U P T

UNIVERSIDAD POLITÉCNICA DE TULANCINGO

grafos: Dijkstra, BFS, DFS, arbol binario, ejercicios 5 pag 229 y ejercicios 1-2 pag 248

por

Guerrero Jiménez Jesse Isaac

Ingeniería en Sistemas Computacionales

Asignatura:

Estructura de datos

Nombre del Catedrático:

Mtra. Carlos Enrique Ramirez

Cuarto Cuatrimestre

Tulancingo de Bravo, Hidalgo

ISC 44

SEPTIEMBRE - DICIEMBRE 2023.

Introducción a Estructuras de Datos y Algoritmos: Explorando las Fundamentales

Las estructuras de datos y los algoritmos forman el cimiento sólido sobre el cual se construyen las soluciones a los desafíos computacionales. En este vasto universo, destacan diversas estructuras y técnicas, cada una diseñada para abordar problemas específicos de manera eficiente. Entre estas, se destacan los grafos, los recorridos BFS (Breadth-First Search) y DFS (Depth-First Search), el algoritmo de Dijkstra, y los árboles binarios.

Grafos: La Tejedura de Conexiones

Un grafo es una estructura que modela relaciones entre entidades a través de vértices y aristas. Estos vínculos son esenciales para representar situaciones en las que la conexión y dependencia entre elementos son cruciales. Los grafos son empleados en una variedad de contextos, desde redes sociales hasta rutas de vuelo, desempeñando un papel central en la resolución de problemas complejos.

Recorridos BFS y DFS: Navegando las Redes

Los recorridos BFS y DFS son algoritmos fundamentales para explorar grafos. BFS prioriza la expansión a lo ancho, explorando todos los vecinos antes de avanzar a niveles más profundos, ideal para encontrar el camino más corto entre dos nodos. Por otro lado, DFS adopta una estrategia en profundidad, adentrándose tan profundamente como sea posible antes de retroceder. Estos métodos son cruciales para análisis de redes, topología y resolución de laberintos.

Algoritmo de Dijkstra: Navegando Caminos Óptimos

Cuando se trata de encontrar la ruta más corta en un grafo ponderado, el algoritmo de Dijkstra brilla con luz propia. Su enfoque meticuloso prioriza la selección de los caminos más cortos en cada paso, garantizando la determinación eficiente de rutas óptimas. Desde planificación de rutas en logística hasta protocolos de enrutamiento en redes, el algoritmo de Dijkstra es una herramienta esencial.

Árboles Binarios: Ramificando la Lógica

Los árboles binarios son estructuras jerárquicas donde cada nodo tiene, como máximo, dos hijos. Estos árboles son fundamentales en la representación de jerarquías y la búsqueda eficiente de datos. Su aplicabilidad abarca desde la construcción de bases de datos hasta la implementación de algoritmos de búsqueda y ordenamiento, convirtiéndolos en un pilar esencial en la optimización de procesos computacionales.

En resumen, las estructuras de datos y algoritmos mencionados son como las herramientas de un arquitecto, proporcionando los medios para construir soluciones robustas y eficientes en el vasto paisaje de la informática. Su comprensión y aplicación hábil son esenciales para abordar problemas complejos y optimizar el rendimiento de los sistemas informáticos modernos.

GRAFOS

Algoritmo de Dijkstra:

El método Dijkstra resuelve el problema de encontrar el camino más corto entre dos nodos en un grafo ponderado. Comienza inicializando distancias y padres para cada nodo, utilizando infinito para representar distancias iniciales desconocidas. Luego, en un bucle, selecciona el nodo no visitado con la distancia mínima, actualizando las distancias acumulativas a sus nodos adyacentes. Este proceso continúa hasta que todos los nodos han sido visitados, y finalmente, se imprime el camino más corto utilizando el método construir_camino.

Además, cabe destacar que en el código proporcionado se observa la importación de un módulo denominado "Cola". La inclusión de estructuras de datos, como colas, es esencial en algoritmos de búsqueda, como el recorrido en amplitud (BFS) implementado en la clase Graph. La capacidad de importar y utilizar herramientas personalizadas, como la cola, subraya la modularidad del código y la adaptabilidad de la clase Graph para integrar soluciones específicas.

Método de Construir Camino:

El método construir_camino se encarga de reconstruir el camino más corto encontrado por el algoritmo de Dijkstra. Comenzando desde el nodo final, sigue retrocediendo a través de los nodos padres hasta alcanzar el nodo inicial. Este camino se construye de manera eficiente utilizando la información almacenada en el diccionario de padres. El resultado final es un camino legible que representa la ruta más corta entre los nodos especificados.

Recorrido en Profundidad (DFS):

El método DFS implementa el recorrido en profundidad en el grafo. Utiliza una función recursiva que explora tan profundamente como sea posible desde un nodo dado antes de retroceder y explorar otros nodos. Durante este proceso, cada nodo visitado se imprime, lo que permite seguir la secuencia de nodos en el orden en que fueron descubiertos.

Recorrido en Amplitud (BFS):

El método recorrido_BFS realiza el recorrido en amplitud, explorando todos los nodos a la misma profundidad antes de avanzar a la siguiente capa. Utiliza una cola para manejar el orden de exploración, y cada nodo visitado se añade a la lista de nodos, permitiendo observar la secuencia de nodos visitados en el orden en que se descubren.

```python
from cola import Cola

class Graph:
    def __init__(self, Nodos, dirijido = False):
        self.Nodos = Nodos
        self.aristas = {}
        self.dirijido = dirijido

        for nodo in self.Nodos:
            self.aristas[nodo] = []

    def grado(self):
        grados = {}
        for nodo, aristas in self.aristas.items():
            grado = len(aristas)
            grados[nodo] = grado
        return grados

    def imprimir_aristas(self):
        for nodo in self.Nodos:
            print(f"[{nodo}] --->",self.aristas[nodo])

    def unir_nodos(self, nodoA, nodoB, peso = None):
        self.aristas[nodoA].append((nodoB, peso))
        if not self.dirijido:
            self.aristas[nodoB].append((nodoA, peso))
```

```python
def Dijktra(self,nodo_inicial, nodo_final):
    nodos_no_visitados = set(self.Nodos)
    distancias = {nodo: float('inf') for nodo in self.Nodos}
    distancias[nodo_inicial] = 0
    padres = {nodo: None for nodo in self.Nodos}

    while nodos_no_visitados:
        nodo_actual = min(
            nodos_no_visitados,
            key=lambda n: distancias[n]
        )

        adyacentes = self.nodos_adyacentes(nodo_actual)

        for nodo_a, peso in adyacentes:
            distancia_acumulada = distancias[nodo_actual] + peso

            if distancia_acumulada < distancias[nodo_a]:
                distancias[nodo_a] = distancia_acumulada
                padres[nodo_a] = nodo_actual

        nodos_no_visitados.remove(nodo_actual)

    # Imprimir el camino
    camino = self.construir_camino(nodo_inicial, nodo_final, padres)
    if camino is not None:
        print(f"Camino de {nodo_inicial} a {nodo_final}: {camino}")

def construir_camino(self, nodo_inicial, nodo_final, padres):
    camino = [nodo_final]
    while camino[-1] != nodo_inicial:
        padre = padres.get(camino[-1])
        if padre is None:
            print(f"No hay camino de {nodo_inicial} a {nodo_final}")
            return None
        camino.append(padre)
    return ' -> '.join(reversed(camino))
```

```
    def calcular_peso(self, nodoA, nodoB):
        for nodo, peso in self.aristas[nodoA]:
            if nodo == nodoB:
                return peso
        return float('inf')

    def salidas(self, nodo):
        return len(self.aristas[nodo])

    def nodos_adyacentes(self,nodo):
        adyacentes = []
        for vecino, peso in self.aristas[nodo]:
            adyacentes.append((vecino,peso))
        return adyacentes

    def recorrido_BFS(self, nodo_inicial):
        c = Cola()
        visitados =  set()
        nodos = []
        c.push(nodo_inicial)
        while not c.empty():
            nodo_actual = c.peek_pop()
            if nodo_actual not in visitados:
                nodos.append(nodo_actual)
                visitados.add(nodo_inicial)
                for vecinos,_ in self.nodos_adyacentes(nodo_actual):
                    if vecinos not in  visitados:
                        c.push(vecinos)
        return nodos
```

```
    def recorrido_DFS(self, nodo_inicial):
        visitados = set()

        def dfs_recursivo(nodo):
            print(nodo, end=' ')
            visitados.add(nodo)

            for vecino, _ in self.nodos_adyacentes(nodo):
                if vecino not in visitados:
                    dfs_recursivo(vecino)

        dfs_recursivo(nodo_inicial)

Nodos = ["A", "B", "C", "D", "E"]
aristas = [("A","B", 5),
        ("A","C", 6),
        ("B","D", 1),
        ("C","E", 3),
        ("E","D", 7)]
graph = Graph(Nodos, dirijido = True)

for v,u,p in aristas:
    graph.unir_nodos(v,u,p)

graph.imprimir_aristas()
grados = graph.grado()
for nodo,grado in grados.items():
    print(f"[{nodo}] ---> {grado}")
graph.Dijktra("B", "E")
```

Estructura del Árbol:

La clase Arbol_Binario presenta una estructura básica con nodos representados por la clase Nodo. Cada nodo contiene un dato, enlace al padre, indicadores sobre su posición (si es menor o mayor que el padre), y enlaces a los nodos izquierdo y derecho. La clase también incluye métodos para verificar si el árbol está vacío y para agregar nodos, manteniendo la propiedad de árbol de búsqueda binaria.

Lambda en el Método Agregar Nodo:

La función lambda se utiliza en la línea if dato <= nodo.dato dentro del método agregar_nodo. Esta función anónima compara el valor del nuevo dato con el valor del nodo actual. Si el resultado es verdadero, el nuevo nodo se añade a la izquierda; de lo contrario, se añade a la derecha. La función lambda aquí sirve como un criterio de comparación personalizado.

Recorridos en Profundidad y Amplitud:

La clase proporciona métodos para realizar recorridos en profundidad (DFS) y en amplitud (BFS). El recorrido DFS se implementa en tres formas: in-order, pre-order y post-order, cada uno visitando los nodos en un orden específico. Por otro lado, el recorrido BFS utiliza una cola para explorar niveles completos antes de avanzar al siguiente, proporcionando una visión más amplia y horizontal del árbol.

Métodos Específicos:

Hermanos: El método hermanos devuelve los nodos hermanos de un nodo dado, organizados por niveles.
Padres: El método padres identifica los padres de cada nodo, excluyendo la raíz y agrupando los nodos por niveles.
Grado: El método grado calcula y muestra el grado de cada nodo, indicando si es hoja o el número de hijos.

```python
from cola import Cola

class Nodo:
    def __init__(self, dato=None, parent=None, is_root=False, es_menor=False, es_mayor=False, nivel=None):
        self.dato = dato
        self.parent = parent
        self.is_root = is_root
        self.es_menor = es_menor
        self.es_mayor = es_mayor
        self.izquierda = None
        self.derecha = None
        self.nivel = nivel

class Arbol_Binario:
    def __init__(self):
        self.root = None

    def vacio(self):
        if self.root == None:
            return True
        else:
            return False

    def agregar_nodo(self, dato):
        if self.vacio():
            self.root = Nodo(dato = dato, is_root=True, nivel=0)
        else:
            nodo = self.posicion_nueva(dato)
            if dato <= nodo.dato:
                nodo.izquierda = Nodo(dato=dato, parent=nodo, es_menor=True, nivel = (nodo.nivel)+1 )
            else:
                nodo.derecha = Nodo(dato=dato, parent=nodo, es_mayor=True, nivel = (nodo.nivel)+1 )
```

```python
    #   con esta fucion identifica el nodo padre del nuevo nodo
    #   dependiendo si es menor o mayor que la raiz
    def posicion_nueva(self, dato):
        aux = self.root
        while aux is not None:
            padre = aux
            if dato <= aux.dato:
                aux = aux.izquierda
            else:
                aux = aux.derecha
        return padre

    def buscar(self,nodo,valor):
        if not nodo:
            return False
        elif nodo.dato == valor:
            return nodo
        elif valor <= nodo.dato:
            return self.buscar(nodo.izquierda, valor)
        else:
            return self.buscar(nodo.derecha, valor)

    def recorrido_DFS(self, nodo):
        nodos_visitados = []
        if nodo:
            nodos_visitados.append(nodo.dato)
            nodos_visitados.extend(self.recorrido_DFS(nodo.izquierda))
            nodos_visitados.extend(self.recorrido_DFS(nodo.derecha))
        return nodos_visitados
```

```
def recorrido_BFS(self, nodo):
    c = Cola()
    c.push(nodo)
    nodos_visitados = set() #conjunto desordenado de elementos unicos
    nodos = []
    while not c.empty():
        nodo_actual = c.peek_pop()
        if nodo_actual not in nodos_visitados:
            nodos.append(nodo_actual.dato)
            nodos_visitados.add(nodo_actual)
            if nodo_actual.izquierda:
                c.push(nodo_actual.izquierda)
            if nodo_actual.derecha:
                c.push(nodo_actual.derecha)
    return nodos

def hermanos(self, nodo):
    niveles = {}
    lista = self.recorrido_BFS(nodo)
    for valor in lista:
        elemento = self.buscar(nodo, valor)
        nivel = elemento.nivel
        if nivel in niveles:
            niveles[nivel].append(elemento.dato)
        else:
            niveles[nivel] = [elemento.dato]
    return niveles
```

```
def padres(self, root):
    padres = {"Raiz": []}
    lista = self.recorrido_BFS(root)
    for valor in lista:
        elemento = self.buscar(root, valor)
        nivel = elemento.nivel
        if nivel != 0:
            if elemento.parent.dato in padres:
                padres[elemento.parent.dato].append(elemento.dato)
            else:
                padres[elemento.parent.dato] = [elemento.dato]
        else:
            padres["Raiz"].append(elemento.dato)
    return padres

def grado(self, root):
    lista = self.recorrido_BFS(root)
    for valor in lista:
        elemento = self.buscar(root, valor)
        grado = 0
        if elemento.izquierda:
            grado += 1
        if elemento.derecha:
            grado += 1

        if grado == 0:
            print(f"Nodo hoja: [{elemento.dato}]")
        else:
            print(f"Nodo: [{elemento.dato}] --> Grado: {grado}")
```

```python
#   recorridos de el arbol
    def in_order(self,nodo):
        if nodo:
            self.in_order(nodo.izquierda)
            print(nodo.dato)
            self.in_order(nodo.derecha)

    def pre_order(self,nodo): #busqueda en profundidad
        if nodo:
            print(nodo.dato)
            self.pre_order(nodo.izquierda)
            self.pre_order(nodo.derecha)

    def pos_order(self,nodo):
        if nodo:
            self.pos_order(nodo.izquierda)
            self.pos_order(nodo.derecha)
            print(nodo.dato)
```

Salida por consola de los métodos

```
El dato raiz del arbol es:

Nivel: 0
    - [15]
Nivel: 1
    - [10, 23]
Nivel: 2
    - [11, 18, 25]
Nivel: 3
    - [13, 43]
Nivel: 4
    - [30]


Padre:   Raiz
    - [15]
Padre:   15
    - [10, 23]
Padre:   10
    - [11]
Padre:   23
    - [18, 25]
Padre:   11
    - [13]
Padre:   25
    - [43]
Padre:   43
    - [30]


Nodo: [15] --> Grado: 2
Nodo: [10] --> Grado: 1
Nodo: [23] --> Grado: 2
Nodo: [11] --> Grado: 1
Nodo hoja: [18]
Nodo: [25] --> Grado: 1
Nodo hoja: [13]
Nodo: [43] --> Grado: 1
Nodo hoja: [30]
```

```python
from arbol_binario import Arbol_Binario

tree = Arbol_Binario()

tree.agregar_nodo(15)
tree.agregar_nodo(10)
tree.agregar_nodo(23)
tree.agregar_nodo(11)
tree.agregar_nodo(13)
tree.agregar_nodo(25)
tree.agregar_nodo(43)
tree.agregar_nodo(18)
tree.agregar_nodo(30)

print(f"\nEl dato raiz del arbol es: {tree.root.dato}\n") #identificar raiz principal

nodos_hermanos = tree.hermanos(tree.root) #identificar nodos hermanos y altura
for nivel, nodos in nodos_hermanos.items():
    print(f"Nivel: {nivel}")
    print(f"    - {nodos}")
print("\n")

nodos_padre = tree.padres(tree.root) #identificar quien es padre de quien
for padre, nodos in nodos_padre.items():
    print(f"Padre:   {padre}")
    print(f"    - {nodos}")
print("\n")

tree.grado(tree.root) #identificar el grado de los nodos junto con las hojas
```

Descripción del Programa para el Manejo de la Red de Equipos: Algoritmos en Grafos

El programa se centra en la representación y manipulación de una red de equipos informáticos mediante un grafo no dirigido. Cada nodo del grafo almacena información adicional, incluyendo el nombre del equipo y su tipo, que puede ser pc, notebook, servidor, router, switch o impresora. A continuación, se describen los nuevos métodos implementados para abordar las tareas propuestas:

a. cada nodo además del nombre del equipo deberá almacenar su tipo: pc, notebook, servidor, router, switch, impresora;
b. realizar un barrido en profundidad y amplitud partiendo desde la tres notebook:
Red Hat, Debian, Arch;
c. encontrar el camino más corto para enviar a imprimir un documento desde la pc: Manjaro,
Red Hat, Fedora hasta la impresora;
d. encontrar el árbol de expansión mínima;
e. determinar desde que pc (no notebook) es el camino más corto hasta el servidor “Guaraní”;
f. indicar desde que computadora del switch 01 es el camino más corto
al servidor “MongoDB”;
[230]
g. cambiar la conexión de la impresora al router 02 y vuelva a resolver el punto b;
h. debe utilizar un grafo no dirigido.

El programa amplía la funcionalidad del manejo de una red de equipos mediante un grafo no dirigido. En comparación con la implementación anterior, se han introducido nuevos métodos que abordan tareas específicas relacionadas con el tipo de equipo y la optimización de rutas.

Nuevo Atributo Tipo en los Nodos:
Cada nodo en el grafo ahora incluye un atributo adicional que representa el tipo de equipo. Esta información permite clasificar los nodos según su función, como pc, notebook, servidor, router, switch o impresora.

Impresión de Aristas con Tipos:
El método imprimir_aristas se ha actualizado para mostrar no solo los nodos y sus conexiones, sino también el tipo de equipo asociado. Esto proporciona una visión más detallada de la red, facilitando la identificación de roles y conexiones específicas.

Árbol de Expansión Mínima:
Se ha introducido el método arbol_de_exprecion_minima que implementa el algoritmo de Prim para encontrar el árbol de expansión mínima en la red. Este algoritmo es crucial para identificar la infraestructura más eficiente y de menor costo para conectar todos los equipos en la red.

Búsqueda por Tipo:
El método buscar permite buscar nodos específicos en la red según su tipo. Esto facilita la clasificación y recuperación de nodos específicos, brindando una herramienta útil para gestionar y analizar equipos según sus roles.

Optimización de Dijkstra para Rutas con Suma de Pesos:
En el método Dijktra, se ha mejorado el algoritmo de Dijkstra para no solo encontrar el camino más corto, sino también calcular la suma total de pesos en la ruta. La adición de la variable suma brinda información adicional sobre la eficiencia global del camino.

```python
from cola import Cola

class Graph:
    def __init__(self, Nodos, dirijido = False):
        self.Nodos = Nodos
        self.aristas = {}
        self.dirijido = dirijido

        for nodo, *_ in self.Nodos:
            self.aristas[nodo] = []

    def imprimir_aristas(self):
        for nodo, tipo in self.Nodos:
            print(f"[{tipo}] >> [{nodo}] --->",self.aristas[nodo])

    def unir_nodos(self, nodoA, nodoB, peso = None):
        self.aristas[nodoA].append((nodoB, peso))
        if not self.dirijido:
            self.aristas[nodoB].append((nodoA, peso))
```

```python
def Dijktra(self, nodo_inicial, nodo_final):
    nodos_no_visitados = set(nodo[0] for nodo in self.Nodos)
    distancias = {nodo[0]: float('inf') for nodo in self.Nodos}
    distancias[nodo_inicial[0]] = 0
    padres = {nodo[0]: None for nodo in self.Nodos}
    suma = {nodo[0]: 0 for nodo in self.Nodos}

    while nodos_no_visitados:
        nodo_actual = min(
            nodos_no_visitados,
            key=lambda n: distancias[n]
        )

        adyacentes = self.nodos_adyacentes((nodo_actual,))

        for nodo_a, peso in adyacentes:
            distancia_acumulada = distancias[nodo_actual] + peso

            if distancia_acumulada < distancias[nodo_a]:
                distancias[nodo_a] = distancia_acumulada
                padres[nodo_a] = nodo_actual
                suma[nodo_a] = suma[nodo_actual] + peso

        nodos_no_visitados.remove(nodo_actual)

    camino = self.construir_camino(nodo_inicial[0], nodo_final[0], padres)
    if camino is not None:
        suma_total = suma[nodo_final[0]]
        return camino, suma_total
```

```python
def construir_camino(self, nodo_inicial, nodo_final, padres):
    camino = [nodo_final]
    while camino[-1] != nodo_inicial:
        padre = padres.get(camino[-1])
        if padre is None:
            print(f"No hay camino de {nodo_inicial} a {nodo_final}")
            return None
        camino.append(padre)
    return ' -> '.join(reversed(camino))

def calcular_peso(self, nodoA, nodoB):
    for nodo, peso in self.aristas[nodoA]:
        if nodo == nodoB:
            return peso
    return float('inf')

def salidas(self, nodo):
    return len(self.aristas[nodo])
```

```python
def nodos_adyacentes(self,nodo):
    adyacentes = []
    for vecino, peso in self.aristas.get(nodo[0],[]):
        adyacentes.append((vecino,peso))
    return adyacentes

def recorrido_BFS(self, nodo_inicial):
    c = Cola()
    visitados =  set()
    nodos = []
    c.push(nodo_inicial)
    while not c.empty():
        nodo_actual = c.peek_pop()
        if nodo_actual not in visitados:
            nodos.append(nodo_actual)
            visitados.add(nodo_actual)
            for vecinos,_ in self.nodos_adyacentes((nodo_actual,)):
                if vecinos not in  visitados:
                    c.push(vecinos)
    return nodos

def recorrido_DFS(self, nodo_inicial):
    visitados = set()
    recorrido = []
    def dfs_recursivo(nodo):
        recorrido.append(nodo)
        visitados.add(nodo)

        for vecino,_ in self.nodos_adyacentes((nodo,)):
            if vecino not in visitados:
                dfs_recursivo(vecino)

    dfs_recursivo(nodo_inicial)
    return recorrido
```

```python
def arbol_de_exprecion_minima(self, nodo_inicial):
    visitados = set()
    arbol_expansion_minima = []
    distancias = {nodo[0]: float('inf') for nodo in self.Nodos}
    distancias[nodo_inicial[0]] = 0
    while len(visitados) < len(self.Nodos):
        nodo_actual = min(
            (nodo for nodo in self.Nodos if nodo[0] not in visitados),
            key=lambda n: distancias[n[0]]
        )
        visitados.add(nodo_actual[0])
        if distancias[nodo_actual[0]] != float('inf'):
            arbol_expansion_minima.append((nodo_actual[0], distancias[nodo_actual[0]]))
        for vecino, peso in self.nodos_adyacentes(nodo_actual):
            if vecino not in visitados and peso < distancias[vecino]:
                distancias[vecino] = peso
    return arbol_expansion_minima

def buscar(self, tipo):
    clasificados = []
    for nodo in self.Nodos:
        if nodo[1] == tipo:
            clasificados.append(nodo[0])
    return clasificados
```

```
Nodos = [("Ubuntu", "PC"), ("Mint", "PC"), ("Manjaro", "PC"), ("Parrot", "PC"), ("Fedora", "PC"),
        ("Debian", "Notebook"), ("Red Hat", "Notebook"), ("Arch", "Notebook"),
        ("Impresora","Impresora"),
        ("Guarani", "Servidor"), ("MongoDB", "Servidor"),
        ("Switch1","Switch"), ("Switch2","Switch"),
        ("Router1","Router"), ("Router2","Router"), ("Router3","Router")]

aristas = [
    ("Switch1", "Debian", 17),
    ("Switch1", "Ubuntu", 18),
    ("Switch1", "Impresora", 22),
    ("Switch1", "Mint", 80),
    ("Switch1", "Router1", 29),
    ("Router1", "Router2", 37),
    ("Router1", "Router3", 43),
    ("Router2", "Router3", 50),
    ("Router2", "Red Hat", 25),
    ("Router2", "Guarani", 9),
    ("Switch2", "Manjaro", 40),
    ("Switch2", "Parrot", 12),
    ("Switch2", "MongoDB", 5),
    ("Switch2", "Arch", 56),
    ("Switch2", "Fedora", 3),
    ("Switch2", "Router3", 61)
]
```

```
graph = Graph(Nodos, dirijido=False)
for v,u,p in aristas:
    graph.unir_nodos(v,u,p)
print("\n")
graph.imprimir_aristas()

print("\nEJERCICIO B:\nBarrido DFS")
red_hat_BFS = graph.recorrido_BFS("Red Hat")
debian_BFS = graph.recorrido_BFS("Debian")
arch_BFS = graph.recorrido_BFS("Arch")
print(f"\n{red_hat_BFS}\n")
print(f"{debian_BFS}\n")
print(f"{arch_BFS}\n")
print("Barrido DFS")
red_hat_DFS = graph.recorrido_DFS("Red Hat")
debian_DFS = graph.recorrido_DFS("Debian")
arch_DFS = graph.recorrido_DFS("Arch")
print(f"\n{red_hat_DFS}\n")
print(f"{debian_DFS}\n")
print(f"{arch_DFS}\n")
```

```
print("EJERCICIO C:\n \nDesde la pc: Manjaro")
lista = graph.Dijktra(("Manjaro", "PC"), ("Impresora","Impresora"))
camino,suma_total = lista
print(f"Camino de Manjaro a Impresora: {camino}")
print(f"Suma total de los pesos: {suma_total}")

print("\nDesde la notebook: Red Hat")
lista = graph.Dijktra(("Red Hat","Notebook"), ("Impresora","Impresora"))
camino,suma_total = lista
print(f"Camino de Red Hat a Impresora: {camino}")
print(f"Suma total de los pesos: {suma_total}")

print("\nDesde la pc: Fedora")
lista = graph.Dijktra(("Fedora","PC"), ("Impresora","Impresora"))
camino,suma_total = lista
print(f"Camino de Fedora a Impresora: {camino}")
print(f"Suma total de los pesos: {suma_total}")

print("\nEJERCICIO D:\nArbol de expancion minima")
nodo_origen = "Parrot"
arbol_expansion_minima = graph.arbol_de_exprecion_minima(("Parrot", "PC"))
print(f"{arbol_expansion_minima}\n")
```

```
tipo = "PC"
sistemas = graph.buscar(tipo)
menor = float('inf')
camino_mas_corto = None
for elemento in sistemas:
    lista = graph.Dijktra((elemento,tipo), ("Guarani","Servidor"))
    camino,distancia = lista
    if distancia < menor:
        menor = distancia
        camino_mas_corto = camino
if camino_mas_corto:
    print(f"El camino mas corto de todas las: {tipo} a Guarani es: {camino_mas
    print(f"Con una distancia de: {menor}")
else:
    print("No se encontro ningun camino")
```

```
tipo = "PC"
sistemas = ["Ubuntu", "Mint", "Debian"]
menor = float('inf')
camino_mas_corto = None
for elemento in sistemas:
    lista = graph.Dijktra((elemento,tipo), ("MongoDB","Servidor"))
    camino,distancia = lista
    if distancia < menor:
        menor = distancia
        dispositivo = elemento
if dispositivo:
    print(f"La {tipo} del Switch1 mas cercano a MongoDB es: {dispositivo}")
    print(f"Con una distancia de: {menor}\n")
else:
    print("No se encontro ningun camino\n")
```

SALIDAS POR CONSOLA

```
[PC] >> [Ubuntu] ---> [('Switch1', 18)]
[PC] >> [Mint] ---> [('Switch1', 80)]
[PC] >> [Manjaro] ---> [('Switch2', 40)]
[PC] >> [Parrot] ---> [('Switch2', 12)]
[PC] >> [Fedora] ---> [('Switch2', 3)]
[Notebook] >> [Debian] ---> [('Switch1', 17)]
[Notebook] >> [Red Hat] ---> [('Router2', 25)]
[Notebook] >> [Arch] ---> [('Switch2', 56)]
[Impresora] >> [Impresora] ---> [('Switch1', 22)]
[Servidor] >> [Guarani] ---> [('Router2', 9)]
[Servidor] >> [MongoDB] ---> [('Switch2', 5)]
[Switch] >> [Switch1] ---> [('Debian', 17), ('Ubuntu', 18), ('Impresora', 22), ('Mint', 80), ('Router1', 29)]
[Switch] >> [Switch2] ---> [('Manjaro', 40), ('Parrot', 12), ('MongoDB', 5), ('Arch', 56), ('Fedora', 3), ('Router3', 61)]
[Router] >> [Router1] ---> [('Switch1', 29), ('Router2', 37), ('Router3', 43)]
[Router] >> [Router2] ---> [('Router1', 37), ('Router3', 50), ('Red Hat', 25), ('Guarani', 9)]
[Router] >> [Router3] ---> [('Router1', 43), ('Router2', 50), ('Switch2', 61)]
```

```
EJERCICIO B:
Barrido DFS

['Red Hat', 'Router2', 'Router1', 'Router3', 'Guarani', 'Switch1', 'Switch2', 'Debian', 'Ubuntu', 'Impresora', 'Mint', 'Manjaro', 'Parrot', 'MongoDB', 'Arch', 'Fedora']

['Debian', 'Switch1', 'Ubuntu', 'Impresora', 'Mint', 'Router1', 'Router2', 'Router3', 'Red Hat', 'Guarani', 'Switch2', 'Manjaro', 'Parrot', 'MongoDB', 'Arch', 'Fedora']

['Arch', 'Switch2', 'Manjaro', 'Parrot', 'MongoDB', 'Fedora', 'Router3', 'Router1', 'Router2', 'Switch1', 'Red Hat', 'Guarani', 'Debian', 'Ubuntu', 'Impresora', 'Mint']

Barrido DFS

['Red Hat', 'Router2', 'Router1', 'Switch1', 'Debian', 'Ubuntu', 'Impresora', 'Mint', 'Router3', 'Switch2', 'Manjaro', 'Parrot', 'MongoDB', 'Arch', 'Fedora', 'Guarani']

['Debian', 'Switch1', 'Ubuntu', 'Impresora', 'Mint', 'Router1', 'Router2', 'Router3', 'Switch2', 'Manjaro', 'Parrot', 'MongoDB', 'Arch', 'Fedora', 'Red Hat', 'Guarani']

['Arch', 'Switch2', 'Manjaro', 'Parrot', 'MongoDB', 'Fedora', 'Router3', 'Router1', 'Switch1', 'Debian', 'Ubuntu', 'Impresora', 'Mint', 'Router2', 'Red Hat', 'Guarani']
```

```
EJERCICIO C:

Desde la pc: Manjaro
Camino de Manjaro a Impresora: Manjaro -> Switch2 -> Router3 -> Router1 -> Switch1 -> Impresora
Suma total de los pesos: 195

Desde la notebook: Red Hat
Camino de Red Hat a Impresora: Red Hat -> Router2 -> Router1 -> Switch1 -> Impresora
Suma total de los pesos: 113

Desde la pc: Fedora
Camino de Fedora a Impresora: Fedora -> Switch2 -> Router3 -> Router1 -> Switch1 -> Impresora
Suma total de los pesos: 158

EJERCICIO D:
Arbol de expancion minima
[('Parrot', 0), ('Switch2', 12), ('Fedora', 3), ('MongoDB', 5), ('Manjaro', 40), ('Arch', 56), ('Router3', 61), ('Router1', 43), ('Switch1', 29), ('Debian'
, 17), ('Ubuntu', 18), ('Impresora', 22), ('Router2', 37), ('Guarani', 9), ('Red Hat', 25), ('Mint', 80)]

EJERCICIO E:

El camino mas corto de todas las: PC a Guarani es: Ubuntu -> Switch1 -> Router1 -> Router2 -> Guarani
Con una distancia de: 93


EJERCICIO F:

La PC del Switch1 mas cercano a MongoDB es: Debian
Con una distancia de: 155

[7282 preload-host-spawn-strategy] Warning: waitpid override ignores groups
sh-5.1$
```

ESTRUCTURA DE COLA IMPLEMENTADA EN OTROS ALGORITMOS

```python
class Cola:
    def __init__(self, size = 5):
        self.lista = []
        self.size = size
        self.tope = 0

    def empty(self):
        if self.lista == []:
            return True
        else:
            return False

    def push(self, dato):
        if self.tope < self.size:
            self.lista.append(dato)
            self.tope += 1
        elif self.tope == self.size:
            self.size += 5
            self.lista.append(dato)
            self.tope += 1

    def pop(self):
        if not self.empty():
            self.lista = self.lista[1:]
            self.tope -= 1

    def show(self):
        for i in range(self.tope-1,-1,-1):
            print(f"[{i}] -> {self.lista[i]}")

    def peek(self):
        if not self.empty():
            return self.lista[0]

    def peek_pop(self):
        elemento = self.peek()
        self.pop()
        return elemento
```

Resuelva el problema de dar los siguientes cambios de monedas diseñando un algoritmo voraz:

a. el costo es 4.01 y se paga con 10,
b. el costo es 10.75 y se paga con 20,
c. el costo es 0.93 y se paga con 5.
Para ello, considerar los siguientes sistemas de monedas:
d. monedas griegas: 0.01, 0.02, 0.05, 0.10, 0.20, 0.50, 1.00, 2,00 euros;
e. monedas japonesas: 1, 5, 10, 50, 100, 500 yen;
f. monedas rusas: 0.01, 0.05, 0.1, 0.5, 1, 2, 5, 10 rublo;
g. monedas tailandesas: 0.01, 0.05, 0.1, 0.25, 0.5, 1, 2, 5 baht.
Responda las siguientes preguntas:
h. ¿el mismo algoritmo funciona para todos los sistemas monetarios?;
i. ¿todos los sistemas monetarios tiene solución?;
j. ¿si se obtiene una solución siempre es óptima?

Estructura y Organización del Código:

El programa se estructura en torno a tres funciones principales, cada una desempeñando un papel crucial en la resolución del problema. La función dar_cambio constituye el núcleo del algoritmo voraz, calculando la cantidad óptima de monedas para un cambio dado. Además, la función imprimir_cambio y calcular_y_mostrar_cambios proporcionan un marco para visualizar los resultados de manera clara y sistemática.

Algoritmo Voraz y Lógica de dar_cambio:

El algoritmo voraz aplicado en la función dar_cambio demuestra una estrategia eficiente para minimizar el número de monedas utilizadas en el cambio. La lógica subyacente aborda cada situación de cambio dividiendo la tarea en subproblemas más pequeños, priorizando las denominaciones de monedas de mayor valor. Además, se implementa una cuidadosa validación para garantizar que el pago sea suficiente para cubrir el costo, y se manejan posibles problemas de redondeo para mejorar la precisión de los cálculos.

Presentación y Flexibilidad del Código:

La función imprimir_cambio desempeña un papel crucial al presentar los resultados de manera clara y detallada. La información sobre el sistema monetario, costo, pago y el cambio dado se exhibe de manera estructurada, facilitando la comprensión y evaluación de los resultados. Además, la función calcular_y_mostrar_cambios demuestra la flexibilidad del código al adaptarse a diversos sistemas monetarios y escenarios específicos de costo y pago.

```python
def dar_cambio(costo, pago, monedas):
    if pago < costo:
        print("El pago es insuficiente para cubrir el costo.")
        return
    cambio = pago - costo
    cambio_dado = {}
    for moneda in reversed(monedas):
        cantidad_monedas = round(cambio // moneda)
        cambio_dado[moneda] = cantidad_monedas
        cambio -= cantidad_monedas * moneda
    return cambio_dado

def imprimir_cambio(sistema, costo, pago, cambio_dado):
    print(f"Sistema monetario: {sistema}")
    print(f"Costo: {costo}, Pago: {pago}")
    print("Cambio dado:")
    for moneda, cantidad in cambio_dado.items():
        print(f"{moneda} --> {cantidad:.0f}")
    print()

def calcular_y_mostrar_cambios():
    sistemas_monedas = {
        'monedas_griegas': [0.01, 0.02, 0.05, 0.10, 0.20, 0.50, 1.00, 2.00],
        'monedas_japonesas': [1, 5, 10, 50, 100, 500],
        'monedas_rusas': [0.01, 0.05, 0.1, 0.5, 1, 2, 5, 10],
        'monedas_tailandesas': [0.01, 0.05, 0.1, 0.25, 0.5, 1, 2, 5]
    }
    costos_pagos = [
        {'costo': 4.01, 'pago': 10},
        {'costo': 10.75, 'pago': 20},
        {'costo': 0.93, 'pago': 5}
    ]
    for sistema, monedas in sistemas_monedas.items():
        for cp in costos_pagos:
            cambio_dado = dar_cambio(cp['costo'], cp['pago'], monedas)
            imprimir_cambio(sistema, cp['costo'], cp['pago'], cambio_dado)

calcular_y_mostrar_cambios()
```

```
Sistema monetario: monedas_griegas
Costo: 4.01, Pago: 10
Cambio dado:
2.0 --> 2
1.0 --> 1
0.5 --> 1
0.2 --> 2
0.1 --> 0
0.05 --> 1
0.02 --> 2
0.01 --> 0

Sistema monetario: monedas_griegas
Costo: 10.75, Pago: 20
Cambio dado:
2.0 --> 4
1.0 --> 1
0.5 --> 0
0.2 --> 1
0.1 --> 0
0.05 --> 0
0.02 --> 2
0.01 --> 0

Sistema monetario: monedas_griegas
Costo: 0.93, Pago: 5
Cambio dado:
2.0 --> 2
1.0 --> 0
0.5 --> 0
0.2 --> 0
0.1 --> 0
0.05 --> 1
0.02 --> 1
0.01 --> 0
```

```
Sistema monetario: monedas_japonesas
Costo: 4.01, Pago: 10
Cambio dado:
500 --> 0
100 --> 0
50 --> 0
10 --> 0
5 --> 1
1 --> 0

Sistema monetario: monedas_japonesas
Costo: 10.75, Pago: 20
Cambio dado:
500 --> 0
100 --> 0
50 --> 0
10 --> 0
5 --> 1
1 --> 4

Sistema monetario: monedas_japonesas
Costo: 0.93, Pago: 5
Cambio dado:
500 --> 0
100 --> 0
50 --> 0
10 --> 0
5 --> 0
1 --> 4
```

```
Sistema monetario: monedas_tailandesas
Costo: 4.01, Pago: 10
Cambio dado:
5 --> 1
2 --> 0
1 --> 0
0.5 --> 1
0.25 --> 1
0.1 --> 2
0.05 --> 0
0.01 --> 4

Sistema monetario: monedas_tailandesas
Costo: 10.75, Pago: 20
Cambio dado:
5 --> 1
2 --> 2
1 --> 0
0.5 --> 0
0.25 --> 1
0.1 --> 0
0.05 --> 0
0.01 --> 0

Sistema monetario: monedas_tailandesas
Costo: 0.93, Pago: 5
Cambio dado:
5 --> 0
2 --> 2
1 --> 0
0.5 --> 0
0.25 --> 0
0.1 --> 0
0.05 --> 1
0.01 --> 2
```

```
Sistema monetario: monedas_rusas
Costo: 4.01, Pago: 10
Cambio dado:
10 --> 0
5 --> 1
2 --> 0
1 --> 0
0.5 --> 1
0.1 --> 4
0.05 --> 1
0.01 --> 4

Sistema monetario: monedas_rusas
Costo: 10.75, Pago: 20
Cambio dado:
10 --> 0
5 --> 1
2 --> 2
1 --> 0
0.5 --> 0
0.1 --> 2
0.05 --> 0
0.01 --> 4

Sistema monetario: monedas_rusas
Costo: 0.93, Pago: 5
Cambio dado:
10 --> 0
5 --> 0
2 --> 2
1 --> 0
0.5 --> 0
0.1 --> 0
0.05 --> 1
0.01 --> 2
```

Implementar un algoritmo que permita determinar qué elementos debe llevar un Jedi en su mochila de manera que se optimice el beneficio, con las siguientes consideraciones:

a. los elementos tienen una cantidad máxima y no son fraccionables;
b. la mochila tiene una capacidad de 27 kilos;
c. los elementos son los siguientes:

Elemento	Peso	Beneficio	Cantidad
frutas	0.73	50	8
carpa	3	90	2
termo stanley	1.5	75	3
sable de luz	1.3	175	1
mapa holograma	1.1	93	1
traje Jedi	0.9	87	4
bolsa de créditos galácticos	5	100	3
lata de alimento	0.79	75	10
galletitas	1	60	15
yerba canarias	0.64	70	7
pan	2.5	65	5
botella de agua	1.9	99	5
soga	2.3	40	3
mate	0.8	75	6
blaster	8.3	85	2

El código presentado implementa un algoritmo para resolver el problema de la mochila, un escenario común en la optimización combinatoria donde se busca maximizar el beneficio, dado un conjunto de elementos con diferentes pesos y valores, y una capacidad máxima de carga. A continuación, se realiza un análisis del código, destacando las partes más relevantes y, posiblemente, menos intuitivas.

Descripción General:
El algoritmo emplea un enfoque de programación dinámica para construir una tabla (tabla) que representa el beneficio máximo alcanzable para diferentes combinaciones de elementos y capacidades de la mochila. La tabla se actualiza mediante un proceso iterativo que considera la inclusión o exclusión de cada elemento en la mochila, evaluando las opciones óptimas.

Partes Destacadas:

Generación de la Tabla:
Se utiliza un bucle anidado para recorrer cada elemento y cada capacidad de la mochila. Se calcula el beneficio máximo posible para cada combinación, considerando las diferentes cantidades de cada elemento que podrían incluirse.

Selección de Elementos:
Después de construir la tabla, el algoritmo retrocede para determinar qué elementos deben incluirse en la mochila y en qué cantidad. Se utiliza un segundo bucle para iterar a través de los elementos y sus cantidades, eligiendo las combinaciones que maximizan el beneficio total.

Impresión de Resultados:
Finalmente, se imprimen en pantalla los elementos seleccionados junto con la cantidad de cada uno, así como el beneficio total alcanzado.

```python
def mochila(elementos, capacidad_mochila):
    num_elementos = len(elementos)
    tabla = [[0] * (capacidad_mochila + 1) for _ in range(num_elementos + 1)]

    for i in range(1, num_elementos + 1):
        nombre, peso, beneficio, cantidad = elementos[i - 1]
        for capacidad in range(capacidad_mochila + 1):
            for k in range(min(cantidad, int(capacidad // peso)) + 1):
                tabla[i][capacidad] = max(
                    tabla[i][capacidad],
                    tabla[i - 1][int(capacidad - k * peso)] + k * beneficio
                )

    resultado = []
    capacidad_restante = capacidad_mochila
    for i in range(num_elementos, 0, -1):
        nombre, peso, beneficio, cantidad = elementos[i - 1]
        for k in range(min(cantidad, int(capacidad_restante // peso)), 0, -1):
            if k * peso <= capacidad_restante and \
                tabla[i][int(capacidad_restante)] == tabla[i - 1][int(capacidad_restante - k * peso)] + k * beneficio:
                resultado.append((nombre, k))
                capacidad_restante -= k * peso

elementos = [
    ('frutas', 0.73, 50, 8),
    ('carpa', 3, 90, 2),
    ('termo stanley', 1.5, 75, 3),
    ('sable de luz', 1.3, 175, 1),
    ('mapa holograma', 1.1, 93, 1),
    ('traje Jedi', 0.9, 87, 4),
    ('bolsa de créditos galácticos', 5, 100, 3),
    ('lata de alimento', 0.79, 75, 10),
    ('galletitas', 1, 60, 15),
    ('yerba canarias', 0.64, 70, 7),
    ('pan', 2.5, 65, 5),
    ('botella de agua', 1.9, 99, 5),
    ('soga', 2.3, 40, 3),
    ('mate', 0.8, 75, 6),
    ('blaster', 8.3, 85, 2),
]

capacidad_mochila = 27

resultado = mochila(elementos, capacidad_mochila)

print("Elementos seleccionados:")
for nombre, k in resultado:
    print(f"{k} unidades de {nombre}")

beneficio_total = sum(elemento[2] * k for elemento, k in zip(elementos, [cantidad for _, cantidad in resultado]))
print(f"\nBeneficio total: {beneficio_total}")
```

```
Elementos seleccionados:
6 unidades de mate
7 unidades de yerba canarias
10 unidades de lata de alimento
2 unidades de termo stanley
8 unidades de frutas

Beneficio total: 2774
[7282 preload-host-spawn-strategy] Warning: waitpid override ignores groups
sh-5.1$
```

CONCLUSIONES

En este recorrido a través de distintos temas, desde la implementación de grafos y árboles binarios hasta la aplicación de algoritmos como BFS, DFS, y Dijkstra, hemos explorado la versatilidad y poder de la programación, específicamente utilizando Python como nuestro lenguaje dinámico. La implementación de estas estructuras de datos y algoritmos no solo destaca la elegancia y simplicidad del código en Python, sino también cómo estas herramientas pueden abordar una variedad de problemas complejos en la informática y más allá.

El manejo de grafos, con su capacidad para modelar relaciones y conexiones, se ha demostrado esencial en problemas de rutas óptimas, redes, y organización de datos complejos. Los algoritmos de búsqueda, como BFS y DFS, nos permiten navegar eficientemente por estas estructuras, descubriendo patrones y relaciones clave.

El algoritmo de Dijkstra, con su enfoque voraz, ha sido presentado como una solución efectiva para encontrar los caminos más cortos en grafos ponderados, abriendo la puerta a aplicaciones en logística, redes de transporte, y más.

El estudio de árboles binarios y sus recorridos (in-order, pre-order, y post-order) nos proporciona herramientas para organizar y procesar datos jerárquicos de manera sistemática, con aplicaciones en estructuras de datos y algoritmos de búsqueda eficientes.

En cuanto a la implementación de cambios de monedas, hemos explorado cómo un algoritmo voraz puede abordar problemas prácticos y cotidianos, adaptándose a diferentes sistemas monetarios y ofreciendo soluciones eficientes.

En última instancia, el aprendizaje y aplicación de un lenguaje dinámico como Python ha sido un hilo conductor en todos estos temas. Python no solo facilita la implementación de algoritmos y estructuras de datos de manera clara y concisa, sino que también fomenta un enfoque exploratorio y experimental, lo que es esencial para resolver problemas complejos y encontrar soluciones innovadoras.