

ZERTIFIKATSKURS DATA LIBRARIAN 2021/22

Modul 6

Erstellung eines Netzwerkgraphen aus Personendaten
der Hamburg-Bibliographie mit der
Python-Bibliothek Pyvis

Heinke Schumacher
Staats- und Universitätsbibliothek Hamburg
Von-Melle-Park 3
20146 Hamburg



Inhaltsverzeichnis

1. Einleitung.....	1
2. Daten.....	1
2.1 Originaldaten	1
2.2 Transformation	2
3. Erstellung und Visualisierung des Graphen	2
4. Diskussion.....	3
5. Schluß	5
Quellen.....	6
Anhang.....	7
Anhang 1: Beispiel Originaldaten	8
Anhang 2: Beispiel JSON-Datei	9
Anhang 3: Hauptskript „data_transformation.ipynb“	10
Anhang 4: Hilfsskript „hamburg_edges.ipynb“	23
Anhang 5: Skripte variante Visualisierungen „variant_visualizations.ipynb“	25

Materialien und Ergebnisse:

- schumahe.github.io/hhbib_networkGraph
- github.com/schumahe/hhbib_networkGraph

1. Einleitung

Die Hamburg-Bibliographie ist das zentrale Nachweisinstrument der Freien und Hansestadt Hamburg für die Vermittlung von Literatur-, Personen- und Körperschaftsdaten zur Stadt und ihrer Region. Sie verzeichnet die Daten seit etwa 1994 im Produktivsystem des Gemeinsamen Bibliotheksverbundes GBV, der jetzigen Verbunddatenbank K10plus. Die dort als Normdaten erstellten Personen- und Körperschaftsdaten bilden u.a. die Grundlage für Recherchen in den Discoverysystemen der Bibliothek. Von der Gesamtzahl der in der Verbunddatenbank nachgewiesenen Normdaten der Hamburg-Bibliographie sind etwa 20.000 Hamburger Personendaten, rund 600 davon verzeichnen etwa 900 Verwandtschaftsbeziehungen in ihren jeweiligen Normdatensätzen, nahezu alle Berufsangaben. Diese Datenlage könnte für eine visuelle Vermittlung dieses informativen Datenbestandes zu Hamburger Persönlichkeiten in Form eines Netzwerkgraphen genutzt werden. Als Idee zur Verwendung eines solchen Netzwerkgraphen ist ein Teasern und Eyecatchen etwa auf der Plattform des eigenen, sich momentan im Relaunch befindenden Discoverysystems, gut denkbar. Dabei kann der Graph z.B. eine Verknüpfung zusammengehöriger Familienmitglieder darstellen oder aber auch die Verbindung von Personen und Berufe visualisieren. Für die Visualisierung von Netzwerkgraphen gibt es eine Reihe von Software-Tools und -Bibliotheken, wie etwa Gephi, Cytoscape, networkX, Plotly oder Pyvis. Die vorliegende Arbeit soll den Prozess der Erstellung eines solchen Netzwerkgraphen mit Hilfe der Python-Bibliothek Pyvis aufzeigen.

2. Daten

Zur Erstellung eines Netzwerkgraphen werden Daten in einer bestimmten Datenstruktur benötigt. Den Ausgangspunkt zur Herstellung dieser Datenstruktur bildet der von der Verbundzentrale des GBV an die Bibliothek gelieferte Datenabzug der K10plus-Normdaten der Hamburg-Bibliographie. Durch eine Transformation werden diese Originaldaten zur späteren Verwendung in der Software-Bibliothek Pyvis vorbereitet. Im Folgenden sollen die Originaldaten sowie der Transformationsprozess beschrieben werden.

2.1 Originaldaten

Die Originaldaten werden in einer Datei geliefert, die sämtliche, Normdatensatzarten enthält, die die Hamburg-Bibliographie erzeugt, sie beinhaltet also nicht nur Personennormdaten. Es handelt sich um eine Textdatei in dem verbundeigenen Datenformat .pp (picaplus). Die Textdatei bildet die Datensatzstruktur des Produktivkataloges K10plus ab (vgl. Anhang 1). So sind die Inhalte der Normsätze entsprechend der Katalogisierungsrichtlinie des K10plus in einzelnen Feldern abgelegt, ggf. durch Codierungen unterteilt in Subfelder. Jedes Feld entspricht einer Zeile in der Textdatei, strukturiert in Feldbenennung und Feldinhalt, Unterfelder werden durch schwarze Buttons (Sectionsplitter) markiert, die später im Transformationsprozess als Leerzeichen interpretiert werden. Anfang und Ende eines Datensatzes sind gekennzeichnet durch je zwei Leerzeilen. Die Abfolge der Felder ist für jeden Datensatz gleich. Alle Datensätze beginnen mit dem gleichen Feld. Für die Erzeugung des Netzwerkgraphen zwingend benötigt werden die Felder „041A“, „047A“ (Berufsangaben aus Altdaten) sowie „041R“ (mit der Codierung „4bezf“ bzw. „4beru“). Sie sind die Felder, die für die spätere Visualisierung des Netzwerkgraphen direkt in Beziehung gesetzt werden müssen. Potentiell verwendbar sind „047A/02“, „041R“ (mit der Codierung „4ortg“/ „4orts“, „4affi“, „4bezb“) und „060R“

für zusätzliche biographische Informationen der Personen (vgl. Anhang 1). Sie werden bei der Transformation im Skript berücksichtigt, auch wenn sie zunächst keine direkte Verwendung bei der Erstellung der Visualisierung finden sollten, quasi als vorausgedachte Ausbaustufe. Zusätzlich wird die Ziffernfolge in Feld „003@“ (die ID jedes Datensatzes) beibehalten als Anker zu den Originaldaten, denn während des Wandlungsprozesses ist ein Prüfen und Abgleichen mit den Originaldaten unumgänglich, die ID ermöglicht einen schnellen Zugriff.

2.2 Transformation nach JSON

Die als Textdatei gelieferte Ausgangsdatei mit den Normdaten der Hamburg-Bibliographie muss zur späteren Verwendung in der Python-Bibliothek Pyvis in eine JSON- oder csv-Datei transformiert werden, hier wird in eine JSON-Datei gewandelt. Diese Umformung wurde in Jupyter-Notebooks mit der Programmiersprache Python vorgenommen. Zum detaillierten Nachvollziehen und besseren Verständnis der folgenden Erläuterungen kann das als Anhang 3 beigegebene annotierte Skript sowie ein Ausschnitt der JSON-Enddatei in Anhang 2 hinzugezogen werden. An dieser Stelle werden die nötigen Umwandlungsschritte nur grob umrissen, jedoch nicht unbedingt in der Abfolge wie sie im Skript erfolgen.

In der Ausgangsdatei „hhb_td_2022-01-19“, die die gesamten Normdatensatzarten der Hamburg-Bibliographie enthält, sind außer den Normsätzen zu Personen also ebenso weitere Normsatzarten verzeichnet, die für die Visualisierung nicht benötigt werden, sie müssen im Umwandlungsprozess herausgefiltert werden. Gleichermaßen gilt für nicht zu nutzende Felder eines jeden Personendatensatzes, erhalten bleiben die in Anhang 1 aufgeführten. Weiterhin müssen die Zeilen, die zu einem Datensatz gehören, in die JSON-gemäßige Struktur von Key/Value-Paaren gewandelt sowie die Key/Value-Paare eines Datensatzes zu einem Dictionary zusammengefasst werden, so dass eine Liste von Dictionaries das Endprodukt ist. Zur Vermeidung eines Überschreibens von gleichnamigen Keys (z.B. „041R“) bei der Dictionary-Bildung bedarf es einer Individualisierung dieser Keys, als Anker bei diesem Prozess dient der als erster Key in jedem Datensatz enthaltene Key „001A“. Die Values der entstandenen Dictionaries müssen eine Listenstruktur haben, auch bei fehlenden Values muss eine leere Liste generiert werden, damit bei einer Iteration darauf zugegriffen werden kann. Ist die JSON-Struktur für die Datensätze hergestellt, kann die Einzelbehandlung der jeweiligen Values erfolgen: Befreiung von nicht benötigten Daten aus Subfeldern, Löschen von Codierungen, ggf. Hinzufügen von Text. Dies ist besonders wichtig bei den Values von „041A“ und „041R_bezf“ bzw. „041R_beru“, die später bei der Erstellung des Graphen miteinander in Beziehung gesetzt werden sollen: ihre Values müssen deckungsgleich sein, damit ein Abgleich stattfinden kann und keine Doppeleinträge und dadurch doppelte Nodes entstehen. Als letzter Schritt erfolgt zwecks griffiger Visualisierung eine aussagekräftigere Benennung der Keys.

Der Transformationsprozess hat auch Datenunstimmigkeiten aufgrund von Altdaten, die nicht der normierten Datenstruktur entsprechen, und Eingabefehlern aufgezeigt. Dies verdeutlicht noch einmal profiliert, auch wenn dies im passiven Bewußtsein jedem Datenbibliothekar klar sein sollte, dass für eine automatisierte Behandlung von Daten das A und das O saubere Originaldaten sind, möglichst feingliedrig codiert und in separierten Feldern abgelegt.

3. Erstellung und Visualisierung der Graphen

Die Generierung und die Visualisierung des Graphen mit den dazugehörigen Nodes und Edges wird von der Python-Bibliothek Pyvis übernommen. Pyvis ist eine Bibliothek, die die Bibliothek vis.js als

Grundlage hat, und mit der Intention entwickelt wurde „[...] to build a python based approach to constructing and visualizing network graphs in the same space.“ (Pyvis 2022a). Mit der Bibliothek kann mit den Funktionen des Graphen und denen von Nodes und Edges direkt aus dem Skript heraus eine html-Datei erzeugt werden. Sie kann im Browser geladen werden und das Ergebnis lässt sich als Webpage ansehen. Bei Bedarf können mit networkX erstellte Graphen mit der Funktion from_nx() ins Pyvis-Format übersetzt werden. Besonders erwähnenswert für die Vermittlung der Graphen ist die show_buttons()-Funktion, die Schiebereglern generiert, mit denen Betrachter Werte und Parameter des Graphen GUI-basiert ändern können.

Die im ersten Schritt erstellte JSON-Datei mit den Daten zu den Hamburger Personen bildet die Iterationsgrundlage zur Erzeugung von Nodes und Edges für die Visualisierung (vgl. Anhang 3), nachdem der Graph in einer eigenen Funktion definiert wurde. Ein Netzwerkgraph verknüpft mehrere Informationen visuell miteinander, hier setzt die Iteration über diese Datei die Values der Keys „name“ und „relation_fam“ oder „job“ mit den zugehörigen Funktionen „add_node()“ oder „add_edge()“ in Beziehung. Der Graph kann mit vier verschiedenen Standardalgorithmen generiert werden, die jeweils andere Parameter für die Visualisierung aufweisen und die mit der Funktion „set_options()“ durch erweiterbare Optionen aus der Library vis.js modifiziert werden können (Pyvis 2022b), besonders ansprechend ist dabei, die Möglichkeit mit dem Parameter „physics“ den Nodes eine Bewegungsdynamik zuzuweisen. Besonders erwähnenswert für die Vermittlung der Graphen ist darüber hinaus die show_buttons()-Funktion, die Schiebereglern generiert, mit denen Betrachter Werte und Parameter des Graphen GUI-basiert ändern können, allerdings sind diese Regler visuell nicht besonders ansprechend.

Neben dem Ausloten dieser Standardmöglichkeiten und dem Spielraum bei den Value-Arguments für Nodes und Edges sind zwei Modifikationen hinzugekommen, die weitere Gestaltungsmöglichkeiten für die Visualisierung bieten. Einerseits kann eine Farbgestaltung der Nodes mit zufällig generierten Hex-ColorCodes stattfinden, andererseits kann durch einen einzelnen Node, der mit jeweils einem Node der Familien verknüpft ist, ein visueller Ankerpunkt geschaffen werden.

Die farbliche Gestaltung mit den Hex-Color-Codes wird durch die Definition einer Variablen erzeugt, die mit Hilfe des Python-Moduls „Random“ Farbwerte generiert (vgl. AskPython 2022), die dann über den Wert des „color“-Arguments den Nodes zugewiesen werden. Dadurch dass die Nodes der Hauptpersonen und der in Beziehung zu setzenden Familienmitglieder abhängig iteriert werden, wird allen Familienmitgliedern die gleich Farbe zugewiesen. So gibt es die Möglichkeit, visuell Familiencluster zu schaffen.

Bedingt durch die Datenlage bei den Personen der Hamburg-Bibliographie, in der zwar Familienmitglieder untereinander verknüpft werden, jedoch nicht Familien untereinander, entstehen im Graphen jeweils Subnetze für die einzelnen Familien. Um ein verbindendes Element herzustellen, das allen Familien gemeinsam ist, entstand die Idee eines einzelnen „Hamburg-Nodes“ als Mittelpunkt des Graphen. Jede Familie ist mit einem Familienmitglied mittels eines Edges mit diesem Node verbunden. Es besteht auch die Möglichkeit, alle Personennodes mit dem „Hamburg-Node“ zu verbinden. Das ist zwar gestalterisch interessant, macht aber das Handling des Graphen etwas unübersichtlich, da die Familienverbindungen aufgrund der Vielzahl der Edges dann nur noch mühsam zu erkennen sind. Für diese „Familienverknüpfung“ wurde eine Liste anhand der erstellten JSON-Datei erzeugt (vgl. hamburg_edges.ipynb, Anh. 4), die die Values der Keys „name“ und „relation_fam“, die die Nodes generieren, derart filtert, dass nur noch eine Person einer Personengruppe mit jeweils gleichem Nachnamen in dieser Liste „hamburg_edges.txt“ erhalten bleibt.

Für die Erzeugung eines Graphen mit den zugehörigen Nodes und Edges wurde meistens die JSON-Datei datenreduziert. Bei einer Berücksichtigung aller Daten traten beim Laden des Graphen extrem lange Ladezeiten auf. Daher wurden die Daten dahingehend vermindert, dass die relationierten Angaben bei Familienmitgliedern oder Berufen mindestens eine Anzahl von zwei haben sollten. Das schafft gleichzeitig auch einen interessanteren visuellen Eindruck, da größere Subnetze entstanden.

Als Ergebnisse des Visualisierungsprozesses werden neun Beispielgraphen auf https://schumahe.github.io/hhbib_networkGraph/ präsentiert, Erläuterungen und Parameter ihrer Erstellung finden sich einerseits in den varianten Skripten in Anhang 5, andererseits unter „About“ unter der genannten Webadresse. Da hinter dem Vorhaben, einen Netzwerkgraphen mit den Daten der Hamburg-Bibliographie zu erstellen, die Idee stand, damit auch ein Vermittlungseffekt in die Öffentlichkeit zu ergründen, ist diese Projektarbeit auch als ein Ausloten und Ausprobieren der Visualisierungsmöglichkeiten zu verstehen, es werden als Ergebnisse auf der Präsentationswebsite daher zum Teil auch Visualisierungen aufgeführt, die in der Vermittlung der Information, die über den Graphen stattfinden kann, nicht unbedingt praktikabel sind.

4. Diskussion

Die Erstellung der JSON-Datei aus den Personendaten der Hamburg-Bibliographie sowie der mit dieser Datei generierte Netzwerkgraph bildet den Arbeitsprozess für diese Projektarbeit ab. Der folgende Überblick diskutiert Ergebnisse und zeigt darüber hinaus Ansätze auf, inwieweit und in welche Richtungen dieses Projekt mit den Kenntnissen der Projektausführenden ausbaufähig wäre. Für die im ersten Arbeitsschritt produzierte JSON-Datei haben sich nach der Visualisierung des Netzwerkgraphen zwei potentielle weitere Arbeitsschritte ergeben, die das visuelle Endergebnis verbessern könnten. Zum einen wäre eine Erweiterung der Biogramme unter dem Value des Keys „bio“ um die weiteren biographischen Angaben, die in der Datei mitgenommen wurden, denkbar. Dabei handelt es sich um die zum Wohle der Grapherstellung gelöschten Verwandtschaftsangaben in dem zu relationierenden Value des Keys „relation_fam“ sowie um die Lebens- und Sterbedaten und –orte in „place_birth_death“ und „date_exact“/„date_simple“. Diese Angaben in die Biogramme zu integrieren hätte einen informative Mehrwert, da die Biogramme der Personen/Nodes mit dem Argument „title“ der add_node()-Funktion sichtbar gemacht werden können. Zum anderen haben die Lebensdaten in den Values der Keys „date_*“ in den Originaldaten ausserordentlich verschiedenartige Platzhalter für nicht vorhandene Daten, so dass es sich lohnt - insbesondere bei einer Überführung in die Biogramme – diese einer genauen Analyse zu unterziehen und zu vereinheitlichen. Im Zusammenhang mit ihrer inhaltlichen Erweiterung ist auch die visuelle Verbesserung der Anzeige der Biogramme im Netzwerkgraphen augenfällig. Die Biogramme werden in einem Kästchen beim Hovern über die Nodes angezeigt. Dieses Kästchen ist so lang, wie der ungebrochene String des Values, aus dem sie generiert werden. Da die Biogramme für die Personen der Hamburg-Bibliographie zum Teil relativ lang sind, befinden sich Teile davon außerhalb der Anzeige des Graphen. Eine Veränderung des Textes durch Einfügen von Zeilenbrüchen etwa für den Pool der node.items wäre hierfür eine Lösung.

Ein weiterer Punkt, der die Anzeige der Biogramme betrifft, führt aller Wahrscheinlichkeit nach auf die Wahl des JSON-Formates als Grundlage für die Grapherstellung zurück. Die Biogramme werden nur bei den Nodes angezeigt, von denen die Relation ausgeht (Key „name“), nicht aber bei den relationierten Nodes (Key „relation_fam“) bzw. dort werden bei Aktivierung des Funktionsarguments durch die Abhängigkeit bei der Iteration die der übergeordneten Person angezeigt. Datenstrukturell ist das plausibel, denn im Dictionary einer Person gibt es nur ein Biogramm, jenes der Person, die im Value „name“ steht. Zwar ist jede Person mit einem eigenen Dictionary in der JSON-Datei vorhanden, weil es für jede auch einen eigenen Datensatz in der Hamburg-Bibliographie gibt, aber beim Iterationsabgleich wird vermutlich beim ersten Auftreten eines Namens, egal in welchem Key, ein Node erzeugt und bei allen späteren Vorkommnissen mit gleichem Valueinhalt nur noch abgeglichen, um keine doppelten Nodes entstehen zu lassen. Möglicherweise wäre dies bei einer csv-

Struktur anders, in der in der „Source“-Spalte (hier also der Referenzvalue „name“) die Einträge gedoppelt werden müssen, sollten sie mehrere Relationentsprechungen haben. Dies auszuprobieren wäre einen Versuch wert.

Betrachtenswert ist auch das Ergebnis für die Visualisierung des „Hamburg-Nodes“ und seinen Edges. Wünschenswert wäre dort, den Node an den unteren Rand der Graphleinwand zu lokalisieren und dann in einem Halbkreis die Edges erstrahlen zu lassen. Dies kann durch die Besetzung des Positionsarguments erzielt werden, ist aber bislang bei allen meinen Versuchen mißlungen. In Bezug auf den „Hamburg-Node“ ist die Liste zur Erzeugung der dazugehörigen Edges zu betrachten (vgl. „hamburg_edges.ipynb“): dort wird der Pool der Gesamtpersonen auf jeweils ein Familienmitglied reduziert, der dann das Ziel für das vom „Hamburg-Node“ ausgehenden Edge ist. Im gewählten Verfahren wird nicht berücksichtigt, dass wenn zwei Familien den gleichen Nachnamen haben, eine von ihnen nicht mit einem Edge verbunden wird, weil doppelte Nachnamen überschrieben werden. Darüberhinaus bekommen Familien mit Mitgliedern unterschiedlicher Namen zwei Edges. Diesbezüglich muss noch einmal überlegt werden, wie dies bei der Listenbildung zu vermeiden ist.

In einem letzten Punkt soll erneut die Wichtigkeit von sauberen Originaldaten in den Fokus gestellt werden. In den Daten der Hamburg-Bibliographie gibt es Personen mit exakt gleichen Namen, die jedoch unterschiedliche Individuen sind. Da diese Personen kein Unterscheidungsmerkmal – etwa einen Homonymzusatz – haben, werden für sie selbstreferenzielle Nodes erstellt, also Nodes, die mit ihrem Edge auf sich selbst weisen. Dieses Problem kann nur durch die Veränderung der Originaldaten behoben werden, es sei denn, man wolle auf diese Personen in der Anzeige verzichten.

5. Schluß

Die Erfahrung mit der Python-Bibliothek Pyvis in dieser Projektarbeit hat sich für mich als einer Person mit sehr geringen Programmierkenntnissen als sehr wirkungsmächtig herausgestellt. Mit relativ wenigen Stellschrauben lassen sich mit den angebotenen Funktionen und Parametern Netzwerkgraphen erstellen, die interaktiv mit Drag-, Drop- Move-Möglichkeiten ausgestattet und visuell ansprechend sind. Bei der Erstellung war dieser visuelle Aspekt ein Ankerpunkt, um ggf. eine Gelegenheit zu schaffen, auf die Daten der Hamburg-Bibliographie aufmerksam machen zu können, die Optik der Graphen spielte also keine untergeordnete Rolle. In diesem Sinne sind die Ergebnisse auch als ein Ausloten der Visualisierungsoptionen zu verstehen mit dem Hintergrund, eine Einschätzung für eine Anwendung im Ernstfall zu gewinnen. Und sollten gar Personen damit arbeiten, die das Programmierinstrumentarium virtuos beherrschen, so können unter Einbeziehung vielfältiger Erweiterungen visuelle Eindrücke erschaffen werden, die bemedenswert jenseits der Ergebnisse dieser Projektarbeit liegen.

Quellen

- [AskPython 2022]** AskPython. Customizing the Pyvis Interactive Network Graphs. 2022. URL: <https://www.askpython.com/python/examples/customizing-pyvis-interactive-network-graphs>
- [Devaux 2019]** Devaux, Elise: List of graph visualization libraries. 2019. URL: <https://elise-deux.medium.com/the-list-of-graph-visualization-libraries-7a7b89aab6a6>
- [Grandjean 2015]** Grandjean, Martin: Network visualization: mapping Shakespeare's tragedies. 2015. URL: <http://www.martingrandjean.ch/network-visualization-shakespeare/>
- [Mayank, Mohit 2021]** Mayank, Mohit: Visualizing Networks in Python. A practical guide to tools which helps you "see" the network. 2021. URL: <https://towardsdatascience.com/visualizing-networks-in-python-d70f4cbeb259>
- [Nápoles Duarte 2021]** Nápoles Duarte, José Manuel: Making network graphs interactive with Python and Pyvis. 2021. URL: <https://towardsdatascience.com/making-network-graphs-interactive-with-python-and-pyvis-b754c22c270>
- [NetworkX 2022]** NetworkX. 2022. URL: <https://networkx.org/documentation/stable/>
- [Perrone 2020]** Perrone, Giancarlo; Unpingco, Jose; Lu, Haww-min: Network visualizations with Pyvis and VisJS. In: Proceedings of the 19th Python in Science Conference (SciPy 2020). Austin 2020, S. 58-62. URL: https://conference.scipy.org/proceedings/scipy2020/pdfs/giancarlo_perrone.pdf. DOI: 10.25080/Majora-342d178e-008
- [Python 2022a]** Python.Wiki. 2022. URL: <https://docs.python.org/3/>
- [Python 2022b]** Python 3.10.6 documentation. 2022. URL: <https://wiki.python.org>
- [Python and Pyvis 2020]** Python and PyVis for Data Visualization. 2020. PythonHumanities.com. URL: <https://pythonhumanities.com/python-and-pyvis-for-data-visualization/>
- [Python Tutorials 2022]** Python Tutorials for Digital Humanities. 2022. URL: <https://www.youtube.com/c/PythonTutorialsforDigitalHumanities>
- [Pyvis 2022a]** Pyvis. Read the docs. Introduction. 2022. URL: <https://pyvis.readthedocs.io/en/latest/introduction.html>
- [Pyvis 2022b]** Pyvis. Read the docs. 2022. URL: <https://pyvis.readthedocs.io/en/latest/index.html>
- [Pyvis 2022c]** Westhealth Institute. Pyvis github repository. 2022. URL: <https://github.com/West-Health/pyvis>
- [Realpython 2022]** Realpython.com. 2022. URL: <https://realpython.com/>
- [Stackoverflow 2022]** Stackoverflow. 2022 URL: <https://stackoverflow.com/>
- [Vis.js 2022a]** Vis.js. 2022. URL: <https://visjs.org/>
- [Vis.js 2022b]** Vis.js. Network. 2022. URL: <https://visjs.github.io/vis-network/docs/network/>
- [Wang 2019]** Wang, Jiahui: Python Interactive Network Visualization Using NetworkX, Plotly, and Dash. 2019 URL: <https://towardsdatascience.com/python-interactive-network-visualization-using-networkx-plotly-and-dash-e44749161ed7>
- [Weng 2020a]** Weng, Rebecca. Midsummer_network. 2020. URL: https://github.com/rweng18/midsummer_network
- [Weng 2020b]** Weng, Rebecca: Tutorial: Network Visualization Basics with Networkx and Plotly in Python. From a Shakespeare script to a network graph. 2020. URL: <https://towardsdatascience.com/tutorial-network-visualization-basics-with-networkx-and-plotly-and-a-little-nlp-57c9bbb55bb9>

Anhang

Anhang 1

Datensatz der Ausgangsdatei „hhb_td_2022-01-19“

GS
RS001A US03550:18-02-96
RS001B US03550:19-04-21US t 16:16:07.000
RS001D US03550:08-03-19
RS001U US0utf8
RS001X US033
RS002@ US0Tdv
RS003@ US0050582658
RS041@ US Sp USA Lichtwark , Cstn. Fried. Dang. Alfred
RS041@ US Sp USA Lichtwark , Friedrich Christian Danger Alfred US4prov
RS041A US Sp USA Lichtwark , Alfred
RS041P US0118572652US2GND
RS041R US9050597272USVTdvUS Sg USA Reitbrook US4ortg
RS041R US9193050153USVTdvUS Sg USA Hamburg US4orts
RS041R US9657469386USVTdvUS Sp USA Lichtwark , Marianne US4bezf USvSchwester
RS041R US9050582666USVTdvUS Sk USA Hamburger Kunsthalle US4affiUSvDirektor
RS041R US9522063322USVTdvUS Ss USA Kunsthistoriker US4beru
RS041R US9786519266USVTdvUS Ss USA Museumsdirektor US4beru
RS041R US9050583824USVTdvUS Ss USA Schriftsteller US4beru
RS041R US9050654780USVTdvUS Ss USA Hochschullehrer US4beru
RS047A/02 USA Direktor der Hamburger Kunsthalle (seit 1886). Baute die Gemäldeksam
RS047A/53 USA HANS : [Portrait] P 22:L 72
RS050C USAHHBIB/Ad. SUB-HANS ; HHBIB/Ma, mil
RS050G USA Deutsche Biographische Enzyklopädie <DBE> Bd 6. München 1997, s. 377
RS060R USA1852US b1914 US4datl
RS060R USA14.11.1852US b13.01.1914 US4datx

A. Zwingend benötigte Datenfelder zur Erstellung des Netzwerkgraphen:

- 041A:** Name der Person (nicht wiederholbar)
- 041R + Codierung „4bezi“:** familiäre Beziehungsangaben (wiederholbar)
- 041R + Codierung „4beru“:** Berufsangaben (wiederholbar)
- 047A:** Berufsangaben aus Altdaten (nicht wiederholbar)

B. Potentiell verwertbare Datenfelder zur Erstellung des Netzwerkgraphen:

- 047A/02:** Biogramm (nicht wiederholbar)
- 041R + Codierung „4ortg“/“4orts“:** Geburts-/Sterbeort (wiederholbar)
- 041R + Codierung „4bezb“:** berufliche Beziehungsangaben
- 041R + Codierung „4affi“:** institutionelle Beziehungsangaben (wiederholbar)
- 060R:** Geburts-/Sterbedaten (wiederholbar)

C. Nicht nötiges Datenfeld, berücksichtigt aus datenorganisatorischen Gründen:

- 003@: Datensatz-ID, ppn

Anhang 2

Datensatz der JSON-Datei

```
{  
    "hamburg": [  
        {  
            "ppn": [  
                "0050582658"  
            ],  
            "name": [  
                "Lichtwark, Alfred"  
            ],  
            "bio": [  
                "Direktor der Hamburger Kunsthalle (seit 1886). Baute die Gemäldesammlung der Hamburger Kunsthalle in wenigen Jahren zu einer der führenden Deutschlands aus und betrieb eine Erwerbungspolitik großen Stils. Initiator der Kunsterziehungsbewegung. Gründete 1886 die 'Gesellschaft Hamburger Kunstfreunde' und 1896 die 'Hamburger Lehrervereinigung zur Pflege der künstlerischen Bildung in den Schulen'. Entwickelte eine lebhafte publizistische Tätigkeit. Veröffentlichte 'Übungen in der Betrachtung von Kunstwerken' (1897). Bruder von Lichtwark, Marianne (1857-1931). Seit 1880 Studium der Kunstgeschichte in Leipzig, 1885 Promotion."  
            ],  
            "date_simple": [  
                "*1852 +1914"  
            ],  
            "date_exact": [  
                "*14.11.1852 +13.01.1914"  
            ],  
            "job": [  
                "Kunsthistoriker",  
                "Museumsdirektor",  
                "Schriftsteller",  
                "Hochschullehrer"  
            ],  
            "relation_fam": [  
                "Lichtwark, Marianne"  
            ],  
            "place": [  
                "Reitbrook",  
                "Hamburg"  
            ],  
            "relation_inst": [  
                "Hamburger Kunsthalle"  
            ],  
            "relation_job": [ ],  
            "date": [ ]  
        }  
    [...]  
    ]  
}
```

Erstellung eines Netzwerkgraphen aus Personendaten der Hamburg-Bibliographie mit der Python-Bibliothek Pyvis.

A. Der Weg von der Ausgangsdatei zur json-Datei für die Nutzung in Pyvis.

Dieses Skript ist im Zusammenhang mit der Ausarbeitung für dieses Projekt zu lesen, Angaben zu Datenstruktur der Ausgangsdatei sowie Transformationsanforderungen im allgemeinen finden sich auch dort im Text und in den Anhängen.

1. Importieren der nötigen Bibliotheken¶

```
In [1]: from pyvis.network import Network
import random
import json
```

2. Aufrufen der gelieferten Ausgangsdatei

```
In [2]: with open ("hhb_td_2022-01-19.pp", "r+", encoding = "utf-8") as infile:
    file = infile.readlines()
```

3. Wandlung der Lines in einzelne Dictionaries, Teilung der Lines in Keys und Values

Die einzelnen Zeilen der Ausgangsdatei teilen sich in Feldbezeichnung und Inhalt, die Feldbezeichnung ist durch ein Leerzeichen von dem Inhalt des Feldes getrennt. Diese Struktur kann dazu genutzt werden, eine Liste "entries" mit Key-Value-Paaren als Dictionaries für jede Zeile herzustellen. Dabei wird die Feldbenennung zum Key, der Feldinhalt zum Value der Dictionaries. Entstehen wird die Liste von Dictionaries "entries".

```
In [3]: entries = []

for line in file:

    # Das "*" vermeidet einen ValueError bei der Übertragung der Strings in eine Liste beim Splitten
    dict_line = {}
    key, *value = line.strip().replace("\x1f", " ").split(" ", 1)
    dict_line[key] = value

    entries.append(dict_line)
```

4. Gesammelte Bearbeitung der Values von Key "041R"

Die Entscheidung, den Beginn der Values der Keys "041R" an dieser Stelle schon endzubearbeiten, hat praktische Gründe: Die Values der Keys "041R" enthalten Daten, die später bei Erstellung des Graphen mit dem Value von Key "041A" (Person des Datensatzes) relationiert werden müssen. Sie sind in den Originaldaten mehrfach vorhanden mit unterschiedlichen Value-Inhalten, die erhalten bleiben müssen (vgl. Anhang 1). Für die Zusammenfassung zu Dictionaries müssen sie individualisiert werden, damit sie nicht überschrieben werden. Um eine spätere Einzelbehandlung der dann individualisierten Keys zu vermeiden und diese in einem Rutsch zu erledigen, geschieht dies an dieser Stelle vor der Individualisierung. Die gleiche Datenstruktur der Values begünstigt dies. Das Strippen bzw. Replacen des Beginns der Values wird von links bis zum Subfield des Inhaltes durchgeführt. Restlich zu löschen Codierungen rechts vom Inhalt werden in späteren Arbeitsschritten erledigt, die Codierungen werden noch als Anker zum Vereinigen der Inhalte benötigt.

```
In [4]: for item in entries:
    for key in item:
        item[key] = str(item[key]).strip("[ '']").strip("[]").strip("\\").replace("\\"", "")

    if "041R" in item:
        item[key] = item[key].lstrip(
            "[0123456789]X").lstrip().lstrip(
            "VTdv").lstrip().lstrip(
            "6gnd\\/[0123456789]-[0123456789]X").lstrip().lstrip(
            "S[pkcshtfz]").lstrip().replace("a", "", 1)
```

5. Individualisierung gleichnamiger Keys I:

Umbenennung der verschiedenen Keys des Feldes "041R" und der Keys "060R", die mehrfach vorkommen können, anhand der Codierungen in den Inhalten ihrer Values. Die Codierungen, die den Inhalt definieren, werden als Keyergänzung angehängt:

Key "041R": "4beru" = Berufsangaben
"4bezf" = familiäre Beziehung
"4bezb" = berufliche Beziehung
"4ort" = Geburts-/Sterbeort
"4affi" = institutionelle Affiliation/Zugehörigkeit
Key "060R": "4datl" = Lebensdaten
"4datx" = exakte Lebensdaten

```
In [5]: for item in entries:
    for key, value in list(item.items()):

        if "041R" == key:
            if "4beru" in value:
                item["041R_beru"] = item.pop("041R")

            if "4bezf" in value:
                item["041R_bezf"] = item.pop("041R")

            if "4bezb" in value:
                item["041R_bezb"] = item.pop("041R")

            if "4ort" in value:
                item["041R_ort"] = item.pop("041R")

            if "4affi" in value:
                item["041R_affi"] = item.pop("041R")

        if "060R" == key:
            if "4datl" in value:
                item["060R_datl"] = item.pop("060R")

            if "4datx" in value:
                item["060R_datx"] = item.pop("060R")
```

6. Individualisierung gleichnamiger Keys II

Nach dem ersten Individualisierungsschritt für die Keys "041R" und "060R" braucht es einen zweiten, denn noch ist nicht vermieden, dass Keys doppelt vorhanden sind. Dazu erhalten die gefährdeten Keys einen durchlaufenden Nummernzusatz und werden so individualisiert.

```
In [6]: i=0

for item in entries:

    if "041R_beru" in item:
        item[str("041R_beru-" + str(i))] = item.pop("041R_beru")

    if "041R_bezf" in item:
        item[str("041R_bezf-" + str(i))] = item.pop("041R_bezf")

    if "041R_bezb" in item:
        item[str("041R_bezb-" + str(i))] = item.pop("041R_bezb")

    if "041R_ort" in item:
        item[str("041R_ort-" + str(i))] = item.pop("041R_ort")

    if "041R_affi" in item:
        item[str("041R_affi-" + str(i))] = item.pop("041R_affi")

    i=i+1
```

7. Zusammenfassung der zu einem Personendatensatz gehörigen Einzel-Dictionaries zu gruppierten Dictionaries

Die einzelnen Zeilen-Dictionaries werden nun zu gruppierten Items einer List of Dictionaries transformiert, bei der nicht mehr jedes Item einer Zeile der Ausgangsdatei entspricht, sondern dem Datensatz einer Person. Dabei dient das Feld "001A" als Anker, dessen Dictionary anzeigt, dass die folgenden Dictionaries, bis zum nächsten Vorkommen des Feldes "001A" zu einem Personendatensatz gehören. Entstehen wird neu die Liste mit Dictionaries "grouped_dicts".

```
In [7]: grouped_dicts = []
current_entry = {}

for item in entries:
    if "001A" in item:

        # erzeugt ein Dictionary für die Keys eines jeden Datensatzes, wenn das if-Statement wahr ist
        current_entry = {}

        # fügt das dict "current_entry" in die Liste "grouped_dicts" an
        grouped_dicts.append(current_entry)

    # liest die Einzeldictionaries aus "entries" nach "current_entry" in "grouped_dicts"
    current_entry.update(item)
```

```
In [8]: # Zählung Gesamtmenge der Datensätze der Ausgangsdatei
count_items = 0
for item in grouped_dicts:
    if "003@" in item.keys():
        count_items += 1
count_items
```

Out[8]: 72125

8. Vereinigung zusammengehöriger Values unter einen Key

Durch das Individualisieren der gleichnamigen Keys wurden deren Values gesichert. In einem nächsten Schritt müssen die Values der einzelnen Keys nun in einen Gesamtvalue zusammengefügt werden. Für diese Filterung werden wieder die oben erwähnten Codierungen genutzt. Um unterschiedliche Werte gleichnamiger Keys in einen Wert zu integrieren, werden neue Values generiert und die alten gelöscht. Danach erfolgt die Löschung der alten, durchnummerierten Key/Value-Paare.

```
In [9]: for item in grouped_dicts:

    item["joined_val-041R_beru"] = []
    item["joined_val-041R_bezf"] = []
    item["joined_val-041R_bezb"] = []
    item["joined_val-041R_ort"] = []
    item["joined_val-041R_affi"] = []

    for key, value in item.items():
        if "041R_beru-" in key:
            item["joined_val-041R_beru"].append(value)

        if "041R_bezf-" in key:
            item["joined_val-041R_bezf"].append(value)

        if "041R_bezb-" in key:
            item["joined_val-041R_bezb"].append(value)

        if "041R_ort-" in key:
            item["joined_val-041R_ort"].append(value)

        if "041R_affi-" in key:
            item["joined_val-041R_affi"].append(value)

    # Löschung der alten Key/Value-Paare
    old_keys = ["041R_beru-", "041R_bezf-", "041R_bezb-", "041R_ort-", "041R_affi-"]

    for key in item.copy():
        for element in old_keys:
            if element in key:
                item.pop(key)

    # Rückbenennung der Keys
    if key == "joined_val-041R_beru":
        item["041R_beru"] = item.pop("joined_val-041R_beru")

    if key == "joined_val-041R_bezf":
        item["041R_bezf"] = item.pop("joined_val-041R_bezf")

    if key == "joined_val-041R_bezb":
        item["041R_bezb"] = item.pop("joined_val-041R_bezb")

    if key == "joined_val-041R_ort":
        item["041R_ort"] = item.pop("joined_val-041R_ort")

    if key == "joined_val-041R_affi":
        item["041R_affi"] = item.pop("joined_val-041R_affi")
```

9. Aufräumen I: leere Values löschen

```
In [10]: for item in grouped_dicts:  
    item.pop("", None)
```

```
In [11]: for item in grouped_dicts:  
    for key, value in list(item.items()):  
        if not value:  
            del item[key]
```

10. Aufräumen II: überflüssige Satzarten löschen

Die Ausgangsdatei beinhaltete sämtliche Normdatensätze der Hamburg-Bibliographie, nicht nur Personen. Im nächsten Schritt werden alle Items, die keine Personennormdaten verzeichnen, gelöscht. Um folgende Satzarten handelt es sich, sie werden mittels eindeutigen Zeichenabfolgen in den Values des Keys "041A" gegriffen.

Zeichenfolge: "Sk a" = Körperschaftsdaten
"St a" = Titeldaten
"Sf a" = Medienartdaten
"Sg a" = Gebietsdaten
"Ss a" = Sachdaten
"Sz a" = Zeitdaten
"Sc a" = Gebietskörperschaftsdaten

```
In [12]: for item in grouped_dicts.copy():  
  
    data_type = ["Sk a", "St a", "Sf a", "Sg a", "Ss a", "Sz a", "Sc a"]  
    for key in item:  
        if "041A" == key:  
            for character in data_type:  
                if character in item[key]:  
                    grouped_dicts.remove(item)
```

Für die nächsten Filterungen der Datenatztypen, die keine (relevanten) Personendaten verzeichnenen, wie Datensätze von Schlagwortketten, Personendatensätze von Nicht-Hamburgern sowie Datensätzen von Familien, muss bei meinen Programmierkenntnissen einzeln iteriert werden, da die Filterkriterien nicht ausschließlich einzeln in den Values vorkommen können. Folgende Filterkriterien befinden sich in den Values der Keys "041A" und "050D".

Filterkriterien: "Sp a" und " x" = Datensätze von Schlagwortketten
"Sp a" und " gFamilie" = Familiennormdaten
"Sp a" und "<Familie>" = Familiennormdaten
"abio-" = Personennormdaten von Nicht-Hamburgern

```
In [13]: for item in grouped_dicts.copy():  
    for key in item:  
        if key == "041A":  
            if "Sp a" in item[key]:  
                if " x" in item[key]:  
                    grouped_dicts.remove(item)
```

```
In [14]: for item in grouped_dicts.copy():  
    for key in item:  
        if key == "041A":  
            if "Sp a" in item[key]:  
                if " gFamilie" in item[key]:  
                    grouped_dicts.remove(item)
```

```
In [15]: for item in grouped_dicts.copy():  
    for key in item:  
        if key == "041A":  
            if "Sp a" in item[key]:  
                if "<Familie>" in item[key]:  
                    grouped_dicts.remove(item)
```

```
In [16]: for item in grouped_dicts.copy():  
    for key in item:  
        if "050D" == key:  
            if "abio-" in item[key]:  
                grouped_dicts.remove(item)
```

In [17]:

```
# Zählung der Gesamtmenge der verzeichneten Personendaten in der Hamburg-Bibliographie
count_items = 0
for item in grouped_dicts:
    if "003@" in item.keys():
        count_items += 1
count_items
```

Out[17]: 19869

11. Aufräumen III: überflüssige Key/Value-Paare löschen

Anmerkung zu Feld 041R: es kann gelöscht werden, da die relevanten Inhalte in neue neubenannte Key/Value-Paare überführt wurden (Feld "041R" ist ein Sammelfeld für viele relationierte Daten, die jeweils durch Codierungen - weitaus mehr als die hier relevanten sind möglich - inhaltlich zugeordnet wurden, vgl. Anhang 1). Es befinden sich also nur noch irrelevante Inhalte im Feld sowie relevante Inhalte, die nicht zugeordnet werden können, weil bei ihnen die Codierung nicht aussagekräftig ist, die Codierung fehlt oder diese unkorrekt eingegeben wurde. Auf diese relevanten Inhalte könnte erst nach einer händischen Umarbeitung der originalen Ausgangsdaten zugegriffen werden.

In [18]:

```
for item in grouped_dicts:

    item.pop("001A", None)
    item.pop("001B", None)
    item.pop("001D", None)
    item.pop("001U", None)
    item.pop("001X", None)
    item.pop("002@", None)
    item.pop("041@", None)
    item.pop("041P", None)
    item.pop("047A/53", None)
    item.pop("050C", None)
    item.pop("050G", None)
    item.pop("006Y", None)
    item.pop("047A/04", None)
    item.pop("050E", None)
    item.pop("003D", None)
    item.pop("0410", None)
    item.pop("050H", None)
    item.pop("041R", None) # Löschung des Keys (Erklärung siehe vorige Zelle)
    item.pop("007K", None)
```

Im nächsten Abschnitt werden die Values der übrig gebliebenen Keys endbehandelt, d.h. von allen restlichen Codierungen befreit, bei Bedarf gestript, replaced, gesliced, gesplittet u.a., so dass der reine Inhalt übrig bleibt. Neben des Präsentationszweckes im Graphen und der Verhinderung von Whitespace-Fehlermeldungen ist dieser Schritt auch besonders wichtig für die Values der Keys "041A" und "041R_beru" bzw. "041R_bezf", die bei der Generierung des Graphen direkt abgeglichen werden und daher deren Inhalte in beiden Values gleichlauten müssen. Die Values der übrigen Keys für relationierte Angaben "041R_bezb" und "041R_affi" werden mitbehandelt für eine eventuelle spätere Nutzung. Weiterhin werden, um Fehlermeldungen bei der Erzeugung des Graphen zu vermeiden, für alle Keys leere Values erzeugt, sollten in den Originaldaten keine Inhalte vorhanden gewesen sein, damit in jedem Dictionary ein Wert angesteuert werden kann. Dies gilt nicht für den Key "041A". Um dort Fehlermeldungen durch fehlenden Values aufgrund falsch eingegebener Originaldaten zu vermeiden, werden in diesem Fall keine leeren Values erzeugt, sondern die Dictionaries ganz gelöscht, da ohne einen Inhalt für den Ankerkey "041A" diese Person quasi ja gar nicht vorhanden ist und ein leerer Node für den Graphen generiert werden würde.

Im Folgenden wird aufgrund der Fülle nicht die Gesamtheit aller zu entfernenden Codierungen in den jeweiligen Values näher erläutert, dazu empfiehlt sich bei Interesse ein Blick in die Ausgangsdaten.

12. Abschlussbehandlung Value "041A" (Personenname)

In [19]:

```
for item in grouped_dicts:
    for key in item:
        if key == "041A":
            item[key] = str(item[key][4:]).lstrip().replace("^", "").replace(" g", "; ")
```

In [20]:

```
for item in grouped_dicts.copy():
    if "041A" not in item:
        grouped_dicts.remove(item)
```

13. Abschlussbehandlung Value "047A/02" (Biogramm)

```
In [21]:  
for item in grouped_dicts:  
    for key in item:  
        if key == "047A/02":  
            item[key] = str(item[key]).replace("a", "", 1)  
            item[key] = [item[key]]
```

```
In [22]:  
for item in grouped_dicts:  
    for key in item.copy():  
        if "047A/02" not in item:  
            item["047A/02"] = []
```

14. Abschlussbehandlung Value "047A" (Berufsangaben Altdaten)

Aufgrund von Altdatentransformationen existieren auch Berufsangaben in einem anderen Feld als dem richtlinienvorgeschriebenen "041R". Glücklicherweise können nicht beide Felder mit Berufsangaben in einem Datensatz der Ausgangsdaten vorkommen, es gibt sie nur je und je, das hat Arbeitsschritte gespart. "047A" wird hier bearbeitet und zum Key "041R_beru" gewandelt.

```
In [23]:  
for item in grouped_dicts:  
    for key in item:  
        if key == "047A":  
            item[key] = str(item[key]).lstrip(  
                "a").replace(  
                ".", "").replace(  
                ";", "").replace(  
                "und ", "").replace(  
                ",", "").split()
```

```
In [24]:  
for item in grouped_dicts:  
    if "047A" in item:  
        item["047A_beru"] = item.pop("047A")
```

15. Abschlussbehandlung Value "041R_beru" (Berufsangaben)

```
In [25]:  
for item in grouped_dicts:  
    for key in item.copy():  
        if key == "041R_beru":  
            item["temp_041R_beru"] = []  
            for element in item[key]:  
                item["temp_041R_beru"].append(element.replace(" 4beru", "").replace(" v", " ", 1))  
  
            item.pop(key)
```

```
In [26]:  
for item in grouped_dicts:  
    if "temp_041R_beru" in item:  
        item["041R_beru"] = item.pop("temp_041R_beru")
```

Umbenennen des Altdatenfeldes "047A" in das gleichnamige Feld "041R_beru", damit alle Berufsdaten die gleiche Keybenennung haben (siehe oben).

```
In [27]:  
for item in grouped_dicts:  
    for key in item.copy():  
        if key == "047A_beru":  
            item["041R_beru"] = item.pop("047A_beru")  
  
        if "041R_beru" not in item:  
            item["041R_beru"] = []
```

16. Abschlussbehandlung Value "041R_bezf" (familiäre Beziehungsangaben)

Eine Besonderheit für die Inhalte von Key "041R_bef" (familiäre Beziehungsangaben) sind erwähnenswert: in den Originaldaten können neben den Namen der Familienmitglieder auch die jeweilige Beziehungskategorie (Schwester, Vater, Sohn u.a.) erfasst werden. Leider darf diese nicht erhalten bleiben, da zum Abgleich mit Key "041A" bei der Generation des Graphen gleichlautende Einträge vorhanden sein müssen (vgl. oben). Das ist schade, denn so gehen Informationen verloren. In einer Ausbaustufe des Projektes ist geplant, diese Angaben in die Biogramme ("047A/02") zu integrieren, um sie zu erhalten.

In [28]:

```
for item in grouped_dicts:
    for key in item.copy():
        if key == "041R_bezf":
            item["temp_041R_bezf"] = []
            for element in item[key]:
                element = element.rsplit("4bezf", 1)[0]
                element = element.rstrip("4bezf").rstrip().replace("^", "")
            item["temp_041R_bezf"].append(element)

item.pop(key)
```

In [29]:

```
for item in grouped_dicts:
    for key in item.copy():
        if key == "temp_041R_bezf":
            item["041R_bezf"] = item.pop("temp_041R_bezf")

        if "041R_bezf" not in item:
            item["041R_bezf"] = []
```

17. Abschlussbehandlung Value "041R_bezb" (berufliche Beziehungsangaben)

Vgl. Punkt 16. Diese Angaben gehören zu dem Kreis der für den Graphen potentiell nutzbaren. Da sie in den Originaldaten jedoch nicht häufig besetzt sind, wird hier zunächst davon Abstand genommen.

In [30]:

```
for item in grouped_dicts:
    for key in item.copy():
        if key == "041R_bezb":
            item["temp_041R_bezb"] = []
            for element in item[key]:
                element = element.rsplit("4bezb", 1)[0]
                element = element.rstrip("4bezb").rstrip().replace("^", "")
            item["temp_041R_bezb"].append(element)

item.pop(key)
```

In [31]:

```
for item in grouped_dicts:
    for key in item.copy():
        if key == "temp_041R_bezb":
            item["041R_bezb"] = item.pop("temp_041R_bezb")

        if "041R_bezb" not in item:
            item["041R_bezb"] = []
```

18. Abschlussbehandlung Value "041R_affi" (institutionelle Beziehungsangaben)

Vgl. Punkt 17.

In [32]:

```
for item in grouped_dicts:
    for key in item.copy():
        if key == "041R_affi":

            item["temp_041R_affi"] = []
            for element in item[key]:
                item["temp_041R_affi"].append(
                    element.replace(
                        " 4affi v", "; ").replace(
                        " 4affi", "").replace(
                        " Z", " ").replace(
                        " g", " , ").replace(
                        " x", " , "))

item.pop(key)
```

In [33]:

```
for item in grouped_dicts:  
    for key in item.copy():  
  
        if key == "temp_041R_affi":  
            item["041R_affi"] = item.pop("temp_041R_affi")  
  
        if "041R_affi" not in item:  
            item["041R_affi"] = []
```

Im folgenden werden Angaben endbehandelt, die zu den biographischen Angaben zählen (Geburts-/Sterbeorte, Lebensdaten). Sie sind für eine Ausbaustufe des Projektes zur Integration in die Biogramme eingeplant. Dafür werden ggf. auch schon Angaben hinzugefügt.

19. Abschlussbehandlung Value "041R_ort" (Geburts-/Sterbeort)

In [34]:

```
for item in grouped_dicts:  
    for key in list(item):  
        if key == "041R_ort":  
            item["temp_041R_ort"] = []  
            for element in list(item[key]):  
                if "4ortg" in element:  
                    item["temp_041R_ort"].append("geb. " + element.replace(" 4ortg", ""))  
                if "4orts" in element:  
                    item["temp_041R_ort"].append("gest. " + element.replace(" 4orts", ""))  
            item.pop(key)
```

In [35]:

```
for item in grouped_dicts:  
    for key in item.copy():  
  
        if key == "temp_041R_ort":  
            item["041R_ort"] = item.pop("temp_041R_ort")  
  
        if "041R_ort" not in item:  
            item["041R_ort"] = []
```

20. Abschlussbehandlung Value "060R_datl" (einfache Lebensdaten)

In [36]:

```
for item in grouped_dicts:  
    for key in item:  
        if key == "060R_datl":  
            item[key] = item[key].replace(" 4datl", "").replace(" v", " ", 1).replace(" b", " +", 1)  
  
            if item[key][0] == "a":  
                item[key] = "*" + item[key][1:]  
  
            if item[key][0] == "b":  
                item[key] = "+" + item[key][1:]  
  
            item[key] = [item[key]]
```

In [37]:

```
for item in grouped_dicts:  
    for key in item.copy():  
        if "060R_datl" not in item:  
            item["060R_datl"] = []
```

21. Abschlussbehandlung Value "060R_datx" (exakte Lebensdaten)

In [38]:

```
for item in grouped_dicts:  
    for key in item:  
        if key == "060R_datx":  
            item[key] = item[key].replace(" 4datx", "").replace(" v", " ", 1).replace(" b", " +", 1)  
  
            if item[key][0] == "a":  
                item[key] = "*" + item[key][1:]  
  
            if item[key][0] == "b":  
                item[key] = "+" + item[key][1:]  
  
            item[key] = [item[key]]
```

In [39]:

```
for item in grouped_dicts:
    for key in item.copy():
        if "060R_datx" not in item:
            item["060R_datx"] = []
```

22. Abschlussbehandlung Value "060R" (falsch codierte Lebensaten / uncodierte Lebensdaten aus Altdaten)

In diesem Feld befinden sich nur noch, nachdem datl- und datx-codierte Angaben in eigene Key/Value-Paare gewandert sind, durch Eingabefehler falsch codierte datl-, datx-Daten sowie Daten ohne Codierung aus Altdaten. Es wird versucht, so viele Falschcodierungen zu löschen wie bei einem Datenscan gefunden werden konnten.

In [40]:

```
for item in grouped_dicts:
    for key in item:
        if key == "060R":
            item[key] = item[key].replace(
                " 4datw", "").replace(
                " 4dats", "").replace(
                " 4datb", "").replace(
                " bdatx", "").replace(
                " bdatl", "").replace(
                " datz", "").replace(
                " datl", "").replace(
                " 4dtl", "").replace(
                " v", ", ", 1).replace(
                " b", "+", 1)

        if item[key][0] == "a":
            item[key] = "*" + item[key][1:]

        if item[key][0] == "b":
            item[key] = "+" + item[key][1:]

        if item[key][0] == "":
            item[key] = "*" + item[key][1:]

        item[key] = [item[key]]
```

In [41]:

```
for item in grouped_dicts:
    for key in item.copy():
        if "060R" not in item:
            item["060R"] = []
```

23. Abschlussbehandlung Value "003@"

Der Value von Key "003@" ist die Datensatz-ID für jeden Personendatensatz. Für die Generierung des Netzwerkgraphen ist sie nicht relevant, aus datenpraktischen Gründen wird sie aber mitgeführt, um etwa Fehler in den Originaldaten schnell lokalisieren zu können.

In [42]:

```
for item in grouped_dicts:
    for key in item:
        if key == "003@":
            item["003@"] = [item["003@"]]
```

24. Keys bekommen sprechende Namen

```
In [43]: for item in grouped_dicts:
    for key in item.copy():

        if key == "060R":
            item["date"] = item.pop("060R")

        if key == "060R_datl":
            item["date_simple"] = item.pop("060R_datl")

        if key == "060R_datx":
            item["date_exact"] = item.pop("060R_datx")

        if key == "041R_beru":
            item["job"] = item.pop("041R_beru")

        if key == "041R_bezf":
            item["relation_fam"] = item.pop("041R_bezf")

        if key == "041R_bezb":
            item["relation_job"] = item.pop("041R_bezb")

        if key == "041R_affi":
            item["relation_inst"] = item.pop("041R_affi")

        if key == "041R_ort":
            item["place_birth_death"] = item.pop("041R_ort")

        if key == "047A/02":
            item["bio"] = item.pop("047A/02")

        if key == "041A":
            item["name"] = item.pop("041A")

        if key == "003@":
            item["ppn"] = item.pop("003@")
```

25. Erstellung json-Dateien

25.1 Gesamtdatei

```
In [44]: with open("hhbib_persons.json", "w") as outfile:  
    json.dump(grouped_dicts, outfile, indent=4, ensure_ascii=False)
```

25.2 Gesamtdatei zur Erstellung des Graphen in Pyvis

Als darstellerische Variante für den Graphen kann ein übergreifendes Item "Hamburg" gedacht werden, das als "Urpunktnode" alle Personen miteinander verbindet. Damit kann ein Graph mit einem Bezugspunkt für alle relationierten Daten erstellt werden (vgl. Punkt 26). Dies ist als darstellerische Variante gedacht. Daher wird ein übergeordnetes Dictionary mit der Keybenennung "Hamburg" angelegt, das bei Bedarf bei der Erstellung des Graphen berücksichtigt werden kann.

```
In [45]: hamburg_node = {"Hamburg": grouped_dicts}
```

```
In [46]: # Überprüfung der Anzahl der Items = Personen  
count_items = 0  
for item in hamburg_node["Hamburg"]:  
    if "ppn" in item.keys():  
        count_items += 1  
count_items
```

Out[46]: 19865

```
In [47]: with open("hh_persons.json", "w") as outfile:  
    json.dump(hamburg_node, outfile, indent=4, ensure_ascii=False)
```

25.3 Datei zur Darstellung der Beziehungskonstellation "Person/familäre Beziehung":

Probedurchläufe haben ergeben, dass durch die Berücksichtigung der Gesamtpersonendaten mit familiärem Bezug lange Ladezeiten für die Anzeige des Graphen entstehen. Daher wird die Datenmenge in einer Datei auch auf Personendaten mit mindestens zwei Verwandtschaftsbeziehungen reduziert.

```
In [48]: with open("hhbib_persons.json", "r+", encoding = "utf-8") as infile:  
    hh_persons_fam = json.load(infile)
```

```
In [49]: # Anzahl der Items, in denen Verwandtschaftsbeziehungen vorliegen  
count_items = 0  
for item in hh_persons_fam:  
    for key, value in item.items():  
        if key == "relation_fam":  
            if value:  
                count_items += 1  
count_items
```

```
Out[49]: 571
```

```
In [50]: for item in hh_persons_fam.copy():  
    for key, value in item.items():  
        if key == "relation_fam":  
            if not value:  
                hh_persons_fam.remove(item)
```

```
In [51]: with open("hh_persons_fam_all.json", "w") as outfile:  
    json.dump({"Hamburg" : hh_persons_fam}, outfile, indent=4, ensure_ascii=False)
```

```
In [52]: for item in hh_persons_fam.copy():  
    for key, value in item.items():  
        if key == "relation_fam":  
            if len(value) <= 1:  
                hh_persons_fam.remove(item)
```

```
In [53]: # Anzahl der Personen mit mindestens zwei Verwandtschaftsbeziehungen  
count_items = 0  
for item in hh_persons_fam:  
    if "ppn" in item.keys():  
        count_items += 1  
count_items
```

```
Out[53]: 191
```

```
In [54]: with open("hh_persons_fam_2.json", "w") as outfile:  
    json.dump({"Hamburg" : hh_persons_fam}, outfile, indent=4, ensure_ascii=False)
```

25.4 Datei zur Darstellung der Beziehungskonstellation "Person/Berufsangaben"

Auch hier ist wie unter Punkt 25.3 eine Datenreduzierung nötig, jedoch noch drastischer. Es werden zwei Dateien erstellt, die eine mit der Begrenzung auf mindestens vier Berufsangaben, die andere auf mindestens fünf.

```
In [55]: with open("hhbib_persons.json", "r+", encoding = "utf-8") as infile:  
    hh_persons_job = json.load(infile)
```

```
In [56]: # Gesamtanzahl der Items mit verzeichneten Berufsangaben  
count_items = 0  
for item in hh_persons_job:  
    for key, value in item.items():  
        if key == "job":  
            if value:  
                count_items += 1  
count_items
```

```
Out[56]: 16757
```

Beschränkung auf Personen mit mindestens vier Berufsangaben

In [57]:

```
for item in hh_persons_job.copy():
    for key, value in item.items():
        if key == "job":
            if len(value) <= 3:
                hh_persons_job.remove(item)
```

In [58]:

```
# Anzahl der Items mit mindestens vier verzeichneten Berufsangaben
count_items = 0
for item in hh_persons_job:
    if "ppn" in item.keys():
        count_items += 1
count_items
```

Out[58]: 673

In [59]:

```
with open("hh_persons_job_4.json", "w") as outfile:
    json.dump({"Hamburg" : hh_persons_job}, outfile, indent=4, ensure_ascii=False)
```

Beschränkung auf Personen mit mindestens fünf Berufsangaben

In [60]:

```
for item in hh_persons_job.copy():
    for key, value in item.items():
        if key == "job":
            if len(value) <= 4:
                hh_persons_job.remove(item)
```

In [61]:

```
# Anzahl der Items mit mindestens vier verzeichneten Berufsangaben
count_items = 0
for item in hh_persons_job:
    if "ppn" in item.keys():
        count_items += 1
count_items
```

Out[61]: 197

In [62]:

```
with open("hh_persons_job_5.json", "w") as outfile:
    json.dump({"Hamburg" : hh_persons_job}, outfile, indent=4, ensure_ascii=False)
```

B. Erstellung des Netzwerkgraphen

Wie das Skript zur Erstellung der json-Datei ist auch dieses in Zusammenhang mit der Ausarbeitung zu lesen. Über die einfache Erzeugung eines Graphen hinaus, werden hier zwei darstellerische Optionen zusätzlich angeboten:

1. Da die Personen als Familienmitglieder zwar untereinander durch Edges verknüpft werden, aber nicht die Familien untereinander (das spiegeln die Originaldaten nicht wieder), entstehen Familiensubnetze. Um ein verbindendes Element herzustellen, was allen Familien gemeinsam ist, kam die Idee eines übergeordneten "Hamburg"-Nodes auf, das mit allen Familien mittels eines Edges verbunden ist. Dazu wurde eine Liste mit jeweils nur einem Mitglied einer Familie erstellt, deren Einträge im Skript zur Erzeugung der Edges eingesetzt werden (hamburg_edges.txt). Es ist auch eine Variante möglich, in der alle Personen (Nodes) mit diesem Hamburg-Node verbunden sind. Das ist allerdings zwar hübsch anzusehen, aber etwas unpraktisch, weil dann die Familienverbindungen visuell nur schwer erkennbar und auseinander zu klabüstern sind.

2. Zur besseren visuellen Erfassbarkeit der einzelnen Familiennetze entstand weiterhin die Idee, diese farblich unterschiedlich zu erzeugen. Dazu wurden zufällig generierte Hex-ColorCodes erzeugt und diese dann den Nodes zugewiesen.

In der folgenden Skriptzelle sind die gesamten, ausprobierten Optionen zur Visualisierung des Netzwerkes dargestellt. Für die Generierung varianter Visualisierungen/html-Dateien damit - auch mit den oben erstellten verschiedenen Dateien - werden einige entsprechend ausgeblendet oder Werte geändert. Die Einzelskripte dazu werden in der gesonderten Datei "variant_visualization_options.ipynb" notiert und annotiert.

```
In [63]: with open("hh_persons_fam_2.json", "r", encoding = "utf-8") as infile:
    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
        for line in infile_2:
            hh_name = line[:-1]
            hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#FBFCFC", font_color="black")

    # Erzeugung "Hamburg-Node"
    for hamburg in data_fam_2:
        g.add_node(hamburg, color="#2471A3", size=100)

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["relation_fam"]
        bio = person["bio"]

        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        random_colnumber = random.randint(0, 17687459)
        hex_number = str(hex(random_colnumber))
        hex_number = "#" + hex_number[2:]

        # Erzeugung der Hauptnodes
        g.add_node(name, color=hex_number, title=bio, size=50)

    # Erzeugung der relationierten Nodes sowie den Edges
    for rela in relation:
        g.add_node(rela, color=hex_number, size=50)
        g.add_edge(name, rela)

        # Erzeugung der "Hamburg-Edges"
        for item in hamburg_edges:
            if item == name:
                g.add_edge(hamburg, name)
            if item == rela:
                g.add_edge(hamburg, rela)

    # Layout-Algorithmen
    g.barnes_hut()
    #g.force_atlas_2based()
    #g.hrepulsion()
    #g.repulsion()
    #g.show_buttons()
    #g.set_options(''' var options = {
    #    "nodes": {"font": {"size": 20}},
    #    "physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}} } ''')

    # Erzeugung der Visualisierung und der html-Datei
    g.show("hhbib_networkGraph_1.html")
```

In []:

Liste zur Generierung von Edges vom "Hamburg-Node" zu jeder Familie

Erläuterungen zum Hintergrund dieses Skriptes finden sich in der Ausarbeitung sowie in den Anmerkungen im Hauptskript in Anhang 3.

1. Laden der nötigen Bibliotheken

```
In [1]: import json
```

2. Öffnen der Datei "hh_persons_fam_2.json" mit zwei Verwandtschaftsbeziehungen pro Person

```
In [3]: with open("hh_persons_fam_2.json", "r+", encoding = "utf-8") as infile:
    daten = json.load(infile)
    hamburg = daten["Hamburg"]
```

3. Filtern der Namen der Haupt- und relationierten Nodes

```
In [4]: hamburg_edges = []

for item in hamburg:
    for key in item:
        if key == "name":
            for element in item[key]:
                hamburg_edges.append(element)

        if key == "relation_fam":
            for element in item[key]:
                hamburg_edges.append(element)
```

```
In [5]: # hamburg_edges
```

4. Doppelte Einträge löschen

Um doppelte Einträge zu eliminieren, wird die Liste einmal in ein Dictionary verwandelt und wieder zurück in eine Liste.

```
In [6]: hamburg_edges_new = list(dict.fromkeys(hamburg_edges))
```

```
In [7]: # hamburg_edges_new
```

5. Löschung aller doppelten Nachnamen

Um tatsächlich nur ein Mitglied einer Familie als Endeintrag der Liste zu generieren, wird wiederum ein Dictionary aller Einträge erzeugt, wobei der Nachname der Key, der Vorname der Value ist, damit dann gleichnamige Keys elöscht werden.

```
In [8]: entries = []

for item in hamburg_edges_new:
    dict_line = {}
    key, *value = item.strip().split(", ", 1)
    dict_line[key] = value

    entries.append(dict_line)
```

```
In [9]: # entries
```

```
In [10]: for item in entries:
    for key in item:
        item[key] = str(item[key]).lstrip("[ ' ").rstrip("']").lstrip()
```

```
In [11]: # entries
```

Durch Bildung nur eines Dictionarys werden die doppelten Keys/Nachnamen überschrieben

In [12]:

```
current_entry = {}  
  
for item in entries:  
    current_entry.update(item)
```

In [13]:

```
# current_entry
```

In [14]:

```
# sorted(current_entry)
```

In [15]:

```
print(len(current_entry))
```

106

6. Endbearbeitung und Erstellung der Datei "hamburg_edges.txt"

In [17]:

```
with open("hamburg_edges.txt", "w") as outfile:  
    for key, value in current_entry.items():  
        print(key.lstrip(  
            "") .rstrip(  
            "") + ", " + value.lstrip(  
            "") .rstrip(  
            ""), file=outfile)
```

In []:

Variante Visualisierungen: Dokumentation der Optionen

Im Folgenden werden beispielhaft neun verschiedene Visualisierungen eines Netzwerkgraphens vorgestellt, die mit den Personendaten der Hamburg-Bibliographie und der Python-Bibliothek Pyvis erstellt wurden. Dabei werden auch drei Varianten vorgestellt, die in der tatsächlichen Nutzung weniger gut verwendbar wären, um zu verdeutlichen, dass die Darstellung der Daten innerhalb der angebotenen Optionen immer auch ein Abgleichen mit den individuellen Bedürfnisanforderungen ist. Datenpool für die Erstellung sind die mit dem Skript in Anhang 1 erzeugten json-Dateien: in Beziehung gesetzt werden einerseits Personen (Value "name") und ihre relationierten Verwandten (Value "relation_fam") sowie Personen und ihre relationierten Berufe (Value "job"). Bezug genommen wird u.a. auf die Ausarbeitung und ihren Anhängen sowie an dieser Stelle vor allem auf die Dokumentation der Software Pyvis (Pyvis 2022b, <https://pyvis.readthedocs.io/en/latest/documentation.html>). Flankierend zu nutzen sind die Visualisierungen unter https://schumahe.github.io/hhbib_networkGraph/. Es wird jeweils der Code für den Graphen und folgend eine Visualisierung dessen aufgeführt.

```
In [2]: from pyvis.network import Network
import random
import json
```

Ausgangsskript

```
In [ ]: with open("hh_persons_fam_2.json", "r", encoding = "utf-8") as infile:
    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
        for line in infile_2:
            hh_name = line[:-1]
            hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#FBFCFC", font_color="black")

    # Erzeugung "Hamburg-Node"
    for hamburg in data_fam_2:
        g.add_node(hamburg, color="#2471A3", size=100)

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["relation_fam"]
        bio = person["bio"]

        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        random_colnumber = random.randint(0, 17687459)
        hex_number = str(hex(random_colnumber))
        hex_number = "#" + hex_number[2:]

        # Erzeugung der Hauptnodes
        # Erzeugung der relationierten Nodes sowie den Edges
        for rela in relation:
            g.add_node(rela, color=hex_number, size=50)
            g.add_edge(name, rela)

        # Erzeugung der "Hamburg-Edges"
        for item in hamburg_edges:
            if item == name:
                g.add_edge(hamburg, name)
            if item == rela:
                g.add_edge(hamburg, rela)

    # Layout-Algorithmen
    g.barnes_hut()
    #g.force_atlas_2based()
    #g.hrepulsion()
    #g.repulsion()
    #g.show_buttons()
    #g.set_options('' var options = {
    #  "nodes": {"font": {"size": 20}},
    #  "physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}} } '')
    # Erzeugung der Visualisierung und der html-Datei
    g.show("hhbib_networkGraph_1.html")
```

Graph #1

Der erste Graph zeigt verwandschaftliche Zusammenhänge. Diese werden durch die Farbgebung unterstützt. Die Generierung der zufälligen Hex-Colors für die Nodes wird durch die abhängige Iteration an die Verwandten weitergegeben (Ausnahmen begründen falsche Originaldaten). Ebenso sind die Nodes durch die Visualisierung der Edges von dem Hamburg-Node (vgl. Ausarbeitungstext sowie Anhang 1) zu den Familien gruppiert. Besonders macht diese Visualisierung der hohe Node-Abstand aus, definiert im Layout-Algorithmus "repulsion" und in abstossendem Zusammenhang mit dem Node-Abstandswert, der die Nodes mit Hilfe von geraden Edges weiträumig gruppieren.

```
In [ ]: with open("hh_persons_fam_2.json", "r", encoding = "utf-8") as infile:
    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
        for line in infile_2:
            hh_name = line[:-1]
            hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#f5f5f5", font_color="black")

    # Erzeugung "Hamburg-Node"
    for hamburg in data_fam_2:
        g.add_node(hamburg, color="#cd0000", size=20)

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["relation_fam"]
        bio = person["bio"]

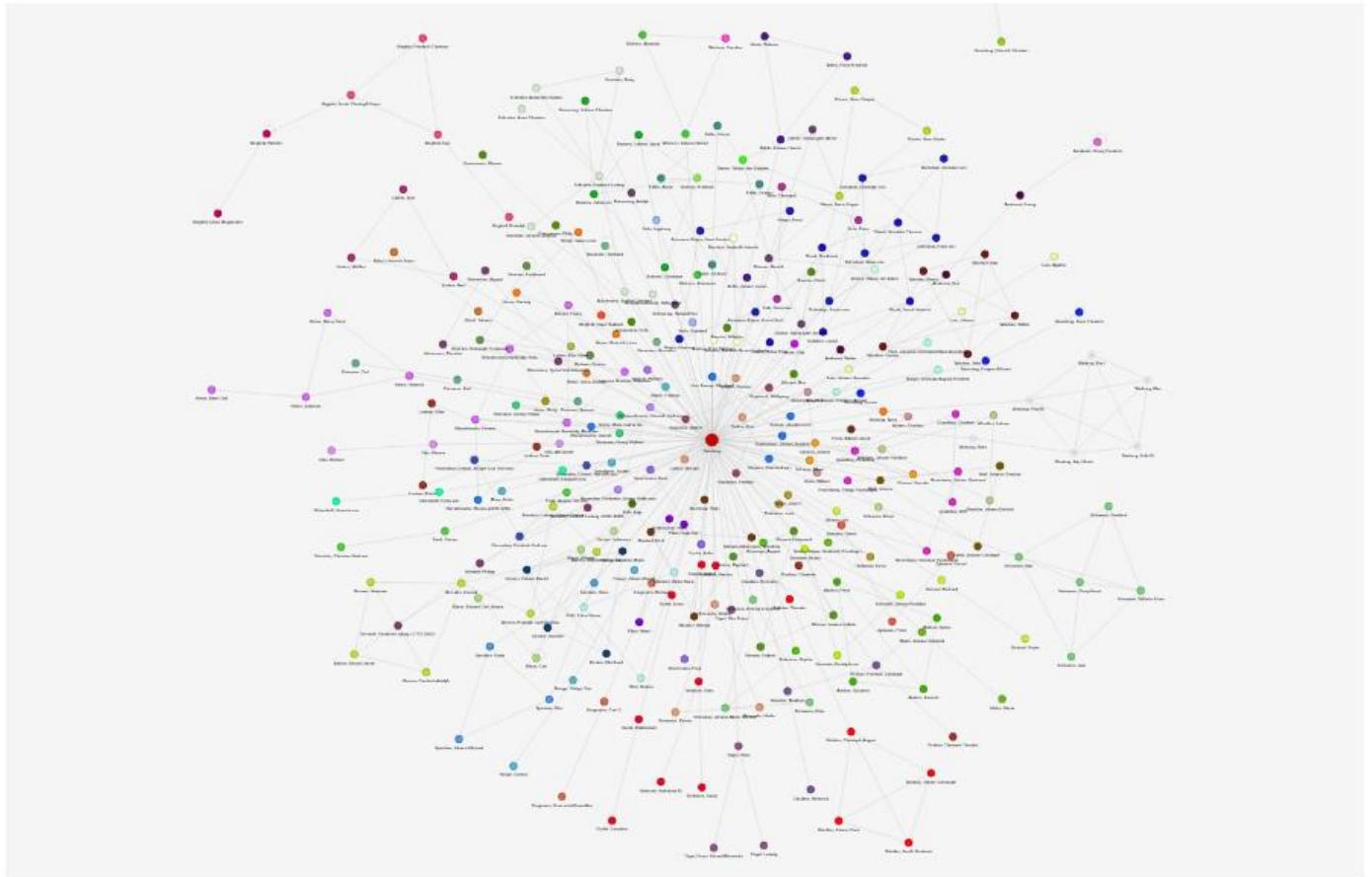
        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        random_colnumber = random.randint(0, 17687459)
        hex_number = str(hex(random_colnumber))
        hex_number = "#" + hex_number[2:]

        # Erzeugung der Hauptnodes
        g.add_node(name, color=hex_number, title=bio, size=12)

        # Erzeugung der relationierten Nodes sowie den Edges
        for rela in relation:
            g.add_node(rela, color=hex_number, size=12)
            g.add_edge(name, rela, color="#b5b5b5", smooth=False)

        # Erzeugung der "Hamburg-Edges"
        for item in hamburg_edges:
            if item == name:
                g.add_edge(hamburg, name, color="#b5b5b5", smooth=False)
            if item == rela:
                g.add_edge(hamburg, rela, color="#b5b5b5", smooth=False)

    # Layout-Algorithmen
    #g.barnes_hut(overlap=0.5, spring_length=870, central_gravity=1.5)
    #g.force_atlas_2based(spring_length=-10, overlap=0, spring_strength=0.099)
    #g.hrepulsion()
    g.repulsion(node_distance=259)
    g.show_buttons()
    g.set_options('' var options = {
    "#nodes": {"font": {"size": 20}},
    "#physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}}})
    #g.show("hhbib_networkGraph_5-65.html")
```



Graph #2

Die zweite Visualisierung zeigt einen Graphen, der ohne die Verbindung von Edges mit dem Hamburg-Node von einzelnen Familiensubnetzen geprägt ist und durch "weiche" Edges einen schwungvollen Charakter erhält. Die Nodes sind dabei durch einen leicht erhöhten Wert der central_gravity als Argument der Algorithmusfunktion "repulsion" in die Mitte gezogen. Die farbliche Dualität spiegelt Nodes mit und ohne "title"-Information wieder.

```
In [ ]:
with open("hh_persons_fam_2.json", "r", encoding = "utf-8") as infile:

    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    #hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    #with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
    #for line in infile_2:
        #hh_name = line[:-1]
        #hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#f5f5f5", font_color="black")

    # Erzeugung "Hamburg-Node"
    #for hamburg in data_fam_2:
        #g.add_node(hamburg, color="#2471A3", size=100)

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["relation_fam"]
        bio = person["bio"]

        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        #random_colnumber = random.randint(0, 17687459)
        #hex_number = str(hex(random_colnumber))
        #hex_number = "#" + hex_number[2:]
```

```

# Erzeugung der Hauptnodes
g.add_node(name, color="#CB4335", title=bio, size=11)

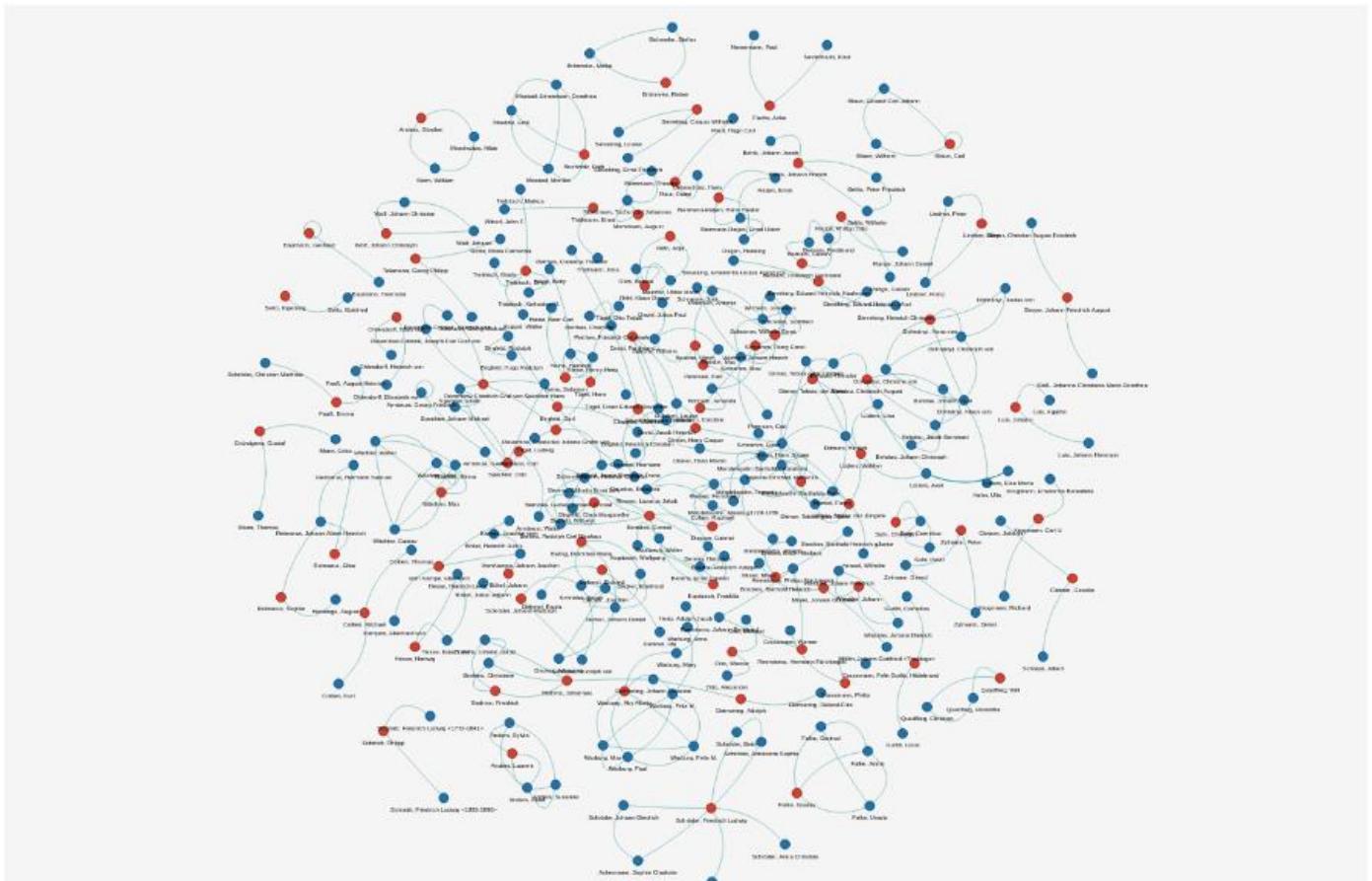
# Erzeugung der relationierten Nodes sowie den Edges
for rela in relation:
    g.add_node(rela, color="#2471A3", size=11)
    g.add_edge(name, rela, color="#018786")

    # Erzeugung der "Hamburg-Edges"
    #for item in hamburg_edges:
        #if item == name:
            #g.add_edge(hamburg, name)
        #if item == rela:
            #g.add_edge(hamburg, rela)

# Layout-Algorithmen
#g.barnes_hut()
#g.force_atlas_2based()
#g.hrepulsion()
g.repulsion(central_gravity=1.1)
#g.show_buttons()
#g.set_options(''' var options = {
#"nodes": {"font": {"size": 20}},
#"physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}}} ''')

# Erzeugung der Visualisierung und der html-Datei
g.show("hhbib_networkGraph_4-2.html")

```



Graph #3

Hier werden die Beziehungen zwischen Personen und Berufen dargestellt. Es sind die Default-Einstellungen des Algorithmus "repulsion" gewählt, der farbliche Unterschied der Nodes scheidet in Personen und Berufe. Es werden bei diesem Graphen andere Netzstrukturen generiert werden, als bei den Graphen mit einzelnen Subnetzen der Familien, die untereinander nicht verknüpft sind, da alle Teilnehmer theoretisch mit allen Berufen verknüpft sein können.

```
In [ ]: with open("hh_persons_job_5.json", "r", encoding = "utf-8") as infile:
    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    #hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    #with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
    #for line in infile_2:
        #hh_name = line[:-1]
        #hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#FBFCFC", font_color="black", layout=False)

    # Erzeugung "Hamburg-Node"
    #for hamburg in data_fam_2:
        #g.add_node(hamburg, color="#2471A3", size=100)

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["job"]
        bio = person["bio"]

        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        #random_colnumber = random.randint(0, 17687459)
        #hex_number = str(hex(random_colnumber))
        #hex_number = "#" + hex_number[2:]

        # Erzeugung der Hauptnodes
        g.add_node(name, color="#228b22", title=bio, size=10)
        #g.add_node(relation, color="#002b55", size=25)

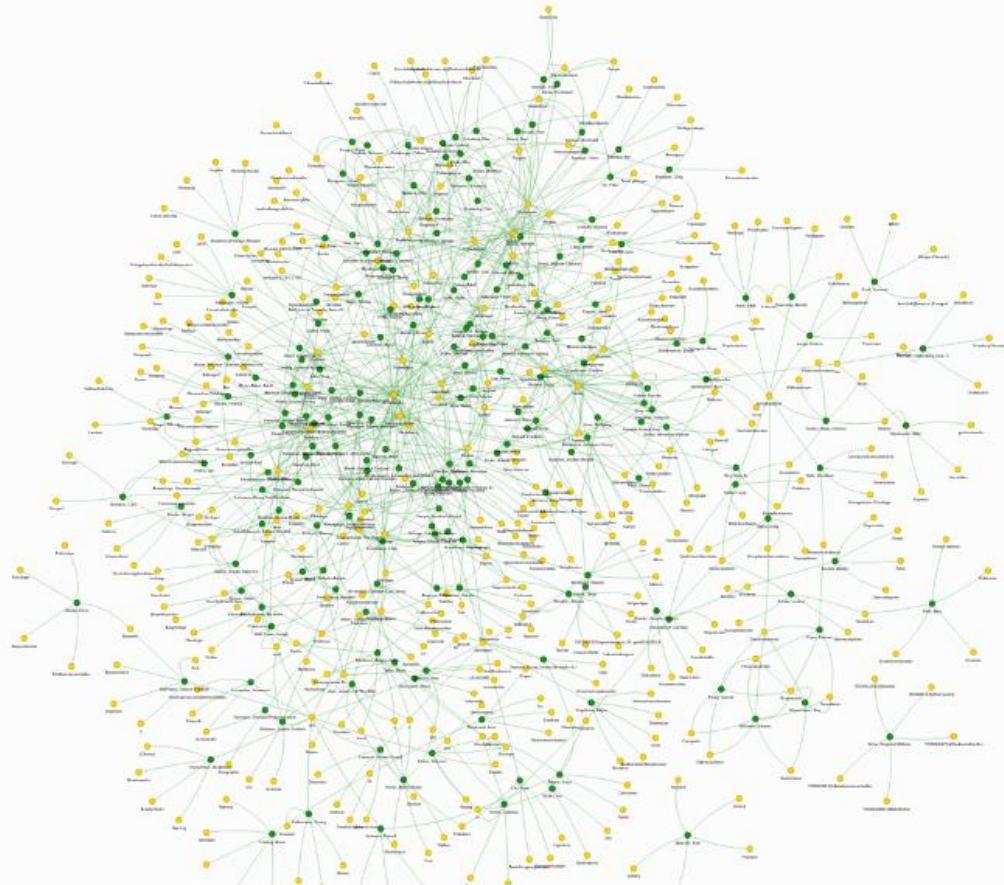
        # Erzeugung der relationierten Nodes sowie den Edges
        for rela in relation:
            g.add_node(rela, color="#ffd700", size=10)
            g.add_edge(name, rela, color="#228b22")

            # Erzeugung der "Hamburg-Edges"
            #for item in hamburg_edges:
                #if item == name:
                    #g.add_edge(hamburg, name)
                #if item == rela:
                    #g.add_edge(hamburg, rela)

    # Layout-Algorithmen
    #g.barnes_hut()
    #g.force_atlas_2based(gravity=-20, spring_length=400)
    #g.hrepulsion()
    g.repulsion()
    #g.show_buttons()
    #g.set_options('''

    #'' var options = {
    #'' "nodes": {"font": {"size": 20}},
    #'' "physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}}}
    #''')

    # Erzeugung der Visualisierung und der html-Datei
    g.show("hhbib_networkGraph_6-17.html")
```



Graph #4

Auch hier sind wieder Berufsbeziehungen dargestellt. Der Layoutalgorithmus "force_atlas_2based" in Kombination mit einem leicht niedrigeren Gravity-Wert als per Default vorgegeben bewirkt hier eine stärkere Anziehung, was den zusammengeballten Eindruck entstehen lässt. Der gravity-Faktor ist bei diesem Layoutalgorithmus zentral "[...] the central gravity model, which is here distance independent, and the repulsion being linear. [...]" (Pyvis 2022b).

```
In [ ]: with open("hh_persons_job_5.json", "r", encoding = "utf-8") as infile:
    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    #hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    #with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
    #for line in infile_2:
        #hh_name = line[:-1]
        #hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#FBFCFC", font_color="black")

    # Erzeugung "Hamburg-Node"
    #for hamburg in data_fam_2:
        #g.add_node(hamburg, color="#2471A3", size=100)

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["job"]
        bio = person["bio"]

        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        random_colnumber = random.randint(0, 17687459)
        hex_number = str(hex(random_colnumber))
        hex_number = "#" + hex_number[2:]
```

```

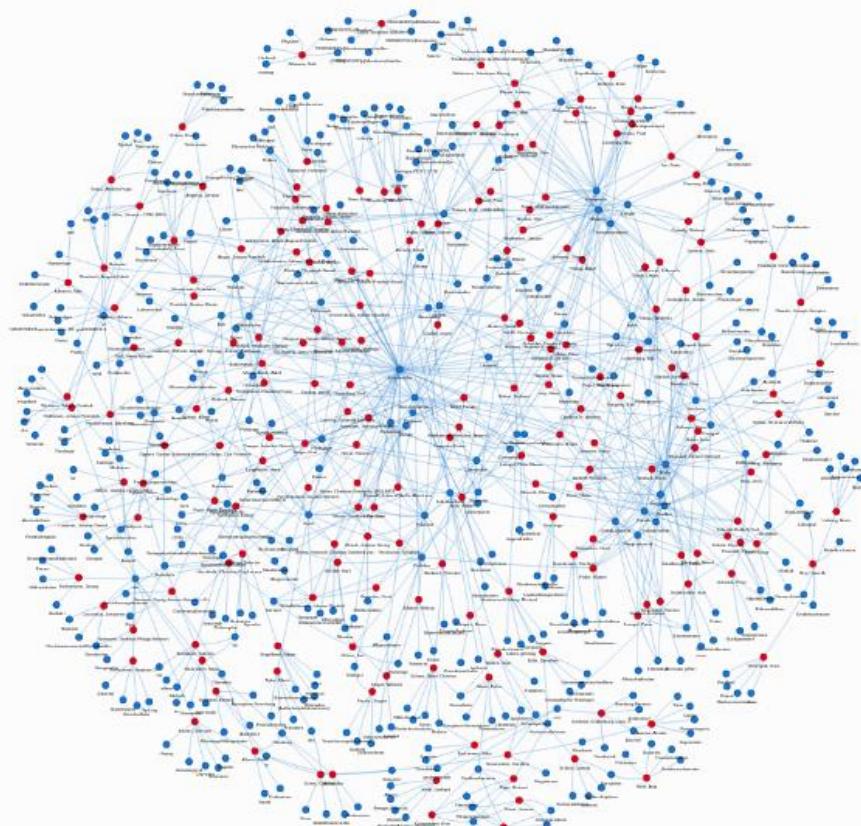
# Erzeugung der Hauptnodes
g.add_node(name, color="#e10019", title=bio, size=10)
#g.add_node(relation, color="#002b55", size=25)

# Erzeugung der relationierten Nodes sowie den Edges
for rela in relation:
    g.add_node(rela, color="#1874cd", size=10)
    g.add_edge(name, rela, color="#1874cd")

    # Erzeugung der "Hamburg-Edges"
    #for item in hamburg_edges:
        #if item == name:
            #g.add_edge(hamburg, name)
        #if item == rela:
            #g.add_edge(hamburg, rela)

# Layout-Algorithmen
#g.barnes_hut()
g.force_atlas_2based(gravity=-10)
#g.hrepulsion()
#g.repulsion()
#g.show_buttons()
#g.set_options('' var options = {
#"nodes": {"font": {"size": 20}},
#"physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}}}} '')
# Erzeugung der Visualisierung und der html-Datei
g.show("hhbib_networkGraph_6-11.html")

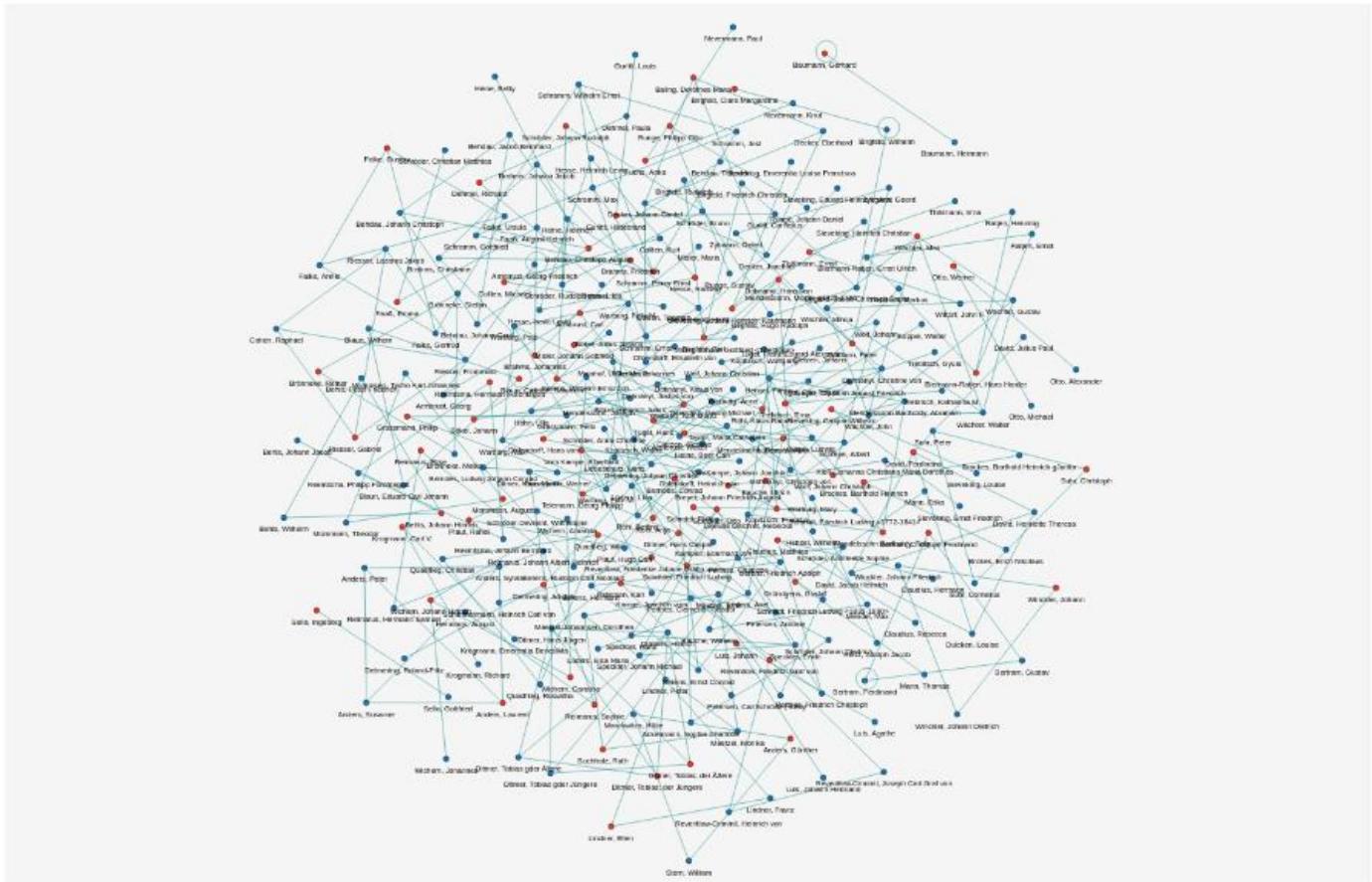
```



Graph #5

Wie stark visuelle Eindrücke durch nur kleine Veränderungen differieren können, zeigt der Graph #5 im Vergleich zu Graph #2. Allein durch die Veränderung der Edges zu geraden Linien entsteht individuell gefühlt ein völlig anderer Graph.

```
In [ ]: with open("hh_persons_fam_2.json", "r", encoding = "utf-8") as infile:  
    data_fam_2 = json.load(infile)  
    hhbib_data = data_fam_2["Hamburg"]  
    hamburg_edges = []  
  
    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"  
    #with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:  
    #for line in infile_2:  
    #    hh_name = line[:-1]  
    #    hamburg_edges.append(hh_name)  
  
    # Erzeugung des Graphen  
    g = Network(height="1000px", width="100%", bgcolor="#f5f5f5", font_color="black")  
  
    # Erzeugung "Hamburg-Node"  
    #for hamburg in data_fam_2:  
    #    g.add_node(hamburg, color="#2471A3", size=100)  
  
    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes  
    for person in hhbib_data:  
        name = person["name"][0]  
        relation = person["relation_fam"]  
        bio = person["bio"]  
  
        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes  
        #random_colnumber = random.randint(0, 17687459)  
        #hex_number = str(hex(random_colnumber))  
        #hex_number = "#" + hex_number[2:]  
  
        # Erzeugung der Hauptnodes  
        g.add_node(name, color="#CB4335", title=bio, size=5)  
  
        # Erzeugung der relationierten Nodes sowie den Edges  
        for rela in relation:  
            g.add_node(rela, color="#2471A3", size=5)  
            g.add_edge(name, rela, color="#018786", smooth=False)  
  
            # Erzeugung der "Hamburg-Edges"  
            #for item in hamburg_edges:  
            #    if item == name:  
            #        g.add_edge(hamburg, name)  
            #    if item == rela:  
            #        g.add_edge(hamburg, rela)  
  
            # Layout-Algorithmen  
            #g.barnes_hut()  
            #g.force_atlas_2based()  
            #g.hrepulsion()  
            g.repulsion(central_gravity=1.1)  
            #g.show_buttons()  
            #g.set_options('' var options = {  
            #    "nodes": {"font": {"size": 20}},  
            #    "physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}}} '')  
  
            # Erzeugung der Visualisierung und der html-Datei  
            g.show("hhbib_networkGraph_4-3.html")
```



Graph #6

Ein gänzlich anderer Ansatz zeigt Graph #6. Er ist einerseits wieder bestimmt durch strahlenförmig auskragende Edges durch die Verbindung mit dem Hamburg-Node bei den Familienrelationen. Die Edges werden hier aber durch den hohen negativen Wert der `spring_length` (wie lang die Edges auseinander "federn/schnellen") im Vergleich zur default-Einstellung bestimmt sowie besonders durch den Null-overlap-Wert, der die höchste Überlappungsstufe der Nodes bedeutet, was den zusammengezogenen Eindruck der Nodes auf einer Linie bewirkt.

```
In [ ]: with open("hh_persons_fam_2.json", "r", encoding = "utf-8") as infile:  
    data_fam_2 = json.load(infile)  
    hhbib_data = data_fam_2["Hamburg"]  
    hamburg_edges = []  
  
    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"  
    with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:  
        for line in infile_2:  
            hh_name = line[:-1]  
            hamburg_edges.append(hh_name)  
  
    # Erzeugung des Graphen  
    g = Network(height="1000px", width="100%", bgcolor="#f5f5f5", font_color="black")  
  
    # Erzeugung "Hamburg-Node"  
    for hamburg in data_fam_2:  
        g.add_node(hamburg, color="#668b8b", size=30)  
  
    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes  
    for person in hhbib_data:  
        name = person["name"][0]  
        relation = person["relation_fam"]  
        bio = person["bio"]  
  
        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes  
        random_colnumber = random.randint(0, 17687459)  
        hex_number = str(hex(random_colnumber))  
        hex_number = "#" + hex_number[2:]
```

```

# Erzeugung der Hauptnodes
g.add_node(name, color=hex_number, title=bio, size=12)

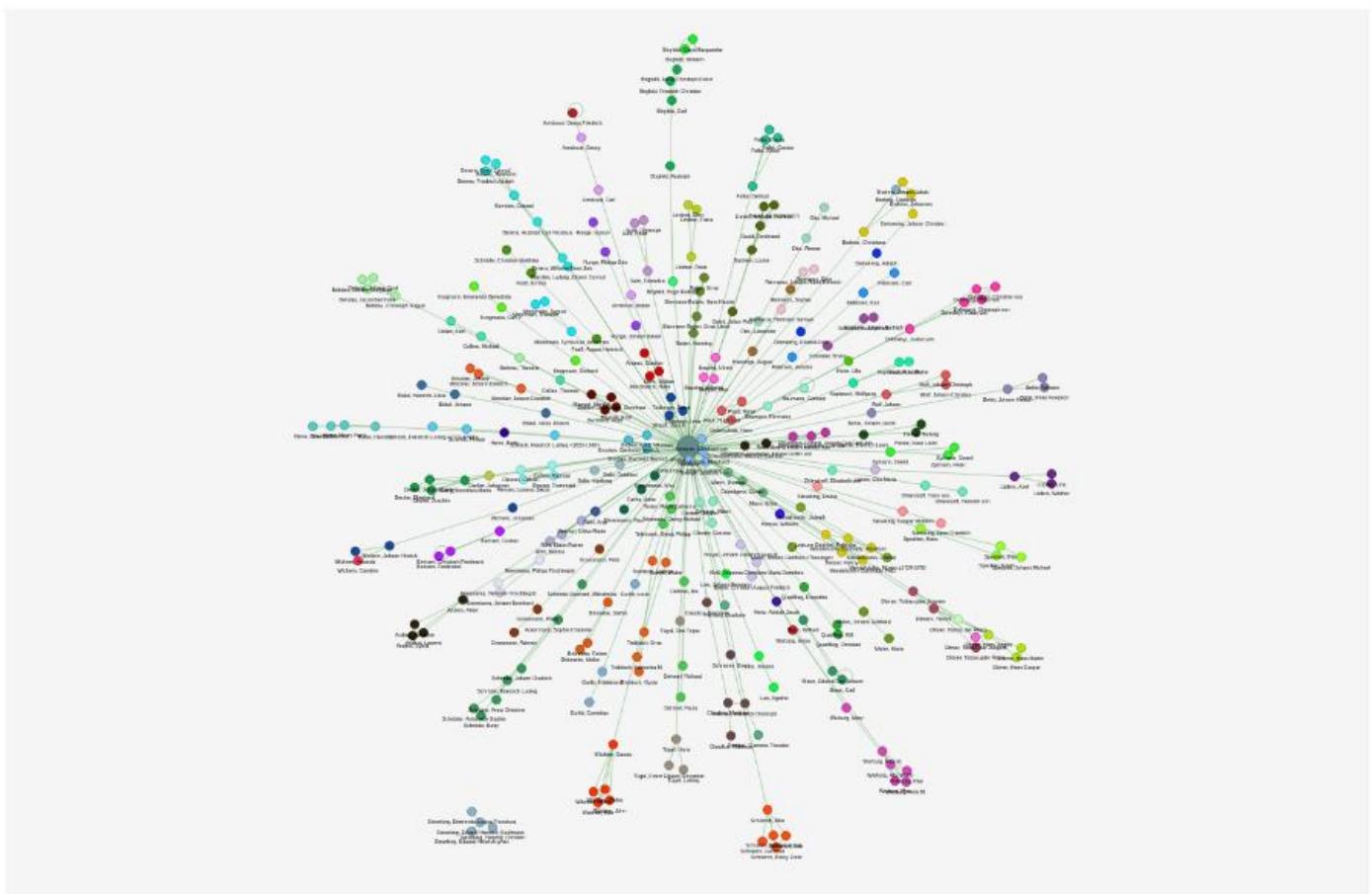
# Erzeugung der relationierten Nodes sowie den Edges
for rela in relation:
    g.add_node(rela, color=hex_number, size=12)
    g.add_edge(name, rela, color="#228b22", smooth=False)

# Erzeugung der "Hamburg-Edges"
for item in hamburg_edges:
    if item == name:
        g.add_edge(hamburg, name, color="#228b22", smooth=False)
    if item == rela:
        g.add_edge(hamburg, rela, color="#228b22", smooth=False)

# Layout-Algorithmen
#g.barnes_hut(overlap=0.5, spring_length=870, central_gravity=1.5)
g.force_atlas_2based(spring_length=-10, overlap=0, spring_strength=0.099)
#g.hrepulsion()
#g.repulsion(central_gravity=1.1)
#g.show_buttons()
#g.set_options(''' var options = {
#"nodes": {"font": {"size": 20}},
#"physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}}}} ''')

# Erzeugung der Visualisierung und der html-Datei
g.show("hhbib_networkGraph_5-39.html")

```



Die nächsten drei Visualisierungen zeigen Varianten, die - will man die Informationen, die in den Daten des Netzwerkgraphen enthalten sind, transportieren - trotz aller Interaktivität der Software dazu nicht so geeignet sind. Zwar können visuell interessante Eindrücke entstehen, aber will man Zusammenhänge oder Node-Labels erkennen, mögen diese Varianten nicht sehr hilfreich sein.

Graph #7

In diesem Graphen werden Familiensubnetze dargestellt, unverbunden mit dem Hamburg-Node. Er ist mit den Default-Einstellungen des Layoutalgorithmus "hrepulsion" erstellt. Dieser Algorithmus ist grundsätzlich schwierig, will man Label-Text lesen können, da sehr stark in den Graphen gezoomt werden muss, bis die Labels lesbar sind. Der Algorithmus basiert ausserdem auf Abstossung der Nodes. Diese funktioniert besser, je mehr Nodes ein Netz hat. Durch die Nutzung des Datenpools, der auch Familienbeziehungen mit nur einem Verwandeten beinhaltet, entstehen "Zweiernetze", deren Nodes dadurch, hier begünstigt durch die Werte von spring_length und spring_strength, übereinandergelagert werden. Auch nicht komfortabel für eine Nutzung.

```
In [ ]: with open("hh_persons_fam_all.json", "r", encoding = "utf-8") as infile:
    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    #with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
    #for line in infile_2:
    #    hh_name = line[:-1]
    #    hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#f5f5f5", font_color="black")

    # Erzeugung "Hamburg-Node"
    #for hamburg in data_fam_2:
    #    g.add_node(hamburg, color="#2471A3", size=100)

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["relation_fam"]
        bio = person["bio"]

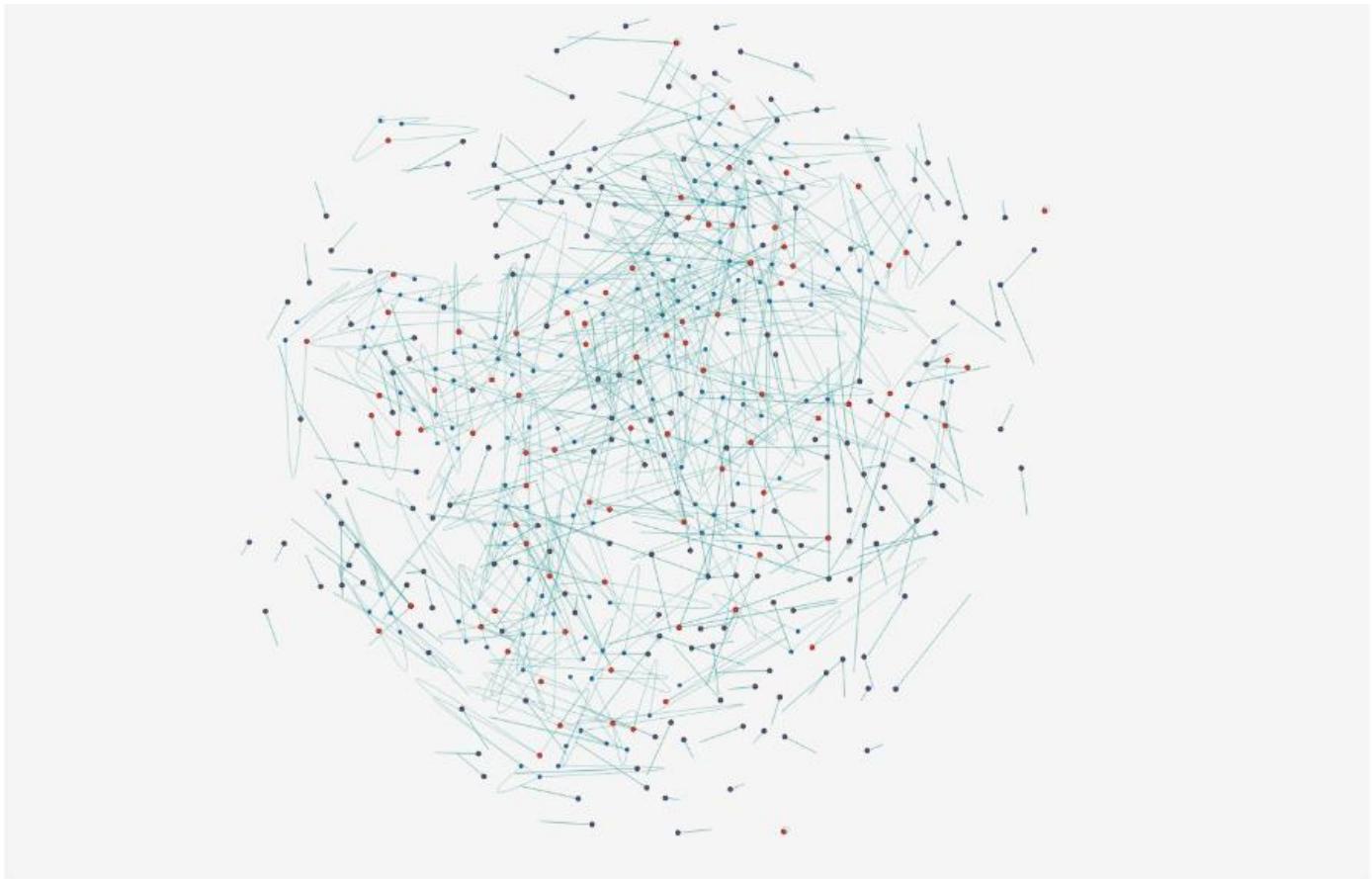
        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        #random_colnumber = random.randint(0, 17687459)
        #hex_number = str(hex(random_colnumber))
        #hex_number = "#" + hex_number[2:]

        # Erzeugung der Hauptnodes
        g.add_node(name, color="#CB4335", title=bio, size=13)

        # Erzeugung der relationierten Nodes sowie den Edges
        for rela in relation:
            g.add_node(rela, color="#2471A3", size=10)
            g.add_edge(name, rela, color="#018786")

            # Erzeugung der "Hamburg-Edges"
            #for item in hamburg_edges:
            #    if item == name:
            #        g.add_edge(hamburg, name)
            #    if item == rela:
            #        g.add_edge(hamburg, rela)

    # Layout-Algorithmen
    #g.barnes_hut()
    #g.force_atlas_2based()
    g.hrepulsion()
    #g.repulsion()
    #g.show_buttons()
    #g.set_options('' var options = {
    #    "nodes": {"font": {"size": 20}},
    #    "physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}} } '')
    # Erzeugung der Visualisierung und der html-Datei
    g.show("hhbib_networkGraph_3-1.html")
```



Graph #8

Hier wird die Unpraktikabilität durch die Unseparierbarkeit der einzelnen Familienubnetze bedingt. Auch hier sind Familienbeziehungen ohne Verbindung zum Hamburg-Node dargestellt. Der Zug der einzelnen Subnetze zum Graphmittelpunkt, will man sie aus dem Konglomerat herausziehen, um zu erkennen, welche Personen dazu gehören, ist so stark, dass dieses Separieren per Drag nicht optimal funktioniert.

```
In [ ]:
with open("hh_persons_fam_2.json", "r", encoding = "utf-8") as infile:

    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    #hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    #with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
    #for line in infile_2:
        #hh_name = line[:-1]
        #hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#f5f5f5", font_color="black")

    # Erzeugung "Hamburg-Node"
    #for hamburg in data_fam_2:
        #g.add_node(hamburg, color="#2471A3", size=100)

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["relation_fam"]
        bio = person["bio"]

        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        random_colnumber = random.randint(0, 17687459)
        hex_number = str(hex(random_colnumber))
        hex_number = "#" + hex_number[2:]
```

```

# Erzeugung der Hauptnodes
g.add_node(name, color="#ff9900", title=bio, size=12)

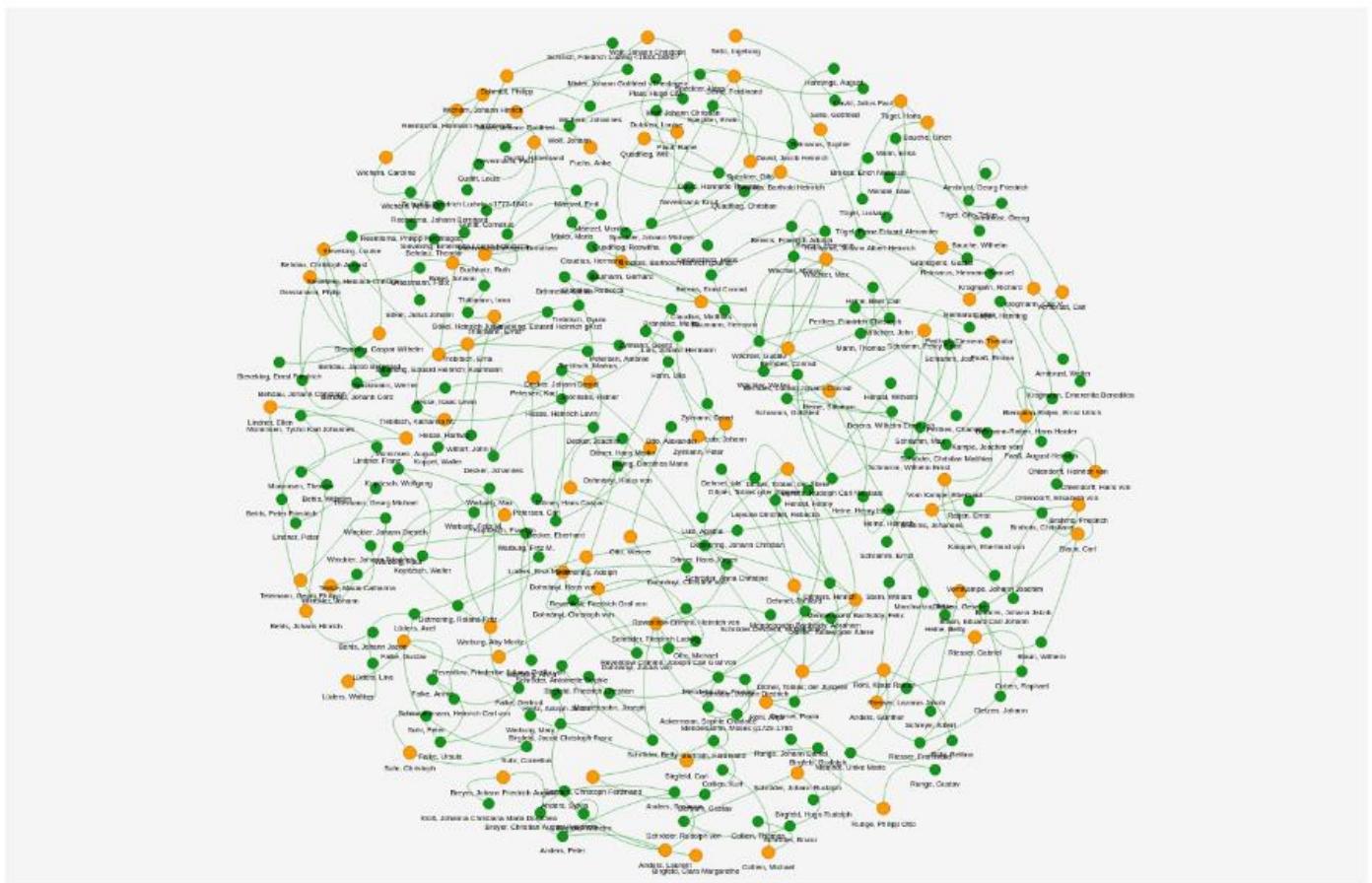
# Erzeugung der relationierten Nodes sowie den Edges
for rela in relation:
    g.add_node(rela, color="#109618", size=10)
    g.add_edge(name, rela, color="#109618")

    # Erzeugung der "Hamburg-Edges"
    #for item in hamburg_edges:
        #if item == name:
            #g.add_edge(hamburg, name)
        #if item == rela:
            #g.add_edge(hamburg, rela)

# Layout-Algorithmen
#g.barnes_hut(overlap=0.5, spring_length=870, central_gravity=1.5)
g.force_atlas_2based(spring_length=220, central_gravity=0.05, overlap=1)
#g.hrepulsion()
#g.repulsion(central_gravity=1.1)
#g.show_buttons()
#g.set_options(''')
#''nodes': {"font": {"size": 20}},
#''physics': {"barnesHut": {"springLength": 400, "springConstant": 0.1}}}
'''')

# Erzeugung der Visualisierung und der html-Datei
g.show("hhbib_networkGraph_5-9.html")

```



Graph #9

Dieser Graph wurde auf Grundlage der Verbindung der Familiennetze mit dem Hamburg-Node mittels den Default-Einstellungen des Layoutalgorithmus "barnesHut" erstellt. Auch hier entsteht eine hübsch anzusehende Sonne mit ihren sich leicht bewegenden Planeten, aber auch hier ist der Zoom-Faktor so groß, dass - ist die Information erst einmal erreicht - sie nicht mehr visuell in den Beziehungszusammenhang gebracht werden kann, ohne wieder zurück zoomen zu müssen.

```
In [16]: with open("hh_persons_fam_2.json", "r", encoding = "utf-8") as infile:
    data_fam_2 = json.load(infile)
    hhbib_data = data_fam_2["Hamburg"]
    hamburg_edges = []

    # Laden der Datei "hamburg_edges.txt" und Erzeugung der Liste für die "Hamburg-Edges"
    with open("hamburg_edges.txt", "r", encoding = "utf-8") as infile_2:
        for line in infile_2:
            hh_name = line[:-1]
            hamburg_edges.append(hh_name)

    # Erzeugung des Graphen
    g = Network(height="1000px", width="100%", bgcolor="#f5f5f5", font_color="black")

    # Erzeugung "Hamburg-Node"
    for hamburg in data_fam_2:
        g.add_node(hamburg, color="#2471A3", size=100, physics=False, color_borderWidth="#00ff00")

    # Erzeugung von Variablen für Nodes, relationierte Nodes, Title der Nodes
    for person in hhbib_data:
        name = person["name"][0]
        relation = person["relation_fam"]
        bio = person["bio"]

        # Erzeugung einer Variablen zur Zuweisung von zufällig generierten Hex-ColorCodes zu den Nodes
        #random_colnumber = random.randint(0, 17687459)
        #hex_number = str(hex(random_colnumber))
        hex_number = "#" + hex_number[2:]

        # Erzeugung der Hauptnodes
        g.add_node(name, color="#CB4335", title=bio, size=70)

        # Erzeugung der relationierten Nodes sowie den Edges
        for rela in relation:
            g.add_node(rela, color="#2471A3", size=70)
            g.add_edge(name, rela, color="#018786")

        # Erzeugung der "Hamburg-Edges"
        for item in hamburg_edges:
            if item == name:
                g.add_edge(hamburg, name)
            if item == rela:
                g.add_edge(hamburg, rela)

    # Layout-Algorithmen
    g.barnes_hut()
    #g.force_atlas_2based(spring_length=0)
    #g.hrepulsion()
    #g.repulsion()
    #g.show_buttons()
    #g.set_options('' var options = {
    #  "nodes": {"font": {"size": 20}},
    #  "physics": {"barnesHut": {"springLength": 400, "springConstant": 0.1}} } '')
    # Erzeugung der Visualisierung und der html-Datei
    g.show("hhbib_networkGraph_5-21.html")
```

