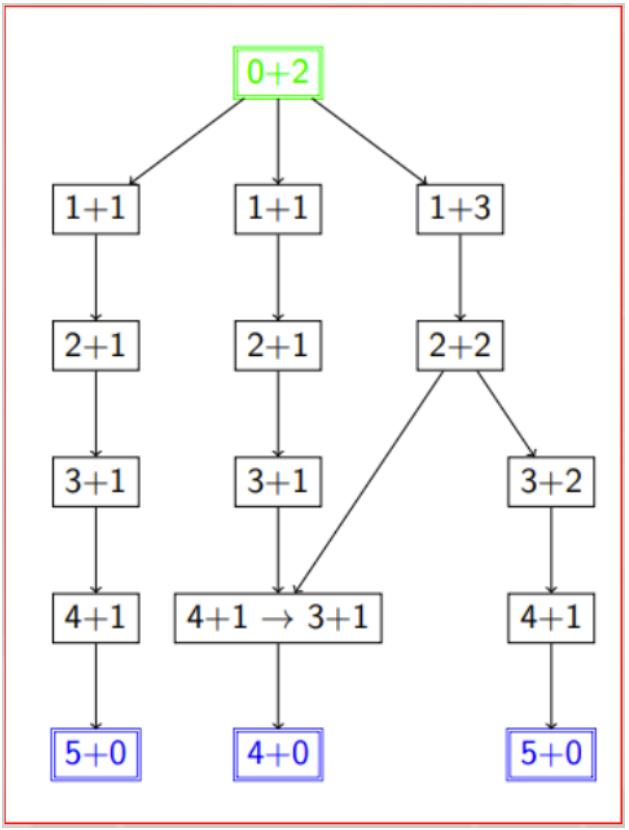
Algoritmo de búsqueda heurística

Elaborado por: Samuel Ibarra
Materia: Control Automático Avanzado
Profesor: Omar Aizpurúa



Enunciado

Con el algoritmo A*, haga un programa que encuentre la solución óptima (menor coste) del problema que se presenta.
El Siguiente recorrido presenta la salida (en verde) y la meta (en azul). Haga su programa utilizando el algoritmo señalado para encontrar el trayecto de menor coste (Óptimo). Presente sus dos estructuras para cada paso (Abiertos y cerrados) y presente además el recorrido final de menor coste. Utilice la plataforma de programación que usted prefiera.

Objetivos

Con el programa que realizaré, buscaré que desde el nodo raíz, podamos encontrar el nodo objetivo que tenga el menor costo. Para esto, voy a utilizar Python3 como lenguaje de programación y programación orientada a objetos como paradigma de programación. Con este programa, espero que se muestre un paso a paso del algoritmo con la lista de nodos en estado abierto y en estado cerrado. Ambas listas se actualizarán en cada iteración, y se mostrará el nodo analizado al momento como nodo actual. El programa luego de encontrar el nodo objetivo, mostrará el camino que ha sido utilizado y el coste total del camino.

Programa

El programa se desarrolló con dos tipos de clases:

- El primer objeto es de la clase Node (Nodo) y representa cada punto en el diagrama con sus respectivas propiedades (su valor h, valor g, sus hijos, etc.)
- El segundo objeto es el algoritmo A* con todo el algoritmo explicado en clase. Tiene como objetivo encontrar la ruta óptima.

Clase Nodo

Como entrada necesita un string con el valor del Nodo. El programa encuentra en este string los valores de g y h para calcular el coste f del nodo, además debe indicarse cual es su hijo. Durante el algoritmo el programa encuentra el nodo activo y cada vez que se necesite actualizar el nodo con un nuevo valor, con el método setNewLink se actualiza su coste f.

La función currentParentF solamente muestra un valor de coste F diferente al coste que se encuentra actualmente en la columna de abiertos.

```
class Node:
    def __init__(self, name:str, value:str, child):
        # propiedades de la clase Node
        self.name = name
        self.__char = value

        self.childs = []
        self.currentParentLink = 0


        # defino los hijos como una lista de nodos
        if type(child) is list:
            self.childs = child
        else:
            self.childs = [child]

        # tratar con el valor introducido, para ver si es de dos valores o uno solo
        if '->' in self.__char:
            vals = self.__char.split('->')
        else:
            vals = [self.__char]
        vals = [val[::2]  for val in vals]
        self.vals = [[int(g), int(h)] for g,h in vals]
        self.setNewLink()

    def __repr__(self):
        return f'({self.name}: {self.g}+{self.h})'

    def currentParentF(self):
        self.currentParentLink += 1
        return sum(self.vals[self.currentParentLink])

    def setNewLink(self):
        self.activeParentLink = self.currentParentLink
        self.g, self.h = self.vals[self.activeParentLink]
        self.f = sum(self.vals[self.activeParentLink])
        return 'Active Link: ' + str(self.activeParentLink)
```

Clase Algoritmo A*

Necesita como parámetros una lista con las raíces si son varias y metas si son varias. Luego, el convierte estas variables a listas y crea las variables opened y closed para que a estas se les añadan los nodos en cada estado.

```
class AStar:
    def __init__(self, root, goal):

        # convierto root y goal a listas
        if type(root) is list:
            self.roots = root
        else:
            self.roots = [root]
        if type(goal) is list:
            self.goals = goal
        else:
            self.goals = [goal]

        # Defino variables del Algoritmo
        self.opened = []
        self.closed = []
```

función runAlgorithm

Añade el nodo raíz a la lista de abiertos, lo pone como nodo actual de estudio y imprime el estado inicial. Luego empieza a buscar la ruta mientras el nodo estudiado no sea el nodo objetivo. Cuando el nodo actual sea el objetivo significa que habremos llegado al final de la búsqueda. Lo que básicamente realiza es una búsqueda de los hijos del nodo actual, para luego añadir estos hijos a la lista de abiertos, si ninguno es repetido, y luego ordena los nodos abiertos por coste, de menor a mayor. El primer nodo, es decir, el menor, es el que se selecciona para ser el siguiente nodo estudiado.

```
def runAlgorithm(self):
        print(diagram+'\n\n')
        self.opened.append(self.roots[0])
        self.currentNode = self.opened[0]
        self.showStep()

        while self.currentNode not in self.goals:
            self.opened.remove(self.currentNode)
            self.closed.append(self.currentNode)
            childs = self.currentNode.childs
            self.manageChilds(childs)
```

```
            self.opened = sorted(self.opened+self.currentNode.childs, key=lambda node:
node.h)
            self.opened = sorted(self.opened, key=lambda node: node.f)

            self.showStep()
            self.currentNode = self.opened[0]
        else:
            self.opened.remove(self.currentNode)
            self.closed.append(self.currentNode)
            self.showStep()
```

Función ManageChilds

Esta función utiliza a los hijos del nodo actual para comparar el nuevo coste con el coste anterior del nodo. Si existe un coste menor, se elimina el nodo anterior para así poder añadir el nuevo nodo con el nuevo coste. Si no es así, se dejará el nodo con el coste anterior. De la misma forma se estudia repetidos en la lista de cerrados.

```
    def manageChilds(self, childs):
        for child in childs:
            if child in self.opened:
                if child.currentParentF() < child.f:
                    self.opened.remove(child)
                    child.setNewLink()
            if child in self.closed:
                if child.currentParentF() < child.f:
                    self.closed.remove(child)
                    child.setNewLink()
```

Función showStep

Solamente funciona para mostrar una representación del estado actual del algoritmo en el programa.

```
    def showStep(self):
        print('Actual: '+str(self.currentNode))
        print('Abiertos: '+str(self.opened))
        print('Cerrados: '+str(self.closed) + '\n\n')
```

Función showPath

Se utiliza para recorrer los nodos desde el objetivo hasta la raíz, utilizando el camino óptimo que fue encontrado por el programa. Cada vez que esta en un nodo encuentra los posibles padres que puede tener este nodo, y luego utiliza el padre que ha quedado activo según el último algoritmo. Este padre es el utilizado en la nueva iteración hasta llegar al nodo raíz. Es entonces que la lista de todos los puntos visitados se invierte y se muestra en el programa. Al final se muestra el coste del camino elegido.

```
    def showPath(self):
        self.path = []
        self.totalCost = 0

        while self.currentNode not in self.roots:
            self.totalCost += self.currentNode.f
            self.path.append(self.currentNode)

            # nodos que podrian ser el padre
            possibleNode = []

            for node in self.closed:
                if self.currentNode in node.childs:
                    possibleNode.append(node)

            # de los posibles padres, elijo el que quedo como link activo en la
busqueda
            self.currentNode = possibleNode[self.currentNode.activeParentLink]

        # una vez llegado al nodo raiz
        self.totalCost += self.currentNode.f
        self.path.append(self.currentNode)
        self.path.reverse()
        print('CAMINO: '+ str(self.path))
        print('COSTE TOTAL: '+ str(self.totalCost))
```

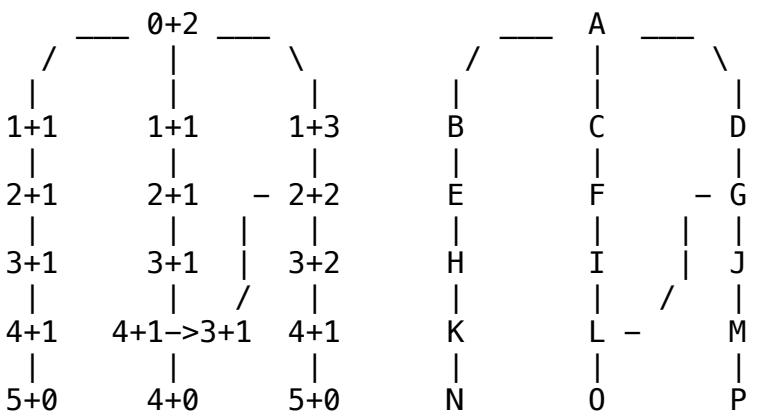
Variables declaradas e inicio del algoritmo

Se declaran los nodos a utilizar, y se muestra una pequeña representación del árbol a utilizar. Luego se crea un objeto llamado algoritmo y se corre el algoritmo, al final se muestra el camino seguido.

```
diagram = """
Diagrama del Problema:

    ___ 0+2 ___              ___  A  ___
  /      |      \\          /      |      \\
 |       |       |        |       |       |
1+1     1+1     1+3       B       C       D
 |       |       |        |       |       |
2+1     2+1   - 2+2       E       F     - G
 |       |   |   |        |       |    |  |
3+1     3+1  |  3+2       H       I    |  J
 |       |   /   |        |       |   /   |
4+1   4+1->3+1  4+1       K       L -     M
 |       |       |        |       |       |
5+0     4+0     5+0       N       O       P

"""

# Name = Node ( Name, Value, Child )
p = Node('P', '5+0', None)
o = Node('O', '4+0', None)
n = Node('N', '5+0', None)
m = Node('M', '4+1', p)
l = Node('L', '4+1->3+1', o)
k = Node('K', '4+1', n)
j = Node('J', '3+2', m)
i = Node('I', '3+1', l)
h = Node('H', '3+1', k)
g = Node('G', '2+2', [j,l])
f = Node('F', '2+1', i)
e = Node('E', '2+1', h)
d = Node('D', '1+3', g)
c = Node('C' ,'1+1', f)
b = Node('B' ,'1+1', e)
a = Node('A', '0+2', [b,c,d]) # Root

algorithm = AStar(a, [n,o,p])
algorithm.runAlgorithm()
algorithm.showPath()
```

Resultados del programa (Output)

```
Samuels-MBP:heuristic_search Samuel$ python3 algoritmo_A_asterisco.py

Diagrama del Problema:
```




```
Actual: (A: 0+2)
Abiertos: [(A: 0+2)]
Cerrados: []


Actual: (A: 0+2)
Abiertos: [(B: 1+1), (C: 1+1), (D: 1+3)]
Cerrados: [(A: 0+2)]


Actual: (B: 1+1)
Abiertos: [(C: 1+1), (E: 2+1), (D: 1+3)]
Cerrados: [(A: 0+2), (B: 1+1)]


Actual: (C: 1+1)
Abiertos: [(E: 2+1), (F: 2+1), (D: 1+3)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1)]


Actual: (E: 2+1)
Abiertos: [(F: 2+1), (H: 3+1), (D: 1+3)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1), (E: 2+1)]


Actual: (F: 2+1)
Abiertos: [(H: 3+1), (I: 3+1), (D: 1+3)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1), (E: 2+1), (F: 2+1)]
```

```
Actual: (H: 3+1)
Abiertos: [(I: 3+1), (D: 1+3), (K: 4+1)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1), (E: 2+1), (F: 2+1), (H: 3+1)]


Actual: (I: 3+1)
Abiertos: [(D: 1+3), (K: 4+1), (L: 4+1)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1), (E: 2+1), (F: 2+1), (H: 3+1),
(I: 3+1)]


Actual: (D: 1+3)
Abiertos: [(G: 2+2), (K: 4+1), (L: 4+1)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1), (E: 2+1), (F: 2+1), (H: 3+1),
(I: 3+1), (D: 1+3)]


Actual: (G: 2+2)
Abiertos: [(L: 3+1), (K: 4+1), (J: 3+2)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1), (E: 2+1), (F: 2+1), (H: 3+1),
(I: 3+1), (D: 1+3), (G: 2+2)]


Actual: (L: 3+1)
Abiertos: [(O: 4+0), (K: 4+1), (J: 3+2)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1), (E: 2+1), (F: 2+1), (H: 3+1),
(I: 3+1), (D: 1+3), (G: 2+2), (L: 3+1)]


Actual: (O: 4+0)
Abiertos: [(K: 4+1), (J: 3+2)]
Cerrados: [(A: 0+2), (B: 1+1), (C: 1+1), (E: 2+1), (F: 2+1), (H: 3+1),
(I: 3+1), (D: 1+3), (G: 2+2), (L: 3+1), (O: 4+0)]


CAMINO: [(A: 0+2), (D: 1+3), (G: 2+2), (L: 3+1), (O: 4+0)]
COSTE TOTAL: 18

--------FIN-------
```

Conclusiones

El programa funciona correctamente, intenté programarlo para problemas más complejos, en donde las rutas no están claramente definidas. El programa se basa en darle prioridad al nodo abierto con menor coste, y en caso de empate, poner en primer lugar al nodo que tenga menor valor h en la ecuación de coste f = g + h. Al final el programa se comporta de manera correcta.

El programa puede encontrarlo en mi repositorio:

https://github.com/sibarras/PythonExamples/blob/master/heuristic_search/algoritmo_A_asterisco.py