Pontificia Universidad Javeriana

Departamento de Ingeniería de Sistemas

Inteligencia Artificial

Informe Proyecto 1

Nombre de integrantes:

Jesús Molina
Santiago Vides
Tomas Figueroa
Santiago Rey

Profesor Ing. Julio Omar Palacio Niño, M.Sc.
Inteligencia Artificial
Septiembre 5, 2023
Bogotá, Colombia

Enlace repositorio del código:

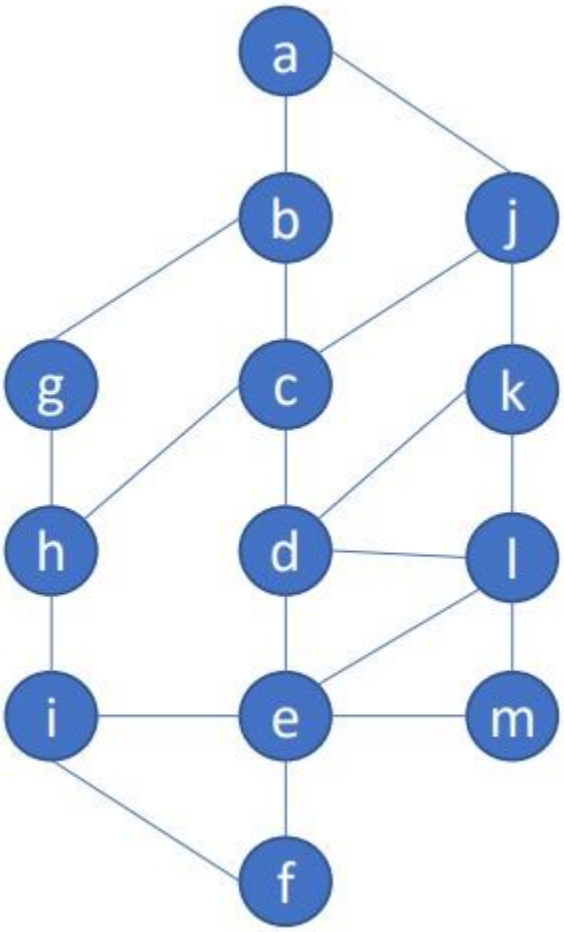
https://replit.com/join/wvmvfynhub-jesusmolina3

Grafo:

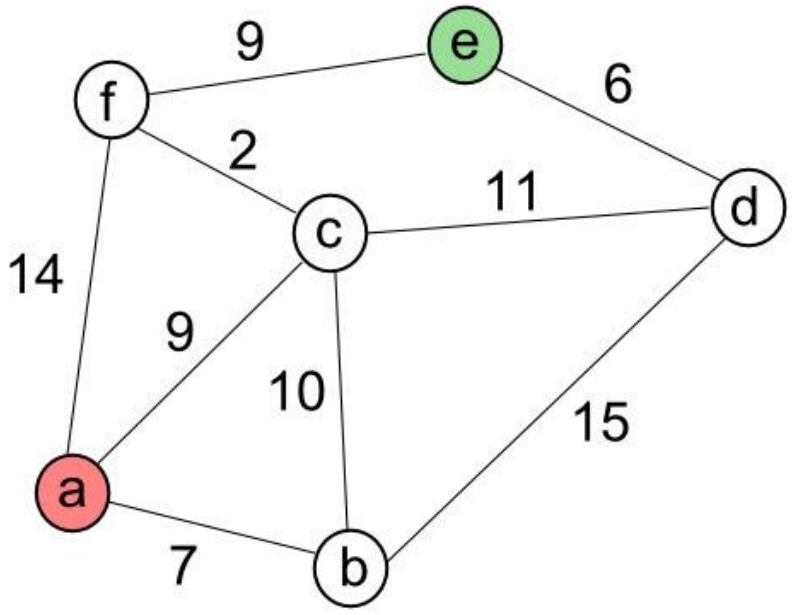
Con el propósito de implementar los algoritmos de búsqueda se creó un grafo en python. El grafo en cuestión está construido a partir de una clase vértice y una clase grafo. La clase vértice contiene el id del nodo y un diccionario que contiene los vértices adyacentes con el peso de sus conexiones, además de funciones para añadir un vecino con su respectivo peso y obtener las conexiones del vértice en cuestión. La clase grafo consiste en un diccionario de vértices y cuenta con funciones para añadir vértices y establecer conexiones. Al usar esta última función se añade una conexión bidireccional entre los vértices ingresados como parámetros.

Haciendo uso de estas estructuras y con la intención de probar los algoritmos, se quemaron en el código dos grafos, uno pequeño de 6 vértices y el grafo utilizado en las sesiones de clase. Este último con el fin de comprobar sin lugar a duda el correcto funcionamiento de los algoritmos.

Grafo visto en clase: con valor unitario de 10



Grafo pequeño:



Algoritmo de búsqueda en profundidad

```
def busqueda_profundidad(self, orig, dest):
```

Entradas:

- `self`: Es el objeto que representa el grafo utilizado. La función se sirve las propiedades y funciones de este objeto para obtener conexiones entre nodos y encontrar los pesos de estas conexiones.

- ` nodo_inicio `: Id del nodo por el que se empieza la búsqueda.

- ` nodo_final`: Id del nodo al cual se quiere llegar.

Salida:

- Imprime por consola la búsqueda los recorridos entre nodos que componen el camino hacia el nodo final, con sus respectivos costos. Incluye también el cálculo del costo total del camino.

```
RECORRIDO EN PROFUNDIDAD DEL NODO: a CON DESTINO: e

a ---> f  Costo: 14
f ---> e  Costo: 9


COSTO DEL RECORRIDO:23
```

Descripción de la función:

El algoritmo de búsqueda recibe como argumentos el vértice inicial y final. A partir del vértice inicial, mientras que no se llegue al vértice final o la lista correspondiente a la agenda de vértices por visitar sea vacía, se obtienen los nodos vecinos no visitados y se añaden a una lista de vértices, de esta lista se van extrayendo los nodos a expandir y se agregan a una lista de vértices expandidos. De la lista de vértices expandidos se obtiene la ruta del origen al destino (siempre y cuando se haya encontrado el vértice destino). Posteriormente se imprime en pantalla la secuencia de nodos que sigue la ruta con sus pesos y con el costo total del camino.

Algoritmo de Primero en anchura

```
def primero_anchura(self, nodo_inicio, nodo_final)
```

Entradas:

- **`self`:** Es un objeto que representa un grafo. La función utiliza las propiedades de este objeto para obtener a las conexiones entre los nodos.

- **` nodo_inicio `:** Es el id del nodo del cual queremos empezar el trayecto.

- **` nodo_final`:** Es el id del nodo al cual se quiere llegar.

Salida:

- Imprime por consola la búsqueda de primero en anchura del nodo inicial al nodo final junto a los precios de las conexiones y el costo total del trayecto.

```
RECORRIDO EN ANCHURA DEL NODO: a CON DESTINO: e

a--->f  Costo: 14
f--->e  Costo: 9


COSTO: 23
```

Descripción de la función:

El algoritmo recibe un grafo, el id del nodo de inicio y el id del nodo destino, después se usa la id del nodo de inicio para encontrar el nodo y así obtenemos a sus vecinos, por otra parte en recorridos guardamos la id del nodo de inicio, declaramos una variable booleana llamada ruta la cual dice si existe una ruta o no hacia el nodo destino que se nos dio, una variable para guardar el costo llamada “costo”, una variable índice la cual nos va a ayudar a poner las conexiones, y una variable de “conec” que nos pondrá las conexiones entre 2 nodos que son vecinos que vayamos encontrando. Después de eso iteramos en la lista de recorridos y buscamos el nodo actual en el grafo que tenemos para obtener sus conexiones es decir sus vecinos, de esta forma recorremos

sus vecinos y si el vecino que estamos recorriendo no está en el arreglo de recorridos lo agregamos. También agregamos al nodo actual y a su vecino para representar una conexión directa dentro de la variable “conec”, una vez se encuentre el nodo final dentro del arreglo de recorridos sí pondrá ruta como true y se romperá el ciclo.

Finalmente usamos el arreglo de “conec” iniciando una alteración desde el último elemento hacía el primero para escoger solamente los que tienen que ver con el trayecto que lleva al destino, como esta lista de conexiones tiene en cada posición 2 vecinos podemos separar esta cadena de caracteres para calcular el costo de esta conexión utilizando el “get_weight” y se usará el nodo primer nodo para buscar su vecino, este proceso se continua hasta que llega al nodo inicial. Ya con todos los datos se imprime el recorrido.

Algoritmo para calcular la funcion heurística:

```python
# función heurística
def heuristica_simple(self, nodo_objetivo):
  nodo_objetivo = self.vert_dict[nodo_objetivo]
  heuristica_simple = {}
  for nodo_id in self.vert_dict:
      nodo_actual = self.vert_dict[nodo_id]
      heuristica = abs(ord(nodo_actual.id) - ord(nodo_objetivo.id))
      heuristica_simple[nodo_id] = heuristica
  return heuristica_simple
```

Entradas:

- `self`: Es un objeto que representa un grafo. La función utiliza las propiedades de este objeto para calcular la heurística para cada nodo en el grafo.

- `nodo_objetivo`: Este es el nodo para el cual se desea calcular la heurística. Debe ser un identificador válido de un nodo en el grafo representado por el objeto "self".

Salida:

- Un diccionario llamado **"heuristica_simple"** que contiene la heurística calculada para cada nodo en el grafo. La clave de cada elemento en el diccionario es el identificador del nodo, y el valor es el valor de la heurística calculada para ese nodo.

Descripción de la función:

Lo primero que hacemos en la función es obtener el objeto que representa el nodo objetivo del grafo almacenado en el objeto **"self"** y se guarda en **“nodo_objetivo”.** Para enseguida inicializar un diccionario vacío llamado **"heuristica_simple"** que se utilizará para almacenar las heurísticas calculadas para cada nodo. Después dentro de un bucle **"for"** itera a través de todos los nodos en el grafo representado por **"self"** se guarda en **‘nodo_id`** el valor de un identificador de nodo en cada iteración y se obtiene el objeto del nodo actual a partir de su identificador. Se calcula la heurística para el nodo actual en relación con el nodo objetivo. La heurística se calcula tomando el valor absoluto de la diferencia entre los valores ASCII de los caracteres que representan los identificadores de los nodos. Esto proporciona una medida simple de la distancia entre los nodos

en función de sus identificadores. Luego se almacena la heurística calculada en el diccionario **"heuristica_simple"** usando el identificador del nodo como clave. Finalmente, la función devuelve el diccionario **"heuristica_simple"** que contiene las heurísticas calculadas para todos los nodos en el grafo.

Ejemplos:

Nodo destino = nodo 'C'

```
Heurísticas:
Nodo a: 2
Nodo b: 1
Nodo c: 0
Nodo d: 1
Nodo e: 2
Nodo f: 3
```

Nodo destino = nodo 'A'

```
Heurísticas:
Nodo a: 0
Nodo b: 1
Nodo c: 2
Nodo d: 3
Nodo e: 4
Nodo f: 5
```

Algoritmo A*

```
def a_estrella(self, nodo_inicial, nodo_final):
```

Entradas:

- **`self`:** Es un objeto que representa un grafo. La función utiliza las propiedades de este objeto para obtener a las conexiones entre los nodos.
- **`nodo_inicio`:** Nodo a partir del cual comenzará la busqueda.
- **`nodo_final`:** Nodo objetivo, la idea es buscar la ruta mas obtima según la heuristica.

Salidas:

- Imprime por consola la ruta resultante, ademas, del costo del recorrido.

```
Recorrido final: a -> c -> d (Costo total: 20)
```

Descripción de la función:

El algoritmo A* recibe como parámetros el grafo, un nodo por el que empezar la búsqueda y otro nodo final que es el destino al cual vamos a llegar. Para el algoritmo, vamos a tener varias estructuras de datos que va a cumplir la función de guardar los nodos ya visitados y los que están por ser explorados. Siempre y cuando tengamos nodos en la agenda, vamos a mantener nos en el

ciclo principal. Dentro de agenda, seleccionamos el nodo que tiene el costo más bajo, lo expandimos y revisamos los nodos a los que esta conectados, si el nodo no es el destino, marcamos el nodo como visitado y calculamos su heurística hasta el destino. Comparamos el costo calculado con los costos anteriores almacenados en el diccionario costo. Si el costo calculado es menor o si el nodo aún no ha sido visitado, actualizamos el costo de fn (fn = gn + hn) y lo agregamos a la agenda, la idea de esto es ir actualizando los costos a medida que vamos explorando. A medida que esto va ocurriendo, vamos almacenando los datos en el arreglo padres, el cual va a guardar los nodos que va a componer nuestra ruta. Al final del todo, usamos una función auxiliar llamada “construir_camino” la cual se encarga de construir la ruta usando el arreglo de padres. Al final, invertimos el arreglo del camino y lo retornamos, teniendo la ruta ya ordenada. En ese momento, imprimimos el orden del recorrido y su costo.