PROYECTO #1

20201005 – Derek Esquivel Díaz

Resumen

El presente proyecto es un programa de apoyo para la expedición del robot de exploración R2E2, con el fin de reducir el combustible usado por este robot, reduciendo así el costo de futuras expediciones.

Para realizar esto primero es necesario almacenar las características del terreno a recorrer en un archivo XML, de esta manera el programa podrá crear una matriz ortogonal del terreno, asignando a cada nodo la cantidad de combustible utilizada por R2E2 al atravesar esa área del terreno.

Una vez construida la matriz se ejecuta una versión modificada del algoritmo de Dijkstra, este algoritmo se encarga de analizar las formas en las que es posible llegar de un nodo a otro, obteniendo la forma más corta de llegar del punto inicial al final en la matriz.

Este algoritmo da como resultado una lista con todos los nodos que R2E2 debe recorrer para llegar a su destino usando la menor cantidad de combustible, dicho camino será mostrado en consola por medio de una sucesión de pasos y por una representación gráfica del terreno.

Palabras clave

Algoritmo de Dijkstra; Lista Enlazada; Python; Tipos de Datos Abstractos; Matriz Dispersa

Abstract

This project is a support program for the expedition of the exploration robot R2E2, in order to reduce the fuel used by this robot, thus reducing the cost of future expeditions.

To do this, first necessary to store the characteristics of the terrain to be traveled through in an XML file, this way is possible for the program to create an orthogonal matrix of the terrain, assigning to each node the amount of fuel used by R2E2 when crossing through that area of the terrain.

Once the matrix is built, a modified version of Dijkstra's algorithm is executed, this algorithm is in charge of analyzing the ways to get from one node to another in the matrix, thus being able to obtain the shortest way to get from the starting point to the end in the matrix.

This algorithm results in a list with all the nodes that R2E2 must travel to reach its destination using the least amount of fuel. This path will be shown on the console through a succession of steps and a graphical representation of the terrain.

Keywords

Dijkstra’s Algorithm, Linked List, Python, abstract data types; Sparse Matrix

Introducción

El presente proyecto es un programa de apoyo para la expedición del robot de exploración R2E2, con el fin de reducir el combustible usado por este robot, reduciendo así el costo de futuras expediciones.

Esto se realiza mediante el escaneo del terreno el cual transitara. Una vez con la información del terreno se divide en casillas de igual tamaño, asignándole un valor de cuanto combustible usaría R2E2 al recorrer dicha casilla.

En este programa se ha decidido usar el algoritmo de Dijkstra, que consiste en encontrar el camino más corto para llegar de un nodo a otro en un grafo, con lo que con algunas modificaciones fue posible aplicar dicho algoritmo para hallar el camino más corto a través del terreno a recorrer por R2E2.

Desarrollo del tema

La primera parte del programa consiste en leer un archivo XML con las características del terreno, este esta dividido en una cuadricula no mayor a 100 filas y 100 columnas, en la que cada casilla contiene las unidades de combustible usadas al transitar por dicha casilla.

El archivo XML consiste en las siguientes partes:

- **terrenos**: Esta es la etiqueta raíz que indica el inicio del documento.
- **terreno**: Esta etiqueta indica el inicio de un terreno nuevo, esta tendrá como atributos el nombre del terreno y la cantidad de filas y columnas del terreno.
    - **posicioninicio:** Esta etiqueta incluirá las coordenadas de donde iniciará el camino
    - **posicionfin**: Esta etiqueta incluirá las coordenadas de donde finalizará el camino

- **posición**: Esta etiqueta contendrá las casillas en las que estará dividido el terreno, además tendrá como valor la cantidad de unidades de combustible en esa casilla, sus atributos son:
    - **x:** Etiqueta con la coordenada X
    - **y:** etiqueta con la Coordenada Y

En un mismo archivo pueden estar incluidos mas de un terreno, por lo a la hora de leerlos cada uno se almacena en una lista, para poder usarlos cuando se necesite.

```
<terrenos>
     <terreno nombre="Matrix_0" n="3" m="3">
          <dimension>
               <n>3</n>
               <m>3</m>
          </dimension>
          <posicioninicio>
               <x>1</x>
               <y>1</y>
          </posicioninicio>
          <posicionfin>
               <x>9</x>
               <y>9</y>
          </posicionfin>
          <posicion x="1" y="1">2</posicion>
          <posicion x="1" y="2">10</posicion>
          <posicion x="1" y="3">3</posicion>
          <posicion x="2" y="1">4</posicion>
          <posicion x="2" y="2">9</posicion>
          <posicion x="2" y="3">12</posicion>
          <posicion x="3" y="1">1</posicion>
          <posicion x="3" y="2">5</posicion>
          <posicion x="3" y="3">20</posicion>
     </terreno>
</terrenos>
```

Figura 1: Estructura del archivo de entrada XML
Fuente: Elaboración propia (2021)

Una vez leído el archivo XML se utiliza la API "ElementTree" para leer las etiquetas y valores del archivo para así poder crear una matriz ortogonal con los valores ingresados.

En programación, una matriz dispersa es una tabla que implementa memoria dinámica que contiene dos cabeceras, una que representa columnas y una que representa Filas.

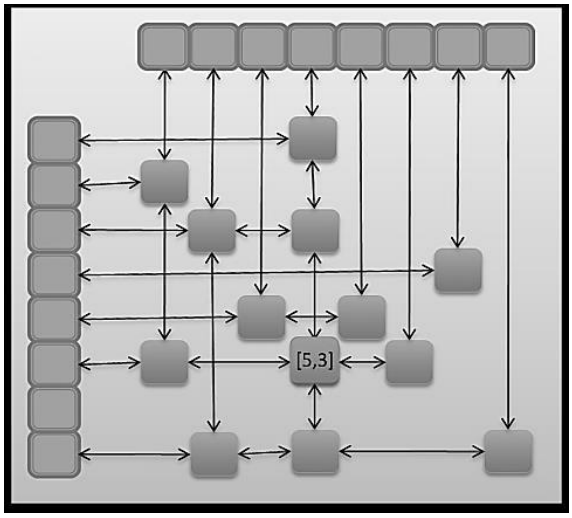

Figura 2: Representación de una matriz dispersa
Fuente: Angela León Mecías (2014)

El código utilizado para introducir los datos del archivo en una matriz dispersa es el siguiente, se guía de las etiquetas y atributos para determinar los valores de las filas y columnas para poder ubicar que valor y ubicación posee cada nodo.

```
    for elemento in root.findall("terreno"):
        if (nombre_terreno) == elemento.attrib["nombre"]:
            nombre = elemento.attrib["nombre"]
            lim_x =  elemento.attrib["n"]
            lim_y =  elemento.attrib["m"]
            m = matriz()
            for i in elemento.findall("posicioninicio"):
                inicio_x = i.find("x").text
                inicio_y = i.find("y").text

            for j in elemento.findall("posicionfin"):
                fin_x = j.find("x").text
                fin_y = j.find("y").text

            for subelemento in elemento.findall("posicion"):
                val_x = subelemento.attrib["x"]
                val_y = subelemento.attrib["y"]
                val = subelemento.text
```

Figura 3: Fragmento del código para leer el archivo XML
Fuente: Elaboración propia (2021)

Cada nodo del tendrá la siguiente estructura:

```
nodos.py > ...
class Nodo:
    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
        self.visitado = False
        self.valido = False

```

Figura 4: Estructura de un Nodo
Fuente: Elaboración propia (2021)

Los nodos para crear la matriz dispersa deberán tener esta estructura para funcionar. Primero tendrán un valor de fila y columna, estos dirán la posición actual del nodo, el valor es la cantidad de combustible que usa el nodo. Los valores derecha. Izquierda, arriba y abajo señalan al nodo adyacente en esa dirección y los valores visitado y valido serán utilizados posteriormente para determinar el camino más corto.
Cada nodo de la matriz representara una casilla en el terreno.

Luego será necesario insertar estos nodos en la matriz dispersa, la matriz dispersa se iniciará de la siguiente manera:

```
class matriz:

    def __init__(self):
        self.eFilas = listaEncabezado()
        self.eColumnas = listaEncabezado()

```

Figura 5: Inicializar Matriz
Fuente: Elaboración propia (2021)

Este método lo que ara será crear los encabezados de la matriz, estos nodos se utilizaran a la hora enlazar los nodos y definir las filas y columnas.

El método utilizado para insertar nodos a la matriz es:

```
10    def insertar(self,fila,columna,valor):
11        nuevo = Nodo(fila,columna,valor)
12
13        eFila = self.eFilas.getEncabezado(fila)
14        if eFila == None:
15            eFila = nodoEncabezado(fila)
16            eFila.accesoNodo = nuevo
17            self.eFilas.setEncabezado(eFila)
18        else:
19            if nuevo.columna < eFila.accesoNodo.columna:
20                nuevo.derecha = eFila.accesoNodo
21                eFila.accesoNodo.izquierda = nuevo
22                eFila.accesoNodo = nuevo
23            else:
24                actual = eFila.accesoNodo
25                while  actual.derecha != None:
26                    if nuevo.columna < actual.derecha.columna:
27                        nuevo.derecha = actual.derecha
28                        actual.derecha.izquierda = nuevo
29                        nuevo.izquierda = actual
30                        actual.derecha = nuevo
31                        break
32                    actual = actual.derecha
33                if actual.derecha == None:
34                    actual.derecha = nuevo
35                    nuevo.izquierda = actual
36
```

Figura 6: Inserción de nodo a la matriz
Fuente: Elaboración Propia (2021)

Este método lo que hará será buscar la posición en la que el nodo será insertado, una vez obtenida la posición procederá a enlazar dicho nodo con sus nodos adyacentes, este proceso se realiza primero en las filas y luego se repite el proceso para insertarlo en las columnas.

Utilizando una de las opciones de reporte es posible imprimir esta matriz como un grafo, utilizando la librería *Graphviz.*

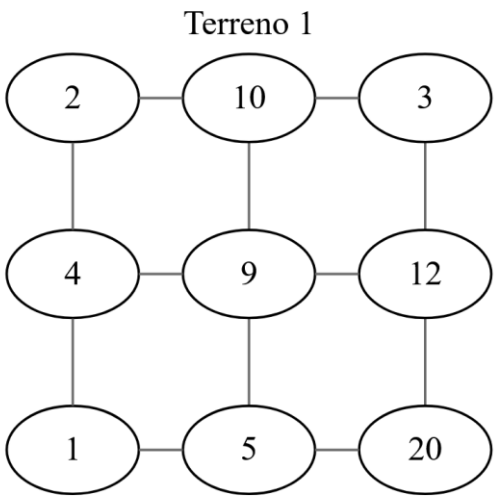

Figura 7: Representación de la matriz en Graphviz
Fuente: Elaboración propia (2021)

Una vez todos los datos han sido introducidos en una matriz ortogonal se procederá a buscar el camino mas corto utilizando el algoritmo de Dijkstra.

Este algoritmo tiene como finalidad determinar el camino mas corto de un nodo origen al resto de los nodos de un grafo donde la unión entre cada nodo (Arista) tendrá un valor (Peso).

Consiste en recorrer los nodos adyacentes al origen, tomando en cuenta los pesos de las aristas, para así poder encontrar la ruta mas corta entre un nodo a otro, este proceso se repite hasta alcanzar el nodo destino.

Si se tiene un grafo ponderad con nodos N, donde O es el nodo inicial y D un vector que guardara las distancias desde O, el algoritmo se ejecutaría de la siguiente manera:

1. Inicializar todas las distancias en D (exceptuando O) con infinito, debido a que estas distancias son desconocidas.
2. Creamos un nodo “A”, este será un nodo auxiliar para llevar control del nodo actual.
3. Se recorren todos los nodos adyacentes a “A”, siempre y cuando no hayan sido marcados como visitados.
4. Se calcula la distancia actual del nodo A desde el origen mas la distancia hacia un nodo adyacente. Si esta distancia es menor que la almacenada en el vector D se actualiza el vector con esta nueva distancia.
5. Una vez analizados todos los nodos adyacentes a “A” se marca “A” como visitado.
6. Se cambia el nodo actual al nodo con menor valor en el vector D, luego se regresa al paso 3 y se repite hasta que ya no haya nodos no visitados o hasta que se alcance el nodo destino.

7. Una vez terminado el algoritmo, el vector D contendrá la ruta mas corta a los nodos del grafo.

Este algoritmo encuentra la ruta mas corta a todos los nodos del grafo, pero para disminuir la complejidad del algoritmo y reducir el tiempo de ejecución se realizaron modificaciones. Por lo que en el Paso 6 se colocó una condición que detiene la ejecución del algoritmo una vez se haya encontrado la ruta al nodo destino.

Este algoritmo se pasó a código de la siguiente manera:

Primero se realizó una búsqueda de los nodos adyacentes al nodo actual, si es posible visitar el nodo adyacente (no es Null) este nodo se agregará a la lista de caminos disponibles.

```
        caminos_disponibles = LL.LinkedList()

        if nodo_actual.arriba != None:
            caminos_disponibles.insertar(nodo_actual.arriba)

        if nodo_actual.derecha != None:
            caminos_disponibles.insertar(nodo_actual.derecha)

        if nodo_actual.abajo != None:
            caminos_disponibles.insertar(nodo_actual.abajo)

        if nodo_actual.izquierda != None:
            caminos_disponibles.insertar(nodo_actual.izquierda)

```

Figura 8: Búsqueda de nodos adyacentes.
Fuente: Elaboración propia (2021)

Luego se procede a realizar el paso 4 del algoritmo, en la cual se comprueba la distancia a los nodos adyacentes y se agregan a la lista de las rutas todos aquellos nodos con una distancia tentativa menor.

```
        while aux is not None:
            peso = int(peso_a_nodo_actual) + int(aux.valor.valor)
            if lista_ruta.existe(aux.valor) == False:
                lista_ruta.insertar(aux.valor,nodo_actual,peso)
            else:
                peso_menor_actual = lista_ruta.buscarPeso(aux.valor)
                if int(peso_menor_actual) > int(peso):
                    lista_ruta.insertar(aux.valor,nodo_actual,peso)
            aux = aux.sig
```

Figura 9. Paso 4 del algoritmo de Dijkstra
Fuente: Elaboración propia (2021)

Una vez se ha actualizado la lista de rutas se filtrarán los nodos adyacentes por los que ya fueron visitados y lo que aun no han sido visitados, se eliminaran los visitados.

Posteriormente se marcará el nodo actual como ya visitado para no repetir el ciclo y se cambiará el nodo actual la ruta mas pequeña de la lista. Luego se repetirá el algoritmo hasta llegar al nodo destino.

Una vez completado el recorrido se procede a imprimir la ruta con el siguiente método:

```
    def imprimir(self, matriz):
        printval = self.head
        gasolina = 0
        print("-----------------------------------------")
        print("||       CALCULANDO LA MEJOR RUTA        ||")
        print("-----------------------------------------")
        time.sleep(0.7)
        while printval != None:
            print("X: " + str(printval.valor.columna) + "    " + "Y: " + str(printval.valor.fila) )
            gasolina += int(printval.valor.valor)
            printval = printval.sig
        print("")
        print("-----------------------------------------")
        print("||     CALCULANDO COMBUSTIBLE USADO      ||")
        print("-----------------------------------------")
        time.sleep(0.5)
        print("Gasolina usada: " + str(gasolina))
        print("")

        print("-----------------------------------------")
        print("||            CAMINO USADO               ||")
        print("-----------------------------------------")
        time.sleep(0.5)

        eFila = matriz.eFilas.primero
        if eFila == None:
            print("ERROR: Matriz Vacia")

        while eFila != None:
            actual = eFila.accesoNodo
            while actual != None:
                if actual.valido:
                    print(Fore.BLUE + "| 1 |" + Style.RESET_ALL,  end=" " )
                else:
                    print("| 0 |", end=" ")
                actual = actual.derecha
            eFila = eFila.siguiente
            print("")
```

Figura 10: Imprimir la ruta
Fuente: Elaboración propia (2021)

Este método imprimirá la ruta utilizada por el robot, el combustible usad y así una representación gráfica en consola del camino recorrido:

```
-----------------------------------------
||            CAMINO USADO             ||
-----------------------------------------
| 0 | | 1 | | 1 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 |
| 0 | | 0 | | 1 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 |
| 0 | | 0 | | 1 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 |
| 0 | | 0 | | 1 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 |
| 0 | | 0 | | 1 | | 1 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 |
| 0 | | 0 | | 0 | | 1 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 |
| 0 | | 0 | | 0 | | 1 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 |
| 0 | | 0 | | 0 | | 1 | | 1 | | 1 | | 1 | | 0 | | 0 | | 0 | | 0 |
| 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 0 | | 1 | | 0 | | 0 | | 0 |
```

Figura 11: Representación gráfica del camino
Fuente: Elaboración propia (2021)

Luego de haber generado la ruta más corta a través del terreno se puede generar un archivo de salida en formato xml con la ruta utilizada, el código para generar dicho archivo es el siguiente:

```
def reporte(self, nombre, inicio_x, inicio_y, final_x, final_y,filas, columnas, path):
    root = etree.Element('terreno')
    tree = etree.ElementTree(root)
    root.set('nombre',nombre)
    root.set('n',str(columnas))
    root.set('m',str(filas))

    posicionInicio = etree.Element('posicionInicio')
    root.append(posicionInicio)
    xInicio = etree.Element('x')
    xInicio.text = str(inicio_x)
    yInicio = etree.Element('y')
    yInicio.text = str(inicio_y)
    posicionInicio.append(xInicio)
    posicionInicio.append(yInicio)

    posicionFinal = etree.Element('posicionFinal')
    root.append(posicionFinal)
    xFinal = etree.Element('x')
    xFinal.text = str(final_x)
    yFinal = etree.Element('y')
    yFinal.text = str(final_y)
    posicionFinal.append(xFinal)
    posicionFinal.append(yFinal)

    combustible = etree.Element("combustible")
    root.append(combustible)
    aux = self.head
    gasolina = 0
    while aux != None:
        gasolina += int(aux.valor.valor)
        aux = aux.sig
    combustible.text = str(gasolina)

    aux = self.head

    while aux != None:
        posicion = etree.Element("posicion")
        posicion.set('x',str(aux.valor.columna))
        posicion.set('y',str(aux.valor.fila))
        posicion.text = str(aux.valor.valor)
        root.append(posicion)
        aux = aux.sig
```

Figura 12: Exportar ruta en XML
Fuente: Elaboración propia (2021)

Este código lo que hace es tomar los valores dentro de los nodos de la ruta usada para escribir la ruta usada, detallando las coordenadas del nodo y el combustible usado, para así saber que acciones tomo el robot. El archivo final sería como este:

```
<?xml version="1.0"?>
- <terreno m="5" n="10" nombre="Matrix_0">
  - <posicionInicio>
      <x>4</x>
      <y>0</y>
    </posicionInicio>
  - <posicionFinal>
      <x>5</x>
      <y>2</y>
    </posicionFinal>
    <combustible>26</combustible>
    <posicion y="2" x="5">0</posicion>
    <posicion y="2" x="4">6</posicion>
    <posicion y="1" x="4">11</posicion>
    <posicion y="0" x="4">9</posicion>
</terreno>
```

Figura 13: Archivo de salida
Fuente: Elaboración propia (2021)

Conclusiones

El uso de nodos y listas enlazadas en un problema de camino mas corto hace su resolución mucho mas fácil, debido a que un nodo puede almacenar mas de un dato, evitando el uso de mas variables, además como estos están enlazados unos con otros es posible generar una jerarquía que permite utilizarlos de mejor manera.

Es posible dar solución al problema del camino con la ayuda de algoritmos, en este caso el algoritmo de Dijkstra. Aunque este algoritmo no es mas adecuado en algunas circunstancias.

Un ejemplo de esto es si algún nodo fuera a tener un peso negativo, pues esto cambia el peso total de la ruta, pudiendo atrapar al algoritmo en un ciclo infinito.

Citas

- Dijkstra, E. W. (1959). «A note on two problems in connexion with graphs». Numerische Mathematik
- Miller,B y Ranum, D (2006) «Problem Solving with Algorithms and Data Structures Using Python», Decorah, Iowa, Estados Unidos de America: Luther College