Instituto Tecnológico de Costa Rica

Sede Central

Escuela de Ingeniería en Computación

IC-6200: Inteligencia Artificial

Proyecto 2: All Roads Lead to Bucharest

2015012856 - Nickolas Rodriguez Cordero

2017075170 - Andres Miranda Arias

2018102354 - Jorge Arturo Vasquez Rojas

14/07/2020

I semestre 2020

Prof. Jorge Arturo Vargas Calvo

Resumen

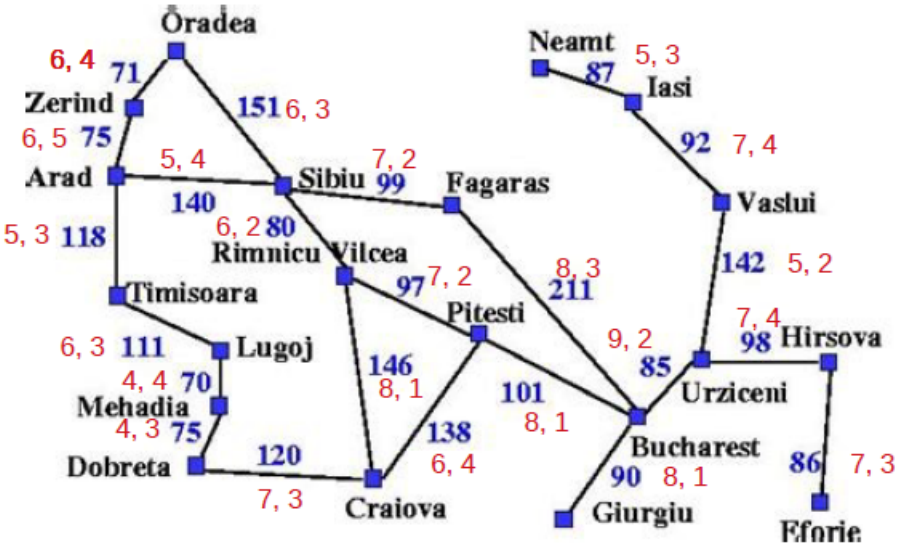
A Python implementation of an A* algorithm was made to be able to solve a graph problem, that searches for the minimum optimal route, according to three different parameters given. The main problem looks for a route from any city A, to a specific destination, Bucharest. The algorithm must be able to take in three parameters, road *state*, road *dangerousness*, and *distance*, to be able to deterime which route is best to take.

Índice

1. Introducción

El siguiente es un mapa de Rumania:

Hay carreteras que unen esas ciudades. Un esquema de los caminos existentes, con las distancias en kilómetros entre algunas de ellas (azul), el estado de la carretera y su nivel de peligrosidad (rojo) se muestran en la siguiente ilustración:



El estado de la carretera es un indicador entero de 1 a 10, donde 10 es un estado óptimo y 1 es un estado muy malo. La peligrosidad es un indicador entero de 1 a 5, donde 5 es muy peligrosa y

1 es totalmente segura.

El trabajo a realizar es escribir un programa, en el lenguaje de programación de elección propia, que, usando el algoritmo A*, encuentre la ruta óptima de cualquier ciudad del mapa a Bucarest, tomando en cuenta los tres componentes dados, y tomando en cuenta que el factor de peligrosidad es más importante que los otros dos factores.

2. Planteamiento del problema como resoluble mediante A*

El algoritmo A* es un algoritmo de búsqueda que se puede aplicar para el cálculo de rutas en un grafo. Este algoritmo be búsqueda es muy bueno al momento de determinar la mejor ruta desde A hacia B, debido a que este, mediante su función de evaluación $f(n) = g(n)+h'(n)$, donde $h'(n)$ representa el valor heurístico del nodo a evaluar desde el nodo actual, n, hasta el nodo final, y $g(n)$ representa el coste real del camino recorrido para llegar a dicho nodo n, desde el nodo inicial.

A* mantiene dos estructuras de datos auxiliares, que podemso denominar *abiertos*, implementado como una cola de prioridad, ordenada por el valor de $f(n)$ de cada nodo, y *cerrados*, donde se guarda la información de los nodos que ya han sido visitados. En cada paso del algoritmo, se expande el nodo que esté primero en abiertos, y en caso de que no sea un nodo objetivo, calcula el valor de $f(n)$ de todos sus hijos, los inserta en abiertos, y pasa el nodo evaluado a cerrados.

Para el caso de este proyecto, el grafo sobre el que se trabaja es una representación del país Rumania, en donde sus principales ciudades representaban un nodo y las conexiones entre ellas son las rutas, las cuales a su vez tienen tres atributos:

- Peligrosidad
- Estado de la Carretera
- Distancia

Con lo anterior claro, las funciones que se deben desarrollar para implementar el algoritmo A* deben tomar en cuenta estos tres factores, dando mayor importancia a la *peligrosidad*. El desarrollo de estas funciones se detalla a profundidad más adelante.

3. Definición de la función de costo entre nodos

Para la función de costo entre nodos, se decidió utilizar el Índice del Componente que se utiliza comúnmente para la realización del cálculo del Índice del Desarrollo Humano. La fórumla del mismo cálculo se muestra a continuación:

$$\text{Indice del componente} = \frac{\text{valor real} - \text{valor minimo}}{\text{valor maximo} - \text{valor minimo}}$$

A partir de esta función como base, se proceden a crear tres diferentes subíndices para cada componente del problema, y por último un índice general para integrar cada uno de los índices anteriores.

$$\text{Indice de distancia (ID)} = \frac{\text{distancia de la ruta} - 71}{211 - 71}$$

$$\text{Indice de peligrosidad (IP)} = \frac{\text{peligrosidad de la ruta} - 1}{5 - 1}$$

$$\text{Indice del estado de la carretera (IEC)} = 1 - \frac{\text{estado de la ruta} - 1}{10 - 1}$$

Con estos tres índices, utilizados para calcular los diferentes componentes de la ruta, se procede a definir el índice general que integra cada uno de los tres anteriores para dar un valor a utilizar en la toma de decisiones del algoritmo A*.

$$\text{Indice de Ruta (IR)} = \frac{1}{4}\text{ID} + \frac{1}{4}\text{IEC} + \frac{1}{2}\text{IP}$$

Como se puede apreciar, el Índice de Ruta le da más importancia al IP, esto para cumplir con la indicación del proyecto que estipula que se debe dar prioridad a la peligrosidad en comparación con los otros dos factores, de distancia y del estado de la carretera; el problema estipula que *"el factor de peligrosidad es más importante que los otros dos factores"*

De esta manera, se procede a definir la función g como:

$$g(d, p, ec) = \frac{1}{4}\text{ID}(d) \times \frac{1}{4}\text{IEC}(ec) \times \frac{1}{2}\text{IP}(p)$$

Según esta función anterior, se transforma cada índice a una función, tomando los parámetros d (distancia de la ruta), p (peligrosidad de la ruta) y ec (estado de la carretera) para proceder a realizar los cálculos respectivos.

4. Definición de función heurística

La función heurística para este problema va a recibir el nombre de IDLR, o Índice de Distancia en Línea Recta. Este se calcula únicamente con la fórmula del índice del componente mostrada anteriormente. Para realizar el cálculo, se utiliza el valor del DLR, o Distancia en Línea Recta, de cada ciudad con destino a Bucharest que se muestra en la tabla abajo. De ahí se toman los valores que nos dieron en el problema, y también se muestra el cálculo del IDLR para cada ciudad.

La fórmula utilizada para calcular la función heurística es la siguiente, siendo el valor x el DLR de la ciudad que se está evaluando,

$$h'(x) = \frac{x - 0}{380 - 0}$$

Datos del DLR e IDLR		
Ciudad	DLR	IDLR
Arad	366	0.96316
Bucarest	0	0.00000
Craiova	160	0.42105
Dobreta	242	0.63684
Eforie	161	0.42368
Fagaras	176	0.46316
Giorgiu	77	0.20263
Hirsova	151	0.39737
Lasi	266	0.70000
Lugoj	244	0.64211
Mehadia	241	0.63421
Neamt	234	0.61579
Oradea	380	1.00000
Pitesti	100	0.26316
Rimnicu Vilcea	193	0.50789
Sibiu	253	0.66579
Timisoara	329	0.86579
Urziceni	80	0.21053
Vaslui	199	0.52368
Zerind	374	0.98421

5. Justificación de la admisibilidad de la función heurística

La función heurística implementada se acepta como valida ya que cumple con distintas características. La primera, de suma importancia, es que una función heurística que se catalogue como admisible, no debe sobreestimar el costo para alcanzar el objetivo, es decir, el costo que estima dicha función para alcanzar el objetivo no debe ser mayor que el valor mínimo posible para el costo desde ese punto en el recorrido.

Con base en lo anterior, si la función $h'(x)$ es admisible, se concluye, por lo tanto, que la función $f(x)$ es, a su vez, también admisible. Esto permite que la función $f(x)$ sea monotónica, es decir, que el valor de $f(x)$ asociado al costo no decrece nunca.

Siguiendo esta lógica, si la función heurística $h'(x)$ es admisible, el árbol de búsqueda se expande, o ramifica, de manera finita, lo que garantiza que efectivamente existe una solución al problema, lo cual permite, a su vez, que el algoritmo A* encuentre la solución óptima al problema.

6. Código Fuente

6.1. GitHub

Todo el código fuente del proyecto se encuentra en la dirección: `https://github.com/andresm07/all-roads-lead-to-bucharest`

6.2. Estructuras de Datos

6.2.1. City

```
class City:

    def __init__(self, name):
        self.__name = name
        self.__near = {}

    def __iter__(self):
        return iter(self.__near.values())

    def add_neighbor(self, neighbor, data):
        self.__near[neighbor] = data

    def get_connections(self):
        return self.__near.keys()

    def get_name(self):
        return self.__name
    #Data: [Distance,Status,Danger]
    def get_data(self, neighbor):
        return self.__near[neighbor]

    def get_near(self):
        return self.__near

    def get_nearNames(self):
        listkeys = list(self.__near.keys())
        keysname = []
```

```
        for key in listkeys:
            name = key.get_name()
            keysname.append(name)
        return keysname
```

6.2.2. Country

```
from city import City

class Country:

    def __init__(self):
        self.__cities = {}
        self.__amount = 0

    def __iter__(self):
        return iter(self.__cities.values())

    def get_cities(self):
        return self.__cities

    def get_amount(self):
        return self.__amount

    def add_city(self, name):
        self.__amount += 1
        new_city = City(name)
        self.__cities[name] = new_city
        return new_city

    def get_city(self, n):
        if (n in self.__cities):
            return self.__cities[n]
        else:
            return None

    def add_route(self, frm, to, data):
        if (frm not in self.__cities):
            self.add_city(frm)
        if (to not in self.__cities):
            self.add_city(to)

        self.__cities[frm].add_neighbor(self.__cities[to], data)
        self.__cities[to].add_neighbor(self.__cities[frm], data)

    def get_routes(self):
        return self.__cities.keys()
```

6.2.3. Distance

```
class Distance:
    distances = {}

    @staticmethod
    def add_distance(cityName, distance):
```

```
        Distance.distances[cityName] = distance

    @staticmethod
    def get_distance(cityName):
        if (cityName in Distance.distances):
            return Distance.distances[cityName]
        else:
            return None

    @staticmethod
    def get_distances():
        return Distance.distances.keys()
    @staticmethod
    def get_distancesValues():
        return Distance.distances.values()
```

6.2.4. Node

```
from city import City
from distance import Distance

class Node:

  # F(n) = g(n) + h'(n)
  def __init__(self, city: City, previusnode=None):
    self.__previusnode = previusnode
    self.__g = 0
    self.__hprime = 0
    self.__f = 0
    self.__city=city

    if(previusnode != None):
      self.data = self.__city.get_data(self.__previusnode.getCity())
      self.__calculateG()
      self.__calculateHprime()
      self.__calculateF()
      #print(previusnode.getCityName(),self.__g,self.__hprime,self.__f)
  def __eq__(self, other):
    try:
      return self.getCityName == other.getCityName
    except:
      return False

  def __lt__(self, other):
    return self.getF() < other.getF()
  def __gt__(self, other):
    return self.getF() > other.getF()

  def __calculateG(self):

    actualG= (1/4)*self.__getID() + (1/4)*self.__getIEC() +(1/2)*self.
   __getIP()
    #(2*self.data[0]*( 1 +  self.__getIEC() + self.__getIP()*4))/6
    self.__g= actualG+ self.__previusnode.getG()

```

```
  def __getID(self):
    return (self.data[0]-71)/(211-71) #Distance

  def __getIP(self):
    return (self.data[2] - 1) / (5 - 1)  # Distance

  def __getIEC(self):
    return 1 - ((self.data[1] - 1) / (10 - 1))  # Distance

    self.data[1] #Status
    self.data[2] #Danger




  def __calculateHprime(self):
    minV = min(list(Distance.get_distancesValues()))
    maxV = max(list(Distance.get_distancesValues()))
    self.__hprime=(Distance.get_distance(self.getCityName())-minV)/(maxV-
     minV)

  def __calculateF(self):
    self.__f=self.__g+self.__hprime

  def isFirstNode(self):
    return self.__previusnode==None


  def getG(self):
    return self.__g

  def getPreviusNode(self):
    return self.__previusnode

  def getF(self):
    return self.__f

  def getCityName(self):
    return self.__city.get_name()

  def getCity(self):
    return self.__city

  def getConnections(self):
    return list(self.__city.get_connections())
```

6.2.5. ListOpenNodes

```
from node import Node
class ListOpenNodes:

  # F(n) = g(n) + h'(n)
  def __init__(self,node):
    self.__opennodes=[node]

```

```
8
9   def addNode(self,node):
10    try:
11      eqnode = self.__opennodes[self.__opennodes.index(node)]
12      if (eqnode > node):
13        self.__opennodes.remove(eqnode)
14        self.__opennodes.append(node)
15    except:
16      self.__opennodes.append(node)
17
18  def expandNode(self,node,closedNodes):
19    self.__opennodes.remove(node)
20    closedNodes.closeNode(node)
21    l = node.getConnections()
22    for n in l:
23      if(not closedNodes.exists(n)):
24        self.addNode(Node(n,node))
25
26  def getMinNode(self):
27    return min(self.__opennodes)
28  def getL(self):
29    return self.__opennodes
```

6.2.6. ListOpenNodes

```
from node import Node
class ListOpenNodes:

  # F(n) = g(n) + h'(n)
  def __init__(self,node):
    self.__opennodes=[node]


  def addNode(self,node):
    try:
      eqnode = self.__opennodes[self.__opennodes.index(node)]
      if (eqnode > node):
        self.__opennodes.remove(eqnode)
        self.__opennodes.append(node)
    except:
      self.__opennodes.append(node)

  def expandNode(self,node,closedNodes):
    self.__opennodes.remove(node)
    closedNodes.closeNode(node)
    l = node.getConnections()
    for n in l:
      if(not closedNodes.exists(n)):
        self.addNode(Node(n,node))

  def getMinNode(self):
    return min(self.__opennodes)
  def getL(self):
    return self.__opennodes
```

6.3. Algoritmo A*

```
from listClosedNodes import ListClosedNodes
from listOpenNodes import ListOpenNodes
from node import Node

class  A_StarAlgorithm:
  @staticmethod
  def calculateBestRouteFrom(city):
    firstNode=Node(city)
    closedNodes=ListClosedNodes()
    openNodes=ListOpenNodes(firstNode)
    resultList=[]
    resultDistance=0


    if(firstNode.getCityName() != 'Bucharest'):
      openNodes.expandNode(firstNode, closedNodes)
      while(openNodes!=[] and openNodes.getMinNode().getCityName()!='
   Bucharest'):
        openNodes.expandNode(openNodes.getMinNode(), closedNodes)
      if(openNodes==[]):
        return "Error"

      resultNode=openNodes.getMinNode()
      resultList = [resultNode.getCityName()]
      resultDistance = resultNode.getG()
      PreviousNode=resultNode.getPreviusNode()
      while(PreviousNode!=None):
        resultList =[PreviousNode.getCityName()] +resultList
        PreviousNode = PreviousNode.getPreviusNode()

      return str(round(resultDistance,4))+"\t"+str(resultList)
    return "0\t[Bucharest]"
```

6.4. Programa Principal

```
#!/usr/bin/env python3
import json
from pathlib import Path
import sys
from country import Country
from distance import Distance
from a_StarAlgorithm import A_StarAlgorithm


def main(command, data):
    base_path_graph = Path(__file__).parent
    file_path_graph = (base_path_graph / "./data/country.json")
    jsonFileGraph = open(file_path_graph)

    jsonCountry = json.loads(jsonFileGraph.read())

    country = Country()
```

```
18
19      for city in jsonCountry:
20          city_name = city["name"]
21          for route in city["routes"]:
22              country.add_route(city_name, route["to"], [route["data"]["
    distance"],route["data"]["status"],route["data"]["danger"]])
23
24
25      base_path_table = Path(__file__).parent
26      file_path_table = (base_path_table / "./data/distanceTable.json")
27      jsonFileTable = open(file_path_table)
28
29      jsonDistance = json.loads(jsonFileTable.read())
30
31      for distance in jsonDistance:
32          Distance.add_distance(distance["name"], distance["
    distanceToBucarest"])
33
34      if(command == "todasLasCiudades"):
35          print("Costo\tRuta")
36          for n in list(country.get_routes()):
37              print(A_StarAlgorithm.calculateBestRouteFrom(country.get_city(
    n)))
38          command =""
39
40      if(command == "ciudad"):
41          header= "Costo\tRuta\n"
42          if (len(data)==1):
43              try:
44                  r= A_StarAlgorithm.calculateBestRouteFrom(country.get_city
    (data[0]))
45                  print(header+r)
46              except:
47                  print("Error: ciudad "+data[0]+" no se encuentra en los
    datos")
48          else:
49              print("Error: Formato incorrecto")
50          command =""
51
52      if(command == "ciudades"):
53          header= "Costo\tRuta\n"
54          resultado = ""
55
56          if (len(data)>=1):
57              for n in data:
58                  try:
59                      resultado += A_StarAlgorithm.calculateBestRouteFrom(
    country.get_city(n)) + "\n"
60                  except:
61                      print("Error: ciudad "+n+" no se encuentra en los
    datos")
62              print(header + resultado)
63          else:
64              print("Error: Formato incorrecto")
```

```
        command =""
    if(command == "ayuda"):
        print("Consultar una ciudad: ./main ciudad <nombreCiudad> \
   nConsultar multiples ciudades: ./main ciudades <nombreCiudad1> <
   nombreCiudad2> <nombreCiudad3> ... \nConsultar todas las ciudades: ./
   main todasLasCiudades")
        print("Formato del resultado: Costo [Ciudad1, Ciudad2, ... ,
   Bucharest]")
        command =""
    if(command != ""):
        print("Comando desconocido")


def __main__():
    city = ""
    data = []
    try:
        city = sys.argv[1]
        data = sys.argv[2:len(sys.argv)]
        main(city,data)
    except:
        print("Error en el formato, si necesita ayuda intente ./main ayuda
   ")

__main__()
```

Referencias

[1] "Algoritmo A* — IDELab", Idelab.uva.es. [Online]. Available: http://idelab.uva.es/algoritmo. [Accessed: 14- Jul- 2020].

[2] P. Hart, N. Nilsson and B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, 4th ed. IEEE Transactions of Systems Science and Cybernetics, 1968.

[3] S. Russell and P. Norvig, Artificial intelligence: A Modern Approach, 3rd ed. Pearson, 1994.