```
# Importar las librerías necesarias
import re  # Para trabajar con expresiones regulares
import pyodbc  # Para conectar con bases de datos SQL Server
import pandas as pd  # Para manipulación y análisis de datos en DataFrames
from math import radians, sin, cos, sqrt, asin  # Para cálculos de distancia utilizando la fórmula de Haversine

# Configuración de la conexión a la base de datos SQL Server
db_file = f'DRIVER={{SQL Server}};SERVER={{BOGLSREBOLLEDO\\SQLEXPRESS}};DATABASE={{CDS}};UID={{sa}};PWD={{Admin123}}'

# Función para estandarizar una dirección eliminando espacios, convirtiendo a mayúsculas y eliminando espacios adicionales
def estandarizar_direccion(direccion):
    if direccion is None:
        return ''
    return re.sub(r'\s+', '', direccion.strip().upper())

# Función para corregir el formato de una coordenada
def corregir_coordenada(coordenada):
    if isinstance(coordenada, float):
        return coordenada
    if coordenada is None:
        return 0.0
    coordenada = coordenada.replace(',', '.')# Convierte comas en puntos y asegura que el resultado sea un float
    return float(coordenada)
```

```
# Función para calcular la distancia entre dos puntos geográficos usando la fórmula de Haversine
def haversine(lat1, lon1, lat2, lon2):
    # Convertir grados a radianes
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    # Fórmula de Haversine
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    a = min(1, max(-1, a))
    c = 2 * asin(sqrt(a))
    return 6371000 * c # Retorna la distancia en metros

# Función para ejecutar una consulta SQL y devolver los resultados
def obtener_datos(query, cursor):
    cursor.execute(query)
    return cursor.fetchall()

# Función para buscar coincidencias exactas en la tabla FPostes
def buscar_en_fpostes(direccion_buscar, ciudad_buscar, resultados_fpostes):
    # Inicializa una lista vacía para almacenar las coincidencias encontradas
    distancias = []
    # Estandariza la dirección y ciudad que se buscan para asegurar una comparación uniforme
    direccion_buscar = estandarizar_direccion(direccion_buscar)
    ciudad_buscar = estandarizar_direccion(ciudad_buscar)
```

```
    # Recorre cada fila en los resultados de la tabla FPostes
    for row_fpostes in resultados_fpostes:
        # Asegura que la fila tenga al menos 5 elementos antes de proceder
        if len(row_fpostes) >= 5:
            # Estandariza la dirección y la región (ciudad) de la fila actual
            direccion_fpostes = estandarizar_direccion(row_fpostes[0])
            region_fpostes = estandarizar_direccion(row_fpostes[1])
            # Compara la dirección y la ciudad de la fila actual con las que se buscan
            if direccion_fpostes == direccion_buscar and region_fpostes == ciudad_buscar:
                # Extrae y corrige las coordenadas de la fila actual
                lat_fpostes = corregir_coordenada(row_fpostes[2])
                lon_fpostes = corregir_coordenada(row_fpostes[3])
                # Obtiene el ID del CTO de la fila actual
                id_cto = row_fpostes[4]
                # Establece la distancia como 0, ya que es una coincidencia exacta
                distancia = 0
                # Añade la coincidencia exacta a la lista de distancias
                # Incluye la distancia, la fila completa, el ID del CTO, y la fuente de datos
(tabla FPostes)
                distancias.append((distancia, row_fpostes, id_cto, "FPostes"))

    # Devuelve la lista de coincidencias exactas encontradas
    return distancias



def buscar_en_patchpannel(direccion_buscar, ciudad_buscar,
resultados_patchpannel):
```

```
    # Inicializa una lista vacía para almacenar las coincidencias encontradas
    distancias = []
    # Estandariza la dirección y ciudad que se buscan para asegurar una comparación uniforme
    direccion_buscar = estandarizar_direccion(direccion_buscar)
    ciudad_buscar = estandarizar_direccion(ciudad_buscar)

    # Recorre cada fila en los resultados de la tabla PatchPannel
    for row_patchpannel in resultados_patchpannel:
        # Asegura que la fila tenga al menos 5 elementos antes de proceder
        if len(row_patchpannel) >= 5:
            # Estandariza la dirección y la región (ciudad) de la fila actual
            direccion_patchpannel = estandarizar_direccion(row_patchpannel[0])
            region_patchpannel = estandarizar_direccion(row_patchpannel[1])
            # Compara la dirección y la ciudad de la fila actual con las que se buscan
            if direccion_patchpannel == direccion_buscar and region_patchpannel == ciudad_buscar:
                # Extrae y corrige las coordenadas de la fila actual
                lat_patchpannel = corregir_coordenada(row_patchpannel[2])
                lon_patchpannel = corregir_coordenada(row_patchpannel[3])
                # Obtiene el ID del CTO de la fila actual
                id_cto = row_patchpannel[4]
                # Establece la distancia como 0, ya que es una coincidencia exacta
                distancia = 0
                # Añade la coincidencia exacta a la lista de distancias
                # Incluye la distancia, la fila completa, el ID del CTO, y la fuente de datos (tabla PatchPannel)
```

```
            distancias.append((distancia, row_patchpannel, id_cto, "PatchPannelF"))

    # Devuelve la lista de coincidencias exactas encontradas
    return distancias

# Función para obtener datos de CTO desde la base de datos
def obtener_cto(cursor):
    query_cto = "SELECT ID_CTO, OLT_NOMBRE, CTO, CABLE, PROPIETARIO, OT, PTO FROM OcupTelefonica"
    cursor.execute(query_cto)
    resultados_cto = cursor.fetchall()
    return {row[0]: row[1:] for row in resultados_cto}

# Revisa que solo se crucen ID_CTO que esten en la tabla de Ocupacion
def validar_id_cto(id_cto, cursor):
    query = "SELECT COUNT(1) FROM OcupTelefonica WHERE ID_CTO = ?"
    cursor.execute(query, (id_cto,))
    result = cursor.fetchone()
    return result[0] > 0  # Devuelve True si el conteo es mayor que 0, es decir, el ID_CTO es válido

# Función para guardar los resultados en un archivo Excel
def guardar_resultados(resultados, archivo, resultados_empalmes_finales):
    # Crea un DataFrame de pandas con los resultados finales
    df_resultados = pd.DataFrame(resultados)
    # Define las columnas requeridas para la hoja 'CTO', incluyendo 'RESPUESTA'
```

```
    columnas_requeridas = ['ID-PRE', 'DS', 'NOMBRE CLIENTE','Ciudad', 'Dirección', 'Latitud', 'Longitud', 'ORDER', 'ID_CTO', 'PROPIETARIO', 'CENTRAL', 'OLT_NOMBRE' , 'CABLE' , 'CTO', 'Distancia' , 'PTO_DISPO' , 'OT', 'Tabla', 'RESPUESTA', 'TIPO ZONA']

    # Asegura que el DataFrame tenga todas las columnas requeridas

    for columna in columnas_requeridas:

        if columna not in df_resultados.columns:

            df_resultados[columna] = ''  # Agrega columnas faltantes con valores vacíos

    # Agrega la columna 'ORDER' y asigna un valor secuencial

    df_resultados['ORDER'] = [1, 2] * (len(df_resultados) // 2)

    if len(df_resultados) % 2 != 0:

        df_resultados['ORDER'].iloc[-1] = 1

    df_resultados['RESPUESTA'] = df_resultados['Tabla'].apply(lambda x: 'DIR EXACTA' if x in ['FPostes', 'PatchPannelF'] else 'CTO CERCANA')

    df_resultados['TIPO ZONA'] = df_resultados['Tabla'].apply(lambda x: 'ZA' if x in ['FPostes', 'DivisorTerminal'] else 'ZC')

    # Reorganiza las columnas según el orden especificado en 'columnas_requeridas'

    df_resultados = df_resultados[columnas_requeridas]

    # Guarda los resultados en la hoja 'CTO'

    with pd.ExcelWriter(archivo) as writer:

        df_resultados.to_excel(writer, sheet_name='CTO', index=False)

        # Crea un DataFrame para los resultados de empalmes

        df_empalmes = pd.DataFrame(resultados_empalmes_finales)

        # Agrega la columna 'ORDER' para empalmes con valores alternos 1 y 2

        df_empalmes['ORDER'] = [1, 2] * (len(df_empalmes) // 2)

        if len(df_empalmes) % 2 != 0:

            df_empalmes['ORDER'].iloc[-1] = 1

        # Reorganiza las columnas en el DataFrame de empalmes
```

```
    columnas_empalmes = ['ID-PRE', 'DS', 'NOMBRE CLIENTE','Ciudad', 'Dirección', 'Latitud', 'Longitud', 'ORDER', 'PROPIETARIO' , 'TIPO_PUNTO_DE_ACCESO' , 'ID_EMPALME', 'EMPALME', 'Coordenada_X', 'Coordenada_Y', 'Distancia' , 'Consulta de conexiones Función del Cable lógico de origen' , 'Hilos']

    df_empalmes = df_empalmes[columnas_empalmes].copy()

    # Guarda los resultados en la hoja 'empalmes'

    df_empalmes.to_excel(writer, sheet_name='Empalmes', index=False)


def buscar_en_divisorsitio(lat_buscar, lon_buscar, resultados_divisor_sitio, ciudad_buscar):
  # Inicializa una lista vacía para almacenar las coincidencias encontradas

  distancias = []

  # Estandariza la ciudad que se busca para asegurar una comparación uniforme

  ciudad_buscar = estandarizar_direccion(ciudad_buscar)

  # Recorre cada fila en los resultados de la tabla DivisorSitio

  for row_divisor_sitio in resultados_divisor_sitio:

    # Asegura que la fila tenga al menos 4 elementos antes de proceder

    if len(row_divisor_sitio) >= 4:

      # Estandariza la ciudad de la fila actual

      ciudad_divisor_sitio = estandarizar_direccion(row_divisor_sitio[0])

      # Compara la ciudad de la fila actual con la ciudad buscada

      if ciudad_divisor_sitio == ciudad_buscar:

        # Extrae y corrige las coordenadas de latitud y longitud de la fila actual

        lat_divisor = corregir_coordenada(row_divisor_sitio[2])

        lon_divisor = corregir_coordenada(row_divisor_sitio[1])

        # Obtiene el ID del sitio y lo convierte en un string

        id_sitio = str(int(row_divisor_sitio[3]))
```

```
            # Calcula la distancia entre las coordenadas buscadas y las de la fila actual
            distancia = haversine(lat_buscar, lon_buscar, lat_divisor, lon_divisor)
            # Verifica si la distancia es menor o igual a 500 metros
            if distancia <= 500:
                # Añade la coincidencia a la lista de distancias
                # Incluye la distancia, la fila completa, el ID del sitio, y la fuente de datos (tabla DivisorSitio)
                distancias.append((distancia, row_divisor_sitio, id_sitio, "DivisorSitio"))

    # Devuelve la lista de coincidencias encontradas dentro de 500 metros
    return distancias

def buscar_en_divisorterminal(lat_buscar, lon_buscar, resultados_divisor_terminal, ciudad_buscar):
    # Inicializa una lista vacía para almacenar las coincidencias encontradas
    distancias = []
    # Estandariza la ciudad que se busca para asegurar una comparación uniforme
    ciudad_buscar = estandarizar_direccion(ciudad_buscar)
    # Recorre cada fila en los resultados de la tabla DivisorTerminal
    for row_divisor_terminal in resultados_divisor_terminal:
        # Asegura que la fila tenga al menos 4 elementos antes de proceder
        if len(row_divisor_terminal) >= 4:
            # Estandariza la ciudad de la fila actual
            ciudad_divisor_terminal = estandarizar_direccion(row_divisor_terminal[0])
            # Compara la ciudad de la fila actual con la ciudad buscada
            if ciudad_divisor_terminal == ciudad_buscar:
                # Extrae y corrige las coordenadas de latitud y longitud de la fila actual
```

```
            lat_divisor = corregir_coordenada(row_divisor_terminal[2])
            lon_divisor = corregir_coordenada(row_divisor_terminal[1])
            # Obtiene el ID de la terminal y lo convierte en un string
            id_terminal = str(int(row_divisor_terminal[3]))
            # Calcula la distancia entre las coordenadas buscadas y las de la fila actual
            distancia = haversine(lat_buscar, lon_buscar, lat_divisor, lon_divisor)
            # Verifica si la distancia es menor o igual a 500 metros
            if distancia <= 500:
                # Añade la coincidencia a la lista de distancias
                # Incluye la distancia, la fila completa, el ID de la terminal, y la fuente de datos (tabla DivisorTerminal)
                distancias.append((distancia, row_divisor_terminal, id_terminal, "DivisorTerminal"))
    # Devuelve la lista de coincidencias encontradas dentro de 400 metros
    return distancias


def buscar_en_empalmes(lat_buscar, lon_buscar, resultados_empalmes):
    # Inicializa una lista vacía para almacenar las coincidencias encontradas
    distancias = []
    # Recorre cada fila en los resultados de la tabla Empalmes
    for row_empalmes in resultados_empalmes:
        # Asegura que la fila tenga al menos 5 elementos antes de proceder
        if len(row_empalmes) >= 5:
            # Extrae y corrige las coordenadas de latitud y longitud de la fila actual
            lat_empalmes = corregir_coordenada(row_empalmes[1])
            lon_empalmes = corregir_coordenada(row_empalmes[0])
```

```
        # Extrae otros datos relevantes de la fila
        id_empalme = row_empalmes[2]
        codigo_empalme = row_empalmes[3]
        punto_acceso = row_empalmes[4]
        propietario = row_empalmes[5]
        hilos = row_empalmes[6]
        conexion = row_empalmes[7]
        # Calcula la distancia entre las coordenadas buscadas y las de la fila actual
        distancia = haversine(lat_buscar, lon_buscar, lat_empalmes, lon_empalmes)
        # Añade la coincidencia a la lista de distancias
        # Incluye la distancia, la fila completa, y otros detalles específicos del empalme
        distancias.append((distancia, row_empalmes, id_empalme, codigo_empalme,
lat_empalmes, lon_empalmes, punto_acceso, propietario, hilos, conexion))

    # Ordena las coincidencias por distancia en orden ascendente
    distancias = sorted(distancias, key=lambda x: x[0])
    # Devuelve las dos coincidencias más cercanas
    return distancias[:2]



# Desarrollo del programa
def main():
    connection = None # Inicializa la variable de conexión a la base de datos
    try:
        # Establece una conexión a la base de datos usando pyodbc
        connection = pyodbc.connect(db_file)
```

```
cursor = connection.cursor() # Crea un cursor para ejecutar consultas

print("Conexión exitosa")


# Define las consultas SQL para obtener datos de diferentes tablas

query_fpostes = "SELECT DIRECCIONP, REGION_NAME, COORDENADA_Y, COORDENADA_X, ID_CTO, NAME FROM Postes"

query_patchpannel = "SELECT DIRECCIONP, REGION_NAME, COORDENADA_Y, COORDENADA_X, ID_EQUIPO, NAME FROM PatchPannel"

query_divisor_sitio = "SELECT [Localidad Nombre], [Coordenada_X], [Coordenada_Y], [Sitio ID], [Cable lógico óptico Código] FROM CDS.[dbo].[Divisor_Ampliacion_Sitio_3] WHERE [Divisor óptico - ISP Situación] = '5 - Existente' AND [Sitio Fecha de activación] NOT IN ('')"

query_divisor_terminal = "SELECT [Localidad Nombre], [Coordenada_X], [Coordenada_Y], [Terminal de fibra óptica ID], [Cable lógico óptico Código] FROM CDS.[dbo].[Divisor_Ampliacion_3] WHERE [Divisor óptico - ISP Situación] = '5 - Existente' AND [Terminal de fibra óptica Fecha de la instalación] NOT IN ('')"

query_buscar = "SELECT DIRECCION, CIUDAD, LATITUD, LONGITUD, [ID-PRE], DS, [NOMBRE CLIENTE] FROM VIABILIDAD"

query_empalmes = """

SELECT [Coordenada_X], [Coordenada_Y], [Terminal de fibra óptica ID], [Terminal de fibra óptica Código], [Punto de acceso Tipo de punto de acceso], [Propietario Nombre], [Hilos], [Consulta de conexiones Función del Cable lógico de origen]

FROM [dbo].[Empalmes_VB_V2]

WHERE [Consulta de conexiones Función del Cable lógico de origen] LIKE '11 - %'

OR [Consulta de conexiones Función del Cable lógico de origen] LIKE '18 - %'

OR [Consulta de conexiones Función del Cable lógico de origen] LIKE '19 - %'"""


# Obtiene los datos de las tablas usando la función obtener_datos

resultados_fpostes = obtener_datos(query_fpostes, cursor)
```

```
    resultados_patchpannel = obtener_datos(query_patchpannel, cursor)
    resultados_divisor_sitio = obtener_datos(query_divisor_sitio, cursor)
    resultados_divisor_terminal = obtener_datos(query_divisor_terminal, cursor)
    resultados_empalmes = obtener_datos(query_empalmes, cursor)
    cto_dict = obtener_cto(cursor) # Obtiene un diccionario con datos de CTO
    resultados_buscar = obtener_datos(query_buscar, cursor)
    print(f"Número de filas en resultados_buscar: {len(resultados_buscar)}") # Imprime el número de filas a procesar
    resultados_finales = [] # Lista para almacenar los resultados finales cto
    resultados_empalmes_finales = [] # Lista para almacenar los resultados de empalmes

    # Procesa cada fila de resultados_buscar
    for i, row_buscar in enumerate(resultados_buscar, start=1):
        print(f"Procesando fila {i} de {len(resultados_buscar)}...") # Imprime el progreso

        if len(row_buscar) >= 4:
            direccion_buscar = row_buscar[0] # Obtiene la dirección
            ciudad_buscar = row_buscar[1] # Obtiene la ciudad
            lat_buscar = corregir_coordenada(row_buscar[2]) # Corrige la latitud
            lon_buscar = corregir_coordenada(row_buscar[3]) # Corrige la longitud
            id = row_buscar[4]
            ds = row_buscar[5]
            cliente = row_buscar[6]

            # Lista para almacenar distancias totales
            distancias_totales = []
```

```
        # Buscar coincidencias exactas en FPostes
        distancias_totales.extend(buscar_en_fpostes(direccion_buscar, ciudad_buscar, resultados_fpostes))
        # Si no hay coincidencias exactas, buscar en PatchPannel
        distancias_totales.extend(buscar_en_patchpannel(direccion_buscar, ciudad_buscar, resultados_patchpannel))
        # Si aún no hay coincidencias, buscar e divisor sitio más cercano
        distancias_totales.extend(buscar_en_divisorsitio(lat_buscar, lon_buscar, resultados_divisor_sitio, ciudad_buscar))
        # Si aún no hay coincidencias, buscar en divisor terminal más cercano
        distancias_totales.extend(buscar_en_divisorterminal(lat_buscar, lon_buscar, resultados_divisor_terminal, ciudad_buscar))
        distancias_totales = sorted(distancias_totales, key=lambda x: x[0])
        resultados_seleccionados = []
        id_ctos_usados = set()

        # Procesa las distancias totales para seleccionar los mejores resultados
        for distancia, mejor_row, mejor_id, tabla in distancias_totales:
          if mejor_id not in id_ctos_usados and validar_id_cto(mejor_id, cursor):
            datos_cto = cto_dict.get(mejor_id, [""]*7)
            resultado_dict = {
              'ID-PRE' : id,
              'DS' : ds,
              'NOMBRE CLIENTE' : cliente,
              'Ciudad': ciudad_buscar,
              'Dirección': direccion_buscar,
              'Latitud': lat_buscar,
```

```
                'Longitud': lon_buscar,
                'Resultado': 'Coincidencia encontrada',
                'ID_CTO': mejor_id,
                'CTO': datos_cto[1],
                'OLT_NOMBRE': datos_cto[0],
                'CABLE': datos_cto[2],
                'PROPIETARIO': datos_cto[3],
                'OT': datos_cto[4],
                'PTO_DISPO': datos_cto[5],
                'Tabla': tabla,
                'Distancia': int((distancia*(0.41)+distancia) + 0.999999)
            }
            resultados_seleccionados.append(resultado_dict) # Agrega el resultado a
la lista
            id_ctos_usados.add(mejor_id) # Marca el CTO como utilizado

        if len(resultados_seleccionados) == 2:
            break

    # Asegura que siempre haya al menos 2 resultados por dirección
    while len(resultados_seleccionados) < 2:
        resultados_seleccionados.append({
            'ID-PRE' : id,
            'DS' : ds,
            'NOMBRE CLIENTE' : cliente,
            'Ciudad': ciudad_buscar,
```

```
                'Dirección': direccion_buscar,
                'Latitud': lat_buscar,
                'Longitud': lon_buscar,
                'Resultado': 'No encontrada',
                'ID_CTO': '',
                'CTO': '',
                'OLT_NOMBRE': '',
                'CABLE': '',
                'PROPIETARIO': '',
                'OT': '',
                'PTO_DISPO': '',
                'Tabla': '',
                'Distancia': ''
            })

        # Busca empalmes cercanos y agrega los resultados a la lista final
        resultados_finales.extend(resultados_seleccionados)
        empalmes_cercanos = buscar_en_empalmes(lat_buscar, lon_buscar, resultados_empalmes)
        for distancia, row_empalmes, id_empalme, codigo_empalme,lat_empalmes, lon_empalmes, punto_acceso, propietario, hilos, conexion in empalmes_cercanos:
            resultados_empalmes_finales.append({
                'ID-PRE' : id,
                'DS' : ds,
                'NOMBRE CLIENTE' : cliente,
                'Ciudad': ciudad_buscar,
                'Dirección': direccion_buscar,
```

```
                'Latitud': lat_buscar,
                'Longitud': lon_buscar,
                'ID_EMPALME': id_empalme,
                'EMPALME': codigo_empalme,
                'Distancia': distancia,
                'Coordenada_X':lon_empalmes,
                'Coordenada_Y':lat_empalmes,
                'TIPO_PUNTO_DE_ACCESO':punto_acceso,
                'PROPIETARIO':propietario,
                'Hilos': hilos,
                'Consulta de conexiones Función del Cable lógico de origen': conexion
            })

    archivo_resultados = 'ENTREGA VIABILIDAD FTTH.xlsx' # Nombre del archivo Excel donde se guardarán los resultados
    guardar_resultados(resultados_finales, archivo_resultados, resultados_empalmes_finales)
    print("Resultados guardados en:", archivo_resultados)

  except Exception as e:
    print(f"Error: {e}") # Imprime cualquier error que ocurra durante la ejecución
  finally:
    if connection:
      connection.close()

if __name__ == '__main__':
```

main()