Universidad de San Carlos de Guatemala
Escuela de Ingeniería en Ciencias y Sistemas, Facultad de Ingeniería
Introducción a la programación y computación 2, 2do. Semestre 2021.

PROYECTO #1 IPC2

202004810 – Henry Ronely Mendoza Aguilar

Resumen

Empleando la programación orientada a objetos se desarrollo un software que permite la optimización del uso de combustible para un robot durante la exploración de terrenos específicos.

Para el desarrollo de la aplicación se utilizaron Estructuras de datos abstractos para el manejo de la memoria dentro del programa los cuales fueron una lista doblemente enlazada y una matriz ortogonal. Asimismo, para la optimización del camino con menos consumo de gasolina se emplearon los fundamentos del algoritmo de Dijkstra el cual su concepto es el adecuado para la creación del software solicitado. A su vez, respecto a la generación de reportes se graficó utilizando la aplicación Graphviz. Dentro de Python se utilizó la librería ElementTree para la lectura y escritura de archivos de extensión xml.

Palabras clave
Algoritmo, Dijkstra, POO, software, programación

Abstract

Using object-oriented programming, software was developed that allows the optimization of fuel use for a robot during the exploration of specific terrain.

For the development of the application, abstract data structures were used for memory management within the program, which were a doubly linked list and an orthogonal matrix. Likewise, for the optimization of the road with less gasoline consumption, the fundamentals of the Dijkstra algorithm were used, which its concept is adequate for the creation of the requested software. In turn, regarding the generation of reports, it was graphed using the Graphviz application. Within Python, the ElementTree library was used to read and write files with an xml extension.

Keywords
Algorithm, Dijkstra, OOP, software, programming

Introducción

La optimización y ahorro de recursos es importante en cualquier ámbito científico o industrial, lo que implica que con el uso de los recursos justos se pueden obtener mayor cantidad de resultados.

Por tal motivo fue solicitado el desarrollo de un software que permita la gestión del combustible de un robot para la exploración de terrenos.

Desarrollo

Para el desarrollo de la aplicación solicitada por la Agencia Guatemalteca de Investigación Espacial para el nuevo robot de exploración r2e2, comencé realizando un diagrama sobre las clases que implementaría para el desarrollo del software. (*ver figura 1*).

Dentro de las clases para el almacenamiento de los datos de entrada a través del archivo xml, plantee dos clases, terreno y posición, dentro de la clase terreno en el software final se asignaron once atributos (*ver figura 2*) los cuales fueron obtenidos del archivo de entrada que son: nombre, dimensiones, posición de inicio, posición final, las posiciones que se les nombró nodos, las coordenadas de la ruta más corta y por ultimo la gasolina empleada en cada posición.

Para la clase planteada como posición se había asignados los atributos de posición en x, posición en y, el combustible. La cual en un principio había planteado para agregarlos en el apartado Nodos de la clase terreno, sin embargo, con los avances del proyecto y la implementación de las clases que se mencionara a continuación, se descartó la implementación de esta clase ya que no aportaba al desarrollo de la aplicación.
Teniendo claro la forma con la que se almacenaría los datos extraídos del archivo de entrada con extensión xml. A solicitud para el desarrollo de la aplicación se debían implementar Estructuras de Datos Abstractos o TDA'S para el manejo de la memoria del software, en este caso de forma dinámica y tener un mejor rendimiento durante la ejecución de este.

Durante el inicio del desarrollo de la aplicación se implementó una lista doblemente enlazada ya que la principal idea para el manejo de las posiciones del terreno era hacer una lista dentro de otra lista. No obstante, la implementación final de este TDA fue el manejo de los terrenos para almacenar cada objeto empleando el funcionamiento de la lista con buscadores, ya que una de las principales solicitudes para el desarrollo del programa fue que se pueda escoger el terreno que se desee en cualquier momento para las diferentes funciones que posee el sistema. Por lo que la lista enlazada doble funcionó de forma adecuada para el desarrollo de esta funcionalidad.

La lista enlazada dentro del programa únicamente posee los atributos del dato, el cual almacena los objetos a través de una clase nodo en el apuntador *first* y los apuntadores *next* y *preview*. Dentro de los métodos se obtiene el método *empty* el cual únicamente retorna si la lista está vacía o no. Y el método *append* el cual sirve para añadir los objetos de terreno a la lista enlazada. (*ver figura 3).*

Por otro lado, el manejo de las posiciones del terreno o nodos como se conocen para fines prácticos, no se conseguía un funcionamiento adecuado. Como experiencia personal con el desarrollo de otros proyectos con el lenguaje JAVA, recordé el uso de la estructura de matrices la cual tiene un funcionamiento similar al que poseen los nodos del terreno. Para la implementación de las matrices se desarrollo el TDA de matriz ortogonal o matriz dispersa. La cual se implemento por medio de la clase NodoM (*ver figura 4*) en la cual se asignan los valores de la posición del terreno, esta clase posee un método constructor el cual recibe como parámetros la posición en x, la posición en y y el valor de esa casilla en este caso el combustible. A pesar de ello en el parámetro de valor puede recibir datos de todo tipo como: string, int, boolean y objetos. Como podemos ver los parámetros que recibe la clase NodoM son los mismo que recibía la clase posición por tal razón se decidió el descarte de esa clase durante el desarrollo del software, sin

embargo, a la clase NodoM se le añaden los apuntadores, derecha, izquierda, arriba y abajo que son las posiciones en las cuales se puede recorrer la matriz y es el movimiento del robot.

Como complemento para la construcción de la Matriz se realizó la clase encabezado la cual es una lista doblemente enlazada que como su nombre lo dice funciona como las cabeceras de la matriz ortogonal y a su vez funcionan como el acceso a los datos que esta contiene.

En conjunto con las clases NodoM y encabezados, se implementó la clase Matriz (*ver figura 5*) la cual realiza todo el funcionamiento del TDA dentro del sistema. Esta consta de la función insertar el cual solicita como parámetros de entrada la posición en x, la posición en y, el valor a guardar el cual crea un objeto de tipo NodoM para asignar los valores. Asimismo, instancia dos objetos de tipo encabezado el cual funciona como el eje x y el eje y de la matriz, hablando matemáticamente. Para el llenado de la matriz se necesita que los datos estén ordenados ya que la posición (2,2) no puede estar antes de la (1,1) y viceversa, por lo que dentro de el insertado de datos se realiza un ordenamiento para que los valores siempre estén en la posición adecuada de forma ascendente. Otra función introducida dentro de la clase matriz es valores, el cual se encarga de mostrar en consola el valor de cada nodo que esta guardado dentro de la matriz, este consiste en un doble while que valúa cuando el acceso es nulo y de esta forma sale del ciclo obteniendo como resultado una impresión en consola como la que se visualiza en la figura 6. Una de las funciones que se solicitaron en el programa fue la generación de una gráfica como la figura 7, por medio del software Graphviz esta función fue añadida a la clase matriz ya que se necesitan de los datos de los nodos para generar la gráfica, la función consiste en la escritura de un archivo con extensión “dot” ya que este es el formato que maneja el software de graphviz en este se implemento el doble while que se utiliza para presentar los valores de la matriz para añadirlos a una cadena de texto con la estructura que maneja graphviz para le ejecución de los archivos. Por último, valiéndose de la os importando la función system se genera la imagen con formato png de la gráfica utilizando los comandos de consola que proporciona el software de graficación por otra parte de la misma librería os se utilizó la función startfile para ejecutar el archivo png y que se le muestre al usuario cuando la genere.

Ya que la inserción de datos en la matriz se basa en objetos al ingresar un valor nuevo en unas coordenadas existentes estas no se sobrescriben, sino que se crea un objeto nuevo con las mismas coordenadas, pero con el nuevo valor lo que hace que la estructura de la matriz se rompa y tenga posiciones repetidas. Para solucionar esta situación recurrí a la creación de un algoritmo de actualización de datos similar a los métodos set (*ver figura 8)*, esta función se añadió en la clase Matriz y consiste en buscador ya que recibe como parámetros los mismos que el método insertar los cuales son posición en x, posición en y, el valor que se añadirá en este caso que se actualizará. El buscador se encarga de encontrar las coordenadas repetidas y al encontrarlas extrae el valor del nodo el cual se actualiza al nuevo valor ingresado en la función.

Por parte en el main, se implementaron diferentes métodos los cuales en esencia son los solicitados para el funcionamiento principal de la aplicación y en donde se instancia las clases mencionadas anteriormente ya que por si solas no ejecutan un programa completo. La principal función del main es leer el archivo xml de entrada empleando los métodos “read_xml” esta función a través de la librería ElementTree parsea el documento para su manejo con las funciones de la misma librería así el manejo del archivo es más sencillo. La función retorna el archivo parseado y se ingresa a diferentes funciones para la creación de los espacios en memoria con los TDA creados. La función “nodos” recibe el archivo y una variable entera la cual es la posición del terreno a generar, esta función se encarga de buscar dentro del

xml las coordenadas y el valor del combustible del terreno seleccionado por medio de la función "findall" y de esta forma crear la matriz de nodos que contendrá cada terreno (*ver figura 9)*. Dentro de la función "crear terreno" nuevamente se envía el archivo con el cual por medio de la funcón findall de la librería elementTree busca todos los parámetros relacionados a la creación del objeto de tipo terreno y por medio de listas se añaden a los terrenos correspondientes. Concluyendo el método con la instancia de la clase dlinkedlist para añadir los objetos de tipo terreno como se solicitó para el desarrollo del software.

El objetivo primordial del software era realizar una función para que el robot encontrara la ruta que gastara menos combustible en los terrenos seleccionados. Esto se logró por medio del algoritmo de Dijkstra el cual es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. implementando la clase camino y vértice (*ver figura 10)* En la cual dentro de la clase camino se implementa diversas funciones para la construcción de un grafo basándose en el principio de las listas enlazadas se emplean los métodos agregar vértice y como el algoritmo lo propone los enlaces, en este caso llamados aristas. Por ultimo se crea la función "Dijkstra" como el algoritmo mencionado anteriormente. La implementación del algoritmo no fue sencilla ya que en cada archivo de entrada tenemos nodos y los enlaces son únicamente imaginarios dados por los apuntadores. Esto provoca que al querer implementar enlaces la matriz se reduzca en valores en (n-1) para las columnas y las filas, por lo que la suma de combustible no es el adecuado y como repercusión el camino no puede ser el correcto. Luego de varios planteamientos, propuse la solución creando enlaces sumando los valores de cada nodo, así si el nodo (1,1) es 5 y el nodo (1,2) es 3 el valor del enlace será 8. Por obvias razones al momento de encontrar la ruta la suma de combustible es mayor sin embargo restando los valores de cada nodo sin incluir el valor del primero y el último se obtiene el valor de combustible gastado correcto.

Dentro del main, se creo una función de "recorrido" la cual contiene la instancia de la clase camino, esta función recibe como parámetros el objeto de tipo terreno a analizar, el archivo para crear un auxiliar de coordenadas buscando nuevamente la posición en el archivo xml. Dichas coordenadas se añaden a una única lista de la forma (xy) quedando las coordenadas de la siguiente forma [11,12,13,21,22,23, …, xy] de esta forma se crean los vértices del grafo con la función vértices de la clase camino, para relacionar cada grafo se utiliza un doble while con la matriz del terreno este recorre cada posición por columnas y por filas para relacionar cada nodo y a su vez añadiéndole la suma para formar los enlaces. (*ver figura 11*).

A este momento el grafo esta creado sin embargo la ruta más corta aun no ha sido encontrada. Para ello a través del objeto g que es una instancia de la clase camino se utiliza la función Dijkstra ingresándole la posición de inicio que contiene el objeto de tipo terreno. Posteriormente para encontrar los nodos de la ruta se emplea el método camino que retorna una lista con las coordenadas de los nodos recorridos, esta lista es empleada junto a un diccionario con los datos de las coordenadas relacionados con su valor respecto de cada nodo. Por medio de la función get de los diccionarios fue posible obtener el valor, emplear esto hizo que la resta propuesta anteriormente ya no se realizara ya que la búsqueda con los diccionarios retornaba el valor real del recorrido y por ende el combustible empleado para recorrer esa ruta.

Este método concluye llamando una nueva función llamada "mostrar recorrido" en la que se encarga de mostrar la ruta obtenida, además durante este método se muestra en consola todos los valores que contiene el objeto de tipo terreno seleccionado, las cuales son la dimensión. Posición inicial, posición final, las coordenadas del recorrido, el combustible empleado, una impresión de los valores de los nodos y por último una matriz conformada por 0 y 1 en la cual el

1 representa el nodo donde el robot pasó para encontrar la ruta optima y el 0 donde no se realizó ningún recorrido, para lograr esta matriz se empleó el método actualizar de la clase matriz descrito anteriormente.

A forma de almacenar en forma de reporte y poder realizar el envío de datos al robot para que emplee el camino adecuado, se solicitó que el programa escribiera un archivo de salida con formato xml. Para desarrollar esta función dentro de la aplicación se creó la función "write_xml" (*ver figura 12)* el cual emplea la función que trae la librería ElementTree para la escritura de archivos xml. Con esto, al archivo se añadieron los parámetros del nombre del terreno, la dimensión, la posición de inicio, la posición final, el combustible empleado y la posición de la ruta con menor consumo. La librería de ElementTree no crea un archivo con una lectura agradable al usuario ya que al generar el documento coloca todos los parámetros en una sola línea por lo que dificulta la lectura, para un análisis de parte del usuario. Por ellos se creó la función estructura, el cual por medio de una lista añade los saltos de línea necesarios, además de añadir las tabulaciones para que el identado del archivo sea agradable para la lectura de parte del usuario final. (*ver figura 13*).

Para facilitar el acceso a las funciones de la aplicación se implementó un menú con seis funciones (*ver figura 14*) en las cuales se encuentran la carga del archivo, el procesado del archivo, la escritura del archivo de salida, los datos del autor de la aplicación, la generación de la gráfica del terreno y finalizar el programa.

Cargar Archivo: En esta opción del menú se utiliza la función read_xml junto a crear terreno y nodos, ya que dentro de este apartado del menú su función principal es almacenar los parámetros que contiene el archivo para el funcionamiento correcto del programa.

Procesado del Archivo: En esta opción del menú se solicita que se ingrese el nombre del terreno que se desea procesar y por medio del buscador envía el objeto de tipo terreno correcto a las funciones recorrido y a su vez esta función llama a mostrar recorrido como se explicó anteriormente, de esta forma se presenta en consola los datos referentes al terreno elegido por el nombre. Por otro lado se agregaron distintas verificaciones como detectar si el archivo ya había sido cargado, si el robot aun tiene combustible y si el terreno existe dentro del archivo.

Escritura de Archivo: Para esta función nuevamente se solicita el nombre del terreno al que se desea obtener los datos para crear el archivo. Implementando el buscador se llama a la función write_xml enviándole el objeto de tipo terreno correcto. Dentro de esta función se implementaron varias verificaciones como detectar si se cargó un archivo, detectar si el terreno ya fue procesado y si el terreno existe dentro del archivo.

Mostrar Datos: En esta función únicamente se muestra en consola los datos del creador de la aplicación, carnet, nombre del curso, carrera y el semestre que cursa.

Generar Gráfica: En esta función se solicita el nombre del terreno para generar por medio del software Graphviz la representación del terreno solicitada. Por medio del buscador se selecciona el objeto de tipo terreno adecuado y se llama a la función Graph de la clase matriz. En esta opción solo se implementó la verificación para detectar si el archivo ha sido cargado y si el terreno existe.

Conclusiones:

La gestión de recursos es importante durante la ejecución de proyectos por tal motivo es importante encontrar los algoritmos adecuados que permitan la optimización de recursos en este caso el combustible disponible por el robot.

La implementación de TDA dentro del desarrollo de un software, es importante ya que permite una gestión de memoria más eficiente y no se desperdician tanta memoria como seria empleando la gestión de tipo estática.

Emplear diagramas, lluvia de ideas, bocetos, en otras palabras, el uso UML, es importante para no perderse en el desarrollo del software ya que permite tener ideas claras lo que repercute en la reducción de los tiempos de desarrollo que implica la aplicación

Anexos:

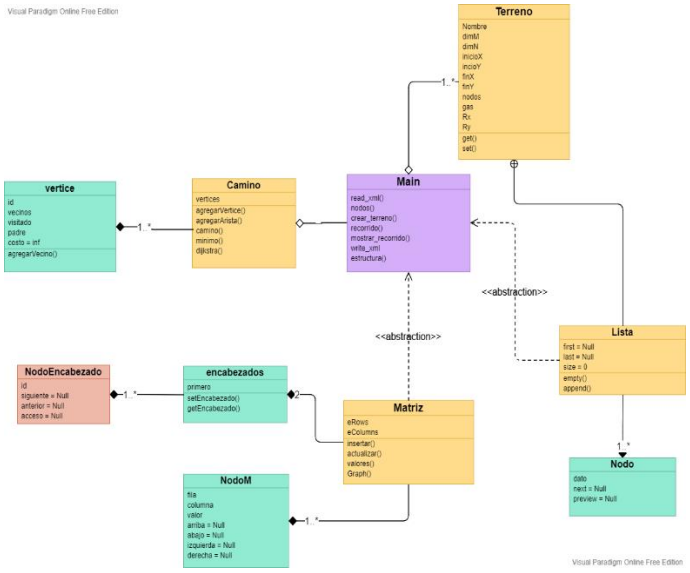

Figura 1. Diagrama de Clases

Fuente: Elaboración propia, 2021

```
class terreno:

    def __init__(self,nombre,dimM,dimN,iniciox,inicioy,finx,finy,nodos,gas,Rx,Ry):
        self.nombre = nombre
        self.dimM = dimM
        self.dimN = dimN
        self.iniciox = iniciox
        self.inicioy = inicioy
        self.finx = finx
        self.finy = finy
        self.nodos = nodos
        self.gas = gas
        self.Rx = Rx
        self.Ry = Ry
```

Figura 2. Clase Terreno
Fuente: Elaboración propia, 2021

```
from nodo import nodo

class dlinkedlist:

    def __init__(self):
        self.first=None
        self.last=None
        self.size=0

    def empty(self):
        return self.first == None

    def append(self,dato):
        if self.empty():
            self.first = self.last = nodo(dato)
        else:
            aux = self.last
            self.last = aux.next = nodo(dato)
            self.last.preview = aux
        self.size+=1
```

Figura 3. Lista doblemente Enlazada
Fuente: Elaboración propia, 2021

```
class NodoM:
    def __init__(self,fila,columna,valor):
        self.fila=fila
        self.columna=columna
        self.valor = valor
        self.derecha=None
        self.izquierda=None
        self.arriba=None
        self.abajo=None
```

Figura 4. Clase NodoM
Fuente: Elaboración propia, 2021

```
class Matriz:

    def __init__(self):
        self.eRows = ListaEncabezado()
        self.eColumns = ListaEncabezado()

    def insertar(self, fila, columna, valor):···

    def actualizar(self,fila,columna,valor):···

    def valores(self):···

    def Graph(self,name):···
```

Figura 5. Implementación de Matriz
Fuente: Elaboración propia, 2021

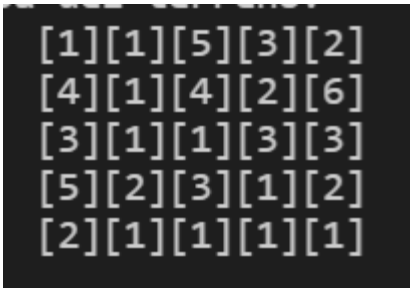

Figura 6. Función valores de la Matriz
Fuente: Elaboración propia, 2021

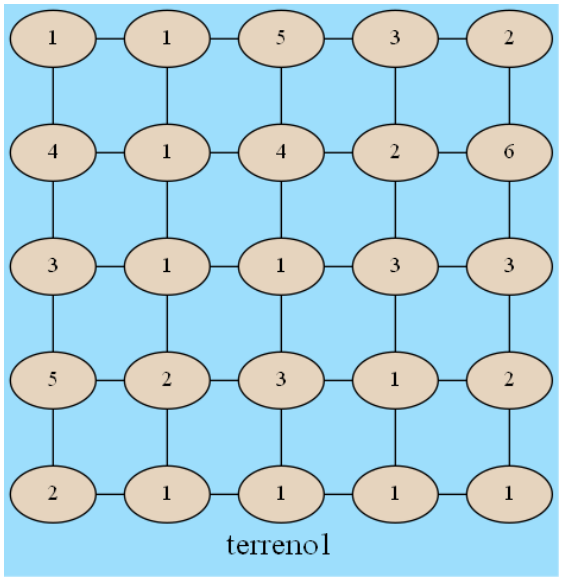

Figura 7. Gráfica usando Graphviz
Fuente: Elaboración propia, 2021

```
def actualizar(self,fila,columna,valor):
    aux = self.eRows.primero
    while aux != None:
        actual = aux.acceso
        while actual != None:
            if actual.fila == fila and actual.columna == columna:
                actual.valor = valor
            actual = actual.derecha
        aux = aux.siguiente
```

Figura 8. Función actualizar de la Matriz
Fuente: Elaboración propia, 2021

```
def nodos(file,n):
    Nodos=Matriz()

    for x in file[n].findall('posicion'):
        Nodos.insertar(x.attrib.get('x'),x.attrib.get('y'),int(x.text))

    return Nodos
```

Figura 9. Función Nodos del Main
Fuente: Elaboración propia, 2021

```
class Vertice:
    def __init__(self, i):...

    def agregarVecino(self, v, p):...

class Camino:
    def __init__(self):...

    def agregarVertice(self, id):...

    def agregarArista(self, a, b, p):...

    def camino(self, a, b):...

    def minimo(self, l):...

    def dijkstra(self, a):...
```

Figura 10. Clase camino y vértice
Fuente: Elaboración propia, 2021

```
for c in file[pos].findall('posicion'):
    coordenadas.append(c.attrib.get('x')+c.attrib.get('y'))
    valor.append(int(c.text))

ruta=dict(zip(coordenadas,valor))

for c in coordenadas:
    g.agregarVertice(c)

aux=dato.getNodos().eRows.primero
while aux != None:
    actual=aux.acceso
    while actual != None and actual.derecha != None:

        g.agregarArista(str(actual.fila+actual.columna),str(actual.derecha.fila+actual.derecha.columna),int(
        actual = actual.derecha

    aux=aux.siguiente
```

Figura 11. Método para crear los vértices y enlaces
Fuente: Elaboración propia, 2021

```
def write_xml(obj):
    gas=0
    for g in obj.getGas():
        gas+=g

    if gas == 0:
        return print('  >>AUN NO SE HA PROCESADO EL TERRENO<<')
    else:
        raiz=ET.Element('terreno',name=obj.getNombre())
        PI=ET.SubElement(raiz,'posicioninicio')
        ET.SubElement(PI,'x').text=str(obj.getIniciox())
        ET.SubElement(PI,'y').text=str(obj.getInicioy())
        PF=ET.SubElement(raiz,'posicionfin')
        ET.SubElement(PF,'x').text=str(obj.getFinx())
        ET.SubElement(PF,'y').text=str(obj.getFiny())

        combustible=ET.SubElement(raiz,'combustible')
        combustible.text = str(gas)

        for i in range(len(obj.getRX())):
            R = ET.SubElement(raiz,'posicion',x=str(obj.getRX()[i]),y=str(obj.getRY()[i]))
            R.text = str(obj.getGas()[i])

        estructura(raiz)
        doc = ET.ElementTree(raiz)
        guardar = input('  >>Ingrese una ruta para guardar el archivo: \n\t')

        try:
            doc.write(guardar)
```

Figura 12. Método para generar archivo xml
Fuente: Elaboración propia, 2021

```
def estructura(root, tab='  '):
    aux = [(0, root)]
    while aux:
        line, e = aux.pop(0)
        lista = [(line + 1, c) for c in list(e)]
        if lista:
            e.text = '\n' + tab * (line+1)
        if aux:
            e.tail = '\n' + tab * aux[0][0]
        else:
            e.tail = '\n' + tab * (line-1)
        aux[0:0] = lista
```

Figura 13. Método Estructura
Fuente: Elaboración propia, 2021

```
------------------MENÚ------------------
|    1. CARGAR ARCHIVO                 |
|    2. PROCESAR ARCHIVO               |
|    3. ESCRIBIR ARCHIVO SALIDA        |
|    4. DATOS DEL ESTUDIANTE           |
|    5. GENERAR GRAFICA                |
|    6. CERRAR PROGRAMA                |
Elija una Opción dentro del menú:
```

Figura 14. Menú de la Aplicación
Fuente: Elaboración propia, 2021