In [0]:

In [115…

```
import matplotlib.pyplot as plt
import numpy as np
import random
from scipy.spatial import Voronoi, voronoi_plot_2d
from matplotlib import style
from sklearn.cluster import KMeans
from sage.graphs.graph_coloring import all_graph_colorings
from sage.graphs.graph_coloring import number_of_n_colorings
from sage.graphs.path_enumeration import feng_k_shortest_simple_paths
style.use("ggplot")
```

Continuing on with Graph Theory

We left off from Graph Theory with the February 5th and 7th lectures. We begin this presentation at looking at a more in-depth look at graph theory

DFS and BFS

Depth-First-Search and Breadth-First-Search are common searching algorithms that involve iterating through a structure in hopes of finding some target value

Most commonly implemented in trees, they vary in implementation when it comes to graphs

According to Niema Moshiri and Liz Izhikevich's course https://stepik.org/course/579

DFS is implemented as:

> - Start at s. It has distance 0 from itself.
> - Consider a node adjacent to s. Call it t. It has distance 1. Mark it as visited.
> - Then consider a node adjacent to t that has not yet been visited. It has distance 2. Mark it as visited.
> - Repeat until all nodes reachable from s are visited.

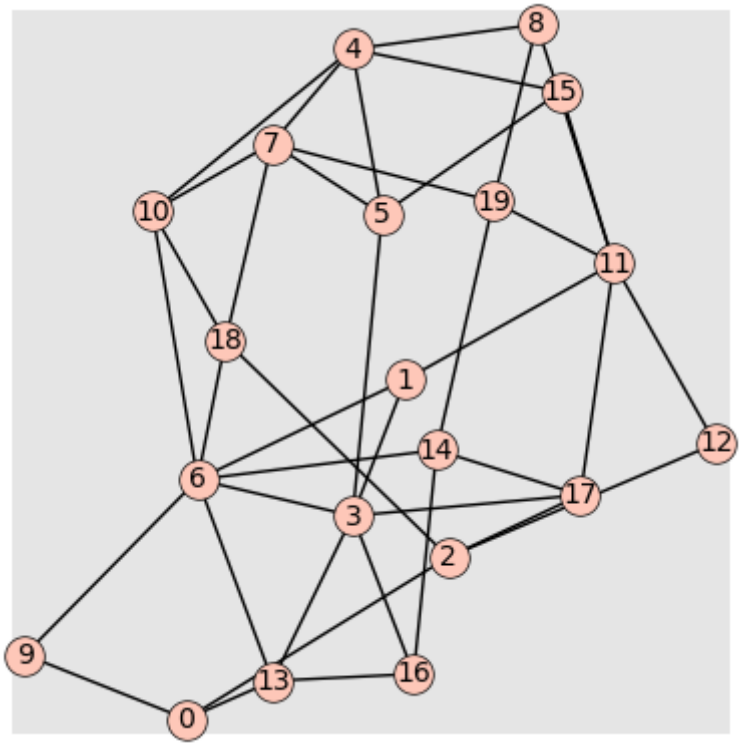
BFS is implemented as:

> - Begin at the starting node s. It has distance 0 from itself.
> - Consider nodes adjacent to s. They have distance 1. Mark them as visited.
> - Then consider nodes that have not yet been visited that are adjacent to those at distance 1. They have distance 2. Mark them as visited.
> - Repeat until all nodes reachable from s are visited.

You can immediately see the stark differences between graph searches and tree searches as tree will usually start from the root and only look at its children and parent, while graphs will look at its adjacent neighbors and so forth

In [110…
```
S = graphs.RandomGNP(20, 0.2)
S.show()
```

Out[110…



In [112…
```
print(S.lex_DFS())
print(S.lex_BFS())
```

```
[0, 2, 12, 11, 17, 3, 1, 6, 13, 16, 14, 19, 8, 4, 7, 10, 18, 5, 15, 9]
[0, 2, 9, 13, 12, 17, 18, 6, 3, 16, 11, 14, 10, 7, 1, 5, 19, 15, 8, 4]
```

4-Color Theorem

Last time we talked about how colorings work in Graph Theory in lecture 2/7

> The chromatic number of a graph has to do with vertex coloring of a graph. A vertex coloring of a graph G is a function from the vertices of G to some target set (the colors) with the property that no two adjacent vertices have the same color. The chromatic number is the minimum number of colors required to achieve a vertex coloring.

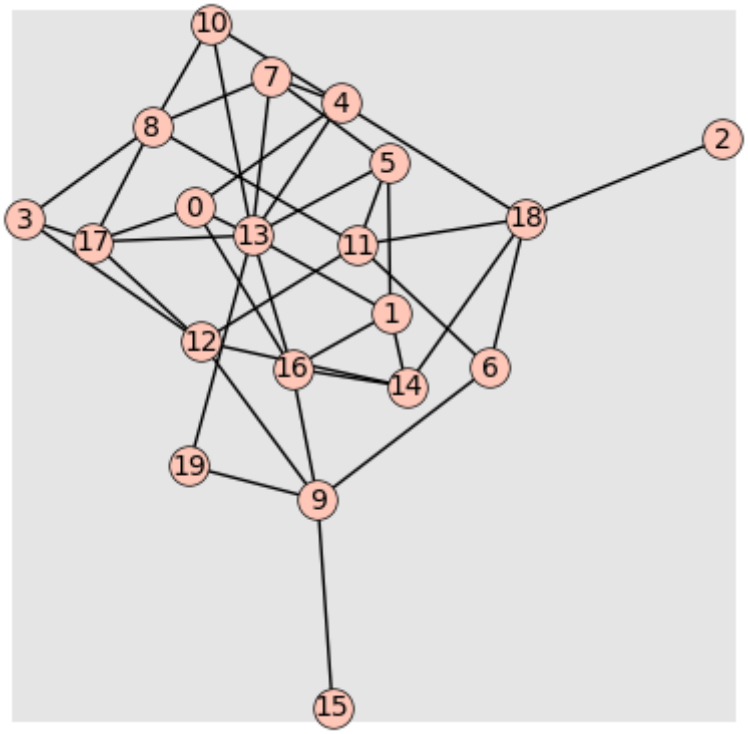
We now delve into it deeper, looking at the 4-color theorem. According to Wikipedia,

> In mathematics, the four color theorem, or the four color map theorem, states that, given any separation of a plane into contiguous regions, producing a figure called a map, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color. Adjacent means that two regions share a common boundary curve segment, not merely a corner where three or more regions meet.[1] It was the first major theorem to be proved using a computer. Initially, this proof was not accepted by all mathematicians because the computer-assisted proof

> was infeasible for a human to check by hand.[2] Since then the proof has gained wide acceptance, although some doubters remain.[3]

In [54]:
```
G = graphs.RandomGNP(20, 0.2)
G.show()
```

Out[54]:



In [108…
```
number_of_n_colorings(G, 4) # Plenty of 4-colorings
```

Out[108… 2241864

In [83]:
```
number_of_n_colorings(G, 5) # Impossible, will run infinitely
```

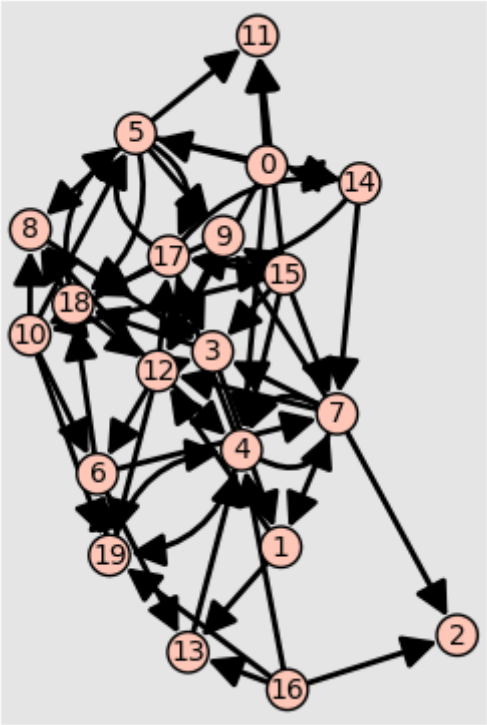
Pathfinding Algorithms

One of the most interesting applications of Graph Theory is figuring out how to find an optimal path given some large graph. Whether the graph is connected or not, these types of applicatons are most interesting when the Graph is weighted. As we have seen in earlier lectures, we are able to find different types of paths including compact paths from our homework.

Lets take a look at how we can use pathfinding algorithms to find optimal paths

In [94]:
```
D = DiGraph(digraphs.RandomDirectedGNP(20,.2),weighted=True)
D.show()
```

Out[94]:



In [102…
```
list(feng_k_shortest_simple_paths(D, 5,11))[0:10]
```

Out[102…
```
[[5, 11],
 [5, 14, 0, 11],
 [5, 17, 14, 0, 11],
 [5, 18, 12, 17, 14, 0, 11],
 [5, 9, 12, 17, 14, 0, 11],
 [5, 9, 12, 4, 17, 14, 0, 11],
 [5, 9, 18, 12, 17, 14, 0, 11],
 [5, 18, 12, 4, 17, 14, 0, 11],
 [5, 18, 12, 19, 4, 17, 14, 0, 11],
 [5, 9, 18, 12, 4, 17, 14, 0, 11]]
```

Dijkstra's Algorithm

Dijkstra's Algorithm is a simple pathfinding algorithm that seeks to use a data structure in storing all weighted edges; based on the weights of those edge, the edge will be restored onto a new graph until a path between u and v is made and a clear path is made

According to https://dev.to/mxl/dijkstras-algorithm-in-python-algorithms-for-beginners-dkc, the pseudocode is as follows:

> - Mark all nodes unvisited and store them.
> - Set the distance to zero for our initial node and to infinity for other nodes.
> - Select the unvisited node with the smallest distance, it's current node now.
> - Find unvisited neighbors for the current node and calculate their distances through the current node. Compare the newly calculated distance to the assigned and save the smaller one. For example, if the node A has a distance of 6, and the A-B edge has length 2, then the distance to B through A will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8.

> - Mark the current node as visited and remove it from the unvisited set.
> - Stop, if the destination node has been visited (when planning a route between two specific nodes) or if the smallest distance among the unvisited nodes is infinity. If not, repeat steps 3-6.

In [106…

```
# In code, from https://dev.to/mxl/dijkstras-algorithm-in-python-algorithms-for-
def dijkstra(self, source, dest):
        assert source in self.vertices, 'Such source node doesn\'t exist'

        # 1. Mark all nodes unvisited and store them.
        # 2. Set the distance to zero for our initial node
        # and to infinity for other nodes.
        distances = {vertex: inf for vertex in self.vertices}
        previous_vertices = {
            vertex: None for vertex in self.vertices
        }
        distances[source] = 0
        vertices = self.vertices.copy()

        while vertices:
            # 3. Select the unvisited node with the smallest distance,
            # it's current node now.
            current_vertex = min(
                vertices, key=lambda vertex: distances[vertex])

            # 6. Stop, if the smallest distance
            # among the unvisited nodes is infinity.
            if distances[current_vertex] == inf:
                break

            # 4. Find unvisited neighbors for the current node
            # and calculate their distances through the current node.
            for neighbour, cost in self.neighbours[current_vertex]:
                alternative_route = distances[current_vertex] + cost

                # Compare the newly calculated distance to the assigned
                # and save the smaller one.
                if alternative_route < distances[neighbour]:
                    distances[neighbour] = alternative_route
                    previous_vertices[neighbour] = current_vertex

            # 5. Mark the current node as visited
            # and remove it from the unvisited set.
            vertices.remove(current_vertex)


        path, current_vertex = deque(), dest
        while previous_vertices[current_vertex] is not None:
            path.appendleft(current_vertex)
            current_vertex = previous_vertices[current_vertex]
        if path:
            path.appendleft(current_vertex)
        return path
```

Other pathfinding algorithms include A *and D*, which are essentially variants of Dijkstra's but involve

using a heuristic to aid with the algorithm Can be read here: \
https://en.wikipedia.org/wiki/A*_search_algorithm and https://en.wikipedia.org/wiki/D*

Application of Graph Theory: Congressional Redistricting

The Census

With the 2020 Census coming on April 1st, we should expect the United States to undergo political change, including determining the number of congressional districts within each of the 50 states; thereby also delegating the number of seats for each state in the House of Representatives.

Some states are expected to lose seats, while other are expected to gain seats. For example, big states like New York and California might lose a seat or two, while growing states such as Texas and Florida are expected to gain a few more seats

As a result of the census, which takes place every 10 years, state officials will begin redrawing the boundaries of their congressional districts to accomodate the new changes in population, also known as the redistricting process.

Redistricting Process

From an outside perspective, it is interesting to see how these officials use the census data to redraw the lines of the congressional districts.

Despite there being many constraints in congressional redistricting, many of these officials undergo gerrymandering, which is the practice of redrawing boundaries with the purpose of giving a political party an advantage come election time. As a result, many political parties will prepare heavily for these once-in-a-decade census', whether it be influencing the committees responsible for the redistricting process, mocking up scenarios see what types of district layouts will give their party a political advantage

How we can apply Graph Theory to this topic

Because of gerrymandering, it is difficult for redistricting committees to abstain from political biases, which makes it difficult to come to bipartisan agreements when checking which masses of land these parties would want to be redrawn. For example, the Democratic Party might want the areas of La Jolla, Del Mar, and Oceanside to be redrawn to their favor, but the Republican party might say otherwise and want some of that land as well.

As it all comes down to demographic information, the most practical way to ensure equal distribution of these districts that provide fairness between all political parties, as well as making each district relatively compact, is to consider a computational approach of redistricting these lands.

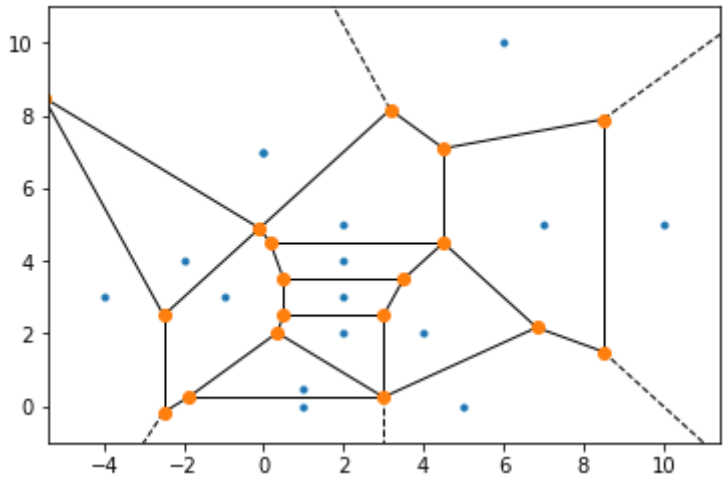
Mathematical Approaches to redistricting apart from Graph Theory: Geometric Approach

One of the ways that mathematicians have been able to characterize congressional districts is through the use of convex polygons. Because these types of polygons follow under the "compact" characteristic of a congressional district, mathematicians look at these adjacent polygons and see

how their 'generating points' compare with their neigboring polygons'. This can be shown mathematically through a Voronoi diagram, which partitions a plane into convex polygons and highlight their generating points

In [17]:
```
points = np.array([[-4,3],[-2,4],[-1,3],[0,7],[1,.5],[1,0],[2,3],[2,4],
                   [0,7],[2,2],[4,2],[5,0],[2,5],[7,5],[6,10],[10,5]])
vor = Voronoi(points)
voronoi_plot_2d(vor)
plt.show()
```

Out[17]:



Voronoi Diagrams help us how understand the redistricting process a little bit more due to how they resemble land masses as a whole; we can see that based on the demographics of an area, the generating points, which represent cities or concentrated populations, generate appropriate-sized polygons. These polygons could then translate to a congressional district based on the generating point's population mass.

To exemplify this even furhter, a team competing in the 2007 COMAP mathematical modeling competition were able to approach redistricting the state of New York using such Voronoi diagrams. They did so by deciding which cities were most densely populated and giving those cities the most generating points. Cities such as New York City received the most, while smaller cities such as Buffalo received less.

Nevertheless, this approach to redistricting isn't as optimal, as giving generating points to any particular city might give a party an unfair advantage compared to the surrounding regions.

Lloyd's Algorithm - K-Means Clustering

When choosing to optimize Voronoi Diagrams for use in redistricting, we turn to K-Means Clustering, a tool commonly used in data analysis to form "clusters" between some set of observations in hopes of finding some inferences from the data set. In our particular case, we can use Clustering to form the polygons of our Voronoi Diagram, and then use Lloyd's Algorithm to compute the centroid of the polygon, representing the concentrated population masses of the land mass. When doing this iteratively, we can move each of the polygon's generating point to an approximate central location of

each polygon. Ultimately, we are able to find a proper configuration where we can appropriately analyze what each distric should look like.

Below is a snippet of how Lloyd's Algorith would be implemented:

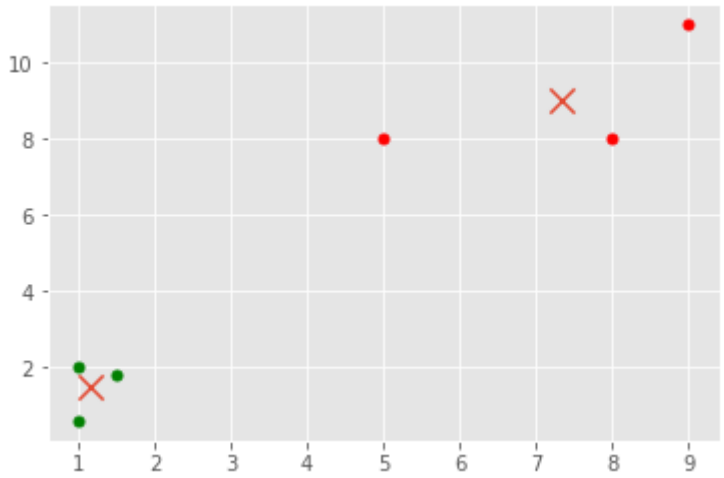
In [42]:
```
# Referenced from https://datasciencelab.wordpress.com/2013/12/12/clustering-wit

# Forms the cluster/polygons
def cluster_points(X, mu):
    clusters  = {}
    for x in X:
        bestmukey = min([(i[0], np.linalg.norm(x-mu[i[0]])) \
                    for i in enumerate(mu)], key=lambda t:t[1])[0]
        try:
            clusters[bestmukey].append(x)
        except KeyError:
            clusters[bestmukey] = [x]
    return clusters

# Checks if centroids are appropriate for the cluster
def reevaluate_centers(mu, clusters):
    newmu = []
    keys = sorted(clusters.keys())
    for k in keys:
        newmu.append(np.mean(clusters[k], axis = 0))
    return newmu

# Check if center of clusters/polygons have converged
def has_converged(mu, oldmu):
    return set([tuple(a) for a in mu]) == set([tuple(a) for a in oldmu])

# Finds the centroid of a cluster/polygon; would be called iteratively until we
def find_centers(X, K):
    # Initialize to K random centers
    oldmu = random.sample(X, K)
    mu = random.sample(X, K)
    while not has_converged(mu, oldmu):
        oldmu = mu
        # Assign all points in X to clusters
        clusters = cluster_points(X, mu)
        # Reevaluate centers
        mu = reevaluate_centers(oldmu, clusters)
    return(mu, clusters)
```

In [43]:
```
# Referenced from https://pythonprogramming.net/flat-clustering-machine-learning
X = np.array([[1, 2],
              [5, 8],
              [1.5, 1.8],
              [8, 8],
              [1, 0.6],
              [9, 11]])
kmeans = KMeans(n_clusters=2)
kmeans.fit(X)

centroids = kmeans.cluster_centers_
```

```
labels = kmeans.labels_


colors = ["g.","r.","c.","y."]

for i in range(len(X)):
    print("coordinate:",X[i], "label:", labels[i])
    plt.plot(X[i][0], X[i][1], colors[labels[i]], markersize = 10)


plt.scatter(centroids[:, 0],centroids[:, 1], marker = "x", s=150, linewidths = 5

plt.show()
```

```
coordinate: [1. 2.] label: 0
coordinate: [5. 8.] label: 1
coordinate: [1.5 1.8] label: 0
coordinate: [8. 8.] label: 1
coordinate: [1.  0.6] label: 0
coordinate: [ 9. 11.] label: 1
```

Out[43]:



Nevertheless, this approach would only work if the populations were uniformly distributed. In the real world, populations differ immensely and we won't be able to get a good enough approximation of these polygons and generating points to get the configuration we want. This is because if we decide place our initial generating points in areas that have uniform distribution but still have a relatively low population, then we would inaccurately create a polygon unrepresentative of that region. That is why we must choose which points to place our generating points in order to avoid bias.

As a whole, we see that although a geometric approach to redistricting might be beneficial for easily computing regions of congressional districts, there still remains a lot of biases when determining the origin of these districts since cities are very hard to represent geometrically via generating points. In that case, we go a more population-centric approach via graph partitioning.

Partitioning Method

This is where we get into the really knitty-gritty behind Graph Theory

In solving the redistricting problem, we now pay less attention on demographics and geography of a region and look at states at a more narrow perspective. The US Census Bureau discretizes states

into block groups; we shall use these block groups to form a graph, where each vertex is a block group and each edge is a connection between an adjacent block group. All vertices have some weight to them, which represent the demographic information of that particular block group. Now, we choose to partition our Graph into some k district with APPROXIMATELY EQUAL WEIGHT. We do this via minimal edge cut. Recall that an minimal edge cut is the smallest set of edges needed to disconnect a graph. In our case, we want the set of edges cut such that our graph is equally partitioned into equally-weighted districts.

From there we will be able to accurately form evenly-distributed districts. However, a problem with this approach is it is NP-Complete, meaning we need to obtain a heuristic to implement this approach. This heuristic will be represented in the form of multilevel algorithm, consisting of 3 phases

Coarsening

Coarsening involves cutting edges of the graph such that the graph still retains its basic structure. This is meant to help the algorithm run computationally smoothly as other tasks can be done while the coarsening is taking place. In essence, the number of vertices of the graph reduce by a power of 2.

Coarsening is done through a maximal matching, which is the maximal set of edges where no edge pair share a common vertex; in other words, all the possible matching within some graph. We are able to find a maximal matching by using a greedy algorithm that builds an unordered list of edges via coarsening coefficients

Coarsening coefficients are denoted by

$$\frac{}{}$$

where u<->v represent the set of edges connecting u and v

Partitioning

The graph is then partitioned once we've eliminated enough edges such that the Graph is split into k separate components, which then would represent the congressional districts. We are able to partition our graph by looking at our table of edges alongside their coarsening coefficients; we run down the table and start adding edges to our matching. If an edge sharing a vertex that has already been added to our graph, we ignore it and move along. We do this until we get our maximal matching, hence giving us our partitioned graph.

Uncoarsening

Lastly, uncoarsening the graph means comparing the newly formed graph with the original graph and seeing which vertices can be replaced to find a better partitioning, essentially leading to evenly-weighted components. We do this in concurrence with making sure that the edge cut is made minimally, so we get the most opitimized graph to analyze.

Examples

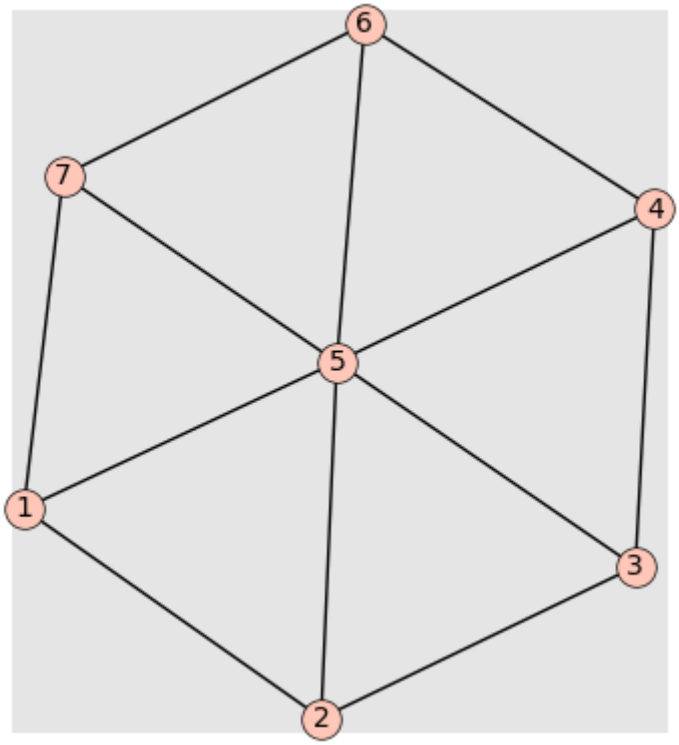
Let's practice this partitioning method by example

Example 1: Coarsening by finding a maximal matching

In [117…

```
# Suppose we have a graph G, which consists of 7 districts with a sum population
# Each vertex represents a district, and the vertex's label is its population
G = Graph([(1,7),(1,2),(1,5),(2,3),(2,5),(3,4),(3,5),(4,5),(4,6),(7,6),(7,5),(6,
G.plot()
```

Out[117…



In [173…

```
# Our goal is to partition into Districts D1 and D2

# Write your functions here

# First create a list that contains each of G's edges, ordered by INCREASING coa
# Remember that each coarsening coefficient is measured by C(uv) = w(u)w(v) ie C
# Remember to include the largest
v = []
def listOrderedByIncreasingCC(G,v):
    pass

# After creating the list, iterate through it and add edges to the matching
# Remember that a matching must contain each vertex at most once
m = []
def addToMatching(v,m):
    pass


# Choose any random initial partition so that we can uncoarsen it in the next ex
# For now lets choose
```

```
District1 = [m[1],m[3]]
District2 = [m[0],m[2]]
```

Example 2: Dijkstra's Algorithm

Given that Sage has no built in Dijkstra's function, can you implement Dijkstra's by taking in the graph from Example 1 and finding the shortest path from vertex 5 to 7? The edges weight will be computed by its connecting vertices coarsening coefficients, meaning you can use the list formed by listOrderedByIncreasingCC() from example 1 to your benefit

In [177…

```
# Write your function here
def Dijkstra(G, v1, v2):
    pass
```

References:

- https://datasciencelab.wordpress.com/2013/12/12/clustering-with-k-means-in-python/
- https://scholar.rose-hulman.edu/cgi/viewcontent.cgi?article=1031&context=rhumj
- https://pythonprogramming.net/flat-clustering-machine-learning-python-scikit-learn/
- https://en.wikipedia.org/wiki/Four_color_theorem
- https://stepik.org/course/579/syllabus
- https://dev.to/mxl/dijkstras-algorithm-in-python-algorithms-for-beginners-dkc
- http://doc.sagemath.org/html/en/reference/graphs/index.html
- http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/traversals.html
- http://doc.sagemath.org/html/en/reference/graphs/sage/graphs/path_enumeration.html
- 2/5 and 2/7 lectures