Pablo Emilio Lopez Avila Matricula 1740087 Matemáticas Computacionales

ALGORITMO DE DIJKSTRA

Es un algoritmo creado por Dijkstra en 1959, él era un “científico de la computación” su algoritmo le llevo a recibir grandes premios; El algoritmo como tal dice que dado un vértice dentro de un grafo que está conectado a algunos-todos los vértices, sus conexiones por primera vez aparece el término “peso” o “costo” el cual mediante su algoritmo; permitía obtener el camino más corto o de menos “peso” para el cual él podría recorrer todos los vértices.

En mis palabras utilizaría un ejemplo para el cual sería de una mejor comprensión; Tomemos el caso que somos repartidores de envíos a todo el país, tenemos bases en la mayoría de las capitales de los estados, el hecho que nos transportemos a tales capitales viajando nos genera un costo de gasolina, por ejemplo de Monterrey a Saltillo un gasto de 2x, pero si vamos a San Luis primero gastamos 1/2x y de San Luis a Saltillo gastamos x, entonces generaríamos un menor costo viajando primero a San Luis y luego Saltillo ; exactamente eso es lo que hace el algoritmo dado un punto de inicio va evaluando siempre las mejores opciones para generar un menor gasto a lo que llaman “camino más corto” o en caso de los negocios más “rentable”.

Complejidad: El peor de sus casos tiene una complejidad de n-1, siendo n el número de vértices.

El algoritmo es:

```
from heapq import heappop, heappush

def flatten(L):
   while len(L) > 0:
      yield L[0]
      L = L[1]
class Grafo:


   def __init__(self):
      self.V = set() # un conjunto
      self.E = dict() # un mapeo de pesos de aristas
      self.vecinos = dict() # un mapeo


   def agrega(self, v):
      self.V.add(v)
      if not v in self.vecinos: # vecindad de v
```

```
        self.vecinos[v] = set() # inicialmente no tiene nada


    def conecta(self, v, u, peso=1):
        self.agrega(v)
        self.agrega(u)
        self.E[(v, u)] = self.E[(u, v)] = peso # en ambos sentidos
        self.vecinos[v].add(u)
        self.vecinos[u].add(v)


    def complemento(self):
        comp= Grafo()
        for v in self.V:
            for w in self.V:
                if v != w and (v, w) not in self.E:
                    comp.conecta(v, w, 1)
        return comp


    def shortest(self, v): # Dijkstra's algorithm
        q = [(0, v, ())] # arreglo "q" de las "Tuplas" de lo que se va a almacenar dondo 0 es la distancia,
v el nodo y () el "camino" hacia el
        dist = dict() #diccionario de distancias
        visited = set() #Conjunto de visitados
        while len(q) > 0: #mientras exista un nodo pendiente
            (l, u, p) = heappop(q) # Se toma la tupla con la distancia menor
            if u not in visited: # si no lo hemos visitado
                visited.add(u) #se agrega a visitados
                dist[u] = (l,u,list(flatten(p))[::-1] + [u])  #agrega al diccionario
            p = (u, p) #Tupla del nodo y el camino
            for n in self.vecinos[u]: #Para cada hijo del nodo actual
```

```
        if n not in visited: #si no lo hemos visitado
            el = self.E[(u,n)] #se toma la distancia del nodo acutal hacia el nodo hijo
            heappush(q, (l + el, n, p)) #Se agrega al arreglo "q" la distancia actual mas la ditanacia hacia el nodo hijo, el nodo hijo n hacia donde se va, y el camino
    return dist #regresa el diccionario de distancias
```

Este es el código general para cualquier grafo, si queremos utilizarlo solo es necesario llamar la función e ir conectando los vértices entre ellos.

Por otra parte subiré los códigos aplicados para 5,10,15,20,25 vértices con el doble de conexiones entre ellos

Resultados 5 nodos.

Nodos Inicio-Fin	# Nodos que recorre	Recorrido	Peso
c-c	0	c	0
c-a	2	c-a	2
C-b	2	c-b	3
C-d	3	c-a-d	3
C-e	2	c-e	3

Algoritmo para 10 vértices

Nodo inicio-fin	# nodos que recorre	Recorrido	Peso
a-a	0	A	0
a-d	1	A-d	1
a-b	1	A-b	2
a-c	1	A-c	2
a-f	2	A-b-f,	3
a-e	2	a-c-e	5
a-h	2	a-c-h	7
a-g	3	a-b-f-g	8
a-i	2	A-c-i,	9
a-k	3	A-b-f-k	11

Algoritmo para 15 vertices:

Nodo inicio-Fin	# Nodos recorridos	Recorrido	Peso
a-a	0	A	0
a-d	1	A-d	1
a-b	1	A-b	2
a-c	1	A-c	2
a-f	2	A-b-f,	3
a-e	2	a-c-e	5
a-h	2	a-c-h	7
a-g	3	a-b-f-g	8
a-i	2	A-c-i,	9
a-k	3	A-b-f-k	11
a-p	2	a-c-p	22
a-l	4	a-b-f-k-l	24
a-o	3	a-c-p-o	25
a-m	4	a-c-p-o-m	28
a-n	5	a-b-f-k-l-n	64

Algoritmo para 20 vértices

Nodo inicio-Fin	# Nodos recorridos	Recorrido	Peso
a-a	0	A	0
a-d	1	A-d	1
a-b	1	A-b	2
a-c	1	A-c	2
a-f	2	A-b-f,	3
a-e	2	a-c-e	5
a-h	2	a-c-h	7
a-g	3	a-b-f-g	8
a-i	2	A-c-i,	9
a-k	3	A-b-f-k	11
a-p	2	a-c-p	22
a-l	4	a-b-f-k-l	24
a-o	3	a-c-p-o	25
a-m	4	a-c-p-o-m	28
a-n	5	a-b-f-k-l-n	64
a-s	3	'a'-'c'-'p',-'s'	32
a-r	3	'a', 'c', 'p', 'r'	35
a-q	4	'a', 'c', 'p', 'r', 'q'	47
a-v	4	'a', 'c', 'p', 'r', 'v'	48
a-u	4	'a', 'c', 'p', 'r', 'u'	56

Algoritmo para 25 vértices:

Nodo inicio-Fin	# Nodos recorridos	Recorrido	Peso
a-a	0	A	0
a-d	1	A-d	1
a-b	1	A-b	2
a-c	1	A-c	2
a-f	2	A-b-f,	3
a-e	2	a-c-e	5
a-h	2	a-c-h	7
a-g	3	a-b-f-g	8
a-i	2	A-c-i,	9
a-k	3	A-b-f-k	11
a-p	2	a-c-p	22
a-l	4	a-b-f-k-l	24
a-o	3	a-c-p-o	25
a-m	4	a-c-p-o-m	28
a-n	5	a-b-f-k-l-n	64
a-s	3	'a'-'c'-'p',-'s'	32
a-r	3	'a', 'c', 'p', 'r'	35
a-q	4	'a', 'c', 'p', 'r', 'q'	47
a-v	4	'a', 'c', 'p', 'r', 'v'	48
a-u	4	'a', 'c', 'p', 'r', 'u'	56
a-w	5	'a', 'c', 'p', 'r', 'v', 'w'	59
a-j	5	'a', 'c', 'p', 'r', 'v', 'j'	61
a-x	5	'a', 'c', 'p', 'r', 'v', 'x'	64
a-z	6	'a', 'c', 'p', 'r', 'v', 'w', 'z'	71
a-y	6	'a', 'c', 'p', 'r', 'v', 'x', 'y'	82

Conclusión:

El algoritmo de verdad es muy rápido, no permite que se cae rodeos sin sentido que puede arruinar el avance, algo que me sorprende que no solo ordena los vértices, sino que imprime todos los posibles casos desde el punto de inicio al de fin, y los ordena según su peso total.

Tarea de Castigo.

Algoritmo de Aproximación

Son algoritmos exactos para encontrar soluciones óptimas requieren tiempo superpolinomial a pesar de que estos algoritmos son más eficientes que la búsqueda exhaustiva pura, su tiempo de corrida es aún exponencial.

La verdad no entendí que era, pero a lo que veo es un recurso que se utiliza para una aproximación rápida y lo más “exacta” posible en un intervalo de tiempo dado la “Complejidad” de otros algoritmos.