Universidad Autónoma de Nuevo León
Facultad de Ciencias Físico Matemáticas
Alumna: **Mayra Alejandra Rivera Rodríguez**
Matricula:**1742899**
Matemáticas Computacionales

Algoritmo de Dijkstra

En este reporte se hablará del funcionamiento de el algoritmo de Dijkstra y además de presentar un seudocódigo del mismo.

¿Qué es el algoritmo de Dijkstra?

Es un algoritmo que determina la distancia más corta, en un grafo, desde un nodo inicial a sus demás nodos de este.

¿Cómo funciona Dijkstra?

La mayoría de las veces este algoritmo se utiliza para problemas de optimización de la vida real, un ejemplo podía ser las rutas viables que tomar para ir a la escuela con la menos distancia, lo que este algoritmo realizaría seria por medio de recorridos de los nodos buscar la distancia mas corta que minimize tiempo y recorrido.

Seudocódigo en Python

```
class Grafo:
    def __init__(self):
        self.V = set()
        self.E = dict()
        self.vecinos = dict()
```

```
def agrega(self, v):
        self.V.add(v)
        if not v in self.vecinos:
                self.vecinos[v] = set()
def conecta(self, v, u, peso=1):
        self.agrega(v)
        self.agrega(u)
        self.E[(v,u)] = self.E[(u,v)] = peso
        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, w):
                q = [(0, v, ())]
                visited = set()
                while len(q) > 0:
                        (l, u, p) = heappop(q)
                        if u not in visited:
                                visited.add(u)
                                if u == w:
                                        return list(flatten(p))[::-1] + [u]
```

```
                    p = (u, p)
                    for n in self[u].neighbors:
                        if n not in visited:
                            el = self.vecinos[u][n]

                            heappush(q, (l + el, n, p))
                return None
```

Prueba

```
g = grafo ()
g.conecta ( ' a ' , ' b ' , 2 )
g.conecta ( ' b ' , ' c ' , 4 )
g.conecta ( ' c ' , ' d ' , 1 )
g.conecta ( ' d ' , ' e ' , 2 )
g.conecta ( ' e ' , ' f ' , 1 )
g.conecta ( ' a ' , ' i ' , 3 )
g.conecta ( ' d ' , ' g ' , 1 )
g.conecta ( ' g ' , ' e ' , 2 )
g.conecta ( ' h ' , ' f ' , 4 )
g.conecta ( ' j ' , ' k ' , 3)
g.conecta ( ' k ' , ' u ' , 2 )
g.conecta ( ' k ' , ' l ' , 2)
g.conecta ( ' u ' , ' t ' , 1 )
g.conecta ( ' l ' , ' m ' , 1 )
g.conecta ( ' m ' , ' d ' , 3 )
g.conecta ( ' n ' , ' g ' , 4 )
g.conecta ( ' n ' , ' o ' , 1 )
g.conecta ( ' o ' , ' h ' , 5 )
```

```
g.conecta ( ' l ' , ' p ' , 2)
g.conecta ( ' p ' , ' t ' , 1 )
g.conecta ( ' p ' , ' q ' , 3 )
g.conecta ( ' q ' , ' n ' , 2 )
g.conecta ( ' q ' , ' s ' , 4 )
imprimir (g.shortest ( ' f ' ))
```

Conclusiones

Después de una serie de pruebas a lo que podemos concluir es que entre menos nodos y menos aristas el programa encuentra mas fácilmente la ruta mas corta o la mejor optimización dependiendo del problema.