SQL and Network Analysis Project - Karl Merisalu

Background: One effective way to simplify understanding of organisational structures (or for any relational dataset) is to produce organisational charts. Many large organisations have such systems in place to explore most frequently accessed reporting lines - for example personnel org charts.

Problem: However, if one is in need of making sense of a large network dataset quickly where such a system is not in place, there is a problem. When the dataset is large, it becomes very time consuming to understand its connection.

Task: This notebook aims to tackle this problem by:
1) Simplifying the data cleanup process by creating an SQL database and a query to request relevant data,
2) Building a hieranchical org chart generator based on the data pulled by the SQL query

This analysis is important, because in my role as a Business Manager I need to create Legal Entity organisational charts of Region EMEA (Europe, Middle East, Africa) countries on a regular basis for presentation to Senior Management. This tool will greatly reduce the time needed for creating and verifying such charts, because as of now, it is being done manually.

Limitations: This analysis is conducted on UBS proprietary data (Legal Entities) and as such data tables will be limited. Where this is not possible, internal data will be edited/hidden to cover sensitive details.

Summary of actions: To summarize, this is an SQL and Network Analysis project. This notebook consists of 2 parts:
1) Manipulating raw excel data in SQL and,
2) Followed by a Python script and a function to process through nodes and edges in a dataset to plot it in a simple hierarchical network chart.
The goal of this project is automating a manual time-consuming task. It will help to avoid errors and save time. The dataset is internal and as such edited to mask sensitive details. In the end, I will conclude with project findings and provide suggestions for improvements.

Following is a step by step guide on how I'm going to accomplish it:
1) Setting up the environment *(importing libraries to enable me to use pre-developed complex functions on the data)*
2) Creating an SQL data pipeline *(importing data, creating a database and manipulating data for it to be suitable for following analysis)*
3) Defining functions to process raw data *(which work through data and transform it to a suitable state for plotting)*
4) Data processing *(using the defined functions to process data)*
5) Plotting the org chart *(visualisation of the outcome)*
6) Conclusion *(summary findings and suggestios for improvements)*

1) Setting up the environment

I will first import all libraries I need to process data and plot the chart

In [1]:
```
# for data manipulation
import numpy as np
import pandas as pd

# for creating nodes, edges and list of paths between nodes
import networkx as nx

# for plotting the network chart
import anytree
from anytree import Node, RenderTree
from anytree.exporter import DotExporter
from anytree.dotexport import RenderTreeGraph
from graphviz import Source

# for SQL queries in python
import sqlite3
```

2) Creating an SQL data pipeline

SQL data pipeline will help me to efficiently join and manipulate the 2 Excel data tables used for the task. This will help me to save a lot of time and avoid manual copy-pasting errors. 'df_det' will include details about Legal Entities, 'df_sub' will include Legal Entity Parent-Subsidiary relationships

In [2]:
```
# importing the 2 required datasets downloaded from company databases
df_det = pd.read_csv("CSVcompaniesDETAIL.csv", encoding = "ISO-8859-1")
df_sub = pd.read_csv("CSVcompaniesSUB.csv", encoding = "ISO-8859-1")

df_det.head(2)
```

Out[2]:

	LSDB ID	Registered Name	Status	Lifecycle	Data Owner	Entity Type	UBS Governed	LSC Process Type	Underlying SPE or Cell	Dome Bra
0										
1										

2 rows × 41 columns

In [3]:
```
# setting matching ID's as index for both datasets
df_det.set_index("LSDB ID", inplace=True)
df_sub.set_index("Subsidiary Entity ID", inplace=True)
```

In [4]:
```
# creating an empty database for data and sql queries
conn = sqlite3.connect("companies.db")
```

In [5]:
```
# adding both datasets from excel as tables to the new database
df_det.to_sql("details", conn, if_exists="replace")
df_sub.to_sql("subsidiaries", conn,  if_exists="replace")
```

```
\\ubsprod.msad.ubs.net\UserData\MERISALK\Home\Miniconda3_64bit\envs\pyt367\lib\site-packages\pandas\core\generic.py:2712: UserWarning: The spaces in these column names will not be changed. In pandas versions < 0.14, spaces were converted to underscores.
  method=method,
```

In [6]:
```
# checking if table shave appeared in the database
pd.read_sql_query("""
SELECT name FROM sqlite_master
WHERE type='table'
ORDER BY name
""", conn)
```

Out[6]:

	name
0	details
1	subsidiaries

In [7]:
```
# performing a quick test query
df = pd.read_sql_query("""
SELECT *
FROM details
""", conn)

df.head(2)
```

Out[7]:

	LSDB ID	Registered Name	Status	Lifecycle	Data Owner	Entity Type	UBS Governed	LSC Process Type	Underlying SPE or Cell	Dome Bra
0										
1										

2 rows × 41 columns

In [8]:
```
# creating a query to combine 2 tables in the database and filter according to required met
rics to make it as easy as possible to manipulate later with python
df2 = pd.read_sql_query("""
SELECT s.'Registered Name of Subsidiary Owner', s.'LSDB ID', s.Subsidiary, s.'Subsidiary En
tity ID', d.Status, d.Lifecycle, d.'Entity Type', d.'Country of Incorporation', d.'Org Leve
l 1 (Primary Business Division)'
FROM subsidiaries s
LEFT JOIN details d
ON s.'Subsidiary Entity ID' = d.'LSDB ID'
WHERE d.Status='Existent' AND d.Lifecycle = 'Operative' AND (d.'Entity Type' = 'Branch (Sig
nificant)' OR d.'Entity Type' = 'Significant Group Entity' OR d.'Entity Type' = 'UBS Group'
OR d.'Entity Type' = 'Branch (Other)' OR d.'Entity Type' = 'Significant Regional Entity' OR
d.'Entity Type' = 'Other UBS Entity' OR d.'Entity Type' = 'Other UBS Entity (Medium Risk)'
 OR d.'Entity Type' = 'Representative Office')

""", conn)

df2.head(2)
```

Out[8]:

	Registered Name of Subsidiary Owner	LSDB ID	Subsidiary	Subsidiary Entity ID	Status	Lifecycle	Entity Type	Country of Incorporation	Org Level 1 (Primary Business Division)
0									
1									

A brief explanation on some of the chosen columns:
Registered Name of Subsidiary Owner - Parent Legal Entity of the subsidiary
Subsidiary - Subsidiary Legal Entity owned by parent in the same row
Status - Whether the subsidiary entity is existing or dormant for example
Lifecycle - Whether the subsidiary entity is operating on being dissolved for example
Entity Type - what kind of subsidiary entity we're dealing with / ent
Org Level 1 - primary business division of the subsidiary entity

Comment: I now have a dataset, which I can directly use later to build a Legal Entity chart. Moreover, the above steps are replicable very quickly - all I need, is to download the 2 updated datasets in Excel/CSV and save them in the right folder and run the algorithms.

3) Defining functions to process raw data

3.1 First, I define a function which takes a list input (all paths from the parent node to all child nodes) and converts the list into suitable format for plotting. I define this function first because it will be used by another function later on.

In [9]:
```
def list_to_anytree(lst):
        root_name = lst[0][0]
        root_node = Node(root_name)
        for branch in lst:
            parent_node = root_node
            assert branch[0] == parent_node.name
            for cur_node_name in branch[1:]:
                cur_node = next(
                    (node for node in parent_node.children if node.name == cur_node_name),
                    None,
                )
                if cur_node is None:
                    cur_node = Node(cur_node_name, parent=parent_node)
                parent_node = cur_node
        return root_node
```

3.2 Second, I define a function which iterates through csv data and creates list of paths from the parent node to all child nodes. Detailed description of how the function works is in code as comments:

In [10]:
```
def org_chart(df, parent_col, child_col, parent, depth=None):

# adding comments below to describe how to use the function
    """Function inputs: df=dataset (dataframe), parent_col="name of parent column in datase
t",
    child_col= name of child column in dataset, parent="root node in parent column",
    depth=max number of children (default=None) """

# Below block of code: 1) creates an empty list (working_list), 2) then iterates through al
l columns in the initial dataset searching for
# user inputted root/parent node in the parent column, 3) once I have root node, I add it
together with it's child node
# into created empty list ([parent, child]). As a result I get a complete list of root nod e
and its child node pairs.

    working_list = []
    for i,j in df.iterrows():
         if df[parent_col][i] == parent:
            working_list.append([df[parent_col][i], df[child_col][i]])

# Below block of code: 1) iterates through populated working_list, 2) takes the child of th
e root node and checks if the child has any
# grandchildren in the initial dataset (df), 3) if grandchildren exist, I add a the pair o n
the bottom of working_list
# as [child, grandchild]. If the pair already exists in working_list I skip adding it an c
ontinue to deal with circular references.
# I repeat this process until there are no more children to add. As a result I have a
# complete list of parent-child relationships existing (only) below the root node.

    for i,j in working_list:
        for k,l in df.iterrows():
            if df[parent_col][k] == j:
                if [j, df[child_col][k]] in working_list:
                    continue
                else:
                         working_list.append([j, df[child_col][k]])

# I need to have complete paths from root to the very grand of children instead of
# pairwise relationships, as I have now. I can achieve this with networkx library.

# To use nx library, I first define all nodes, which is essentially unique values from wor
king_list
    nodes = set(x for l in working_list for x in l)

# Next I define that I'm dealing with a directed graph (DiGraph) - important
# to determine parent and child to avoid vice versa relationship.
    DG = nx.DiGraph()

# Adding all nodes to nx.DiGraph()
    DG.add_nodes_from(nodes)

# Next I add edges (relationship between nodes) to nx.DiGraph()
    for i,j in working_list:
        DG.add_edge(i, j)

# Creating a new list for full paths between root and child nodes.
    path_list = []

# nx.all_simple_paths gives me a path in a list form, from source (root/parent) to target n
ode.
# The loop below iterates through all nodes and adds all created paths from source to the e
nd node,
# to path_list

    for i in nodes:
        for path in nx.all_simple_paths(DG, source=parent, target=i, cutoff=depth):
            path_list.append(path)

# I list_to_anytree function and feed it the path list I just created.
    chart_data = list_to_anytree(path_list)
```

```
# Returning suitably formatted chart_data to create a dotfile and then plot.
    return chart_data
```

4) Data processing

We're going to use org_chart function which I defined in step 2 to process raw data. But first, let's inspect the function to see which variables to select

In [11]:
```
org_chart?
```

Now that I have info about the function I call it with following inputs and save it into a new variable tree_chart:
dataset = df,
parent column = 'parent',
child column = 'subsidiary',
root node of interest = 'XXX',
max children/depth = None *(to see the path all the way down)*

In [12]:
```
tree_chart = org_chart(df2, 'Registered Name of Subsidiary Owner', 'Subsidiary', 'XX
X', depth=None) # use depth = None for max chart depth
```

5) Plotting the org chart

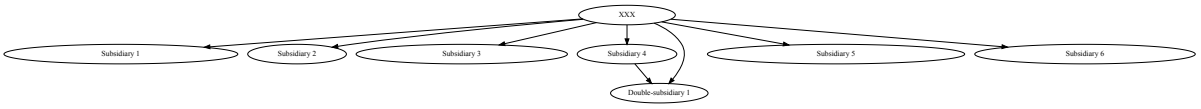
To plot the chart I first create a dotfile out of treechart, which according to http://www.graphviz.org/ (http://www.graphviz.org/) is:
dot - "hierarchical" or layered drawings of directed graphs. This is the default tool to use if edges have directionality.

Then as a final step I plot the chart from dotfile using graphviz Source.

In [13]:
```
DotExporter(tree_chart).to_dotfile('orgchart.dot')

Source.from_file('orgchart.dot')
```

Out[13]:



6) Conclusion

The chart above helps me to get a quick high level overview of UBS' (or any other directional network's) org chart. Higher level parents are on the top and subsidiaries/childs are portrayed down. Root node or ultimate parent is 'XXX' in this case.

This code can save me a lot of time when trying making sense of directional networks as opposed to analysing raw pairwise data manually (with excel for example).

Although the chart conveniently portrays well formatted org chart, there are several further developments that could be considered.

6.1 Further research and improvements

More advanced graphs/specifications:

1) The plot could be used to deliver more information than just the network structure. For example in the case of UBS, it would be interesting to see which of the companies are aligned to 1) IB, 2) WM, 3) based in specific countries and etc. Such node features could be marked with different colours of the chart.
2) Additionally, various filtering options could be added, for example to show a chart of entities in certain countries