinkedList classes are used for dynamic memory management. For the development of the project, traditional data ordering structures were used to generate matrices with the data, graphs were generated through the Graphviz tool, presenting the different parts of the city in the formo of a matrix, so that the user has a better idea of what they are. the patterns extracted from the XML file. The total domain of the project was based on the object-oriented programming paradigm and the manipulation of data structures through the implementation of abstract data types ADTs. In conclusion, it can be said that this project is an introduction to data structures and the

implementation of abstract data types within the object-oriented programming paradigm.

Keywords

- XML
- MiniDom
- Node
- Linked List
- ADTs (Abstract Data Type)
- Graphviz

Introducción

A continuación, se presentará la solución del problema del proyecto, para la cual se utilizó el lenguaje de programación Python y el paradigma de programación orientado objetos, y para el manejo de memoria se utilizó EDD (Estructura de datos) con implementación de tipos de datos abstractos TDAs.
Estas herramientas de programación permitieron manejar la memoria del archivo y poder manipularla en diferentes clases, sabiendo que ya que se usó el paradigma orientado a objetos esto facilitó el orden y la comunicación de datos entre clases y así evitar la perdida de datos enviados y recibidos entre las mismas. Para las distintas clases se utilizaron las importaciones nativas de Python las cuales nos permiten manejar las clases dentro del mismo archivo evitando así importaciones circulares y también falta de importaciones en el manejo de las distintas clases.

Desarrollo del tema

El problema de diseño de tropas, unidades militares, unidades civiles, recursos y puntos de acceso en distintos patrones consiste generalmente en un problema de determinar el tipo de patron obtenido y de esto generar el grafico de cada ciudad, tomando en cuenta aspectos clave como corredores y puntos sin acceso (intransitables), movilizarse dentro del mapa representa la mayor parte de la dificultad del proyecto, tomando en cuenta que las ciudades son distintas cada una de las demás y es imposible predecir el tipo de misión que el usuario deseara realizar y también los puntos de entrada y objetivos que seleccionara, esto nos desafía a determinar el alojamiento de datos de forma en que podamos recorrer los datos buscando los objetivos y descartando aquellas zonas de la cuidad que no son parte del camino que deseamos tomar. Algunas de las situaciones comunes que hemos observado cuando se resuelven instancias muy grandes de un problema de este tipo son: Fuerte requerimiento de tiempo y fuerte demanda de recursos de memoria. Iniciando por extraer los datos del archivo XML y almacenándolos en los objetos de las clases, esta tarea la realizamos por medio de la librería que implementa Python llamada MiniDom, la cual nos permite buscar fuentes de datos por medio de nombres de etiquetas, de esta manera separamos los nombres de las ciudades, sus atributos principales como sus dimensiones, es decir las filas y columnas de la matriz, y también los patrones obtenidos dentro de la etiqueta fila, también se extrajo las unidades militares propias de cada ciudad así como las características propias de cada unidad. De la misma manera se extrajeron los datos de los robots disponibles, estos son generales para cualquier ciudad y pueden seleccionarse desde dentro de cualquier entorno de ciudad por lo que estos se extrajeron como datos independientes a las ciudades. Una vez extraídos los datos y almacenados en las siguientes clases:

- Nodos Ciudad (Node_city)
- Lista Ciudad (Linked_list_city)
- Nodo Patron (Pattern)
- Lista Patrones (Patterns_List)
- Nodo Caja (Box)
- Lista Cajas (Boxes_List)
- Nodo Robot (Node_robot)
- Lista Robot (Linked_list_robot)
- Nodo Militar (Node_military)
- Lista Militar (Linked_list_military)

```
mydoc = minidom.parse(datas)
#Extraer Ciudades
citys = mydoc.getElementsByTagName('ciudad')
for city in citys:
    city_name = city.getElementsByTagName('nombre')
    rowses = city.getElementsByTagName('fila')
    military_unit = city.getElementsByTagName('unidadMilitar')

    for attribs in city_name:
        name = attribs.firstChild.data
        rows = attribs.attributes['filas'].value
        columns = attribs.attributes['columnas'].value
#Extraer Robots
robots = mydoc.getElementsByTagName('robot')
for robot in robots:
    robot_name = robot.getElementsByTagName('nombre')
    for attribs in robot_name:
        robots_names = attribs.firstChild.data
        type_robot = attribs.attributes['tipo'].value
        if type_robot == 'ChapinFighter':
            capacity = attribs.attributes['capacidad'].value
        elif type_robot == 'ChapinRescue':
            capacity = None
        linked_robots.add_to_end(robots_names,type_robot,capacity)
```

Figura 1. Extracción de datos.

Fuente: Elaboración propia.

La manipulación de estas se llevo a cabo por medio de un método llamado menú principal, por medio del cual accedimos a todas las listas y nodos y manejamos ampliamente y a voluntad, nuestros datos almacenados.

Para iniciar con la ejecución se debe ingresar los datos para lo cual se debe cargar un archivo de extensión XML que cumpla con la estructura inicialmente planteada siguiendo las mismas distribuciones de nombre y raíces, por consiguiente podremos observar por separado cada una de las ciudades con sus atributos separados y sus patrones listos para ser impresos, por lo cual realizamos un algoritmo tradicional para contar filas y columnas y generar posiciones para una matriz de m x n dimensiones y con la ayuda de la herramienta graphviz podemos iniciar a escribir un archivo txt por medio de la librería neato, generamos matrices graficas como esta:

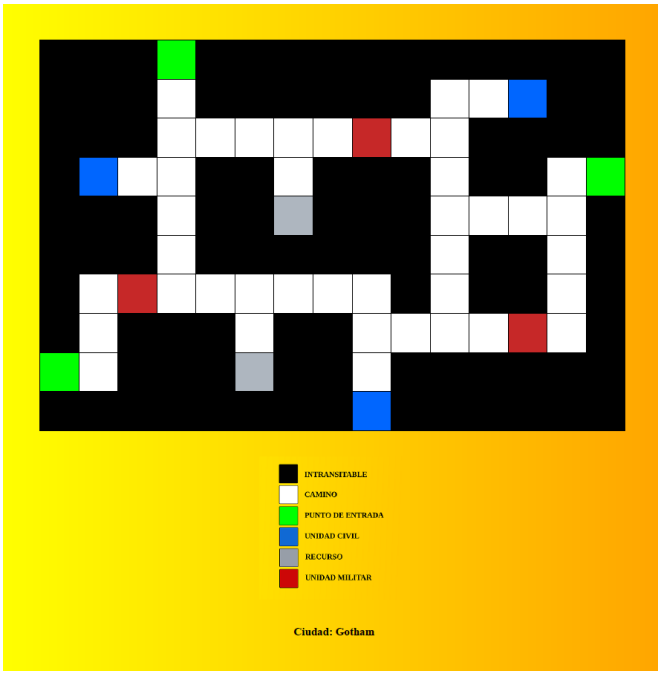

Figura 2. Impresión de Ciudad.

Fuente: Elaboración propia.

De forma mas especifica podemos realizar una inspección minuciosa a la generación grafica en forma de matriz quedando a luz alguna inquietudes como de donde salen las posiciones o como se almacenan los colores, esto se realiza una vez generado la lectura del archivo XML al extraer los patrones, antes de guardarlos en memoria se recorren los patrones y se asignan posiciones en coordenadas (X , Y) de esta forma: Si el carácter corresponde a un espacio “ “, entonces el color de ese nodo será Blanco se le asigna el código de color blanco y si el carácter corresponde a “ * ”, entonces el color de ese nodo será Negro y se le asigna una posición en memoria con dígitos del 1 en adelante, si el carácter corresponde a un “ C ”, entonces el color de ese nodo será Azul, si el carácter corresponde a una “ E ”, entonces el color de este nodo será Verde, si el carácter corresponde a una “ R ”, entonces el color de

ese nodo será Gris. Esto ayudara a graphviz a no confundir nodos del mismo color, para las posiciones comparamos si las columnas son mayores que las filas mientras esto se cumpla se podrá avanzar en las columnas con contadores inicializados en 0,0 para que las coordenadas estén en este orden, una luego se verifica que las filas sean mas grandes que las columnas de esta forma podremos insertar valores en las columnas y una vez que el valor de la columna actual sea menor que el valor de la columna de la matriz entonces se cambiara de fila para poder seguir llenando la matriz con contadores, el contador de columnas se regresa a cero cada vez que se cambia de fila y de esta forma se logra asignar coordenadas a los nodos de colores, una vez asignadas, se envían a la clase Boxes_list, donde se genera un constructor para añadir el nodo a la lista con el método .add_to end().

```
for color in pattern:
    count_rows = 0
    while int(rows) > count_rows:
        count_cols = 0
        if color == ' ':
            color = 'W{}'.format(str(contador))
            codes.color_patterns.add_to_end(color,pos_x,pos_y,Withe,Withe)
            contador += 1
            pos_x += 1
        elif color == '*':
            color = 'B{}'.format(str(contador))
            codes.color_patterns.add_to_end(color,pos_x,pos_y,Black,Black)
            contador += 1
            pos_x += 1
        elif color == 'E':
            color = 'E{}'.format(str(contador))
            codes.color_patterns.add_to_end(color,pos_x,pos_y,Green,Green)
            contador += 1
            pos_x += 1
        elif color == 'C':
            color = 'C{}'.format(str(contador))
            codes.color_patterns.add_to_end(color,pos_x,pos_y,Blue,Blue)
            contador += 1
            pos_x += 1
        elif color == 'R':
            color = 'R{}'.format(str(contador))
            codes.color_patterns.add_to_end(color,pos_x,pos_y,Gray,Gray)
            contador += 1
            pos_x += 1
        while int(columns) > count_cols:
            count_cols += 1
        count_rows += 1
        if pos_x >= int(columns):
            pos_y += 1
            pos_x = 0
```

Figura 3. conversión de XML a matriz de colores.

Fuente: Elaboración propia.

Una vez que tenemos cada nodo como un objeto de nuestra lista con cada uno de sus atributos definidos, podemos pasar a manipularlos en cualquier clase y en cualquier método, de esta premisa se sugiere que lo mejor sea manipularlos en la misma lista enlazada en donde fueron creados, por lo cual en la lista Boxes_list se crea el método de impresión que recibe por parámetros el nombre de la ciudad que se imprimirá y este recorre los nodos hasta el final de la lista y por cada uno se toma el color y sus posiciones en X y en Y para ubicarlos por medio de la herramienta graphviz con lo que nuestro método de impresión queda así:

```
def show_Boxes1(self, name):
    tmp = self.boxes_head

    strGrafica = 'digraph G { \n graph [pad="1" bgcolor="yellow:orange" style="filled" margin="0"  penwidth="3"] \n'
    strGrafica += 'label = "\n \n Ciudad: {}" fontname="times-bold" fontsize="20pt" \n'.format(name)
    strGrafica += 'node [style = filled shape = box height="1" width="1"] \n'

    while tmp != None:
        strGrafica += '{}[fillcolor= "{}" fontcolor = "{}" pos="{},-{}!"] \n'.format(tmp.get_Box(),tmp.getColor_Box(),tmp.getFont_Box(),tmp.getPosX_Box(),tmp.getPosY_Box())
        tmp = tmp.getNext_Box()

    tmp2 = self.boxes_last
    strGrafica += 'node2 [shape = "none", image="AuxFiles\\leyenda.png", label="" pos = "{},-{}!"] \n'.format(int(tmp2.getPosX_Box())/2,int(tmp2.getPosY_Box())+3)

    strGrafica += ' } '
    documentotxt = 'textoplano.txt'
    with open(documentotxt,'w') as grafica:
        grafica.write(strGrafica)
    pdf = '{}.pdf'.format(name)
    os.system( 'neato -Tpdf ' + documentotxt + ' -o ' + pdf )
    webbrowser.open(pdf)
```

Figura 4. Método de impresión en forma de matriz.

Fuente: Elaboración propia.

utilizando el método ‘os.system’, para realizar renderizado se escribe un archivo ‘*.txt, el cual luego es procesado por el software graphviz, generando en este caso un archivo pdf, es decir un ‘*.pdf, el cual luego por medio de la función ‘webbrowser.open(), es abierto por un programa instalado en el sistema, que tenga la capacidad de ejecutarlo.

De esta forma podemos manejar en general nuestros nodos de colores por lo cual de aquí en adelante queda la implementación de la programación orientada objetos por medio de la cual recorrimos cada nodo buscando sus atributos en el caso principal comparamos colores y posiciones de esta forma se facilitó la búsqueda en el algoritmo de búsqueda de caminos disponibles de esta forma implementamos el algoritmo de BackTraking con recursividad para

realizar una sola validación de condición exitosa y el resto es el caso general de misión fallida, este tipo de algoritmo tiene un caso base en el que nos encontramos en el punto objetivo entonces existe un camino, y como caso recursivo tenemos, cuando es una casilla no explorada esta la marcamos como parte del camino y exploramos desde ese punto, si al norte o al sur, o al este o al oste se llega al camino entonces se manda una respuesta verdadera al algoritmo, si en ninguna dirección se llega al destino entonces se desmarca el punto como parte del camino hasta encontrar uno nuevo, y si no se encuentra mas se manda una respuesta Falsa al algoritmo lo cual significa que no hay un camino posible según las condiciones de entorno dadas.

Algo importante que debemos resaltar es que siempre que se ejecuta un algoritmo recursivo debemos tomar en cuenta que el algoritmo no será preciso a al momento de comprender el tipo de condiciones especificadas dentro del mismo por lo cual siempre es necesario y conveniente que fuera de algoritmo se corrijan este tipo de errores que en este caso fueron de cambios de colores innecesarios que luego fueron resueltos con validaciones externas en las cuales se incluyo que como referencia se tomase el color de las fuentes de las cajas y no sus colores propios.

```python
def find_path_Rescue(x,y):

    global citys
    global codes
    global Civil
    global Black
    global Withe
    global Path
    global Gray
    global Red
    global Green

    try:
        nodo: Box = codes.color_patterns.get_Boxs_By_Pos(int(x),int(y))

        if x < 0 or y < 0 or nodo.getColor_Box() == Black \
            or nodo.getColor_Box() == Path or nodo.getColor_Box() == Gray or nodo.getColor_Box() == Red:
            return False

        if (nodo.getPosX_Box() == Civil.getPosX_Box()) and (nodo.getPosY_Box() == Civil.getPosY_Box()):
            return True

        nodo.setColor_Box(Path)

        if find_path_Rescue(x,y+1) or find_path_Rescue(x,y-1) or find_path_Rescue(x+1,y) or find_path_Rescue(x-1,y):
            return True

        nodo.setColor_Box(Withe)

        return False

    except:
        print('Cambiando Ruta')
```

Figura 5. Algoritmo de rescate de Civiles.

Fuente: Elaboración propia.

```python
def find_path_Extract(x,y):

    global citys
    global codes
    global Recurso
    global capacidad_final
    global Black
    global Withe
    global Path
    global Gray
    global Red
    global Green

    try:
        nodo: Box = codes.color_patterns.get_Boxs_By_Pos(int(x),int(y))

        if x < 0 or y < 0 or nodo.getColor_Box() == Black \
            or nodo.getColor_Box() == Path or nodo.getColor_Box() == Blue:
            return False

        if (nodo.getPosX_Box() == Recurso.getPosX_Box()) and (nodo.getPosY_Box() == Recurso.getPosY_Box()):
            return True

        nodo.setColor_Box(Path)

        if find_path_Extract(x,y+1) or find_path_Extract(x,y-1) or find_path_Extract(x+1,y) or find_path_Extract(x-1,y):
            return True

        nodo.setColor_Box(Withe)

        return False

    except:
        print('Cambiando Ruta')
```

Figura 5. Algoritmo de extracción de Recursos.

Fuente: Elaboración propia.

Conclusiones

Se puede concluir que la programación orientada a objetos es fundamental para el manejo y la implementación de estructuras de datos y también para la implementación de tipos de datos abstractos.

También se puede afirmar que las estructuras de datos son la base fundamental para poder construir cualquier tiempo de algoritmo que necesite almacenar datos y manejarlos a voluntad.

Se concluye que los tipos de datos abstractos como las listas enlazadas y las listas doblemente enlazadas y su manejo mediante nodos son la mejor forma de aprender a manejar realmente la programación orientada a objetos y todas sus características poco resaltadas al utilizar librerías nativas en los lenguajes de programación.

Referencias bibliográficas

- https://ebooksonline.es/leer-un-archivo-xml-de-ejemplo-minidom-elementtree/
- https://www.graphviz.org/documentation/
- https://morioh.com/p/9217f30c9d41

Anexos

Diagrama de clases: