Protokoll Fingerübung-Auswertung

Elham Amin
6737322

April 2021

Aufgabe 2

23 - Tool zum Einlesen und Analysieren der XML-Dateien

Das Programm 'exerciseuntenstrich3' ließt .xml Dateien und geht folgende Fragen durch:

- Wie oft kommen welche PoS-Tags vor?
- Wie viele [SpatialEntities, Places, Motions, Locations, Signals, QsLinks, OLinks] gibt es?
- Wie oft kommen welche QsLink Typen vor? (DC,EC, ...)?
- Verteilung der Satzlänge graphisch darstellen (x: Satzlänge, y: Wie häufig)?
- Welche Links (QSLinks, OLinks) werden von welchen Präpositionen (markiert durch SPATIALSIGNAL) getriggert (z.B. wie oft werden QSLinks durch die Präposition „on“ getriggert)?
- Welches sind die fünf häufigsten „MOTION“ Verben (und wie oft kommen diese vor)?

Ich werde im folgenden die Funktionen, die bei der Beantwortung dieser Fragen helfen, erläutern.

function: main()

Die main()-Funktion ist der Hauptpfad unseres Tools. Hier werden die anderen Funktionen aufgerufen und mithilfe der main()-Funktion ausgegeben.
Sie fragt ab (mithilfe der input()-Funktion), welcher Dateipfad (favorisierbar mit xml-Dateien) gewählt werden soll, um die (möglichst) vorhandenen Dateien zu analysieren. Dadurch, dass uns Trainingsdateien zur Verfügung stehen, gehe ich in dieser Dokumentation im Idealfall immer davon aus, dass xml-Dateien im eingefügten Dateipfad gegeben werden und keine Fehlangabe gemacht wird (z.B. kein gültiger Dateipfad). Ich habe dennoch eine Fehlerabfangung (mithilfe einer if-Abfrage in Zeile 197,251) eingebracht, um den Code stabiler zu bauen.

Nach der Eingabe des Dateipfads wird dieser angepasst und alle vorhandenen xml-Dateien werden rausgesucht und in eine Liste gepackt, um die Datei für andere Funktionen bereitzulegen.
Zuallerletzt werden die errechneten Daten von der main()-Funktion genommen und mithilfe der .write-Befehle als Text-Datei in dem pfad abgespeichert, in dem das Programm liegt.

Nun zu den Funktionen, die in der main() aufgerufen werden:

function: readxml()

Die readxml()-Funktion nimmt eine xml-Datei entgegen und ließt hierbei die vorhande Textvorlage der Datei. Dies geschieht mithilfe einer built-in-function aus dem Import xml.etree.ElementTree und spacy. Dabei wird die xml-Datei wie ein Baumgraph durchlaufen, wobei hier ab der Wurzel ¡¡TEXT¿¿ Wort für Wort durchgegangen wird und in einem leeren String namens '**finaltext**' die Wörter einzeln abgespeichert werden.

function: token()

Mithilfe des Moduls 'spacy' wird der Text in der xml-Datei durchgegangen und tokenisiert. Die Funktion hilft dabei, die Anzahl der POS-Tags zu bestimmen.

function: sumpostags()

Für die Summe aller jeweils vorkommender POS-Tags wird diese Funktion genutzt. Sie nimmt den tokenisierten Text der xml-Datei an und überprüft, wie oft welcher POS-Tag im Text vorkommt. Zum einzelnen Summieren aller vorgegebenen POS-Tags habe ich ein Dictionary zur Hilfe genommen, wobei der Key-Value als counter für jeden einzelnen Key-POS-Tag genutzt wurde.

Wenn einer der im Dictionary vorgegeben POS-Tags im tokenisierten Text vorhanden war, wird der Key-Value des passenden Keys um +1 erhöht.
Die Ausgabe ist das fertiggestellte Dictionary mit der Anzahl aller POS-Tags.

function: rel()

Mithilfe der rel()-Funktion wird die Anzahl der relTypes der QS-Links in der xml-Datei bestimmt. Hier ist wieder das gleiche Vorgehen wie in sumpostags():

Ein Dictionary mit allen möglichen relTypes wurde erstellt und dient als counter.

Hier ist der Unterschied zu sumpostags(), dass die xml-Datei von der Funktion genommen wird, nicht der schon tokenisierte Text. In der rel()-Funktion ist die Tokenisierung des Texts auf die QS-Links beschränkt und wird hier einzeln behandelt. Das wird sich in den noch folgenden Funktionen wiederholen.
Die Ausgabe ist das fertiggestellte Dictionary.

function: which_prep()

In dieser Funktion wird die Anzahl aller vorkommender QS-Link Typen bestimmt und von welchen Spatial-Signals sie getriggered wurden. Hierbei wird erneut die xml-Datei durchgegangen, die ID's von den Spatial-Signals, der dazugehörige 'text' und eine Null als Integer (dient als counter) und die ID's der QS-Links mit dem 'trigger'. Die abgefragten Daten werden in zwei verschiedene Listen (qs, spatial) abgespeichert.

Der 'trigger' im QS-Link hilft zu erkennen, um welches Spatial-Signal es sich handelt, denn der 'trigger' ist nichts weiter als die ID des passenden Spatial-Signals. Hier wird mithilfe einer If-Abfrage überprüft, welche 'trigger' gleich mit welchen Spatial-Signal ID's sind. Wenn die Abfrage True ist, wird der counter vom passenden Spatial-Signal um +1 erhöht und die ID des QS-Links in die Liste mit den Angaben der Spatial Signals hinzugefügt.
Die Ausgabe ist eine Liste, welche die QS-Links aufzählt, die von welchem Wort wie oft getriggered wurden, zurückgibt.

function: motion_verbs()

Die Anzahl der 5 am meisten vorkommenden Motionverben wird hier zurückgegeben.

Die Aufsummierung ist hier im Vergleich zu den vorherigen Funktionen etwas anders (in der finalen Ausgabe):

Ich habe ein leeres Dictionary gebaut und bin erneut die xml-Datei durchgegangen, wobei mithilfe einer if-Abfrage die 'texts' der MOTION-Tags als Keys ins Dictionary hinzugefügt wurden. Der Key-Value war hier wieder jeweils 0. Nun konnte ich wie auch vorher in anderen Funktionen erläutert durchzählen, wie oft welches Motionverb in der xml-Datei vorhanden war und habe bei einer Findung den jeweiligen Key-Value um +1 erhöht.
Wichtig ist jetzt die Sortierung und Ausgabe der fünf am meisten vorkommenden Motionverben.

Ich habe hier das Dictionary nach den Key-Values von klein nach groß sortiert und das Dictionary in eine Liste umgewandelt. Mithilfe von Slicing habe ich die letzten fünf Elemente ausgegeben.

function: sentencelenght_graph()

Mithilfe des Moduls 'matplotlib.pyplot as plt' habe ich die Verteilung der Satzlänge pro Text visualisieren können.

Hierbei ist die x-Achse die Satzlänge im Text der xml-Datei und die y-Achse die Häufigkeit jeder vorkommenden Satzlänge. Eine Liste mit allen Satzlängen wird erstellt, indem der Text der xml-Datei (befinden sind in der Liste bloop) durchgegangen wird. Hierbei ist doc.sents (als doc = nlp()) genutzt worden. 'doc' ist hierbei die Liste aller Sätze, wobei dann die Längen mit len() gefunden wurde und in eine seperate Liste abgespeichert worden sind.

Eine weitere Liste mit der Häufigkeit jeder Satzlänge wurde erstellt, indem mit der Hilfsfunktion count() die Anzahl aller Elemente in der Liste mit den Satzlängen einzeln gezählt wurden (z.B.:[1,2,2,3] würde in count für 1 die Summe 1 angeben und für 2 die Summe 3). Beide listen sortiert wurden nun in matplotlib als x-und y-Achse genutzt und mit plt.bar() erstellt. Die Funktion speichert den graph als Bild ab, fragt jedoch vorher, wie die Bilddatei heißen soll.

Die Bilder befinden sich im selben Dateipfad wie das Programm.

0.1 Schlussworte - main()

Nach den Durchgängen aller Funktionen werden die Daten, wie vorher erwähnt, als txt-Datei abgespeichert. zum Schluss wird gefragt, wie die txt-Datei heißen soll und ein neuer Pfad kann direkt angegeben werden.

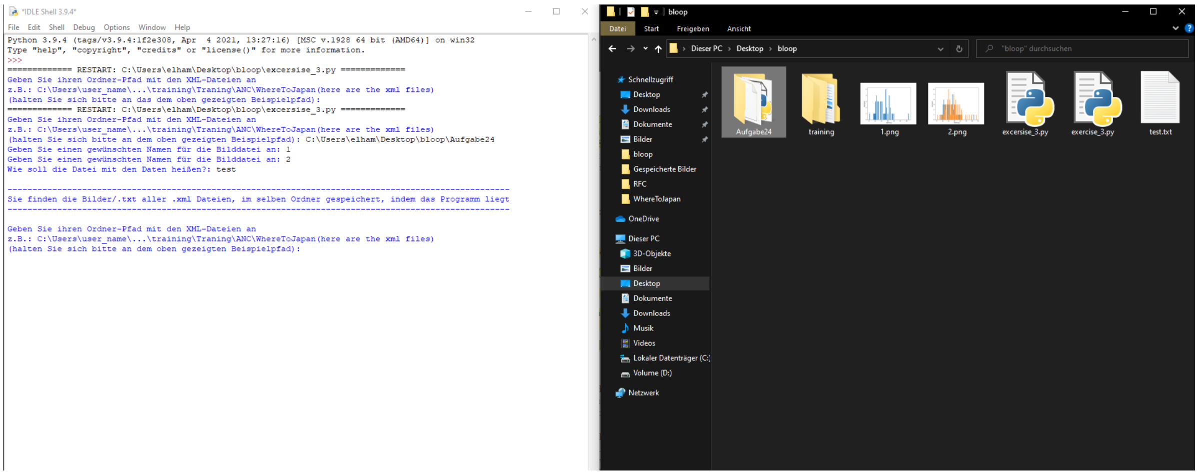
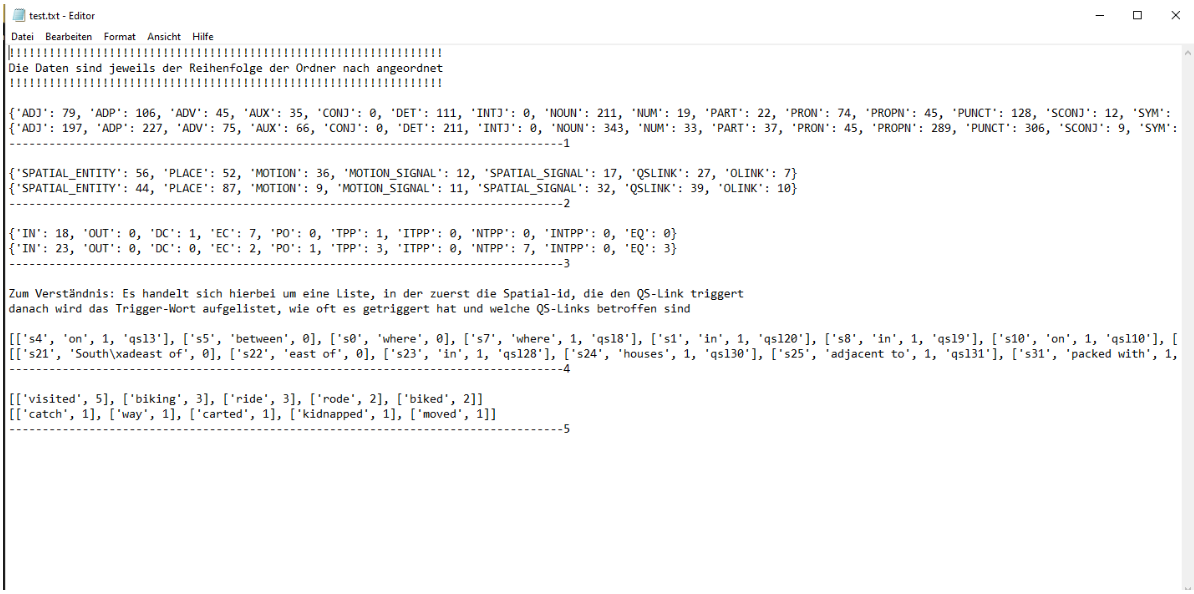


Abbildung 1: Eine Beispielausführung



```
test.txt - Editor
Datei  Bearbeiten  Format  Ansicht  Hilfe
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
Die Daten sind jeweils der Reihenfolge der Ordner nach angeordnet
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

{'ADJ': 79, 'ADP': 106, 'ADV': 45, 'AUX': 35, 'CONJ': 0, 'DET': 111, 'INTJ': 0, 'NOUN': 211, 'NUM': 19, 'PART': 22, 'PRON': 74, 'PROPN': 45, 'PUNCT': 128, 'SCONJ': 12, 'SYM':
{'ADJ': 197, 'ADP': 227, 'ADV': 75, 'AUX': 66, 'CONJ': 0, 'DET': 211, 'INTJ': 0, 'NOUN': 343, 'NUM': 33, 'PART': 37, 'PRON': 45, 'PROPN': 289, 'PUNCT': 306, 'SCONJ': 9, 'SYM':
----------------------------------------------------------------------------------1

{'SPATIAL_ENTITY': 56, 'PLACE': 52, 'MOTION': 36, 'MOTION_SIGNAL': 12, 'SPATIAL_SIGNAL': 17, 'QSLINK': 27, 'OLINK': 7}
{'SPATIAL_ENTITY': 44, 'PLACE': 87, 'MOTION': 9, 'MOTION_SIGNAL': 11, 'SPATIAL_SIGNAL': 32, 'QSLINK': 39, 'OLINK': 10}
----------------------------------------------------------------------------------2

{'IN': 18, 'OUT': 0, 'DC': 1, 'EC': 7, 'PO': 0, 'TPP': 1, 'ITPP': 0, 'NTPP': 0, 'INTPP': 0, 'EQ': 0}
{'IN': 23, 'OUT': 0, 'DC': 0, 'EC': 2, 'PO': 1, 'TPP': 3, 'ITPP': 0, 'NTPP': 7, 'INTPP': 0, 'EQ': 3}
----------------------------------------------------------------------------------3

Zum Verständnis: Es handelt sich hierbei um eine Liste, in der zuerst die Spatial-id, die den QS-Link triggert
danach wird das Trigger-Wort aufgelistet, wie oft es getriggert hat und welche QS-Links betroffen sind

[['s4', 'on', 1, 'qsl3'], ['s5', 'between', 0], ['s0', 'where', 0], ['s7', 'where', 1, 'qsl8'], ['s1', 'in', 1, 'qsl20'], ['s8', 'in', 1, 'qsl9'], ['s10', 'on', 1, 'qsl10'], [
[['s21', 'South\xadeast of', 0], ['s22', 'east of', 0], ['s23', 'in', 1, 'qsl28'], ['s24', 'houses', 1, 'qsl30'], ['s25', 'adjacent to', 1, 'qsl31'], ['s31', 'packed with', 1,
----------------------------------------------------------------------------------4

[['visited', 5], ['biking', 3], ['ride', 3], ['rode', 2], ['biked', 2]]
[['catch', 1], ['way', 1], ['carted', 1], ['kidnapped', 1], ['moved', 1]]
----------------------------------------------------------------------------------5
```

Abbildung 2: Die Text-Datei mit den abgespeicherten Daten der Analyse

4 - Visualisierung von Bicycle.xml und Highlights_of_the_Prado_Museum.xml

Das Programm 'exercise4' visualisiert die ausgewählten xml-Dateien als Graphen unter folgender (zitierten) Kriterien:

'Visualisieren Sie das Dokument Bicycles.xml und HighlightsofthePradoMuseum.xml grafisch. Stellen Sie dazu alle räumlichen Entitäten (PLACE, LOCATION, SPATIALENTITY, NONMOTIONEVENT, PATH) als Knoten da farblich zugeordnet nach Klasse und beschriftet mit deren entsprechenden Textbeschreibung. Entitäten, die dabei mit einem METALINK verknüpft, sollen als ein Knoten dargestellt werden (mergen durch Koreferenz). Als Kanten sollen OLINKS und QSLINKS zwischen den Knoten eingezeichnet werden, wieder farblich erkennbar und relType als Label. Die Trigger müssen dabei nicht beachtet werden. Dabei ist es ihnen überlassen, welches Tool Sie zur Visualisierung verwenden (Neo4j, Graphviz, Networkx, ...).'

Bevor ich erläutere, wie ich logisch vorgegangen bin, möchte ich hier eine kleine Auffassung der Kriterien (wie ich sie verstanden habe) darlegen:

Für die Visualisierung der beiden Dateien habe ich die Entitäten als Farben unterteilt (PLACE=blau, LOCATION=gelb, SPATIALENTITY=rot, NONMOTIONEVENT=grün, PATH=magenta/pink). Die Namen der Knoten sind die Textbeschreibungen als Attribut 'text', die in jeder Entität vorkommen. Sollte es eine Verknüpfung zwischen METALINK und einer Entität geben, so werden die die 'toText's mit den Knoten gemerged.
Die Kanten bilden sich aus den QS- und O-Links, wobei der ''relType' als Kantenbeschriftung genutzt wird und die ID's 'from' und 'to' zur Hilfe genutzt werden.

Anmerkung: Ich habe es nicht hinbekommen, den relType in den Graphen angezeigt zu bekommen (gehe gleich weiter darauf ein und zeige die Zeilen meines versuchten Ansatzes).

Nun weiter zu den Funktionen, die mir bei der Bearbeitung von Nutzen waren:

function: main()

Das Gleiche wie auch im anderen Programm: Der 'Hauptteil' des Programms. Es wird gelegentlich wieder nach dem Dateipfad gefragt (da er von User zu user unterschiedlich sein kann) und die zwei Graphen werden nacheinander ausgegeben.

function: edges_xml()

Wie auch schon oben erwähnt werden wie auch im anderen Programm die xml-Dateien durchgegangen und die O-und QS-Links durchgegangen und dann jeweilig die reltypes,

from und to ID's in eine liste abgespeichert und zurückgegeben.

function: nodes()

Die Knoten werden hier in einer Liste zurückgegeben. Dabei muss jeweils jede toID, jeder fromText und toText aller METALINKS extrahiert werden (die ID's werden nachher für das Mergen gebraucht). Dabei wird auch jedes Attribut Text von allen Entitäten genommen und in einer Liste abgespeichert. Hier wird lediglich abgefragt, ob alle METALINK-Daten, die rausgenommen worden sind (toID's) den ID's der Entitäten entsprechen. Wenn ja, wird gemerged.

Die Ausgabe ist dementsprechend eine Liste mit allen abgespeicherten ID's, dem Text (Knotenname) und dem passenden TAG.

function: graphs_show()

graphsshow() visualisiert den bestimmten Graphen mithilfe des Moduls networkx as nx und matplotlib.pyplot as plt und nimmt die Listen mit den Daten für die Knoten und Kanten entgegen.
Hierbei wird ein leerer Graph erstellt, dem die Kanten und Knoten hinzugefügt werden. Mit einer Liste color wird den Entitäten die oben aufgeführten Farben zugeordnet. Nun werden die Kanten von der entgegengenommen Liste mit den Daten der Kanten entnommen und in einen passenden Typen (Tupel) geformt. Mithilfe einer Erkennungszahl, die ich in der Funktion edgesxml() für jeden O-und QS-Link eingebracht habe(1=O-Link, 2=QS-Link) werden noch die Kanten farblich unterscheidet.

Die Ausgabe erfolgt mit 'nx.draw()'.

relType Fehler

Ich habe versucht, mit dem nx.drawnetworkx-edge-labels(g,..) Aufruf versucht, die relTypes als Labels der Kanten anzugeben, jedoch erscheint nach und nach wieder, dass etwas mit dem Typen nicht stimmt. (siehe meinen Ansatz auf Zeile 68-73). Ich habe nach weiterer Recherche entdeckt, dass das Label ein Tupel darstellt(z.B. von Knoten A zu Knoten B wäre der passende Labeltyp ('a','b')), doch selbst als Tupelangabe kam der gleiche Fehler. Ich habe auch versucht, eine extra Liste mit den passenden relTypes zu erstellen und als edge_labels=[(..,..),...] anzugeben, jedoch kam wieder ein Typfehler.

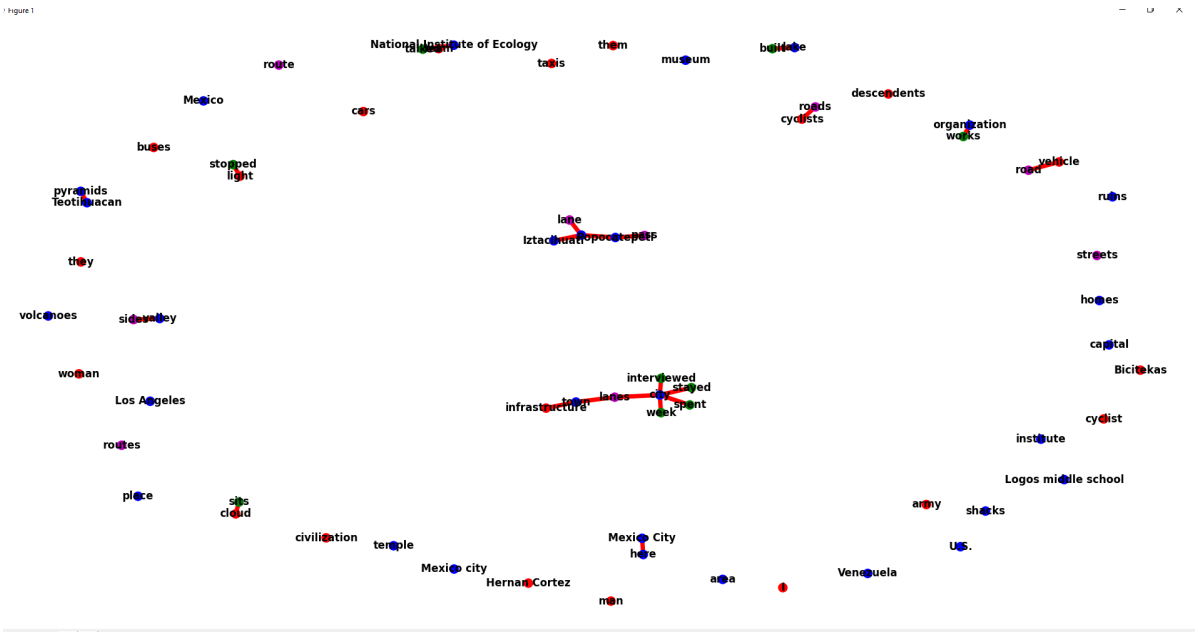


Abbildung 3: Der visualisierte bicyc-Graph

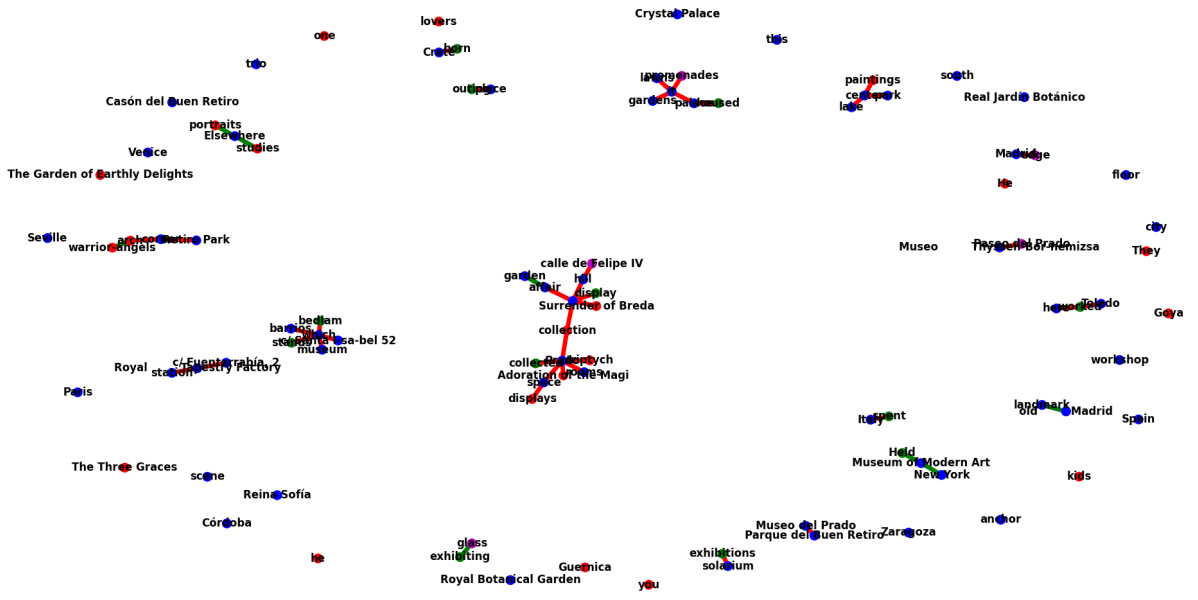


Abbildung 4: Der visualisierte museum-Graph