In [1]:
```
import pandas as pd
import json
import networkx as nx
import nx_altair as nxa
import json
import squarify
import altair as alt
import numpy as np
import matplotlib
from sklearn.cluster import AgglomerativeClustering
from networkx.algorithms.community import *
from scipy.cluster.hierarchy import dendrogram,leaves_list
from scipy.cluster.hierarchy import ClusterWarning
from warnings import simplefilter
simplefilter("ignore", ClusterWarning)
```

Information Visualization II

School of Information, University of Michigan

Week 3:

- Hierarchies and Networks

Assignment Overview

The objectives for this week are for you to:

- Examine practical methods for visualizing hierarchical and network datasets
- Experiment with external libraries (squarify, networkx, and nx-altair)

The total score of this assignment will be

- Problem 1 - 30 points
- Problem 2 - 50 points
- Problem 3 - 20 points

Resources:

- We have created datasets for you for this week based on the Simpsons' TV show (more on this in a moment...). There are a number of files we will use in the assets directory.

Important notes:

1) This assignment will look long, but we've written a lot of helper functions for you. The amount you will need to implement will be relatively small.

2) Grading for this assignment is entirely done by manual inspection.

3) When turning in your PDF, please use the File -> Print -> Save as PDF option ***from your browser***. Do ***not*** use the File->Download as->PDF option. Complete instructions for this are under Resources in the Coursera page for this class.

In [2]:
```
# enable correct rendering (unnecessary in later versions of Altair)
alt.renderers.enable('default')

# uses intermediate json files to speed things up
alt.data_transformers.enable('json')
```

Out[2]:
```
DataTransformerRegistry.enable('json')
```

The Simpsons....

For today's exercise, we're going to be using data from the Simpsons! The Simpsons is an animated comedy that has appeared on American TV continuously since 1989. They are up to nearly 700 episodes over 32 seasons(!) on television, not to mention the movies, books, theme parks, and many other cultural artifacts. Basically, they're everywhere. The main Simpsons family--Homer (the father), Marge (the mother), and the kids: Homer, Lisa, and Maggie--have been joined by 100s of characters over the years.

The Simpsons have lasted long enough that it's probably one of the few shows that are still on the air that need to have older episodes edited or removed because they're no longer appropriate. Nonetheless, jokes and quotes from the Simpson have become engrained in our culture. For this assignment, we will be analyzing key quotes from the history of the Simpsons as a network problem.

We have extracted quote data from the wikiquotes project. For each season, we look at each episode. For each episode, we identify the quotes recorded in wikiquotes.

While some quotes are "one-liners" by a single character:

- **Homer**: Aww, it makes no sense; I haven't changed since high school and all of a sudden I'm uncool.

We are interested in interactions between characters. For example:

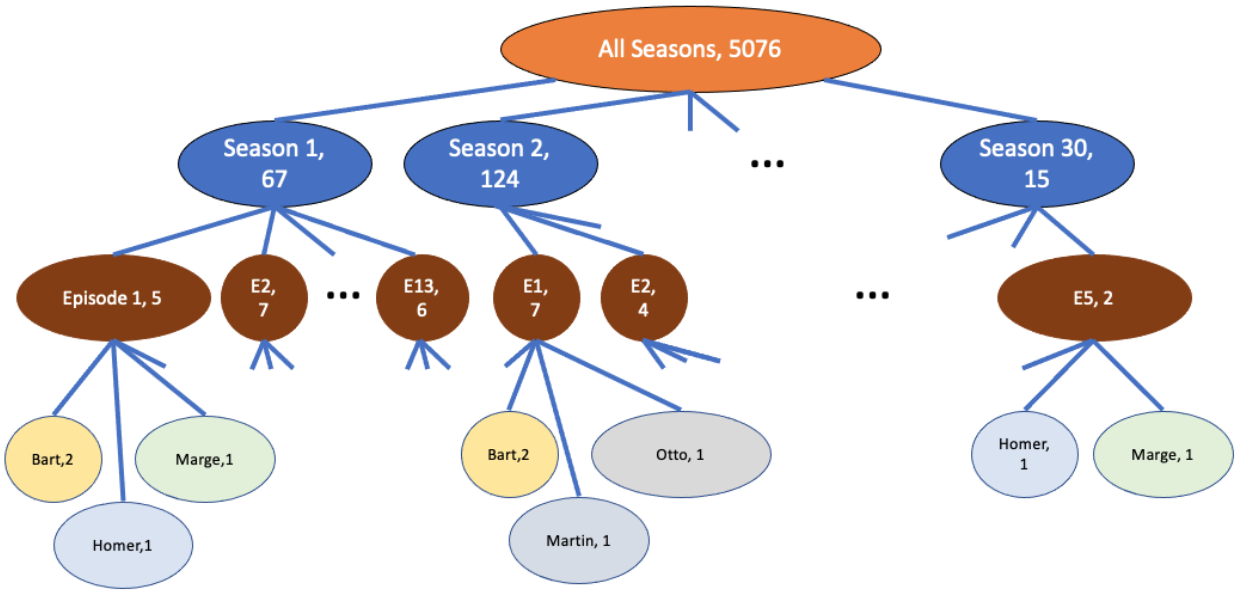
- **Homer**: Doughnut?
- **Lisa**: No, thanks. Do you have any fruit?
- **Homer**: This has purple stuff inside. Purple is a fruit.

These multi-character quotes will become the "edges" in our network, but more on that in a bit. We're going to start with a simpler analysis of the hierarchical data in this database.

Problem 1 (30 points)

For this first problem, we are going to look at the hierarchical representation of the quote database. Specifically, we have multiple seasons. For each of the seasons, we have multiple episodes. Finally, for each of the episodes, we have multiple characters who have participated in funny/memorable scenes. For example, Homer has participated in 972 quoted conversations in the dataset! Bart comes in a remote second at 548. Our goal is to have a visualization that allows us to compare which seasons/episodes/characters had the most quoted conversations. Ideally, we'd also like to know if a certain character had many conversations in one episode and fewer in others? Were there any outlier episodes with lots of conversations? Was there one season with many conversations? All these domain tasks map neatly to abstract problems of hierarchical data. Hierarchical visualization techniques will be a reasonable solution given these questions.

Abstractly, our data looks something like this:



What this tells us is that there are 5076 conversations in our dataset. Season 2, for example, has 124 of those. In Season 2, Episode 1 had 7 quotes, and Bart was responsible for 2 of them (the episode and quote numbers are made up for this example). Notice that the same characters can appear multiple times.

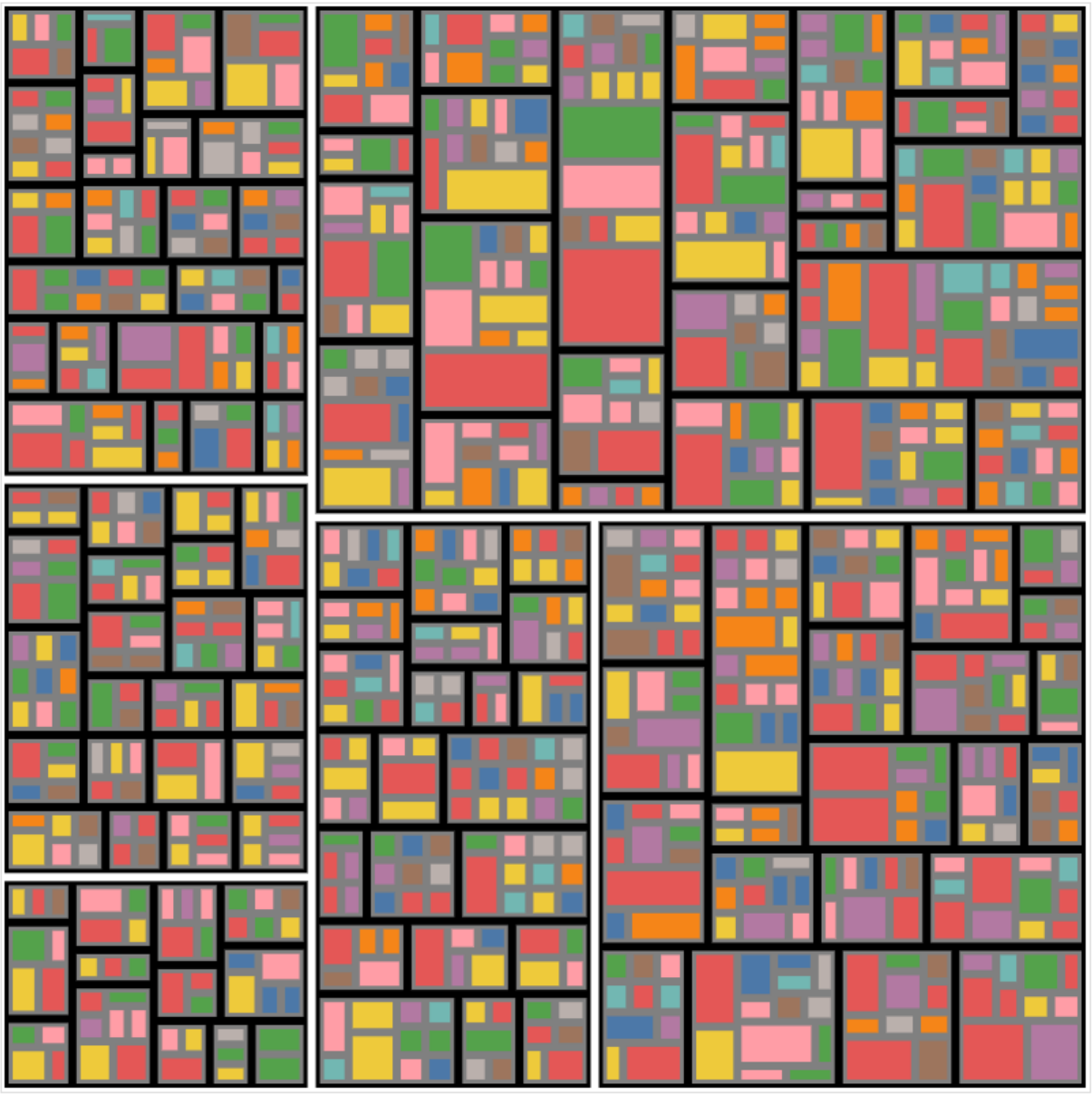
To help us with the real data, we have constructed the hierarchy as a nested data structure. Let's load the data in:

In [3]:
```
with open('assets/simpsonshier.jsonl') as json_file:
    allseasons = json.load(json_file)
```

In [4]:
```
# this variable is a massive JSON object with a hierarchy of seasons -> episodes -> characters. Each "Node"
# hierarchy is a dictionary which has an id, a type (one of 'season','episode', or 'character'), an optional
# label (episodes had names as well as numerical ids), the value (the number of quotes), and children (a
# list of nodes that sit underneath)

allseasons

# for example, we can find season 2's quote quote:
season2 = allseasons[1]
print("season ",season2['id'], "had", season2['value'], "quotes")

# for the first episode of season 1 we see the following
season2e1 = season2['children'][0]
print(season2e1)

# you'll see that this episode was called "Bart Gets and F" and it had 7 quotes. Bart was responsible for 2 of these
```

```
season  s02 had 124 quotes
{'id': 's02e14', 'type': 'episode', 'label': 'Bart Gets an F', 'value': 7, 'children': [{'id': 'Bart', 'type': 'characte
r', 'value': 2}, {'id': 'Mrs. Krabappel', 'type': 'character', 'value': 1}, {'id': 'Martin', 'type': 'character', 'value':
1}, {'id': 'Otto', 'type': 'character', 'value': 1}, {'id': 'Sherri', 'type': 'character', 'value': 1}, {'id': 'Terri', 't
ype': 'character', 'value': 1}]}
```

Recall that we have two main ways to display hierarchies. Node link diagrams (rather like the image above) and space-filling versions such as TreeMaps. We're going to demonstrate how you can build a treemap using a combination of tools. We'll visualize only some of the seasons to keep things a bit sane, but the first 6 roughly look like this:



More on the color encoding later, but you can see the 6 black boxes (seasons), the multiple gray boxes (the episodes), and the colorful objects inside of that are the characters. Unfortunately, Altair doesn't have a treemap layout built in. We'll be using the squarify library to generate the coordinates. Squarify works by generating one level of the hierarchy at a time. So we need a function that lays out the seasons, and then for each episode re-runs squarify but restricts it to the space allocated to the season. After that we re-run squarify to plot the position of each character in that episode. We've written this recursive function for you below. In the end it returns a dataframe with the bits we need to plot:

In [5]:

```
def rectangleIter(data,width,height,xof=0,yof=0,frame=None,level=-1,parentid=""):
    # data: hierarchical structured data
    # width: width we can work in
    # height: height we can work in
    # xof: x offset
    # yof: y offset
    # frame: the dataframe to add the data to, if None, we create one
    # the level of the treemap (will default to 0 on first run)
    # parentid: a string representing the parent of this node
    # returns dataframe of all the rectangles

    if (frame is None):
        frame = pd.DataFrame()
    level = level + 1
    values = []
    children = []
    for parent in data:
        values.append(parent['value'])
        if ('children' in parent):
```

```
            children.append(parent['children'])
        else:
            children.append([])

    # normalize
    values = squarify.normalize_sizes(values, width, height)

    # generate the
    padded_rects = squarify.padded_squarify(values, xof, yof, width, height)

    i = 0
    for rect in padded_rects:
        # adjust the padding and copy the useful pieces of data over
        parent = data[i]
        rect['width'] = rect['dx']
        rect['height'] = rect['dy']
        del rect['dx']
        del rect['dy']
        rect['x2'] = rect['x'] + rect['width'] - 2
        rect['y2'] = rect['y'] + rect['height'] - 2
        rect['x'] = rect['x'] + 2
        rect['y'] = rect['y'] + 2
        rect['width'] = rect['x2'] - rect['x']
        rect['height'] = rect['y2'] - rect['y']
        rect['id'] = parent['id']
        rect['type'] = parent['type']
        rect['value'] = parent['value']
        rect['level'] = level
        if 'label' in parent:
            rect['label'] = parent['label']
        else:
            rect['label'] = parent['id']
        rect['parentid'] = parentid
        frame = frame.append(rect,ignore_index=True)

        # iterate
        frame = rectangleIter(children[i],rect['width'],rect['height'],rect['x'],rect['y'],
                              frame=frame,level=level,parentid=parentid+" → "+rect['label'])
        i = i + 1
    return(frame)
```

In [6]:

```
shortseason = allseasons[0:6]                  # let's grab the first 6 seasons
rect_table = rectangleIter(shortseason,800,800) # and run them through the treemap algorithm
```

In [7]:

```
# let's look at what's inside
rect_table.sample(5)
```

Out[7]:

	height	id	label	level	parentid	type	value	width	x	x2	y	y2
492	12.685570	Barry White	Barry White	2.0	→ s04 → Whacking Day	character	1.0	17.138520	324.707167	341.845688	371.933134	384.618704
296	11.449198	Maude	Maude	2.0	→ s03 → Bart the Lover	character	1.0	17.465608	125.708995	143.174603	615.673510	627.122708
417	15.799832	Aide	Aide	2.0	→ s04 → Lisa's First Word	character	1.0	12.633017	237.571429	250.204446	200.248647	216.048478
656	31.531426	Homer	Homer	2.0	→ s05 → Homer and Apu	character	5.0	54.792480	597.169946	651.962426	221.523143	253.054569
300	11.449198	Lisa	Lisa	2.0	→ s03 → Bart the Lover	character	1.0	17.465608	149.174603	166.640212	633.122708	644.571907

What you'll find inside the table is everything you need to create your visualization using Altair. There are coordinates for the rectangle (the start point: x,y; the end points: x2,y2, and the width, height). You'll also find the id/label for each rectangle, a numerical level (0,1, or 2 in this case), and the type of rectangle (character, episode, or season).

Problem 1.1

Using Altair, generate a treemap using the `rect_table` dataframe. Regenerate the image above. Use the same color scheme for seasons and episodes as we did (your character colors may vary). We used 800 pixels square for the visualization. Complete the function `staticTreemap` that returns this Altair visualization given the rectangles frame.

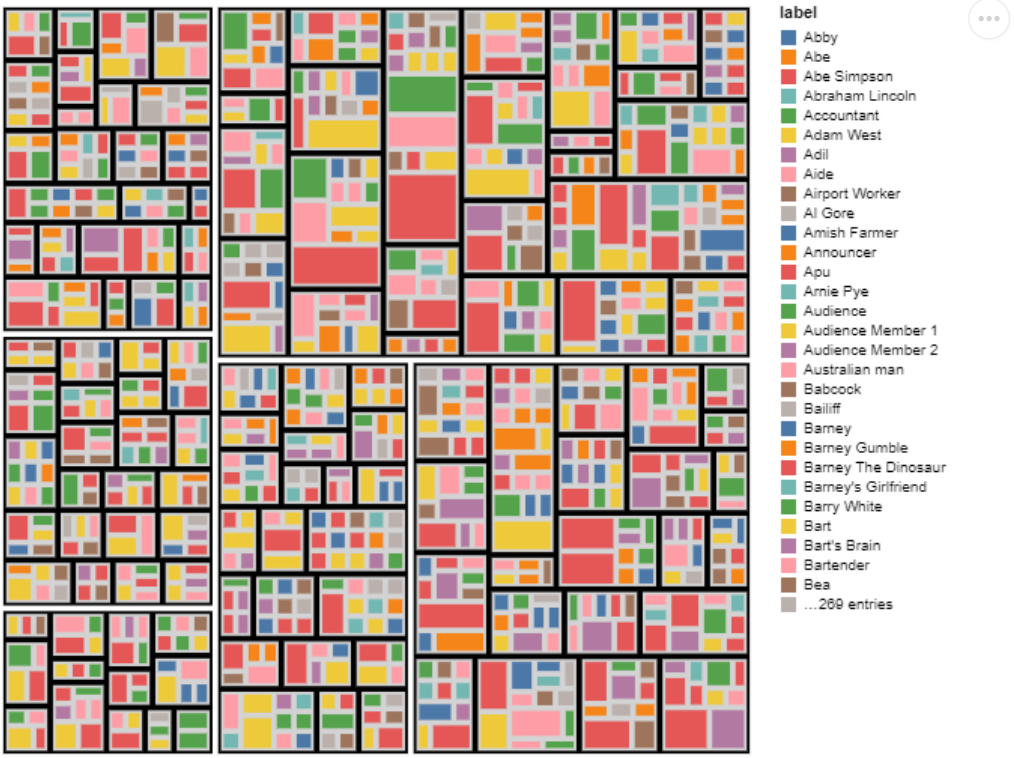
Some hints:

- think layering
- try to get one layer (maybe rectangles for the series objects) to work first and then add more

In [8]:
```
def staticTreemap(inputFrame):
    # input inputFrame the rectangles frame as described above
    # return a static Altair treemap visualization

    # YOUR CODE HERE
    frame_1 = inputFrame.loc[inputFrame['type'] == 'season']

    chart_1 = alt.Chart(frame_1, width=500, height=500).mark_rect(fill='black').encode(
        x=alt.X('x', axis=None),
        y=alt.Y('y', axis=None),
        x2='x2',
        y2='y2'
    )

    frame_2 = inputFrame.loc[inputFrame['type'] == 'episode']

    chart_2 = alt.Chart(frame_2, width=500, height=500).mark_rect(fill='lightgrey').encode(
        alt.X('x', title=None),
        alt.Y('y', title=None),
        x2='x2',
        y2='y2'
    )

    frame_3 = inputFrame.loc[inputFrame['type'] == 'character']

    chart_3 = alt.Chart(frame_3, width=500, height=500).mark_rect().encode(
        alt.X('x', title=None),
        alt.Y('y', title=None),
        x2='x2',
        y2='y2',
        color='label'
    )

    return alt.layer(chart_1, chart_2, chart_3).configure_view(strokeWidth=0).properties(width=800, height=800)
    # raise NotImplementedError()
```

In [9]:
```
# if you did this correctly, the following should work
staticTreemap(rect_table)
```

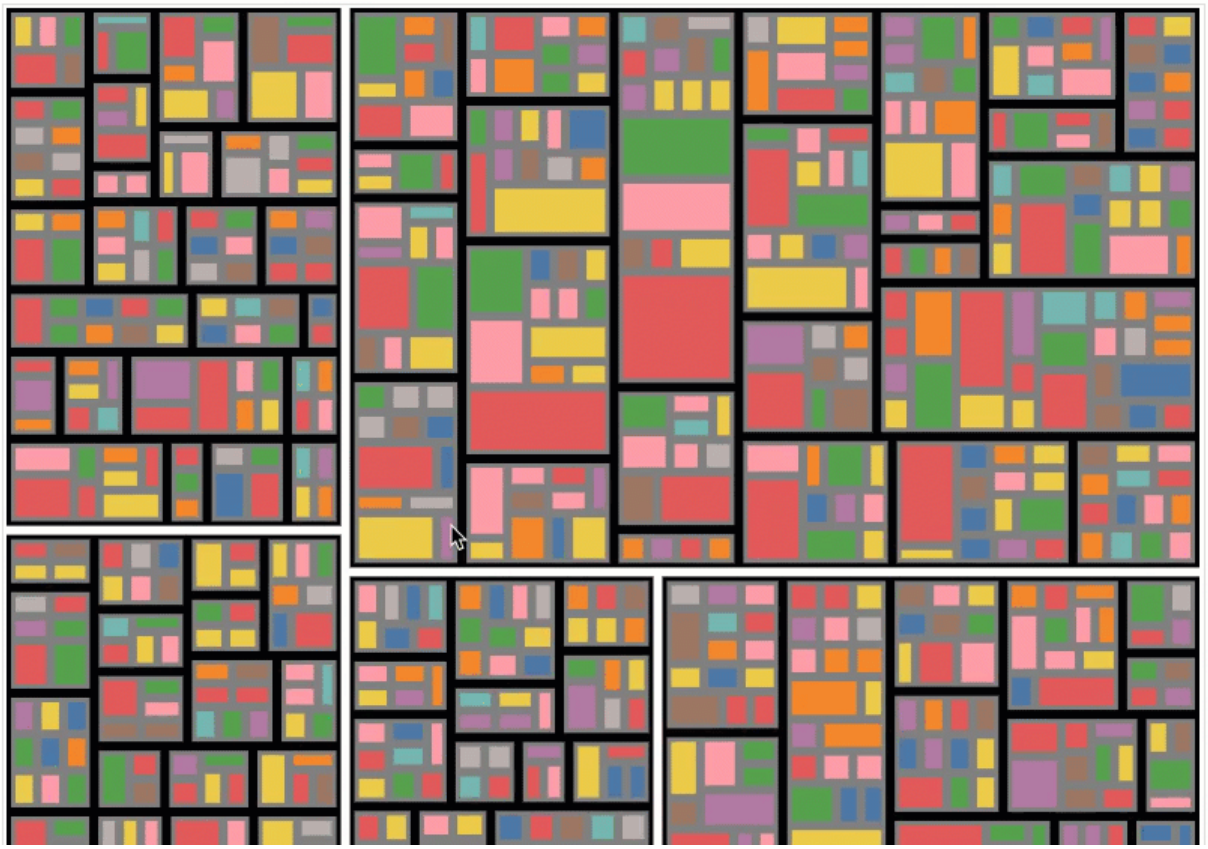
Out[9]:



Problem 1.2

While the solution above looks cool, it's really not so helpful for understanding the data. We can see that seasons are different in terms of the number of quotes, there are differences in episodes, and some characters have many quotes in one episode and few in the others. But we have no idea what these seasons, episodes, or characters are!

We could try to add labels, but that might become unreadable pretty fast. Instead, your task is to add interactivity to make this visualization more usable. You can pick your strategy for doing this. We've created the following example to get you thinking:



To help us determine what we're looking at, we've added tooltips and to aid with comparisons between the same characters over mutliple episodes/seasons we've added the ability to select characters of interest. You are welcome to adopt whatever strategy you think will be expressive/effective here.

Complete the function `interactiveTreemap` to return this interactive Altair visualization.

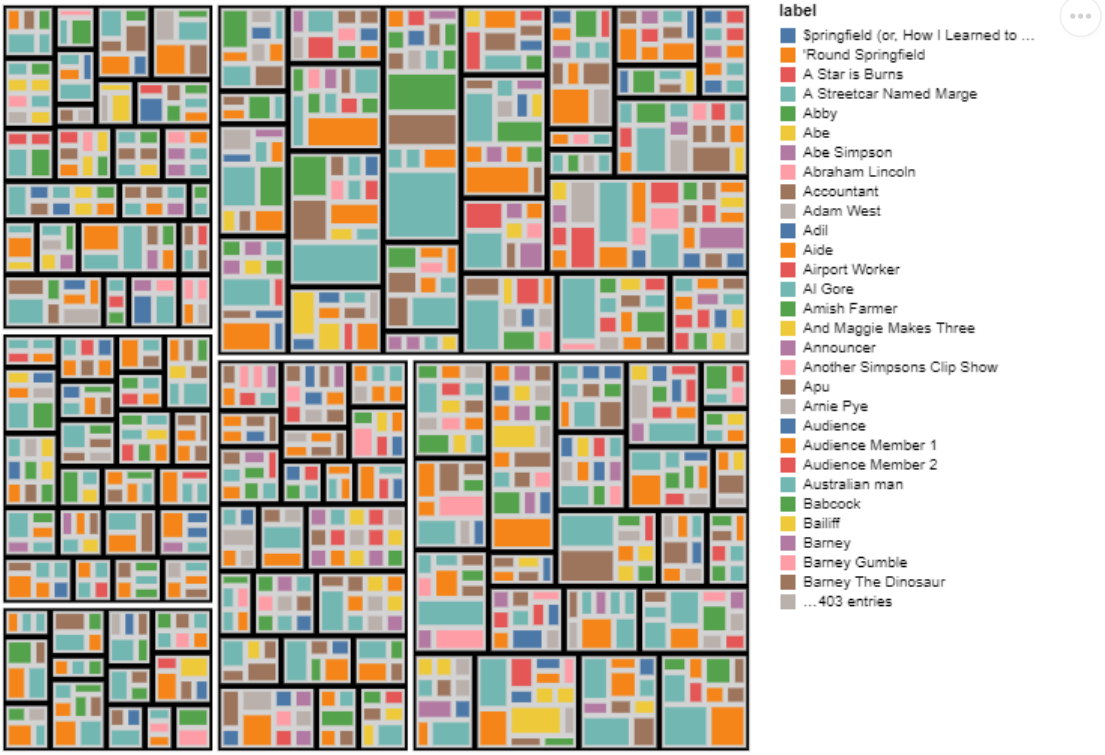
In [10]:

```
def interactiveTreemap(inputFrame):
    # input inputFrame the rectangles frame as described above
    # return a static Altair treemap visualization

    # YOUR CODE HERE
    selection = alt.selection_single(fields=['label'])
    frame_1 = inputFrame.loc[inputFrame['type'] == 'season']

    chart_1 = alt.Chart(frame_1, width=500, height=500).mark_rect(fill='black').encode(
            x=alt.X('x', axis=None),
            y=alt.Y('y', axis=None),
            x2='x2',
            y2='y2',
            color=alt.condition(selection, 'label:N', alt.value('grey')),
            tooltip = [alt.Tooltip('parentid:N'), alt.Tooltip('id:N'), alt.Tooltip('value:Q')]
        ).interactive()

    frame_2 = inputFrame.loc[inputFrame['type'] == 'episode']

    chart_2 = alt.Chart(frame_2, width=500, height=500).mark_rect(fill='lightgrey').encode(
            alt.X('x', title=None),
            alt.Y('y', title=None),
            x2='x2',
            y2='y2',
            color=alt.condition(selection, 'label:N', alt.value('grey')),
            tooltip = [alt.Tooltip('parentid:N'), alt.Tooltip('id:N'), alt.Tooltip('value:Q')]
        ).interactive()
```

```
    frame_3 = inputFrame.loc[inputFrame['type'] == 'character']

    chart_3 = alt.Chart(frame_3, width=500, height=500).mark_rect().encode(
            alt.X('x', title=None),
            alt.Y('y', title=None),
            x2='x2',
            y2='y2',
            color=alt.condition(selection, 'label:N', alt.value('grey')),
            tooltip = [alt.Tooltip('parentid:N'), alt.Tooltip('id:N'), alt.Tooltip('value:Q')]
    ).interactive()

    return alt.layer(chart_1, chart_2, chart_3).configure_view(strokeWidth=0).add_selection(selection).properties(width=80
    # raise NotImplementedError()
```

In [11]:
```
interactiveTreemap(rect_table)
```

Out[11]:



Problem 2 (50 points)

For our next analysis, we're going to look a little more closely at conversation networks. Each quotable conversation can be modeled as a small network. Nodes will correspond to the characters, and edges will be the number of conversations two character's co-occurred in. For example, if Bart, Homer, and Lisa are in the same quote, we would construct 3 undirected edges: Bart to Homer, Bart to Lisa, and Lisa to Homer. By aggregating all these conversations together (over episodes or seasons), we can compute the "weight" of an edge: the total number of quoted conversations those characters interacted over. From this kind of network, we can identify who the central characters are. Who is interacting with the most others in a quotable way? Are there small communities?

For this problem, we are going to use two libraries to help us out: networkx--a library for manipulation and analysis of graph data structures (it will also generate layouts), and nx-altair a library that can generate Altair plots from networkx data. We don't expect you to learn all the pieces of these libraries. However, we'll explore a few functions to get some interesting visualizations.

Let's start by taking a look at this network data.

In [12]:
```
# helper function to load the data
def loadData(filepath):
    articles = []
    with open(filepath) as fp:
        for docid, line in enumerate(fp):
            #print(line)
            #print(docid)
            doc = json.loads(line)
            char1 = doc['c1']
            char2 = doc['c2']
```

```
            articles.append(doc)
    return pd.DataFrame(articles)
```

In [13]:
```
# we're going to load a data frame representation of the network to start
simpsons = loadData('assets/simpsons.jsonl')
```

In [14]:
```
# let's look inside
simpsons.sample(5)
```

Out[14]:

	season	episode	lineid	c1	c2
3809	14	C.E. D'oh	1551	Carl	Lenny
80	2	Bart Gets an F	28	Martin	Sherri
3863	15	The Regina Monologues	1593	Homer	Teen
2810	10	Bart the Mother	1057	Bart	Marge
1590	7	The Simpsons 138th Episode Spectacular	583	Homer	Smithers

We see a row for every edge. The `season` and `episode` column has the season the conversation happened in. The `lineid` is a unique id for the conversation (note that if the conversation involved more than two people, we'd see the same lineid multiple times; see the Bart/Homer/Lisa example above). The columns `c1` and `c2` hold the two characters' names (the name in c1 will always be alphabetically before c2). We have tried to clean up this data as much as possible, but you may see some inconsistencies with names. For example, you might find different entries for "Skinner" and "Principal Skinner" even though they are the same character.

Next, we'll build our network using networkx.

In [15]:
```
# utility classes

def buildNetwork(quoteFrame):
    # takes as input the quote frame (e.g., simpsons) or some subset of it
    # and returns an undirected networkx graph

    weight = quoteFrame.groupby(['c1','c2']).count()
    weight = weight.reset_index()
    toret = nx.Graph()
    for row in weight.iterrows():
        row = row[1]
        if (row['c1'] not in toret.nodes):
            toret.add_node(row['c1'])
            toret.nodes[row['c1']]['appearance'] = 0
            toret.nodes[row['c1']]['label'] = row['c1']
        if (row['c2'] not in toret.nodes):
            toret.add_node(row['c2'])
            toret.nodes[row['c2']]['appearance'] = 0
            toret.nodes[row['c2']]['label'] = row['c2']
        toret.nodes[row['c1']]['appearance'] = toret.nodes[row['c1']]['appearance'] + 1
        toret.nodes[row['c2']]['appearance'] = toret.nodes[row['c2']]['appearance'] + 1
        toret.add_edge(row['c1'],row['c2'])
        toret.edges[row['c1'],row['c2']]['weight'] = int(row['season'])
    return toret

def getLayout(positions):
    # helper function to build a dataframe of positions for nodes
    elems = []
    nodes = list(positions.keys())
    for n in nodes:
        elems.append({'node':n,'x':positions[n][0],'y':positions[n][1]})
    return(pd.DataFrame(elems))

def setCommunityLabels(G,communities):
    # adds community labels to the networkx graph nodes
    id = 0
    for c in communities:
        id = id + 1
        for n in c:
            G.nodes[n]['community'] = id
    return(G)
```

Problem 2.1

We're going to go through a number of steps to build our first network diagram with networkx and nx-altair.

In [16]:
```
# let's start by grabbing only the network for a single season (6)
season6 = buildNetwork(simpsons[simpsons.season == 6])
```

In [17]:
```
# season6 is a networkx object. You can ask for the edges or nodes
season6.nodes
```

Out[17]:
```
NodeView(('Abe', 'Crazy Old Man', 'Family', 'Homer', 'Jasper', 'Quimby', 'Accountant', 'Krusty the Clown', 'Aide', 'Al Gor
e', 'Bart', 'Kool', 'President', 'the Gang', 'Airport Worker', 'Amish Farmer', 'Announcer', 'Godfrey Jones', 'Kent Brockma
n', 'Apu', 'Chief Wiggum', 'Moe', 'Audience Member 1', 'Audience Member 2', 'McBain', 'Rainier Wolfcastle', 'Sherman', 'Wo
lfcastle', 'Australian man', 'Barney', 'Lisa', 'Man', 'Mayor Quimby', 'Woman', "Bart's Brain", 'Database', 'Grampa', 'Grou
ndskeeper Willie', 'Helen', 'Helen Lovejoy', 'Hibbert', 'Jessica', 'Jessica Lovejoy', 'Lunchlady Doris', 'Marge', 'Marin
e', 'Martin', 'Maude', 'Milhouse', 'Mrs. Krabappel', 'Ned', 'Ned Flanders', 'Nelson', 'Principal Skinner', 'Reverend Lovej
oy', 'Server', 'Shelby', 'Sherri', 'Sideshow Bob', 'Skinner', 'TV Announcer', 'Teacher', 'Bartender', 'Bob', 'Boy', 'Broth
er', 'Burns', 'Chespirito', 'Hans Moleman', 'Hopkins', 'Shatner', 'Smithers', 'Spielbergo', 'Carl', 'Lenny', 'Martian', 'M
r. Burns', 'Stonecutters', 'Carla', 'Clavin', 'Norm', 'Sam', 'Woody', 'Dr. Hibbert', 'Homer/Marge', 'Clerk', 'Store Owne
r', 'Comic Book Guy', 'Congressman', 'Speaker', 'Darth Vader', 'James Earl Jones', 'Mufasa', 'Mufasa/Vader/Jones', 'Murph
y', 'Dr. Zweig', 'Euro-Itchy', 'Scratchy Land Ticket Attendant', 'Fat Tony', 'Legs', 'Louie', 'Flanders', 'Frink', 'Girl',
'Pilot 1', 'Pilot 2', 'Hitler', 'Officer', "Homer's Brain", "Homer's Liver", 'Jay', 'Lesbian', 'Maude Flanders', 'Mr. Peab
ody', 'Number One', 'Patty', 'Vendor', 'Hugh', 'Hutz', 'Jay Sherman', 'Jimbo', 'Judge', 'Largo', 'Leopold', 'Miss Hoover',
'Ralph', 'Maggie', 'Willie', 'Park Announcer', 'Nurse', 'Mr Burns', 'Old Woman', 'Scott', 'Selma'))
```

In [18]:
```
# we also calculate a special attribute of nodes called 'appearance' which is equivalent to the
# degree of the node. This will be useful to us when we want to change the visual property of the
# node
season6.nodes['Bart']['appearance']
```

Out[18]:
```
35
```

In [19]:
```
# which is the same as this
print(season6.degree('Bart'))
```

```
35
```

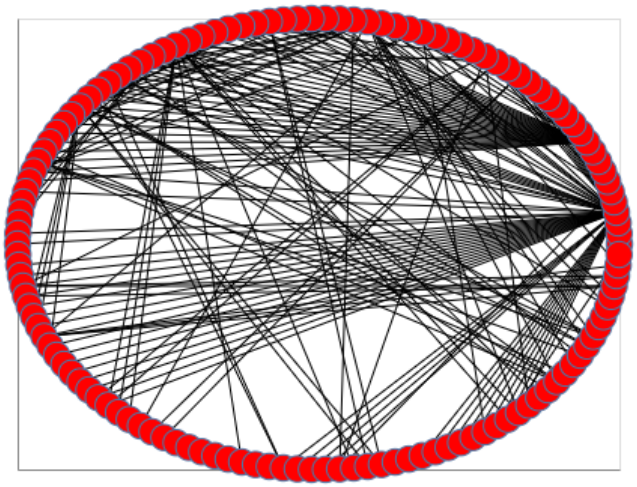
In [20]:
```
# you can ask for the weights of specific edges:
season6.edges['Homer','Bart']['weight']
```

Out[20]:
```
19
```

In [21]:
```
# You can even ask networkx to find the x-y positions for you:
circular_pos = nx.circular_layout(season6)
```

In [22]:
```
# once you have the layout, you can ask nx-altair to draw the graph for you
nxa.draw_networkx(season6, pos = circular_pos)
```

Out[22]:

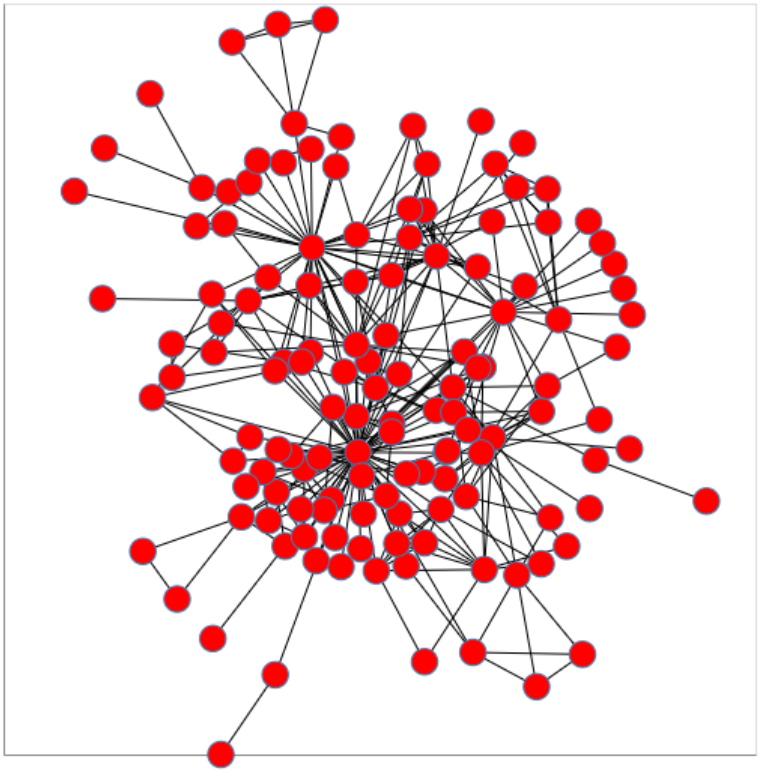


Problem 2.1.1

Clearly, a circular layout isn't going to be great here. Thankfully, networkx has many other layouts: https://networkx.org/documentation/stable//reference/drawing.html#layout. Find one that generates a visualization that satisfies the 'properties of a good graph layout' described in the video lecture:

In [23]:
```
# modify the line below
pos = nx.kamada_kawai_layout(season6)

# remove the exception once you're ready
# YOUR CODE HERE
# raise NotImplementedError()

# to generate a good visualization (you shouldn't need to modify the line below)
nxa.draw_networkx(season6, pos = pos).properties(
    # nx-altair returns an Altair visualization, so we can modify
    # the properties as usual
    width=500,
    height=500
)
```

Out[23]:

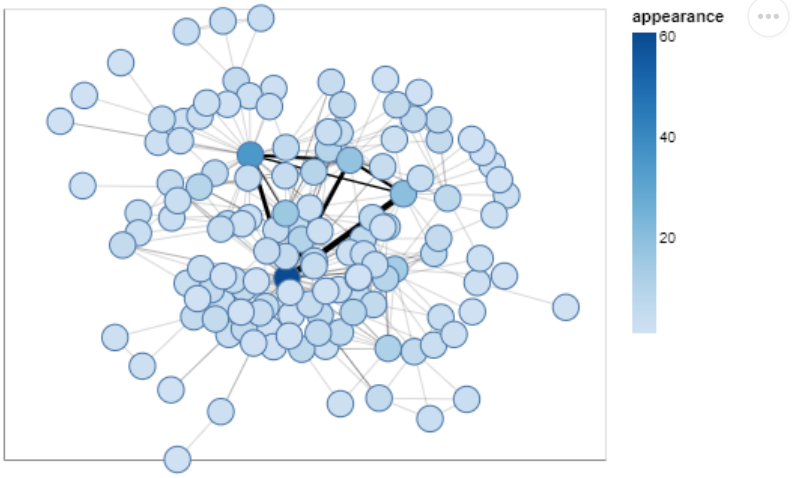


Problem 2.1.2

Now that we have a network visualization that looks ok, we can start to modify it some more. The nx-altair library returns a layered Altair chart when you call `draw_networkx` . The bottom layer is the edges (lines) and the top will be the nodes (circles). The `draw_networkx` function allows you to control **some** properties of the visualization. For example:

In [24]:
```
# recall that we have calculated 'weight' (an edge feature) and 'apperance' (a node feature--which is the degree)
# let's modify our network to visualize these:

nxa.draw_networkx(season6, pos=pos,width='weight',node_color='appearance')
```

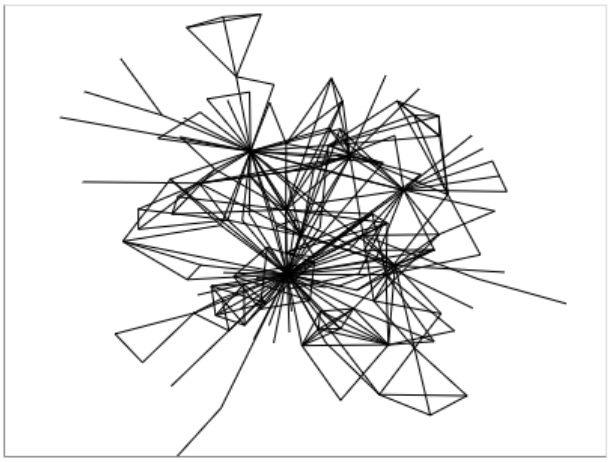
Out[24]:



In some situations, we'll want to get the two different parts of the Layered plot so we can refine them. For that, we have the ability to ask for `draw_networkx_edges` and `draw_networkx_nodes`:

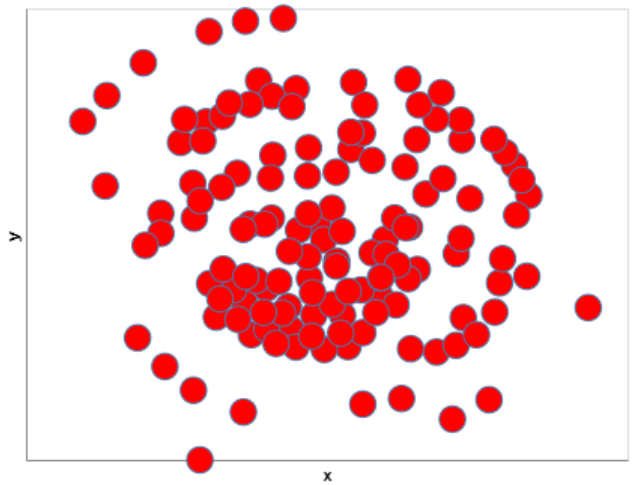
In [25]:
```
e = nxa.draw_networkx_edges(season6, pos=pos)  # get the edge layer
e  # draw it
```

Out[25]:



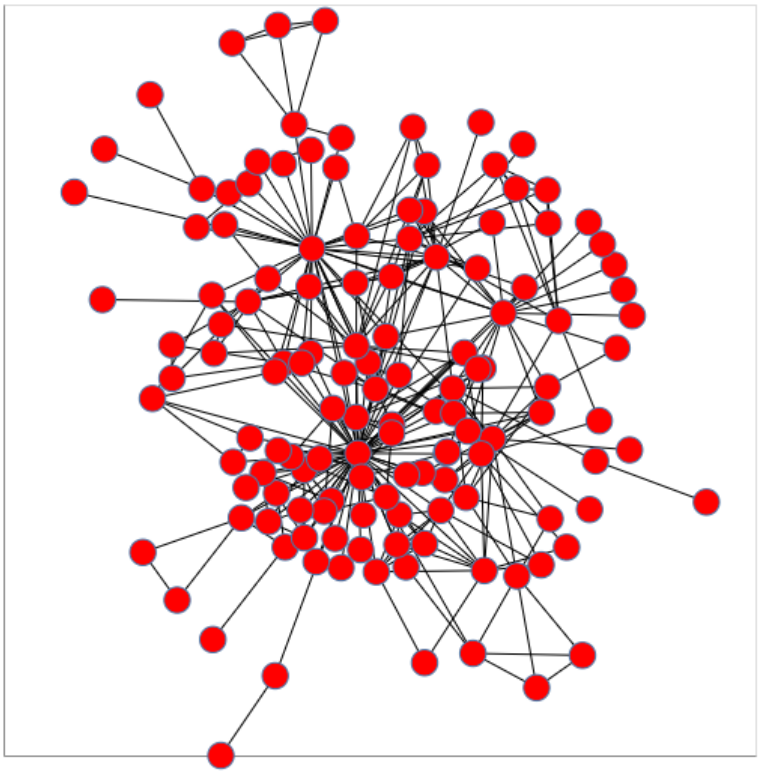
In [26]:
```
n = nxa.draw_networkx_nodes(season6, pos=pos)  # get the node layer
n # draw it
```

Out[26]:



In [27]:
```
# combine them back
(e+n).properties(
    width=500,height=500
)
```

Out[27]:



Problem 2.1.3

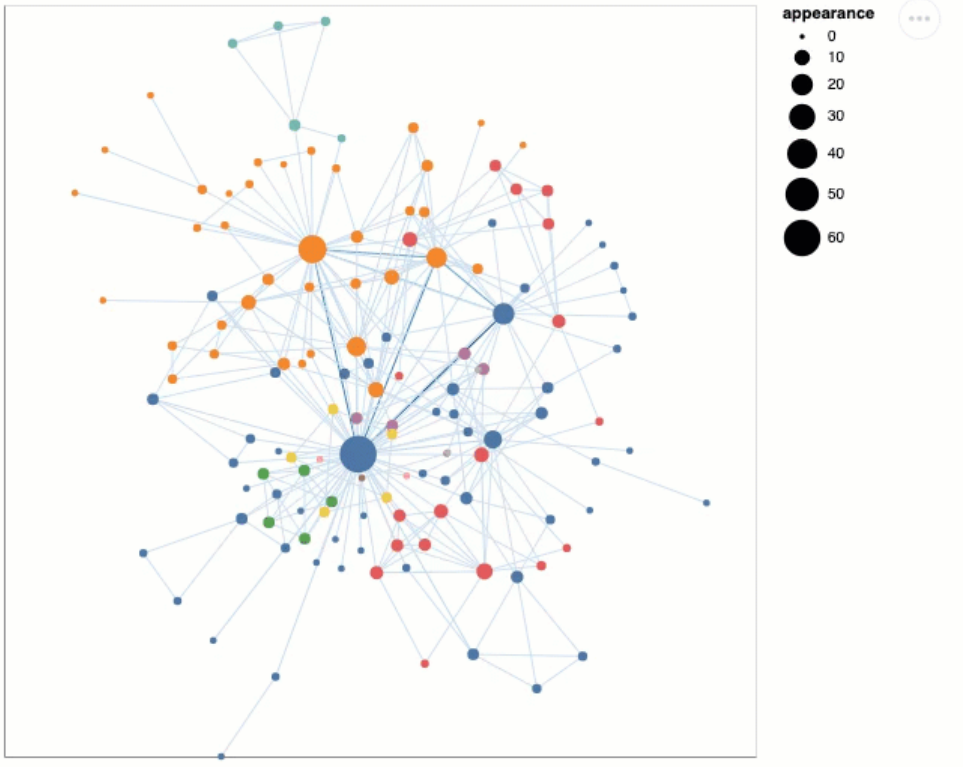
We're going to calculate a new feature of nodes based on a community detection algorithm. If you want to know more read here. We'll augment our networkx object so we have a `community` "column" that can be used as a nominal feature of the nodes in visualization.

In [28]:
```
season6 = setCommunityLabels(season6, greedy_modularity_communities(season6))
```

In [29]:
```
# let's see what community Bart is now in:
season6.nodes['Bart']['community']
```

Out[29]: 2

Modify the node and edge layers to add tooltips for mousing over the nodes. Change the color of the nodes based on the community, the size based on appearance, and change the edge width and color based on weight. Something like this:



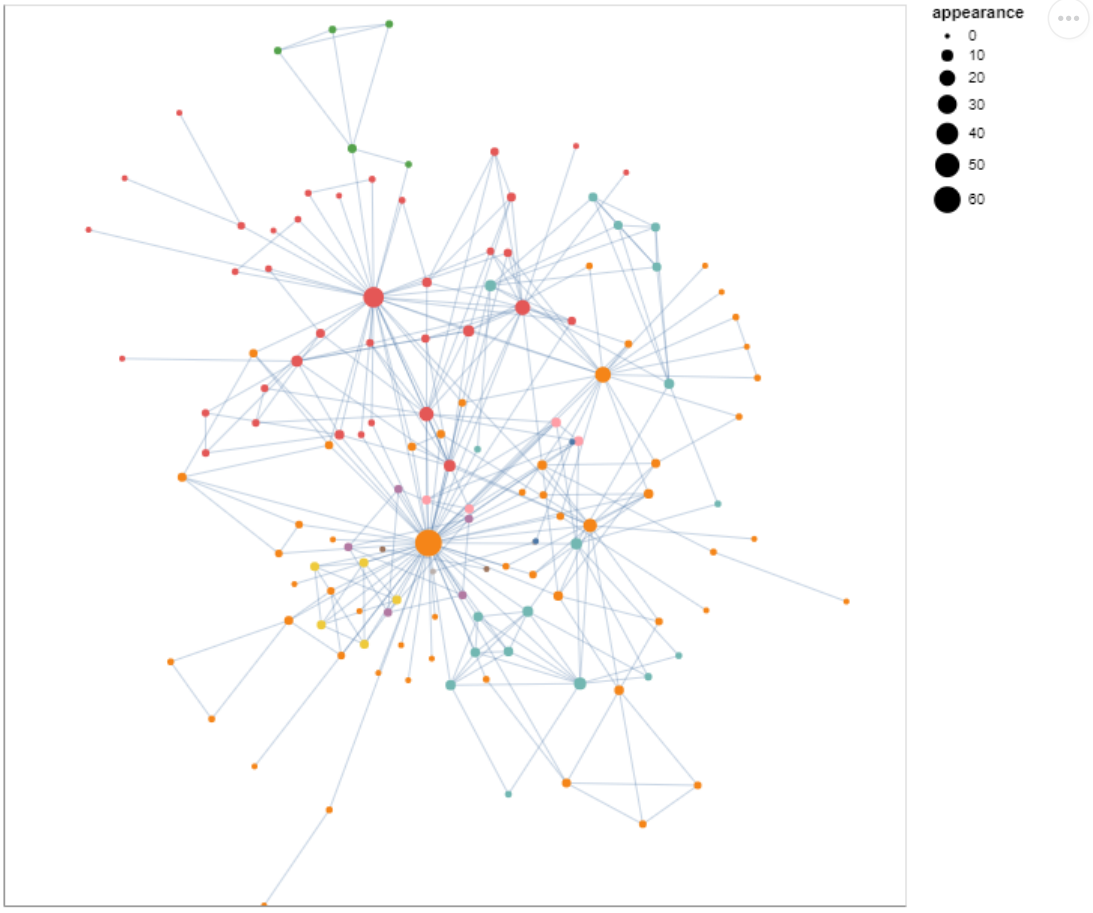
Some hints:

- you may want to re-encode the node layer to mark_circle or mark_point explicitly (`n = n.mark_circle(...)`)
- the edges are built using a mark_line, you should look at the features of this mark type here

In [30]:

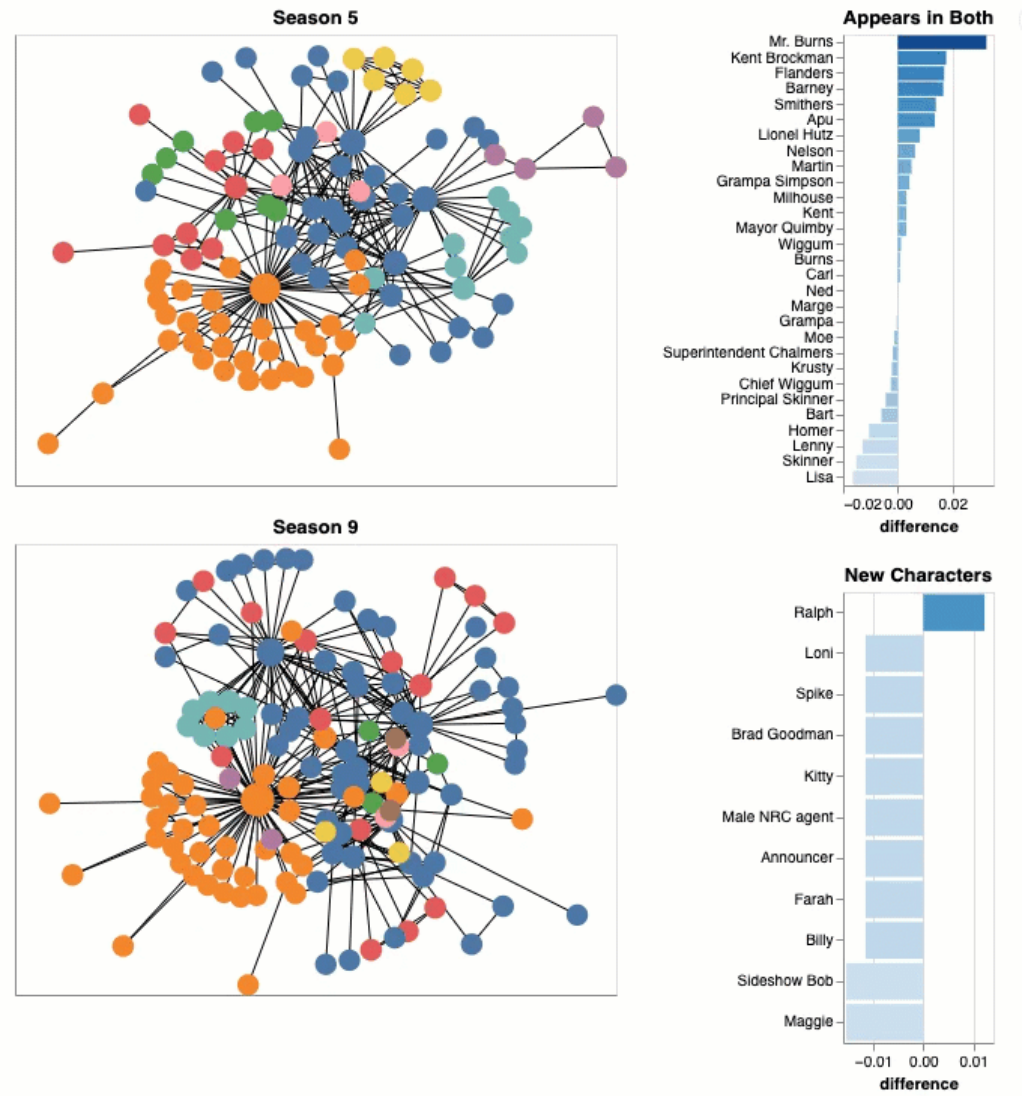
```
e = nxa.draw_networkx_edges(season6, pos=pos)  # get the edge layer
n = nxa.draw_networkx_nodes(season6, pos=pos)  # get the node layer

# modify the code to change the encodings
nearest = alt.selection_single(on='mouseover', nearest=True)

e = e.mark_point(filled=True, opacity=1).encode(
    color=alt.Color('community:N', legend=None),
    size=alt.Size('appearance:Q', scale=alt.Scale(domain=[0,10,20,30,40,50,60], range=[10,60]))
)

n = n.mark_circle(filled=True, opacity=1).encode(
    color=alt.Color('community:N', legend=None),
    size=alt.Size('appearance:Q', scale=alt.Scale(domain=[0,10,20,30,40,50,60], range=[10,60])),
    tooltip=alt.Tooltip('label:N', title=None)
)

l = e.mark_line(color='lightgrey', opacity=0.3).encode()
# remove the exception to render
# YOUR CODE HERE
# raise NotImplementedError()
(e.add_selection(nearest).interactive()+l+n.interactive()).properties(
    width=600,height=600
)
```

Out[30]:



Problem 2.2

Now that we have the basics, we're going to build a visualization to help us compare pairs of seasons. We'd like to understand which characters have been more central to which season and how that's changed. Our final dashboard will look something like this:



Key things to observe: color is based on community labels, there's a mouse over interaction that changes the color of ALL the visualizations if the character appears everywhere. There is also a tooltip over the nodes to get the # of appearaces. The network diagrams are laid out neatly, and the bars are sorted based on changes between seasons (and values double encoded through color).

Notice there are two repeating sets of graphs. So we can try to be efficient in how we code this. Let's get started...

In [31]:
```
# we're going to want the networkx objects for different charts, so let's write a function for that

def getNetwork(season):
    # build a networkx object given the season, annotate with community labels
    toret = buildNetwork(simpsons[simpsons.season == season])
    toret = setCommunityLabels(toret,greedy_modularity_communities(toret))
    return(toret)
```

In [32]:
```
# get the networkx objects for seasons 5 and 9
s5net = getNetwork(5)
s9net = getNetwork(9)
```

In [33]:
```
# we also want the data for the two bar charts, we're going to do that part for you...

def getTotal(G):
    # total appearnce across all characters in a given graph
    app = 0
    for nd in G.nodes:
        app = app + G.nodes[nd]['appearance']
    return(app)

def getComparisonData(G1,G2,threshold=5):
    # generate two dataframes given two graphs
```

```
    # the first is the difference in apparances (normalized) when a character is in both
    # the second is for characters that are either in G1 or G2, but not both
    # the threshold defines the cutoff for how many interactions a character must have
    # to be included in the second ('difference') data frame
    t1total = getTotal(G1)
    t2total = getTotal(G2)
    union = []
    difference = []
    allentities = set(G1.nodes).union(set(G2.nodes))

    for i in allentities:
        if ((i in G1.nodes) & (i in G2.nodes)):
            diff = G1.nodes[i]['appearance']/t1total-G2.nodes[i]['appearance']/t2total
            union.append({'label':i,'difference': diff})
        elif (i in G1.nodes):
            if (G1.nodes[i]['appearance'] > threshold):
                difference.append({'label':i,'difference':-G1.nodes[i]['appearance']/t1total})
        elif (i in G2.nodes):
            if (G2.nodes[i]['appearance'] > threshold):
                difference.append({'label':i,'difference':G2.nodes[i]['appearance']/t2total})

    return(pd.DataFrame(union),pd.DataFrame(difference))
```

In [34]:

```
# let's compare the season 5 and 9 networks
union,difference = getComparisonData(s5net,s9net)

# look inside the union dataframe (the difference one will be similar)
union.sample(5)
```

Out[34]:

	label	difference
16	Ned	0.000070
23	Burns	0.000797
22	Martin	0.005042
18	Smithers	0.013931
25	Moe	-0.001313

This is where you need to start coding. You will modify the two functions `getComparisonBar` and `getNetworkDiagram` based on their specifications. Both should return Altair charts (a bar chart and a modified nx-altair diagram respectively).

We will hand you the Altair Selection object (`sel`), but you will need to use it to implement interactivity in your chart. You can look at the `getNetworkDashboard` to see how we implemented this, but basically, we use:

`alt.selection_single(on='mouseover',fields=['label'])`

In [35]:

```
def getComparisonBar(frame,sel,title):
    # return an Altair chart corresponding to the bar chart example
    # above, given one of the two frames (difference or union)
    # frame is a pandas dataframe
    # sel is the Altair selection object (for interactivity)
    # title is the title for the chart
    nearest = sel

    c1 = alt.Chart(frame).mark_bar().encode(
        x='difference:Q',
        y=alt.Y('label:N', sort='-x', title=None),
        color=alt.condition(nearest, alt.Color('difference:Q', legend=None), alt.value('lightgray'))
    ).properties(title=title, width=80, height=450).add_selection(nearest)

    # YOUR CODE HERE
    # raise NotImplementedError()
    return(c1)
```

In [36]:

```
def getNetworkDiagram(G,season,sel):
    # return an Altair chart corresponding to network diagram above for the given season
    # G is the networkx object
    # season is the season
    # sel is the Altair selection object (for interactivity)
    circular_pos = nx.kamada_kawai_layout(G)

    e = nxa.draw_networkx_edges(G, pos=circular_pos)  # get the edge layer
    n = nxa.draw_networkx_nodes(G, pos=circular_pos)  # get the node layer
```

```
    # modify the code to change the encodings
    nearest = sel

    e = e.mark_point(filled=True, size=250, opacity=1).encode(
        color=alt.condition(nearest, alt.Color('community:N', legend=None), alt.value('lightgray'))
    )

    n = n.mark_circle(filled=True, size=250, opacity=1).encode(
        color=alt.condition(nearest, alt.Color('community:N', legend=None), alt.value('lightgray')),
        tooltip=['label','community','appearance']
    )

    l = e.mark_line(opacity=0.2).encode()
    # remove the exception to render
    # YOUR CODE HERE
    # raise NotImplementedError()
    c2 = (e.add_selection(nearest)+l+n.transform_filter(nearest).interactive()).properties(
        title="Season {}".format(season), width=480,height=400
    )

    # YOUR CODE HERE
    # raise NotImplementedError()
    return(c2)
```

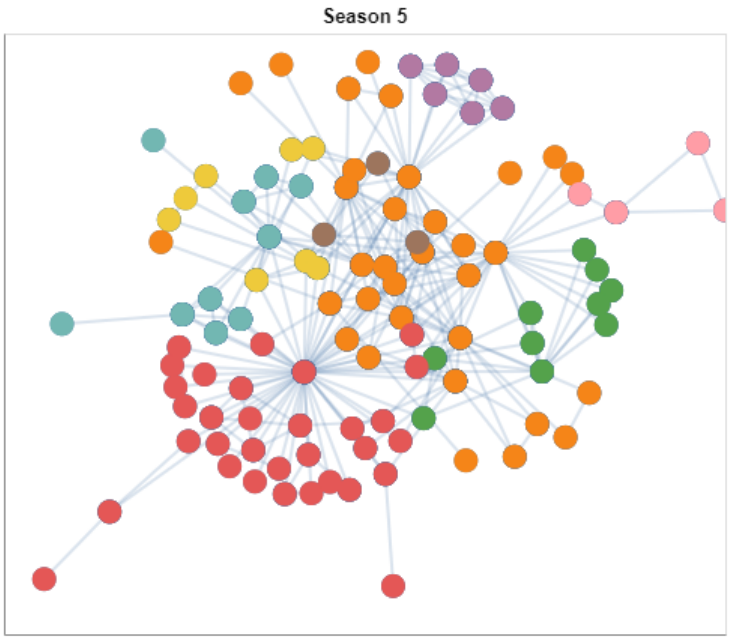
In [37]:

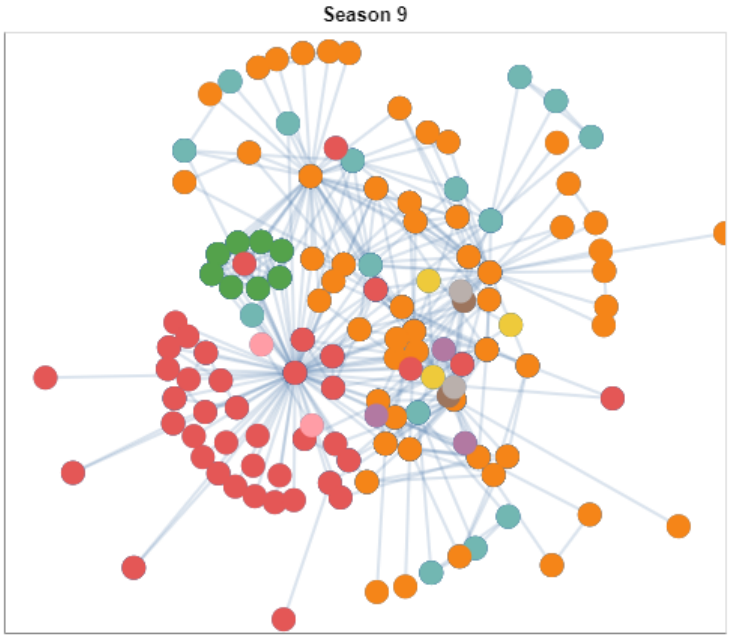
```
# this function will build the dashboard for you assuming you implemented the top two functions correctly
def getNetworkDashboard(season1,season2):

    # create the selection object, based on mouseover. It should look at the "label" of whatever we hover
    # over as a way of deciding other objects with the same label
    single = alt.selection_single(on='mouseover',fields=['label'])

    # get the two networkx objects
    s1net = getNetwork(season1)
    s2net = getNetwork(season2)

    # get the union and difference dataframes
    union,difference = getComparisonData(s1net,s2net)

    # build the top bar chart
    u = getComparisonBar(union,single,"Appears in Both")

    # build the top network
    s1 = getNetworkDiagram(s1net,season1,single)

    # build the bottom network
    s2 = getNetworkDiagram(s2net,season2,single)

    # in some cases, we don't have new characters given the thresold we set
    if (len(difference) == 0):
        # we won't return the bottom chart
        return((s1&s2)|u)
    else:
        # we have both bar charts
        # build the bottom bar chart
        d = getComparisonBar(difference,single,"New Characters")
        # return all charts
        return((s1&s2)|(u&d))
```

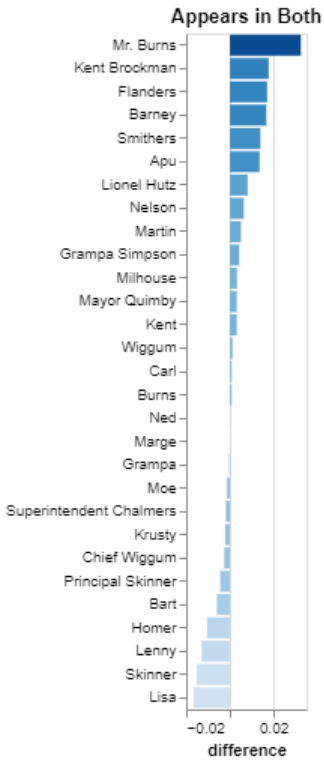
In [38]:

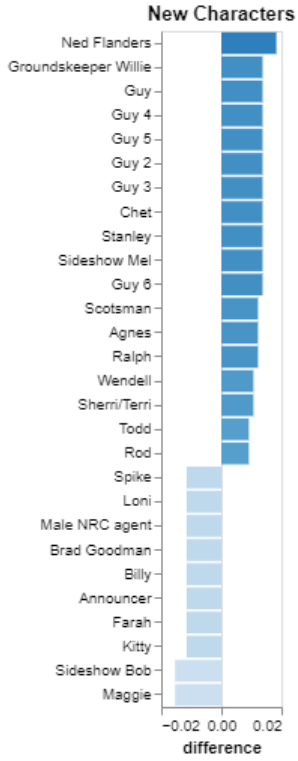
```
# if you implemented everything correctly, this should work
getNetworkDashboard(5,9)
```

Out[38]:





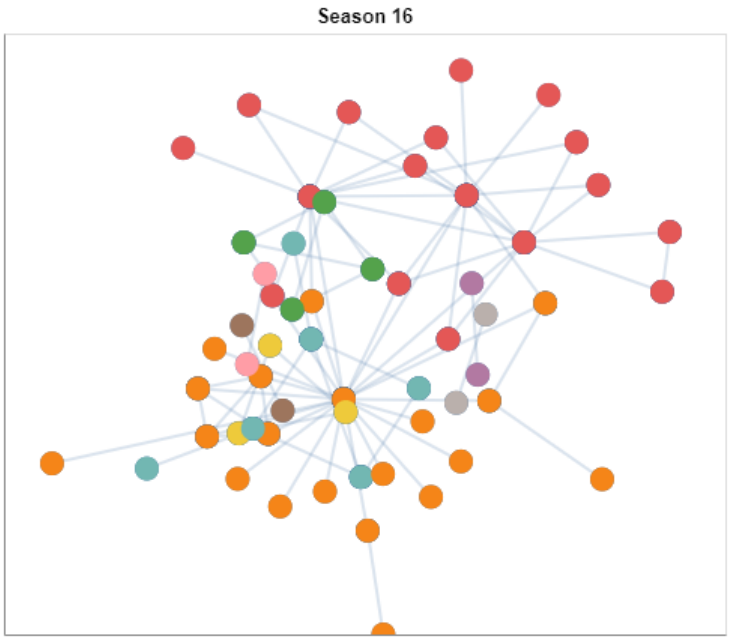


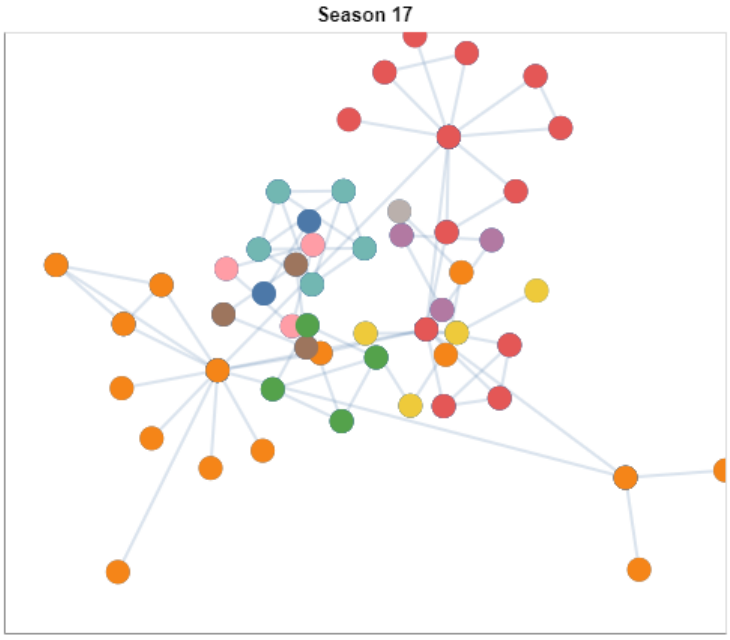


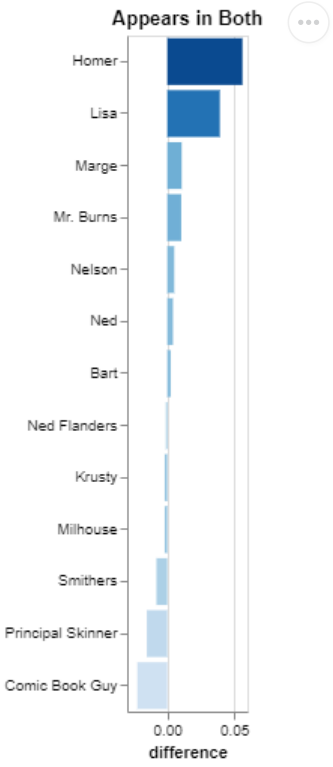
In [39]:

```
# if you implemented everything correctly, this should work
getNetworkDashboard(16,17)
```

Out[39]:

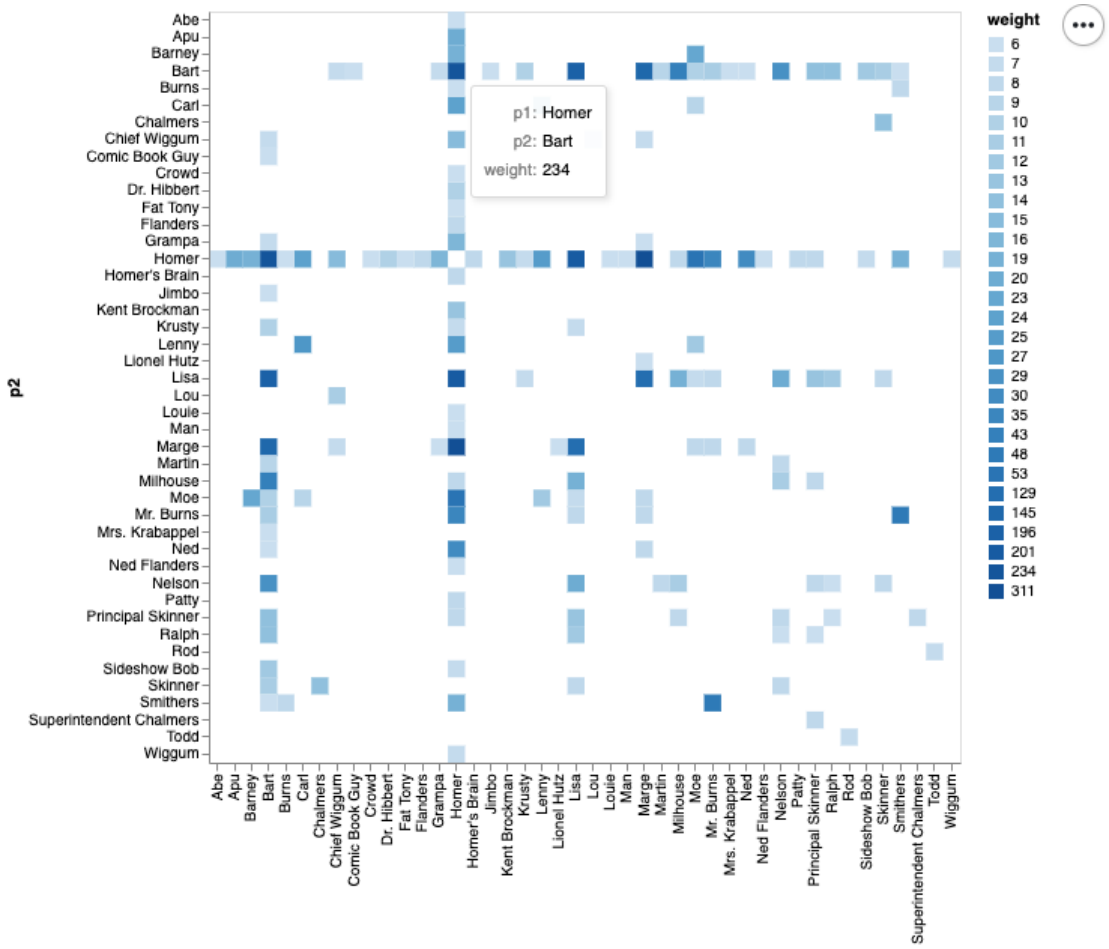






Problem 3 (20 Points)

For our last exercise, we're going to generate a matrix representation of the network:



Each cell will indicate the number of interactions over the given time frame. In the example above, these are all the characters who have interacted 6 or more times with each other. A tooltip gives us a bit more detail since there are so many rows/columns.

To generate this plot in Altair, we're going to reconfigure the data using the following function:

In [40]:

```
def getMatrixDetails(df,threshold=6,removeIsolates=True):
    # given a dataframe with characters (c1,c2, etc.)
    # the returned matrix will find the number of interactions in the dataframe
    # find statistics to generate a matrix representation
    # threshold will be the minimum number of interactions between characters (6)
    # removeIsolates determiens if isolated nodes (nodes not connected to anything) are removed

    # this function returns 3 things
    # the long form dataframe with pairs of nodes and the count
    # the node order of nodes in the matrix given the input
    # a list of list -- an edge list for all nodes
    t = buildNetwork(df)
    for e in t.edges:
        if (t.edges[e]['weight'] < threshold):
            t.remove_edge(e[0],e[1])
    if(removeIsolates):
        t.remove_nodes_from(list(nx.isolates(t)))

    m,names,a,b,w = [],[],[],[],[]

    for n1 in t.nodes:
        e = []
        names.append(n1)
        for n2 in t.nodes:
            if(t.has_edge(n1,n2)):
                a.append(n1)
                b.append(n2)
                w.append(t.edges[n1,n2]['weight'])
                e.append(t.edges[n1,n2]['weight'])
            else:
                e.append(0)
        m.append(e)

    toret = pd.DataFrame()
```

```
    toret['p1'] = a
    toret['p2'] = b
    toret['weight'] = w
    return(toret,names,m)
```

In [41]:
```
# let's call this for the entire dataset
df,names,m = getMatrixDetails(simpsons)
```

In [42]:
```
# we'll get back to names and m in a moment, but let's look at what's inside the df:
df.sample(5)
```

Out[42]:

	p1	p2	weight
120	Skinner	Nelson	8
100	Mr. Burns	Bart	11
97	Patty	Homer	8
95	Ned	Bart	6
69	Lisa	Skinner	8

In the frame above, you'll see `p1` and `p2` which are the two characters in the quoted conversation, and `weight` indicating the number of times they were in conversations.

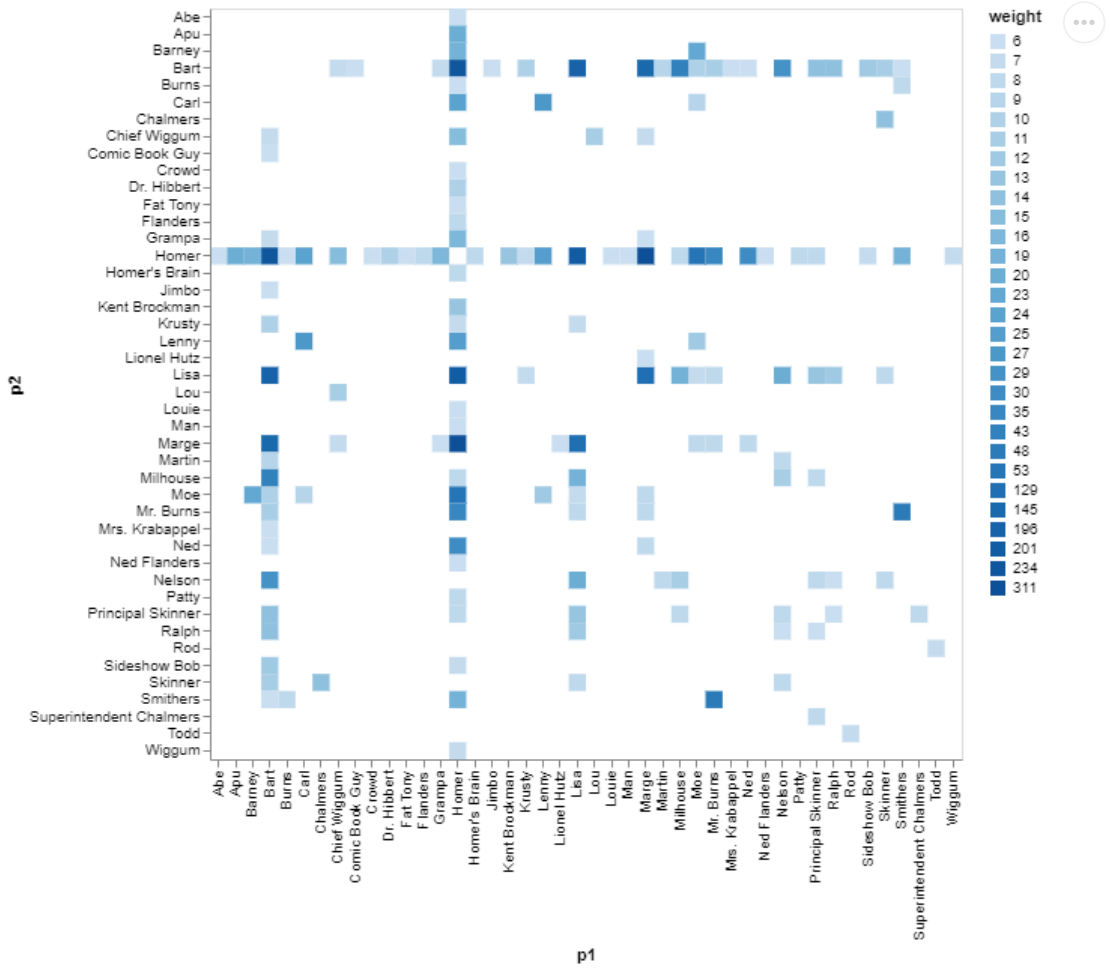
Problem 3.1

Using the dataframe above, generate the matrix representation by completing the `genMatrix1` function below. The input will be the part of the simpsons dataframe we are interested in. This should return an Altair plot as above.

In [43]:
```
def genMatrix1(inframe,threshold=6):
    # takes an input frame as input
    # returns an altair plot for the matrix as described above
    df,names,m = getMatrixDetails(inframe,threshold=threshold)

    # modify the following
    nearest = alt.selection(type='single', on='mouseover')

    toret = alt.Chart(df).mark_rect().encode(
        alt.X('p1:N'),
        alt.Y('p2:N'),
        alt.Color('weight:N', scale=alt.Scale(scheme='blues')),
        tooltip=['p1:N','p2:N','weight:N']
    ).properties(width=500, height=500)

    # YOUR CODE HERE
    # raise NotImplementedError()
    return(toret)
```

In [44]:
```
# If you implemented the above function correctly, this should work
genMatrix1(simpsons)
```

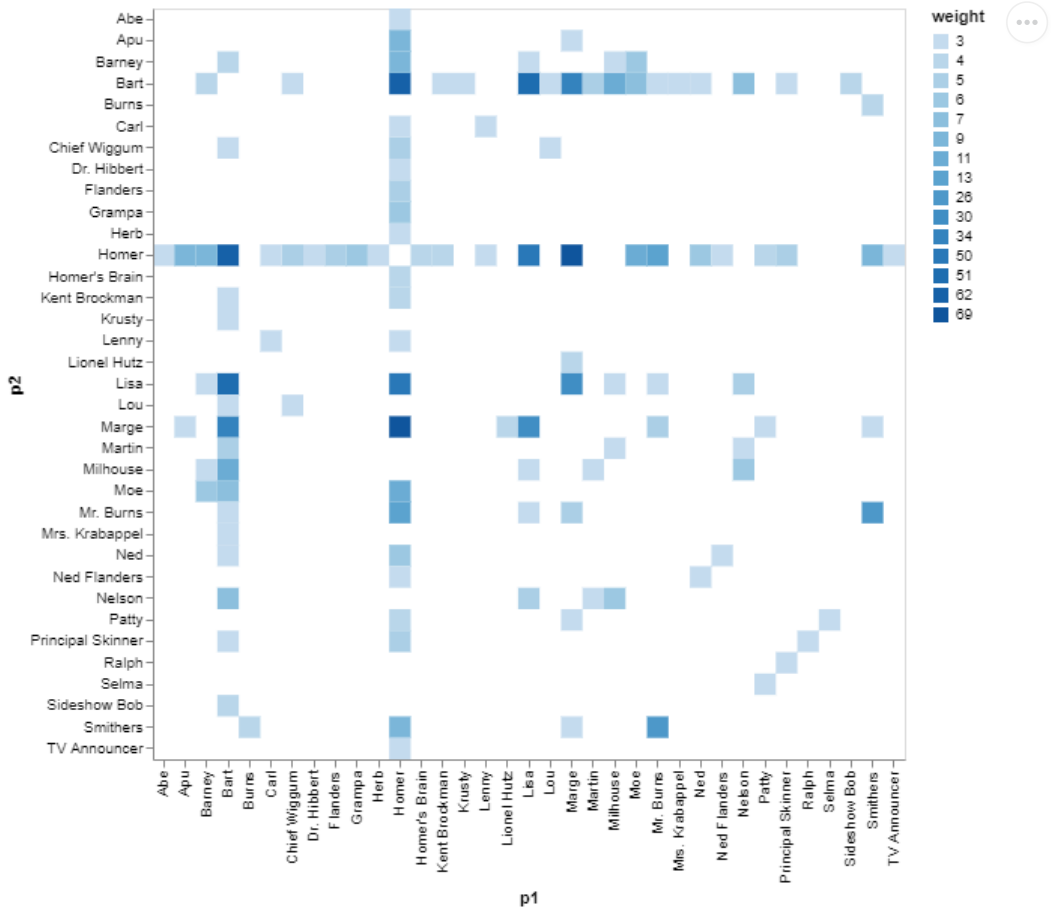
Out[44]:



In [45]:

```
# double check for hard coding by running this with the first 6 seasons and lower threshold
genMatrix1(simpsons[simpsons.season <= 6],threshold=3)
```

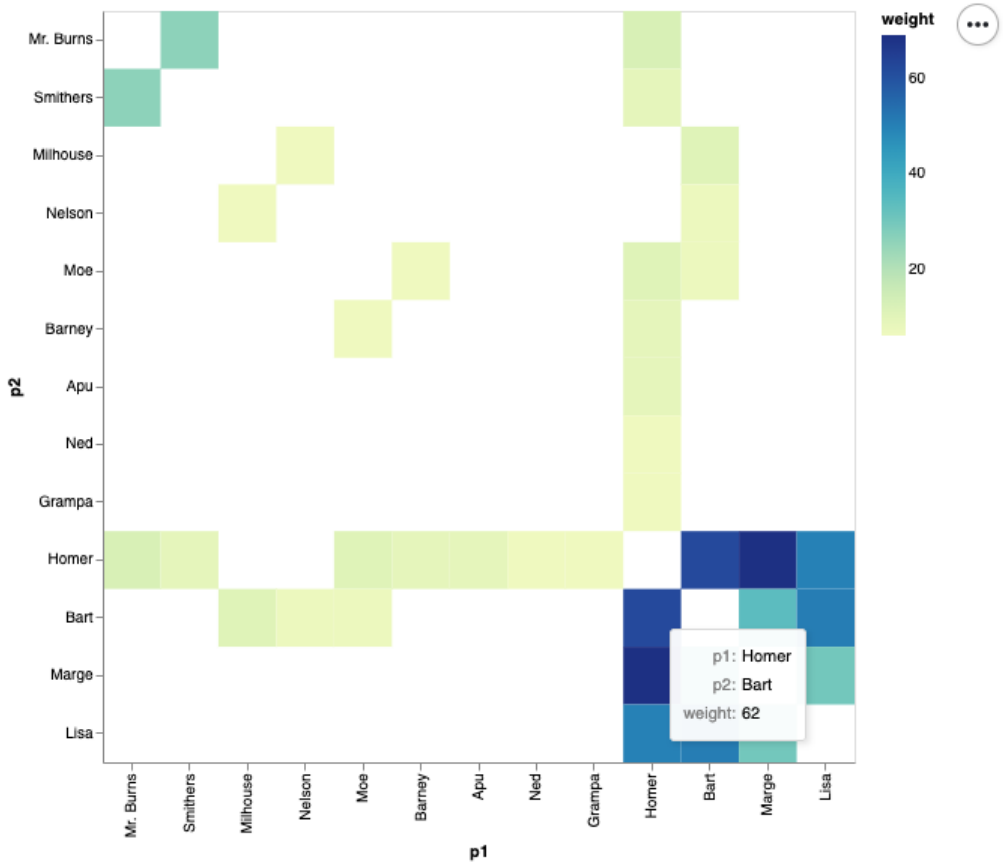
Out[45]:



Problem 3.2

The problem with the above forms is that the layout is rather arbitrary (alphabetical on character names). This makes it hard to see patterns in the data. One solution is to reorder the rows and columns so that those characters that are similar will end up close to each other. If you've ever used Seaborn's clustermap this is essentially what it does. It will run a hierarchical clustering algorithm to find related rows (or columns--in our case, these are the same), and shuffles them around to see patterns in the data.

We can do the same using Scipy's agglomerative clustering and modify the linkage analysis used to generate the dendrogram to find the order of the leaves. This will allow us to generate plots like this:



Notice the new order of the nodes and that we changed the color in this plot (weights are quantitative).

In [46]:

```
# a function to re-order using the agglomerative clustering and dendrogram layout

def getNewOrder(mtrx,originalorder):
    # determine the new order given an "edge list representation"
    # accepts the "original order" returns a new order
    model = AgglomerativeClustering(distance_threshold=0, n_clusters=None)
    model = model.fit(mtrx)
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx - n_samples]
        counts[i] = current_count

    linkage_matrix = np.column_stack([model.children_, model.distances_,
                                      counts]).astype(float)

    leaves = leaves_list(linkage_matrix)
    neworder = []
    for l in leaves:
        neworder.append(originalorder[l])
    return(neworder)
```

Optional detail

If you're curious, the scipy clustering code requires a "vector" representation of each node (which we calculated when we ran `getMatixDetail`). This looks much like the edge list representation. For example, Bart, may have a vector that looks like:

```
Bart -> [1,0,1,4]
```

This means that he is connected to the first and third nodes (whoever those are, it's not important) with a weight of 1, is not connected to node 2, is connected to node 4 with a weight of 4. Strictly speaking, the `m` that is returned by `getMatrixDetail` is a list of lists.

Each vector will be compared to all others giving us the "distance" between characters and that will be used to cluster.

So...

In [47]:
```
df,names,m = getMatrixDetails(simpsons[simpsons.season <= 6],threshold=6)

print("The first character in m is:",names[0])

print("It is represented by the vector:",m[0])
```

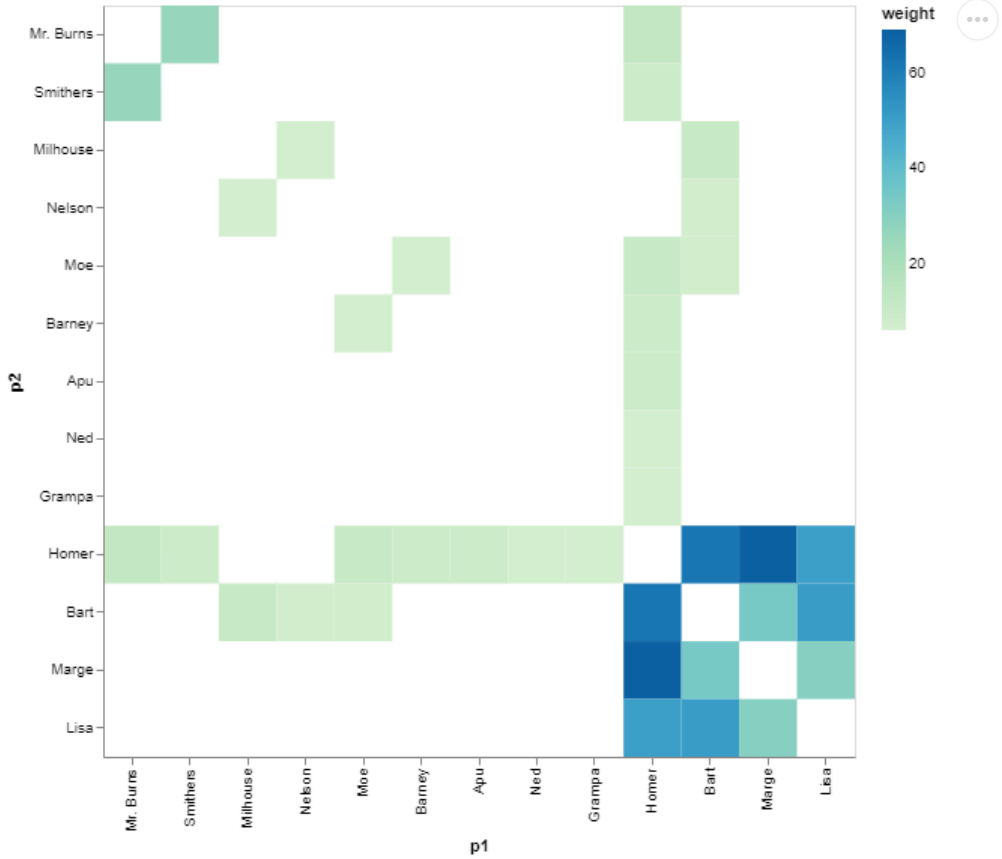
```
The first character in m is: Homer
It is represented by the vector: [0, 62, 11, 9, 9, 69, 6, 50, 13, 9, 0, 0, 6]
```

Your job is to modify `genMatrix1` to make a `genMatrix2` which takes advantage of this new order. We've started the code for you. Return an Altair chart that looks like the example above.

In [48]:
```
def genMatrix2(inframe,threshold=6):
    # takes an input frame as input
    # returns an altair plot for the matrix as described above
    df,names,m = getMatrixDetails(inframe,threshold=threshold)

    neworder = getNewOrder(m,names)

    # modify the following
    toret = alt.Chart(df).mark_rect().encode(
        alt.X('p1:N', sort=neworder),
        alt.Y('p2:N', sort=neworder),
        alt.Color('weight:Q', scale=alt.Scale(scheme='greenblue')),
        tooltip=['p1:N','p2:N','weight:Q']
    ).properties(width=500, height=500)  # alt.Chart...
    # YOUR CODE HERE
    # raise NotImplementedError()
    return(toret)
```

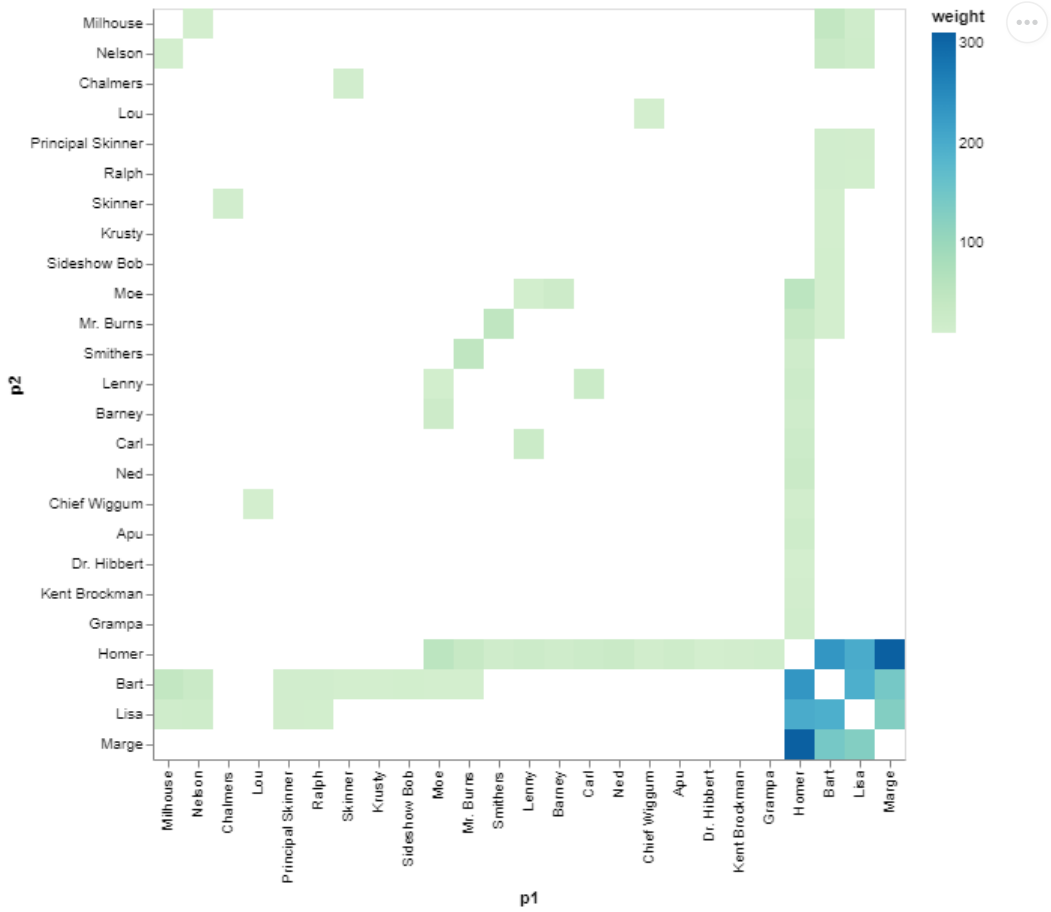
In [49]:
```
# If you im
genMatrix2(simpsons[simpsons.season <= 6])
```

Out[49]:



In [50]:
```
# test your code on the entire series to make sure nothing is hard coded
genMatrix2(simpsons,threshold=10)
```

Out[50]:



Problem 3.3

When we plot the network as a matrix (with the re-ordered rows/columns). You'll see some interesting patterns develop. Describe what these patterns mean. You can add an annotated screenshot with examples if you need to clarify.

A network can be represented by an adjacency matrix, where each cell *ij* represents an edge from vertex *i* to vertex *j*. Here, vertices represent characters in the Simpsons, while edges represent co-occurrence(weight) in a season.

- Pattern 1 - From this pattern, it is a little bit hard to interpret by human beings since the matrix diagram is heaviy dependent on the order of rows and columns. I would like to create a filter by frequency, names, or cluster which is easier to identify cluster and bridges.
- Pattern 2 - Compare with pattern 1, this pattern is easier to classify the co-occurrence among characters. At the right bottom corner, these four characters have strong relationship(weight) with others. Homer has relationship with almost all other characters.