

EDUCATIONAL ARTICLE

Open Access



Visualizing chemical space networks with RDKit and NetworkX

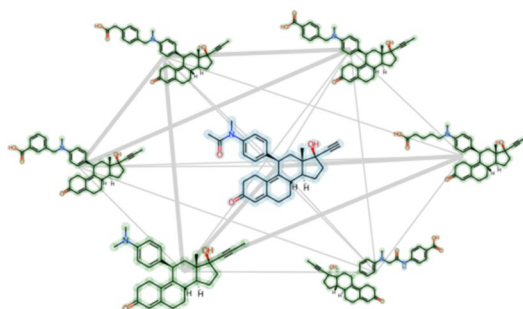
Vincent F. Scalfani*, Vishank D. Patel and Avery M. Fernandez

Abstract

This article demonstrates how to create Chemical Space Networks (CSNs) using a Python RDKit and NetworkX workflow. CSNs are a type of network visualization that depict compounds as nodes connected by edges, defined as a pairwise relationship such as a 2D fingerprint similarity value. A step by step approach is presented for creating two different CSNs in this manuscript, one based on RDKit 2D fingerprint Tanimoto similarity values, and another based on maximum common substructure similarity values. Several different CSN visualization features are included in the tutorial including methods to represent nodes with color based on bioactivity attribute value, edges with different line styles based on similarity value, as well as replacing the circle nodes with 2D structure depictions. Finally, some common network property and analysis calculations are presented including the clustering coefficient, degree assortativity, and modularity. All code is provided in the form of Jupyter Notebooks and is available on GitHub with a permissive BSD-3 open-source license: https://github.com/vfscalfani/CSN_tutorial

Keywords: Chemical space network, Chemical similarity network, CSN, Molecular similarity, Maximum common substructure, RDKit, NetworkX

Graphical Abstract



*Correspondence: vfscalfani@ua.edu

University Libraries, Rodgers Library for Science and Engineering, The University of Alabama, Tuscaloosa, AL 35487, USA



© The Author(s) 2022. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>. The Creative Commons Public Domain Dedication waiver (<http://creativecommons.org/publicdomain/zero/1.0/>) applies to the data made available in this article, unless otherwise stated in a credit line to the data.

Introduction

Over the past decade, chemical space networks (CSNs) have been introduced as a way to visualize and interpret relationships within small molecule datasets. CSNs were designed as an alternative representation to coordinate-based visualizations using molecular descriptors. In a typical CSN, compounds are represented as nodes, and these nodes are connected with edges, where an edge is defined as some type of relationship between the compounds [1–3], including for example, 2D fingerprint-based Tanimoto similarity [4–7], substructure-based similarity [4, 5, 7, 8], or asymmetric Tversky similarity [9, 10].

In cases where the CSN edges represent a range of similarity values such as fingerprint-based Tanimoto similarity, the number of edges can be adjusted by incorporating a minimum threshold value; that is, only draw an edge if the selected relationship between the compounds are equal to or greater than the selected value. This is in contrast to, for example, in the case of matched molecular pair (MMP)-based CSNs, where the edges represent a binary relationship between the compounds (i.e., draw edge for MMP, no edge if not a MMP) [1, 2]. In CSNs, the molecular nodes can be visually represented as symbols such as circles or 2D chemical structure depictions, while edges are represented with lines. Additional layers of complexity can be added to the network visualization, for example, coloring the nodes by property value or changing the line style based on similarity value [1, 3]. Vogt et al. note that CSNs are generally considered most useful to represent compound datasets on the order of 10 s to 1000 s of compounds, and that compounds within the dataset should have some level of similarity or other relationship that can be used to form connections within a network. In addition to the useful visualizations that CSNs provide, one advantage of generating a CSN is that well-established network science algorithms and statistical calculations can be applied in the subsequent analysis [2].

There are a variety of software and techniques reported in the literature used to create CSN visualizations [3]. Specific to small molecule CSNs, reported methods include in-house Java programs and Java Universal network/graph framework [5–7, 11], in-house Python or Java code and Gephi software [4, 8–10], and Python NetworkX and/or Cytoscape [12, 13]. Cheminformatics toolkits reportedly used for computing molecular relationships for use in CSNs include, for example, in-house implementations [9], Molecular Operating Environment [11], OEChem toolkit [7, 8, 10, 11, 13], and CDK [12]. An example has been reported of using RDKit in a chemical network workflow. Dunn et al. used a combination of RDKit and NetworkX to generate Chemical Library

Networks, which are similar to CSNs, with the difference being that the nodes represent compound datasets, not individual compounds [14].

This article will demonstrate how to generate CSNs using RDKit [15], NetworkX [16], and Python. To our knowledge, no such tutorial exists in the published literature nor on the open web. In fact, we were not able to find any step by step code examples for how to generate CSNs regardless of the specific cheminformatics toolkit or other software used. We selected RDKit and NetworkX for this tutorial, in part because of our familiarity with these tools, but also because of the popularity of RDKit and Python in the cheminformatics community.

Overview and organization of this manuscript

Our specific use-case for this manuscript is to create CSNs using RDKit and NetworkX with a dataset collected from ChEMBL associated with the glucocorticoid receptor (adapted from a dataset described by Zhang et al. [7]). We note that the primary focus of this manuscript is to demonstrate a CSN workflow from pairwise calculations to visualization, and not focus on identifying or hypothesizing any specific scientific conclusions from the sample glucocorticoid receptor data. Within the workflow, there are a variety of specific steps necessary including, for example, data curation, computing pairwise relationships of compounds, compiling data into the network data structure, and then plotting the network. Outlined below are the workflow steps we carried out. These workflow steps from loading data to analyzing basic network properties should not be taken as “*the workflow*”, but rather one workflow example of how to approach creating CSNs with RDKit and NetworkX.

Importantly, our methods were heavily influenced and adapted from the published reports by Bajorath and coworkers [1, 2, 5, 7]. From their reported methods, we were able to infer the necessary computations and steps for creating CSNs and turn these steps into Python code. While we used RDKit and NetworkX in this report, we have attempted to present the steps in a way where the methods and code could be adapted with other programming workflows. In fact, our original test implementation of CSN visualizations used Wolfram Mathematica, which we then re-wrote in Python code with RDKit and NetworkX.

For presentation and length considerations within this manuscript, we have focused on discussing selected key parts of the code for CSN calculations and CSN visualizations. The complete code and additional code comments are available as Jupyter Notebooks at https://github.com/vfscalfani/CSN_tutorial. The Jupyter Notebooks contain the same order of steps used in the manuscript. We

recommend readers refer to both this manuscript and the complete code within the Jupyter Notebooks.

Code lines in the manuscript start with the iPython-like cell notation using the symbols, [«], denoting code input, and, [»], denoting code output. Integer values were not used as only selected pieces of the code are shown in this manuscript; that is, the code shown is not necessarily sequential. Lastly, in contrast to traditional scientific manuscripts with separate Methods and Discussion sections, we have combined the Methods with Discussion in an effort to create a readable and easy to follow tutorial style manuscript.

Step 0—hardware, operating system, and conda environment setup

Ubuntu 18.04.6 LTS was used on an Intel-based laptop (Core i9-9980HK CPU @ 2.40 GHz × 16) with 32 GB of RAM. For the development environment, a conda (v4.14.0) python (3.10.6) environment was created with the following recipe in a terminal:

```
$ conda create --name my-awesome-env
$ conda activate my-awesome-env
$ conda install -c conda-forge rdkit jupyterlab \
  numpy matplotlib pandas networkx
```

The specific versions installed were rdkit (2022.03.5), jupyterlab (3.4.7), numpy (1.23.3), matplotlib (3.5.3), pandas (1.4.4), and networkx (2.8.6).

Step 1—ChEMBL data collection

Data was collected by browsing the ChEMBL web interface during September 2022. The data collection procedure was adapted from one of the datasets used by Zhang et al., specifically the glucocorticoid receptor antagonist dataset [7]. To compile a similar dataset, we first searched for "glucocorticoid" in the ChEMBL web interface and then applied several selections and filters in the following order:

1. Selected Targets
 - Added Filters: Homo Sapiens
2. Selected ChEMBL2034 (Glucocorticoid receptor)
3. Selected Assays for Target ChEMBL2034
 - Added Filters: Homo Sapiens, Single Protein Format, Scientific Literature, 9-Direct single protein target assigned, D-Direct protein target assigned
4. Selected Browse Activities
 - Added Filters: Ki, molecular weight under 600, Target - ChEMBL2034
5. Exported and saved data as a CSV file, which contained 459 entries of compounds and associated ChEMBL data.

The exported CSV ChEMBL dataset is provided in the Additional file 1 and is redistributed according to the ChEMBL Data license with a Creative Commons Attribution-Share Alike 3.0 Unported License [17–19].

Step 2—load the data into a Python variable and curate/prepare compound data

We used the Pandas data analysis library [20, 21] to read the glucocorticoid receptor data into a DataFrame variable, and then created a DataFrame with only the data we would be using in the CSNs, including the ChEMBL IDs, SMILES, and Ki values. The Pandas DataFrame variable was selected as it has a number of builtin methods that make data transformations straightforward on delimiter separated data. We applied several data cleanup procedures and checks to the collected glucocorticoid receptor data. First, compounds with no associated Ki values were removed. This was followed by checking for presence of salts as disconnected SMILES by basic string matching a period symbol. As string matching can miss edge-cases, it is preferred to fully parse the SMILES with a cheminformatics toolkit and then perform pattern matching with chemical perception. For example, one potential edge-case with disconnections is if any of the provided SMILES use dot disconnect bonds with ring closures; that is, the presence of a dot disconnect bond does not actually guarantee that the compound has more than one component [22]. As a result, we validated that all compounds in the glucocorticoid dataset did indeed contain only one fragment using the RDKit GetMolFragments function, which can be used to identify the number of molecule fragments:

```
[«]: df = pd.read_csv('glucocorticoid_receptor_2034_2.csv', sep = ',')
[«]: df1 = df[['Molecule ChEMBL ID', 'Smiles', 'Standard Value']].copy()
[«]: df1.dropna(inplace=True)
[«]: df2 = df1[~df1['Smiles'].str.contains("\.")]
[«]: smis = df1['Smiles'].tolist()
    num_frags = []
    for smi in smis:
        mol = Chem.MolFromSmiles(smi)
        num_frags.append(len(Chem.GetMolFragments(mol)))
[«]: all(frag == 1 for frag in num_frags)
[»]: True
```

For datasets containing disconnected structures, there are several well-established procedures and tutorials in the cheminformatics literature for how to handle salts and other standardization protocols (vide infra). The next task was to merge duplicate compound data by averaging Ki values for identical compounds (equivalent ChEMBL ID and SMILES) with multiple Ki values [7]. The final filtered and merged list of the original ChEMBL provided SMILES, now totaling 404, were checked that they were unique from each other using Python sets. We also

double checked this by parsing the ChEMBL SMILES and validating that the RDKit canonicalized SMILES were unique from each other:

```
[*]: df3 = df2.groupby(['Molecule ChEMBL ID', 'Smiles']) \
      ['Standard Value'].mean().reset_index()
[*]: smis = df3['Smiles'].tolist()
[*]: rdkit_can_smiles = []
      for smi in smis:
          mol = Chem.MolFromSmiles(smi)
          rdkit_can_smiles.append(Chem.MolToSmiles(mol, canonical=True))

      set_rdkit_can_smiles = set(rdkit_can_smiles)
      len(set_rdkit_can_smiles) == len(rdkit_can_smiles)
[*]: True
```

We did not perform any specific chemical structure standardization such as the normalization of functional groups, as we relied on the data and small molecule standardization protocols implemented in ChEMBL, for the purposes of this tutorial [23]. For more advanced discussions regarding the importance and methods of data curation in cheminformatics and structure standardization procedures, particularly when compiling data from multiple sources, see the published articles by Fourches et al. [24, 25]. In addition, the open-source presentation materials from the RSC Open Science Workshop from Landrum on molecular standardization, contain several examples with Python and RDKit [26].

Step 3—Compile the network node data (compounds and associated pKi values)

For compiling the final node data, the Ki values in the pandas DataFrame were converted to pKi values ($pKi = -\log_{10}(Ki)$). This was followed by setting the DataFrame index column to the SMILES. In this case, setting the index column to SMILES was appropriate because we already validated that the SMILES strings were all unique from each other. The data was then saved as a dictionary variable, `node_data`, for all downstream calculations. In the `node_data` variable, SMILES are the unique dictionary keys, and the dictionary values are the associated ChEMBL ID and pKi data:

```
[*]: df4 = df3.set_index('Smiles')
[*]: node_data = df4.to_dict('index')
[*]: print(node_data)
[*]:
{'CC(=O)[C@H]1CC[C@H]2[C@@H]3CC(=O)CC[C@]4(C)[C@H]3CC[C@]12C':
 {'Molecule ChEMBL ID': 'CHEMBL103', 'pKi': 7.515700160653214},
 'COC1CC2C(C1)-C1CCC3C(C1)[C@H](C1CCCC1)O2)C(C)=CC(C)(C)N3':
 {'Molecule ChEMBL ID': 'CHEMBL1089811', 'pKi': 7.7447274948966935},
 '#COC1CCC2C(C1OC)-C1CCC3C(C1)[C@H](CC(C)O2)C(C)=CC(C)(C)N3':
 {'Molecule ChEMBL ID': 'CHEMBL1089812', 'pKi': 8.3767507096021},
 ...
 ...
```

Using dictionary variables for both the node data and edge data (Step 4) was a convenient data structure for working with NetworkX and molecular data, as

dictionaries are both straightforward to index and programmatically loop through.

Step 4—compile network edge data by computing pairwise similarity relationships of compounds

We created two different edge datasets for the CSNs, one based on the Tanimoto Similarity of two compounds using RDKit fingerprints, and another CSN based on a maximum common substructure (MCS)-based similarity calculation [7]. To compute these similarity values, we first created a list of all possible unique combinations of the compounds using the Python `itertools combinations` method with the SMILES (reminder: the 404 final SMILES were previously checked for uniqueness from each other). The number of pair combinations, equivalent to possible edges, for 404 unique nodes was 81,406:

```
[*]: smis = []
      for key,value in node_data.items():
          smis.append(key)

      from itertools import combinations
      smis_subsets = list(combinations(smis,2))
      len(smis_subsets)
[*]: 81406
```

The number of pair combinations can quickly become difficult to handle computationally. For example, with 1000 nodes, the unique pair combinations would jump to nearly 500,000. Increasing the number of nodes to 2000 would create about 2 million pair combinations, several orders of magnitude greater than the number of nodes. In addition, as Zhang et al. notes, the MCS-based similarity calculation is computationally intensive, and so it is likely only reasonable to work with around 1000 nodes on standard personal computers, at least for MCS-based calculations [7].

The SMILES subset pairs were converted to a dictionary variable, named `subsets`, and the RDKit mol objects were added to the dictionary such that each subset pair has its own dictionary integer key and the associated SMILES and mol objects as the values:

```
[*]: # add mol objects to our subsets dictionary
      for key,value in subsets.items():
          subsets[key].update({'mol1': Chem.MolFromSmiles(value['sm1'])})
          subsets[key].update({'mol2': Chem.MolFromSmiles(value['sm2'])})
      list(subsets.keys())[0]
[*]: 0
[*]: list(subsets.values())[0]
[*]: {'sm1': 'CC(=O)[C@H]1CC[C@H]2[C@@H]3CC(=O)CC[C@]4(C)[C@H]3CC[C@]12C',
      'sm2': 'COC1CC2C(C1)-C1CCC3C(C1)[C@H](C1CCCC1)O2)C(C)=CC(C)(C)N3',
      'mol1': <rdkit.Chem.rdchem.Mol at 0x7f6b96c0a6e0>,
      'mol2': <rdkit.Chem.rdchem.Mol at 0x7f6b96c0af80>}
```

At this point, we computed the pairwise similarity values for the edges. The Tanimoto Similarity values were

computed as follows using the RDKit fingerprints with default parameters:

```
[*]: for key,value in subsets.items():
    fp1 = Chem.RDKitFingerprint(value['mol1'])
    fp2 = Chem.RDKitFingerprint(value['mol2'])
    tan_sim = round(DataStructs.TanimotoSimilarity(fp1,fp2), 3)
    subsets[key].update({"tan_similarity": tan_sim})

[*]: list(subsets.values())[0]
[*]: {'sm1': 'CC(=O)[C@H]1CC[C@H]2[C@@H]3CC4=CC(=O)CC[C@]4(C)[C@H]3CC[C@]12C',
      'sm2': 'COc1ccc2c(c1)-c1ccc3c(c1)[C@H](c1ccc1)O2)C(C)=CC(C)(C)N3',
      'mol1': <rdkit.Chem.rdchem.Mol at 0x7f6b96c0a6e0>,
      'mol2': <rdkit.Chem.rdchem.Mol at 0x7f6b96c0af80>,
      'tan_similarity': 0.155}
```

The RDKit fingerprint calculations for the ~81,000 pairs of compounds was fast, completing in about 3 min on the Core i9-based laptop used. Next, we wrote a Python RDKit function to compute the MCS-based similarity value using the equation reported by Zhang [7]. For the purposes of this tutorial, we added a 10 s timeout option to the RDKit FindMCS function, which serves to return the best MCS match found within 10 s, and speed up the calculations if certain comparisons are taking too long. This seemed like a reasonable parameter to add as well, as the RDKit “Getting Started” documentation suggests that most of the time, the FindMCS comparison takes less than a second [15].

```
[*]: def tc_mcs(mol1,mol2,key):
    # get maximum common substructure instance
    mcs = rdFMCS.FindMCS([mol1,mol2],timeout=10)

    # get number of common bonds
    mcs_bonds = mcs.numBonds

    # get number of bonds for each
    # default is only heavy atom bonds
    mol1_bonds = mol1.GetNumBonds()
    mol2_bonds = mol2.GetNumBonds()

    # compute MCS-based Tanimoto
    tan_mcs = mcs_bonds / (mol1_bonds + mol2_bonds - mcs_bonds)
    return key, tan_mcs
```

In addition to the FindMCS function being computationally demanding, Python defaults to single core processing. We found that computing 80,000 `tc_mcs` values was incredibly slow with single core processing. As a result, we used the Python multiprocessing module, which allowed us to loop through the 80,000 pairs of mol objects and compute the `tc_mcs` function across multiple processors (Jupyter Notebooks). With this method and using 14 of the 16 cores of an Intel Core i9-9980HK (2.4 GHz × 16) CPU, the computation took about one hour. The final subsets data structure was as follows:

```
[*]: list(subsets.values())[0]
[*]: [{'sm1': 'CC(=O)[C@H]1CC[C@H]2[C@@H]3CC4=CC(=O)CC[C@]4(C)[C@H]3CC[C@]12C',
      'sm2': 'COc1ccc2c(c1)-c1ccc3c(c1)[C@H](c1ccc1)O2)C(C)=CC(C)(C)N3',
      'mol1': <rdkit.Chem.rdchem.Mol at 0x7f6b96c0a6e0>,
      'mol2': <rdkit.Chem.rdchem.Mol at 0x7f6b96c0af80>,
      'tan_similarity': 0.155,
      'tan_mcs': 0.475}]
```

We recognize that not all readers will have access to a core i9 (or faster) computer workstation. As such, we ran

the same computation on a 10 year old laptop with an Intel Core i5-2520 M (2.5 GHz × 4) CPU with 8 GB RAM, using 3 of the 4 cores and all ~80,000 MCS-based similarity values were computed in about four hours. Since the edge data computations can take a while to compute, these were saved as a Python pickle file, eliminating the need to re-compute the edge data for the downstream analysis and CSN visualizations (Additional file 1).

Step 5—select a threshold value for edges to include in the network

The subsets dictionary variable, containing the edge data, was filtered based on the values of the Tanimoto similarity or the MCS-based similarity value; that is, not all possible edges are included in the CSNs. There are two approaches to filtering the edge data, the first would be to select a threshold value of similarity (e.g., >0.5), and the second is to target a specific final edge density (edges / total edges) [7], then calculate the value of similarity needed to match the target edge density. We selected a filtering value of >=0.68 for the RDKit fingerprint Tanimoto-based similarity CSNs, which created an edge density of 0.095 (~10%):

```
[*]: subsets_filtered_ts = {key: value for key, value in subsets.items() if \
    value['tan_similarity'] >= 0.68}
[*]: edge_density_ts = round((len(subsets_filtered_ts) / len(subsets)),3)
```

For the MCS-based similarity CSNs, the edge density of 0.095 was targeted to find the filtering value of 0.635 for the MCS-based similarity value (Jupyter Notebooks).

The selected edge density of ~10% is higher than values used in the literature for CSNs, which are typically 2–5% [2, 5, 6]. For the dataset we used, achieving an edge density of ~3% would require a threshold value of ~0.9 for the Tanimoto and MCS-based similarity CSNs. We selected a higher edge density for the purpose of this tutorial for two main reasons: (1) we wanted to incorporate a range of similarity values into the network visualizations in order to demonstrate creating different edge line styles based on these values, and (2) obtaining a lower edge density created a high number of small individual connected components in the graph, which we wanted to avoid to simplify the visualizations. Tuning and selecting an appropriate edge density is an important consideration when studying CSNs as many network properties are affected by edge density [2, 5–7]. Using the methods and code described in this article, it is straightforward to experiment with different edge densities by simply adjusting the threshold filtering value and executing the code for the downstream visualizations and calculations. Lastly, we removed any nodes (compounds) from the `node_data` dictionary variable that do not appear in

the filtered edge data. This step removes the occurrence of nodes without any edges. For the Tanimoto-based similarity CSNs the final number of nodes was 393 and 7739 edges, and for the MCS-based CSNs, the final number of nodes was 398 and number of edges was 7709.

Step 6—add node and edge data to the network graph and plot

After initializing an empty graph variable, the edge data including the compound SMILES (`smi1`, `smi2`) and their associated similarity value (Tanimoto or MCS-based similarity) were added to the graph by looping through the filtered subset edge data created in Step 5:

```
[«]: G1 = nx.Graph()
[«]: for key, value in subsets_filtered_ts.items():
      G1.add_edge(value['smi1'], value['smi2'], weight=value['tan_similarity'])
```

The NetworkX `Graph.add_edge` method automatically creates the nodes from the edges (nodes are the SMILES strings). However, we also added the node attributes using the `Graph.add_node` method in order to include the associated ChEMBL ID and pKi values:

```
[«]: for key,value in node_data_filtered_ts.items():
      G1.add_node(key, ID=value['Molecule ChEMBL ID'], pKi=value['pKi'])
```

With the edge and node data added to the network graph, we now have all of the pieces in place to start creating visualizations. NetworkX provides graph drawing functionality using the Matplotlib library [27]. To create a basic plot, first a layout algorithm and parameters are selected, followed by the NetworkX draw function:

```
[«]: # added a seed for layout reproducibility
      # k is space between nodes
      pos = nx.spring_layout(G1, k=0.3,seed=40)
      nx.draw(G1,pos)
      plt.axis("off")
      plt.show()
```

The spring 2D layout was selected because of its successful use in reported CSNs [5, 7, 9]. The NetworkX spring layout uses the Fruchterman-Reingold force-directed algorithm, which ultimately creates separate clusters/communities of connected nodes [7, 28]. By default, the NetworkX spring layout uses the weights of the edges (e.g., similarity values) within the force-directed calculations, where a larger weight equates to a stronger attractive force between the nodes in the algorithm. We found that the similarity weight values had minimal visual impact on the overall layout of the CSNs (perhaps due to their similar magnitude). As a result, it is likely that the edge distance between connected nodes

and space between individual clusters of nodes do not scale with similarity relationship value, which is consistent with other applications of the Fruchterman-Reingold algorithm for CSNs without weighted edge values [2, 7].

Step 7—adjust plot settings such as node color based on pKi value, node size, and add legends

Figure 1 is an acceptable basic CSN plot, however, by adjusting a few network drawing plotting parameters, it is possible to improve the amount of information that the CSN visualization conveys. The first adjustment to the CSN network drawing was to incorporate the pKi values as node colors. This was accomplished by looping through the graph's node pKi attributes and creating a list of colors based on a conditional range of values from 4 to 11, with an increment of 1:

```
[«]: color_map = []
      for node in G1.nodes(data=True):
          if 4<= node[1]['pKi'] <= 5:
              color_map.append("darkred")
          elif 5 < node[1]['pKi'] <= 6:
              color_map.append("red")
          ...
          ...
          ...
          elif 10 < node[1]['pKi'] <= 11:
              color_map.append("blue")
          else:
              color_map.append("white")
```

The `color_map` list can then be added into the network drawing with the `node_color` drawing option. Additional adjustments were also added in including a smaller node size, node edge border, and a grey color for the network edges:

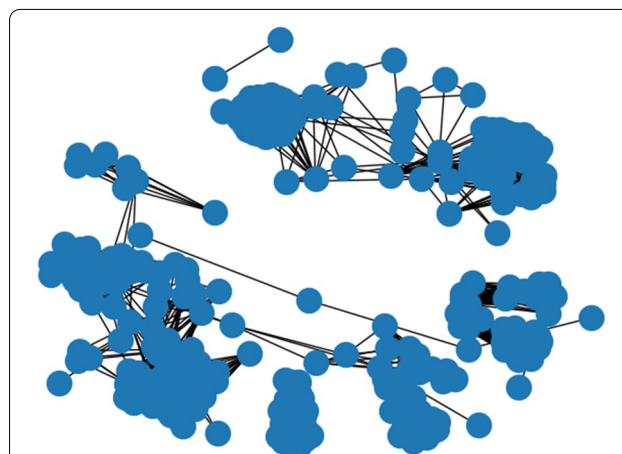


Fig. 1 A basic spring layout CSN (Tanimoto Similarity variant) with the glucocorticoid dataset compounds. The Tanimoto Similarity threshold was set to ≥ 0.68

```
[«]: nx.draw_networkx_nodes(G1, pos, node_size=75, \
    node_color=color_map, edgecolors="black")
    nx.draw_networkx_edges(G1, pos, edge_color="grey")
```

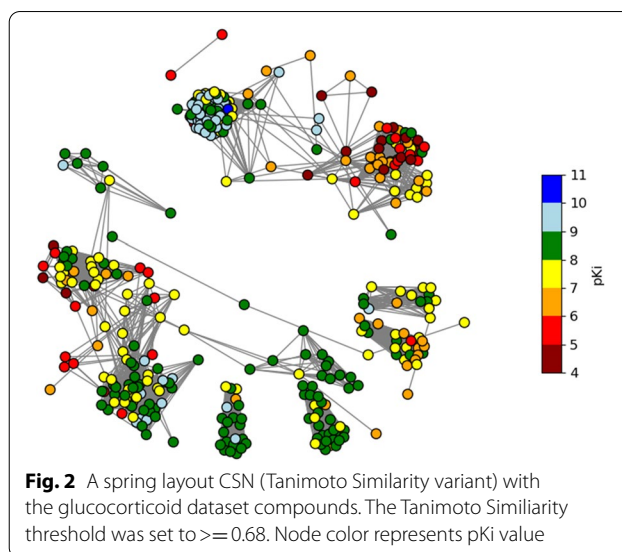
Since NetworkX uses Matplotlib for the drawing functionality, it is also possible to annotate the plot with a color bar legend by adding a second axis and defining the color bar bounds and attributes (Jupyter Notebooks).

Step 8—Identify and plot specific components of the network

Figures 1 and 2 depict the overall CSN visualization, however, it is useful to also identify and explore sections, or individual connected components of the graph. With NetworkX, the individual connected graphs can be enumerated and saved as a list:

```
[*]: connected_graphs1 = [G1.subgraph(c).copy() for c in nx.connected_components(G1)]
```

The list of individual connected graphs can then be plotted similarly to the entire CSN network. However, since the connected graphs contain less nodes than the entire CSN network and offer a “zoomed in” view, it is possible to incorporate features in addition to node color such as node labels and different line styles based on similarity value [3, 29]. Node labels can be added by looping through the nodes and creating, for example an integer label for each node, while the line styles can be adjusted

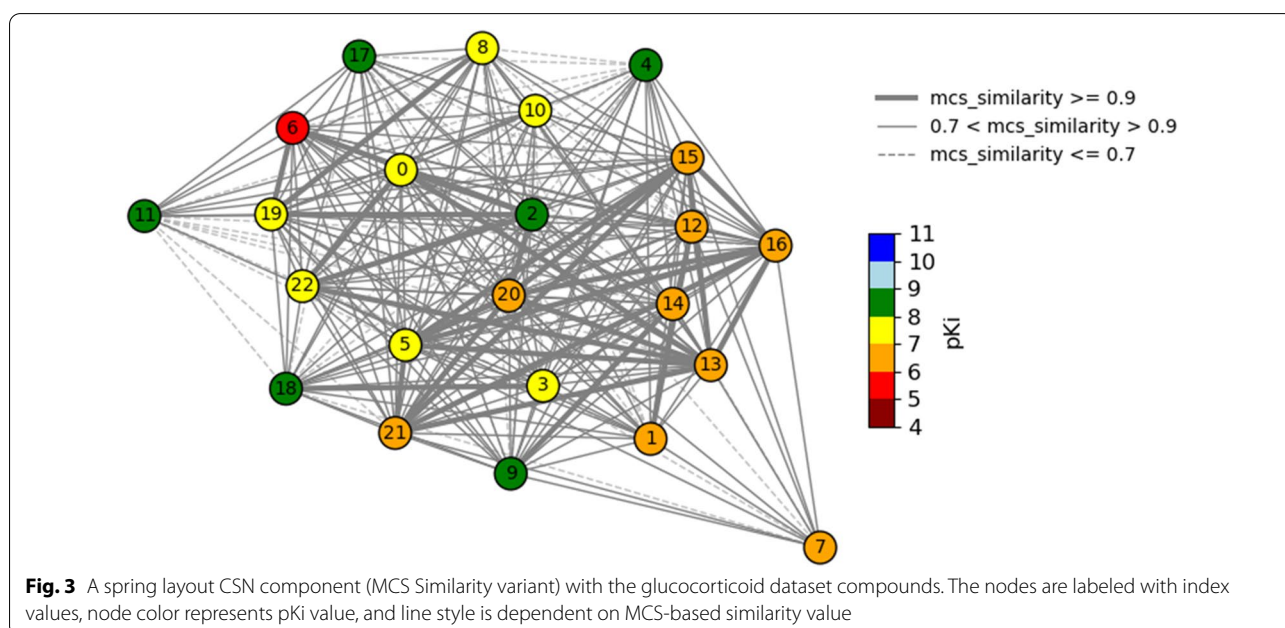


by filtering the edges into individual edge lists based on attribute value (Fig. 3):

```
[*]: thick = [(u, v) for (u, v, d) in H1.edges(data=True) if d["weight"] >= 0.9]
    medium = [(u, v) for (u, v, d) in H1.edges(data=True) if 0.7 < d["weight"] < 0.9]
    thin = [(u, v) for (u, v, d) in H1.edges(data=True) if d["weight"] <= 0.7]
```

The custom node labels and edge lists can then be plotted with the NetWorkX drawing functionality using the `draw_networkx_labels`, `draw_networkx_edges` functions (Jupyter Notebooks).

Finally, the labeled nodes can be identified and further analyzed by looping over the graph nodes and



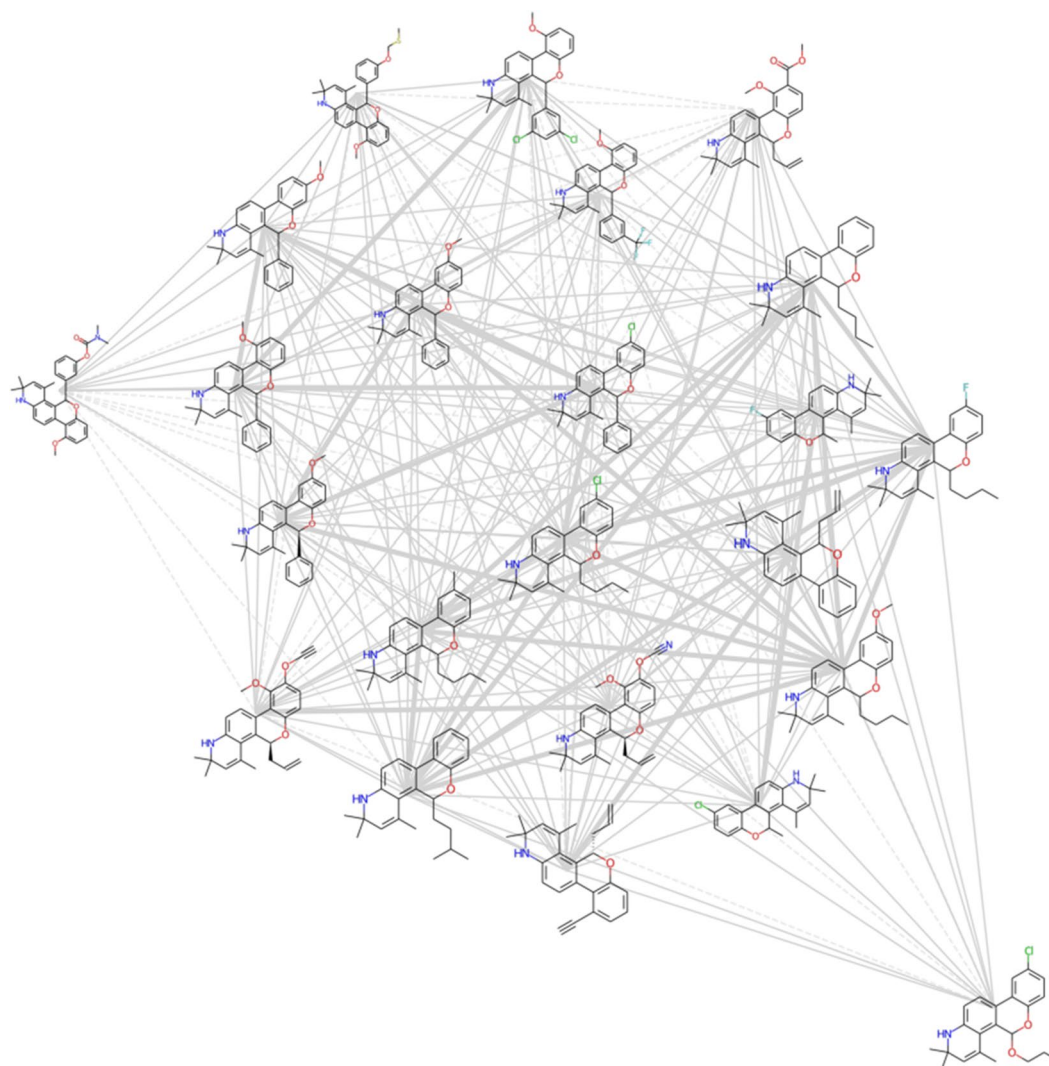


Fig. 4 A spring layout CSN component (MCS Similarity variant) with the glucocorticoid dataset compounds. The nodes are plotted as 2D compound images and line style is dependent on MCS-based similarity value

indexing out the SMILES, ID values, and pKi values, into, for example, a DataFrame variable:

```
[*]: columns = ("Smiles", "ID", "pKi")
index = tuple(custom_labels.values())
table_data = []
for node in H1.nodes(data=True):
    smiles = node[0]
    id_value = node[1]["ID"]
    ki = node[1]["pKi"]
    table_data.append([smiles, id_value, ki])
df_H1 = pd.DataFrame(table_data, columns = columns, index=index)
[*]: print(df_H1)
```

	Smiles	ID	pKi
0	C0c1cccc2c1-c1ccc3c(c1C(c1cccc(OC(=O)N(C)C)c1)...	CHEMBL317578	8.619789
1	C0c1cccc2c1-c1ccc3c(c1C(c1cccc(OC5C)c1)O2)C(C)...	CHEMBL97206	8.397940
2	C0c1cccc2(c1)OC(c1cccc1)c1c-2ccc2c1C(C)=CC(C)...	CHEMBL99128	5.744727
...			

Step 9—replace circle nodes with 2D chemical structures

As images can be plotted on Matplotlib axes and incorporated into network visualizations in NetworkX [30], it is possible to replace the representation of compounds as nodes with 2D structure depictions (Fig. 4). The workflow for plotting the nodes as 2D structures in a NetworkX graph included: (A) obtaining the positions to plot the structure images; (B) looping through the nodes (SMILES), and then; (C) using the RDKit rdMolDraw2D module to create 2D structure drawings as PNG images

with transparent backgrounds. Step C of the workflow is shown below:

```
[«]: for smi in H1.nodes:
    # draw molecule
    drawer = rdMolDraw2D.MolDraw2DCairo(300,300)
    drawer.drawOptions().clearBackground = False
    drawer.drawOptions().addStereoAnnotation = False
    drawer.DrawMolecule(Chem.MolFromSmiles(smi))
    drawer.FinishDrawing()
    mol = drawer.GetDrawingText()

    xf, yf = tr_figure(pos[smi])
    xa, ya = tr_axes((xf, yf))
    # get overlapped axes and plot structure
    a = plt.axes([xa - struct_center, ya - struct_center, \
    struct_size, struct_size])
    a.imshow(show_png(mol))
    a.axis("off")
plt.show()
```

There are a couple limitations to be aware of when plotting nodes as 2D structures using this method. The first limitation is that the scaling size of the 2D structure images may need to be manually adjusted depending on the number of structures in the network (Jupyter Notebooks). This limitation is easily overcome, however, since it only requires adjusting one value manually until an appropriate size is found. We have found scaling values in the range of 0.04 to 0.1 to be appropriate for plotting 5–40 structure images.

One advantage of the Matplotlib/NetworkX CSN plotting method with 2D structure images is that the code to produce the network visualization is relatively straightforward and can be accomplished with about 15 lines of code. The produced static network images work well for creating high quality traditional scientific visualizations of networks on the order of 10 s of structures, however they are not well suited for attempting to navigate through an entire CSN or large component thereof (e.g., 100 s of nodes). For visualizing larger networks with 2D structure images, it is likely that some type of interactive network visualization would be needed. One method for creating interactive CSNs with 2D structures, which would incorporate well into the Python/NetWorkX/RDKit workflow described in this tutorial, would be to use the ipycytoscape Jupyter Notebook widget [31]. The ipycytoscape widget was recently featured in an RDKit User Group Meeting presentation for creating interactive scaffold networks (a type of fragmentation chemical network) [32, 33]. While creating interactive CSNs is beyond the scope of this tutorial, we plan to experiment with the ipycytoscape widget in the near future to evaluate its potential for creating CSNs with RDKit and NetworkX.

Step 10—adjust 2D structure plot settings such as color based on pKi value and add legends

The basic 2D structure network depiction created in Step 9 (Fig. 4) can be enhanced by incorporating in

color based on pKi value and adding in the ChEMBL IDs as labels. To accomplish this, we wrote an RDKit Python function, `highlight_mol`, that accepts SMILES, a string label, and a color name as input. The `highlight_mol` function then returns the depicted 2D drawing PNG image with all atoms and bonds highlighted:

```
[«]: def highlight_mol(smi, label, color):
    mol = Chem.MolFromSmiles(smi)

    if color == 'darkred':
        rgba = (0.55, 0.0, 0.0, 0.3)
    elif color == 'red':
        rgba = (1.0, 0.0, 0.0, 0.2)
    elif color == 'orange':
        rgba = (1.0, 0.65, 0.0, 0.2)
    ...
    else: # no color
        rgba = (1,1,1,1)

    atoms = []
    for a in mol.GetAtoms():
        atoms.append(a.GetIdx())

    bonds = []
    for bond in mol.GetBonds():
        aid1 = atoms[bond.GetBeginAtomIdx()]
        aid2 = atoms[bond.GetEndAtomIdx()]
        bonds.append(mol.GetBondBetweenAtoms(aid1, aid2).GetIdx())

    drawer = rdMolDraw2D.MolDraw2DCairo(300,300)
    drawer.drawOptions().fillHighlights=True
    drawer.drawOptions().setHighlightColour((rgba))
    drawer.drawOptions().highlightBondWidthMultiplier=15
    drawer.drawOptions().legendFontSize=60
    drawer.drawOptions().clearBackground = False
    rdMolDraw2D.PrepareAndDrawMolecule(drawer, mol, \
    legend=label, highlightAtoms=atoms, highlightBonds=bonds)

    mol_png = drawer.GetDrawingText()
    return mol_png
```

The `highlight_mol` function was then incorporated into the workflow for plotting the nodes as 2D structures as described in step 9. The input data for the `highlight_mol` function was obtained by looping through each node in the graph and indexing out the respective pKi and ChEMBL ID value attributes (Jupyter Notebooks). To complete the visualization, a color bar and legend were added (Fig. 5).

Step 11—compute network properties—clustering coefficient, degree assortativity, and modularity

NetWorkX includes many builtin functions for calculating network properties and applying network science algorithms into CSN analysis workflows. In this tutorial, we focused on demonstrating calculations for three common network properties including the clustering coefficient, degree assortativity, and modularity.

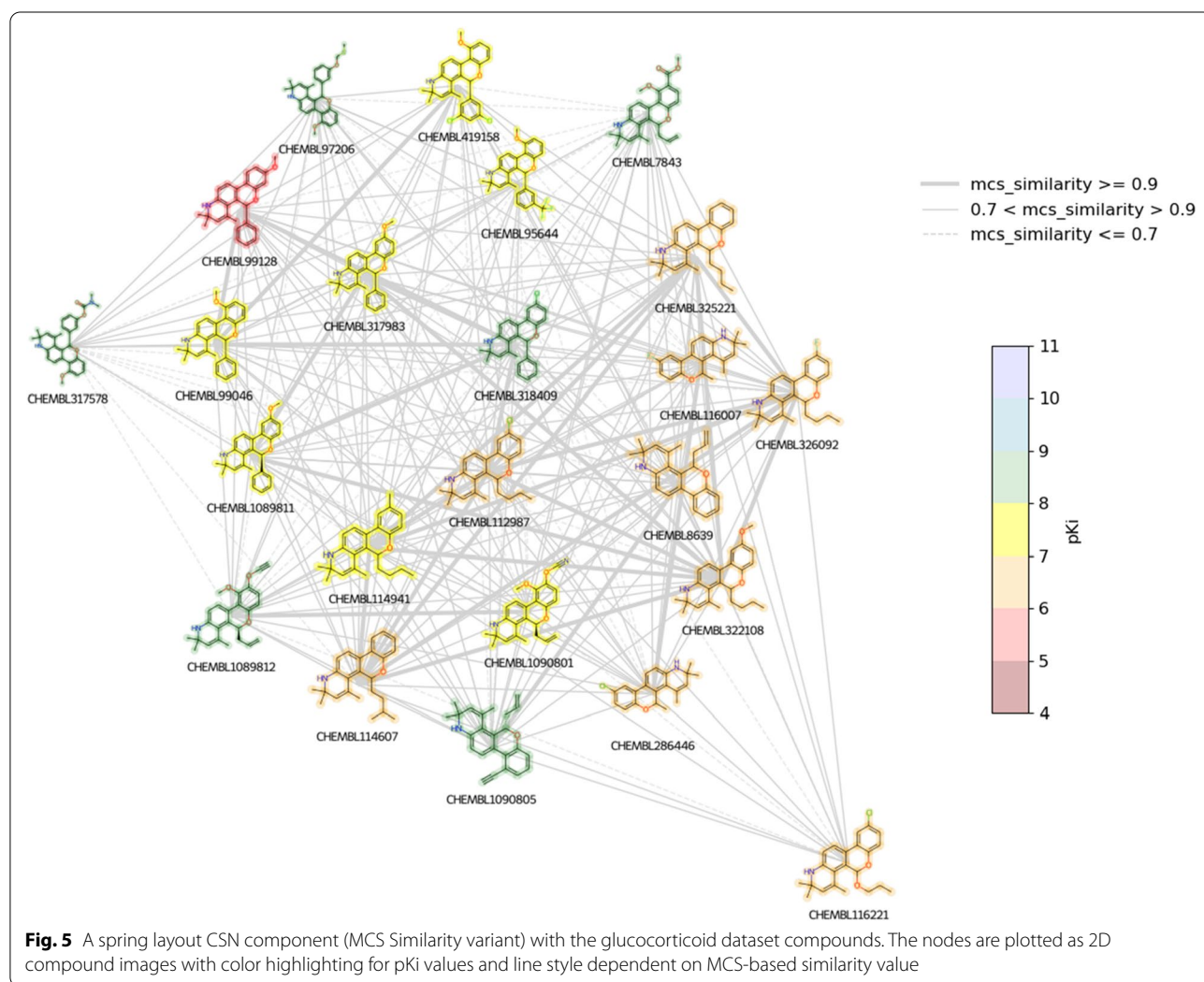
The clustering coefficient is a value between 0 and 1 that quantifies the ratio of the number of closed loops of three nodes that are in the network [34]. For example, in a CSN, for each pair of two compounds connected with an edge, the average clustering coefficient would then quantify the number of occurrences where the two compounds are both connected (similar) to a third neighboring compound [5, 7]. The clustering coefficient in NetworkX can be computed with the builtin clustering

function using the CSN graph as input and optionally the weights of the edges. If the weights of the edges are used (e.g., similarity values), these are incorporated into the clustering coefficient calculation with a weight intensity factor [35, 36]. The average clustering coefficient for the weighted glucocorticoid CSN was 0.76 and 0.71 for the Tanimoto similarity and MCS-based similarity, respectively (Jupyter Notebooks).

The degree assortativity is a measure of correlation between the degrees of nodes (number of edges connected to node). The value ranges from -1 to 1. The larger the value, the more assortative the network, which, in short, indicates that similar node degrees are neighbors (e.g., high degree nodes connected to high degree nodes) [2, 5, 34]. CSNs reported in the literature typically have positive degree assortativity, which is a result of forming individual connected components or clusters of similar compounds in the network [5, 6]. In NetworkX, there is

a builtin function, `degree_assortativity_coefficient`, that can be used with the CSN graph as input and optionally the similarity weights as input. The degree assortativity for the weighted glucocorticoid CSN was 0.96 and 0.84 for the Tanimoto similarity and MCS-based similarity, respectively (Jupyter Notebooks).

Modularity is a value that quantifies the separation of similar nodes (e.g. nodes connected with many edges) into distinct clusters or communities [5, 34]. To compute the modularity of the CSNs in NetworkX, first, communities need to be detected using an optimization function [6], followed by then inputting the detected community component graphs into the NetworkX modularity function. Using the NetworkX `greedy_modularity_communities` function for community detection of the weighted glucocorticoid CSNs, the Tanimoto similarity network had 18 communities with a modularity value of 0.62, and



the MCS-based CSN had 13 communities with a modularity of 0.71 (Jupyter Notebooks).

Since the NetworkX community detection functions allow saving a list of the individual communities as nodes, it is possible to then loop through the list of nodes and add distinct colors to each node depending on the community for the CSN visualization (Fig. 6):

```
[*]: from networkx.algorithms.community import greedy_modularity_communities
communities = greedy_modularity_communities(G1, weight="weight")
[*]: # create a list of colors from a matplotlib colormap
cmap = matplotlib.cm.get_cmap('tab20')
color_intervals = np.linspace(0,1,13) # for 13 colors
colors_tab = []
for value in color_intervals:
    colors_tab.append(cmap(value))
# add color to nodes by community group
community_color_map = {}
for node in G1.nodes:
    for idx,community in enumerate(communities):
        if node in community:
            community_color_map.append(colors_tab[idx])
[*]: # plot nodes with color as community
# added a seed for layout reproducibility
pos = nx.spring_layout(G1, k=0.3, seed=40)
fig, ax = plt.subplots(figsize=(8, 8))
nx.draw(G1, pos, node_color=community_color_map, \
        edgecolor="black", node_size=75)
plt.axis("off")
plt.show()
```

Interestingly, while we included the similarity edge values as an optional weight parameter in the spring layout, clustering coefficient, degree assortativity, and community detection network calculations, CSNs in the literature do not generally include the similarity weights into the network. Studying weighted CSNs is an area of interest and opportunity as pointed out in a 2016 review article by Vogt et. al [2]. However, to our knowledge there has only been a single report studying weighted CSNs. Ito and Ohnishi created weighted CSNs based on Tanimoto similarity relationships without incorporating any minimum threshold to include edges into the network; that is, all edges are included in the network, and the topology of the network is not influenced by a minimum similarity value [37]. As such, there is certainly an opportunity for the cheminformatics community to further evaluate the suitability of weighted CSNs and if a combination of a threshold parameter and weighted edges (as in this tutorial) would afford new information in the analysis of CSNs.

Conclusions

This tutorial demonstrated how to use RDKit and NetworkX to create CSNs based on RDKit fingerprint Tanimoto similarity and a maximum common substructure-based similarity. A variety of CSN visualizations of varying complexity were created including CSNs with circle node representations, node color based on bioactivity attribute, edge thickness dependent on similarity value, and 2D structure depictions as nodes. With this manuscript and the included Jupyter Notebooks (Additional file 1), readers interested in generating CSNs

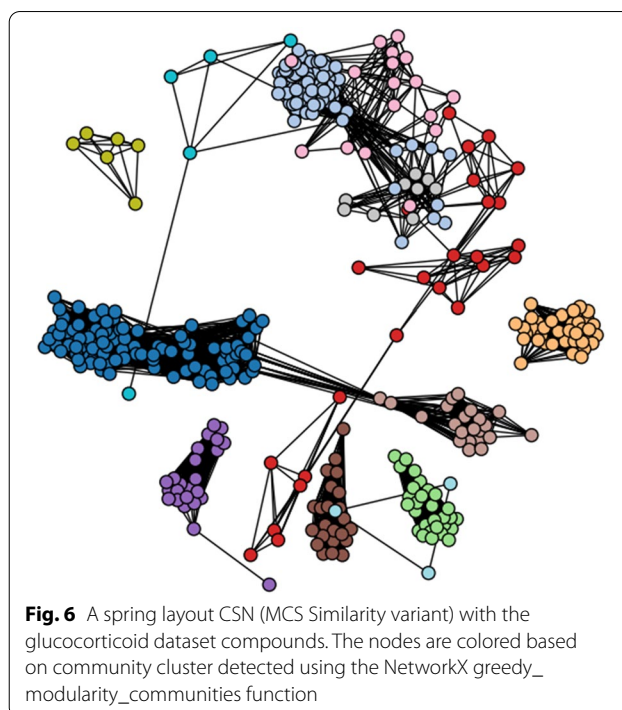


Fig. 6 A spring layout CSN (MCS Similarity variant) with the glucocorticoid dataset compounds. The nodes are colored based on community cluster detected using the NetworkX greedy_modularity_communities function

should be able to adapt the code for their own use in research and teaching. We hope that code in this tutorial will provide a starting point for more CSN related cheminformatics research, including further comparison of CSNs to weighted CSNs (incorporating edge values into network calculations), which, to our knowledge has not yet been extensively studied in cheminformatics [2]. In addition, we suspect that selected parts of the tutorial workflow described in this report could be adapted for generating chemical library networks (where nodes represent collections of compounds) [14]. Finally, we plan to continue our work on CSN tutorials by creating interactive CSNs in the near future.

Abbreviations

CSN: Chemical space network; MMP: Matched molecular pair; MCS: Maximum common substructure.

Supplementary Information

The online version contains supplementary material available at <https://doi.org/10.1186/s13321-022-00664-x>.

Additional file 1. ChEMBL glucocorticoid dataset (CSV file).

Acknowledgements

The work presented in this manuscript was produced with open source software including the Linux Ubuntu operating system, RDKit, NetworkX, Pandas, Matplotlib and other supporting software packages. Thank you to all of the contributors; this work would not have been possible without the excellent software and documentation provided by these open source projects.

Author contributions

VFS directed the project, developed code, and wrote the manuscript. VDP contributed significantly to the workflow and prototyping code. AMF contributed substantial improvements and additions to the code. All authors read and approved the manuscript.

Funding

Not applicable.

Availability of data and materials

All code (as Jupyter Notebooks) needed to reproduce the results in this article are available on GitHub with a BSD-3 License: https://github.com/vfscalfani/CSN_tutorial

Declarations**Ethics approval and consent to participate**

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare they have no competing interests.

Received: 21 October 2022 Accepted: 27 November 2022

Published online: 28 December 2022

References

- Maggiara GM, Bajorath J (2014) Chemical space networks: a powerful new paradigm for the description of chemical space. *J Comput Aided Mol Des* 28:795–802. <https://doi.org/10.1007/s10822-014-9760-0>
- Vogt M, Stumpfe D, Maggiara GM, Bajorath J (2016) Lessons learned from the design of chemical space networks and opportunities for new applications. *J Comput Aided Mol Des* 30:191–208. <https://doi.org/10.1007/s10822-016-9906-3>
- Recanatini M, Cabrelle C (2020) Drug research meets network science: where are we? *J Med Chem* 63:8653–8666. <https://doi.org/10.1021/acs.jmedchem.9b01989>
- Kunimoto R, Bajorath J (2018) Combining similarity searching and network analysis for the identification of active compounds. *ACS Omega* 3:3768–3777. <https://doi.org/10.1021/acsomega.8b00344>
- Zhang B, Vogt M, Maggiara GM, Bajorath J (2015) Comparison of bioactive chemical space networks generated using substructure- and fingerprint-based measures of molecular similarity. *J Comput Aided Mol Des* 29:595–608. <https://doi.org/10.1007/s10822-015-9852-5>
- Zwierzyzna M, Vogt M, Maggiara GM, Bajorath J (2015) Design and characterization of chemical space networks for different compound data sets. *J Comput Aided Mol Des* 29:113–125. <https://doi.org/10.1007/s10822-014-9821-4>
- Zhang B, Vogt M, Maggiara GM, Bajorath J (2015) Design of chemical space networks using a Tanimoto similarity variant based upon maximum common substructures. *J Comput Aided Mol Des* 29:937–950. <https://doi.org/10.1007/s10822-015-9872-1>
- Kunimoto R, Bajorath J (2017) Exploring sets of molecules from patents and relationships to other active compounds in chemical space networks. *J Comput Aided Mol Des* 31:779–788. <https://doi.org/10.1007/s10822-017-0061-2>
- Wu M, Vogt M, Maggiara GM, Bajorath J (2016) Design of chemical space networks on the basis of Tversky similarity. *J Comput Aided Mol Des* 30:1–12. <https://doi.org/10.1007/s10822-015-9891-y>
- Kunimoto R, Vogt M, Bajorath J (2017) Tracing compound pathways using chemical space networks. *Med Chem Commun* 8:376–384. <https://doi.org/10.1039/C6MD00628K>
- de la VegadeLeón A, Bajorath J (2016) Chemical space visualization: transforming multidimensional chemical spaces into similarity-based molecular networks. *Future Med Chem* 8:1769–1778. <https://doi.org/10.4155/fmc-2016-0023>
- Lepp Z, Huang C, Okada T (2009) Finding key members in compound libraries by analyzing networks of molecules assembled by structural similarity. *J Chem Inf Model* 49:2429–2443. <https://doi.org/10.1021/ci9001102>
- Wollenhaupt S, Baumann K (2014) inSARa: intuitive and interactive SAR interpretation by reduced graphs and hierarchical MCS-based network navigation. *J Chem Inf Model* 54:1578–1595. <https://doi.org/10.1021/ci4007547>
- Dunn TB, Seabra GM, Kim TD et al (2022) Diversity and chemical library networks of large data sets. *J Chem Inf Model* 62:2186–2201. <https://doi.org/10.1021/acs.jcim.1c01013>
- Landrum G, contributors Rdk (2022) rdkit/rdkit: 2022_03_5 (Q1 2022) Release
- Hagberg AA, Schult DA, Swart PJ (2008) Exploring network structure, dynamics, and function using NetworkX. In: Varoquaux G, Vaught T, Millman J (eds) Proceedings of the 7th Python in science conference. Pasadena, CA USA, pp 11–15
- ChEMBL Data Licensing. <https://chembl.gitbook.io/chembl-interface-documentation/about#data-licensing>
- Creative Commons—CC BY-SA 3.0 License. <https://creativecommons.org/licenses/by-sa/3.0/>
- Gaulton A, Hersey A, Nowotka M et al (2017) The ChEMBL database in 2017. *Nucleic Acids Res* 45:D945–D954. <https://doi.org/10.1093/nar/gkw1074>
- McKinney W (2010) Data structures for statistical computing in Python. In: Walt S van der, Millman J (eds) Proceedings of the 9th Python in science conference, pp 56–61
- The Pandas development team (2022) pandas-dev/pandas: Pandas 1.4.4. <https://doi.org/10.5281/zenodo.3509134>
- Dalke, Andrew Combinatorial Library Generation with SMILES. In: Combinatorial Library Generation with SMILES. http://www.dalkescientific.com/writings/diary/archive/2004/12/12/library_generation_with_smiles.html. Accessed 20 Oct 2022
- Bento AP, Hersey A, Félix E et al (2020) An open source chemical structure curation pipeline using RDKit. *J Cheminform* 12:51. <https://doi.org/10.1186/s13321-020-00456-1>
- Fourches D, Muratov E, Tropsha A (2010) Trust, but verify: on the importance of chemical structure curation in cheminformatics and QSAR modeling research. *J Chem Inf Model* 50:1189–1204. <https://doi.org/10.1021/ci100176x>
- Fourches D, Muratov E, Tropsha A (2016) Trust, but verify II: a practical guide to chemogenomics data curation. *J Chem Inf Model* 56:1243–1252. <https://doi.org/10.1021/acs.jcim.6b00129>
- Landrum, Greg RSC_OpenScience_Standardization. In: RSC_OpenScience_Standardization. https://github.com/greglandrum/RSC_OpenScience_Standardization_202104. Accessed 20 Oct 2022
- Hunter JD (2007) Matplotlib: a 2D graphics environment. *Comput Sci Eng* 9:90–95. <https://doi.org/10.1109/MCSE.2007.55>
- Noack A (2009) Modularity clustering is force-directed layout. *Phys Rev E* 79:026102. <https://doi.org/10.1103/PhysRevE.79.026102>
- NetworkX Weighted Graph Drawing. In: NetworkX weighted graph drawing. https://networkx.org/documentation/stable/auto_examples/drawing/plot_weighted_graph.html. Accessed 20 Oct 2022
- NetworkX Custom Node Icons Drawing. In: NetworkX custom node icons drawing. https://networkx.org/documentation/stable/auto_examples/drawing/plot_custom_node_icons.html. Accessed 20 Oct 2022
- ipycytoscape. In: ipycytoscape. <https://github.com/cytoscape/ipycytoscape>. Accessed 20 Oct 2022
- Landrum G (2020) RDKit UGM what's new. https://github.com/rdkit/UGM_2020/blob/master/Notebooks/Landrum_WhatsNew.ipynb. Accessed 20 Oct 2022
- Kruger F, Stiefl N, Landrum GA (2020) rdScaffoldNetwork: the scaffold network implementation in RDKit. *J Chem Inf Model* 60:3331–3335. <https://doi.org/10.1021/acs.jcim.0c00296>
- Newman M (2018) Networks. Oxford University Press

35. Onnela J-P, Saramäki J, Kertész J, Kaski K (2005) Intensity and coherence of motifs in weighted complex networks. *Phys Rev E* 71:065103. <https://doi.org/10.1103/PhysRevE.71.065103>
36. Saramäki J, Kivela M, Onnela J-P et al (2007) Generalizations of the clustering coefficient to weighted complex networks. *Phys Rev E* 75:027105. <https://doi.org/10.1103/PhysRevE.75.027105>
37. Ito MI, Ohnishi T (2019) Weighted network analysis of biologically relevant chemical spaces. <https://doi.org/10.48550/ARXIV.1911.05259>

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ready to submit your research? Choose BMC and benefit from:

- fast, convenient online submission
- thorough peer review by experienced researchers in your field
- rapid publication on acceptance
- support for research data, including large and complex data types
- gold Open Access which fosters wider collaboration and increased citations
- maximum visibility for your research: over 100M website views per year

At BMC, research is always in progress.

Learn more biomedcentral.com/submissions

