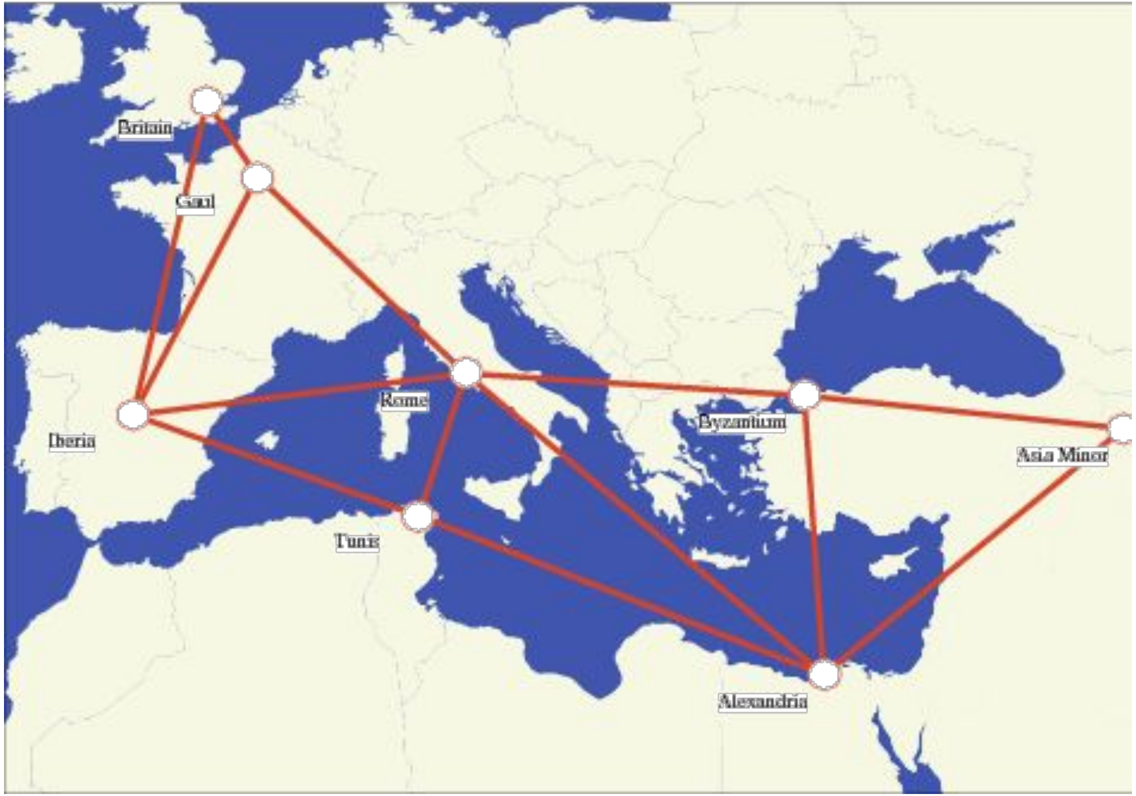
Roman Domination Project

Jim Toledo

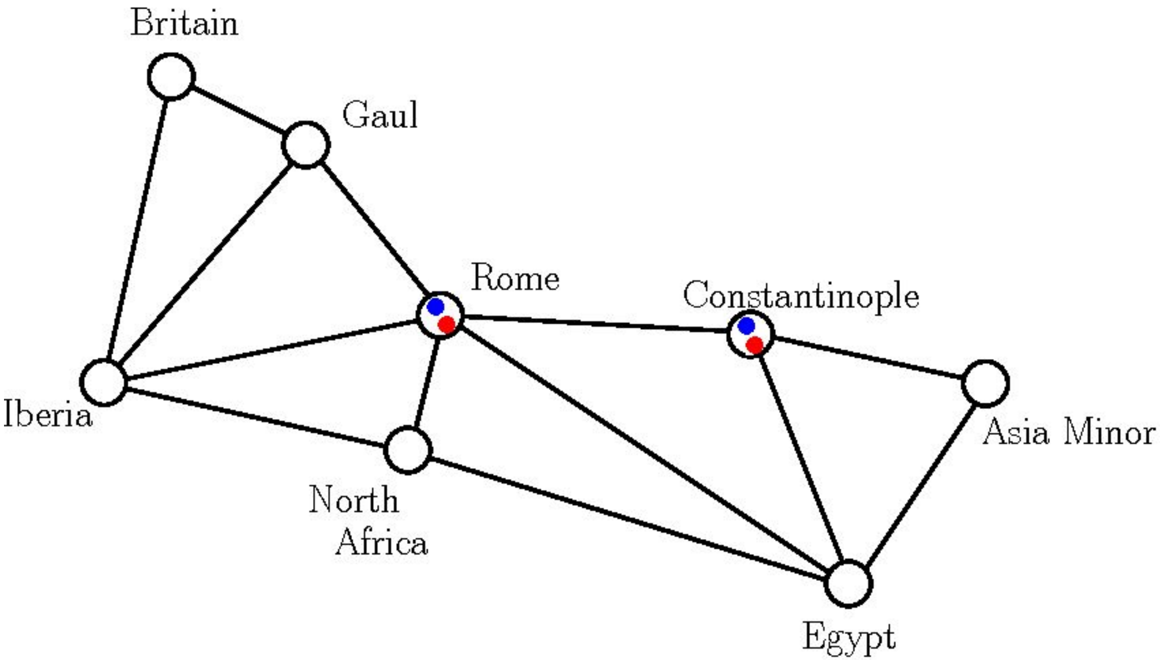
Defend the Roman Empire!

- Article by Ian Stewart in *Scientific American*
- Describes the beginning of the end of the Roman Empire at around 400 AD
- Emperor Constantine had to protect eight regions with four legions
- Constantine devised this strategy:
    - A region is secured if it contains at least one legion
    - A region is unsecured if no legions are stationed there
    - An unsecured region can be secured by sending a legion from an adjacent region
    - However, if that adjacent region only contains one legion then that region will become unsecured

Defend the Roman Empire!



Defend the Roman Empire!



Roman Domination

- Given a graph $G = (V, E)$
  - Roman dominating function = $f : V \rightarrow \{0, 1, 2\}$
  - Every vertex mapped to 0 is adjacent to a vertex mapped to 2
- Weight of a Roman dominating function is simply the accumulation of $f(v)$ over all vertices v of the vertex set of G
- Minimum Roman dominating function of G is the RDF with the smallest weight
  - That weight is the Roman domination number, denoted $\gamma_R(G)$
- Many applications
  - Deploying troops
  - Placement of hospitals, police stations, restaurants, etc.

My Goal

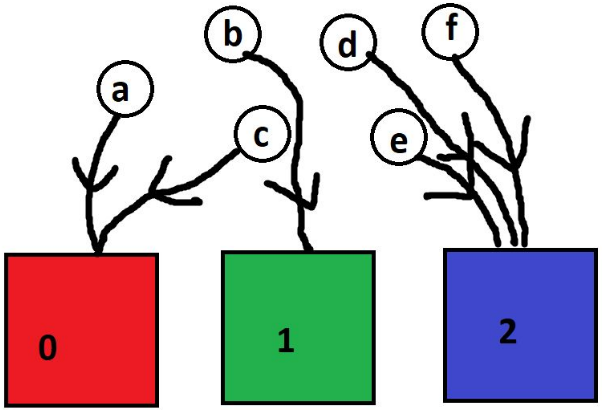
- Create a program that finds both $\gamma_R(G)$ and the minimum RD function
- Have the minimum RD coloring displayed graphically
- Work for any given (simple) graph
- Make it as efficient as possible!

First Step: Verifying if a Function is an RD

```python
def isRomanDomination(G, zeros, ones, twos): #check if graph function f : V(G) -> {0, 1, 2} is a Roman dominating function
    for zero_vertex in zeros:
        connected_to_two = False
        for two_vertex in twos:
            if G[zero_vertex][two_vertex] == 1: #pretty much just check that all zero vertices are connected to a two vertex
                connected_to_two = True
                break
        if not connected_to_two:
            return False
    return True
```

Generating the Functions

- The function $f : V \rightarrow \{0, 1, 2\}$ can be thought of as $f = (V_0, V_1, V_2)$ where V_0, V_1, and V_2 are sets that contain vertices $v \in V$ if $f(v) = 0, 1,$ and 2, respectively
- In other words, divide the vertex set into 3 bins labelled 0, 1, and 2



Generating the Functions

- Turns out, there's a **lot** of different ways to color the vertices
- Still, we can rule out a few functions
  - Impossible for a graph to contain 0-vertices without any 2-vertices
  - Clear upper bound for the Roman domination number is *n* (number of nodes), achieved if all nodes were colored '1'
  - From the previous statement, the minimum Roman domination function cannot contain > *n*/2 vertices colored '2'
- Also we can categorize the functions by the sizes of the respective bins and the sum of the colored vertices
- Test the bin size combinations from lowest sum to highest sum to find the minimum quicker

Generating the Functions: Restrictions

- Ruling out functions and providing a more optimized ordering of what to check

```
def getCombos(n): #all possible sizes of coloring sets given a graph of order n, in order from smallest to largest sum
    combos = []
    for num_twos in range(math.ceil(n/2)): #minimum RD can never contain any more than n/2 2's
        for num_ones in range(n - num_twos + 1):
            num_zeros = n - num_twos - num_ones
            if num_zeros == 0 or num_twos > 0: #impossible to have 0's and no 2's in the coloring
                combos.append((num_ones + 2 * num_twos, num_zeros, num_ones, num_twos)) #tuple where first element is the sum of vertices after coloring
    combos.sort() #test from lowest number to highest
    return combos
```

Generating the Functions: Partitioning

```
def partition(G, V, num_zeros, num_ones, num_twos, zero_list, one_list, two_list): #generate and test all partitions of the vertex set V given #s needed in each set
    if len(V) > 0:
        vertex = V[0]
        #check all ways in which the vertex is in bin 0, bin 1, and bin 2
        if num_zeros > 0:
            zero_list.append(vertex) #color vertex 0 (red)
            copyV = V.copy()
            copyV.remove(vertex) #remove the vertex for recursive call so that it cannot be recolored
            part = partition(G, copyV, num_zeros - 1, num_ones, num_twos, zero_list, one_list, two_list)
            if part[0]:
                return (True, part[1], part[2], part[3]) #exit out of loop early when RD is found
            zero_list.remove(vertex) #if all combos where the vertex was colored '0' fails, remove coloring and try another
        if num_ones > 0:
            one_list.append(vertex)
            copyV = V.copy()
            copyV.remove(vertex)
            part = partition(G, copyV, num_zeros, num_ones - 1, num_twos, zero_list, one_list, two_list)
            if part[0]:
                return (True, part[1], part[2], part[3]) #exit out of loop early when RD is found
            one_list.remove(vertex)

        if num_twos > 0:
            two_list.append(vertex)
            copyV = V.copy()
            copyV.remove(vertex)
            part = partition(G, copyV, num_zeros, num_ones, num_twos - 1, zero_list, one_list, two_list)
            if part[0]:
                return (True, part[1], part[2], part[3]) #exit out of loop early when RD is found
            two_list.remove(vertex)
        return (False, [], [], []) #return false if no partition works
    else: #if no more vertices to color
        if isRomanDomination(G, zero_list, one_list, two_list):
            return (True, zero_list.copy(), one_list.copy(), two_list.copy()) #return the colorings and also the fact that the colorings produce an RD
        else:
            return (False, [], [], [])
```

Generating the Functions: Putting it Together

```
def checkPartitionRD(G, num_zeros, num_ones, num_twos): #generate and test all partitions of the vertex set V given #s needed in each set
    V = list(range(len(G))) #vertex set of G
    return partition(G, V, num_zeros, num_ones, num_twos, [], [], [])

def findMinRD(G): #finds the minimum Roman Domination of graph G
    combos = getCombos(len(G)) #get all possible sizes of coloring sets
    for c in combos:
        pot_RD = checkPartitionRD(G, c[1], c[2], c[3])
        if pot_RD[0]: #if minimum RD is found
            return (pot_RD[1], pot_RD[2], pot_RD[3], c[0]) #return tuple of list of 0s, 1s, and 2s; the last element of the tuple is the RD number
    return ([], [], [], 0) #if no RD can be found
```

Displaying the Function

- **networkx** and **matplotlib.pyplot** libraries

```python
def makeNetworkGraph(G): #make the NetworkX Graph from adjacency matrix representation
    n = len(G)
    G_vis = nx.Graph()
    G_vis.add_nodes_from(list(range(n)))
    for i in range(n):
        for j in range(n):
            if G[i][j] == 1: G_vis.add_edge(i, j)
    return G_vis
```

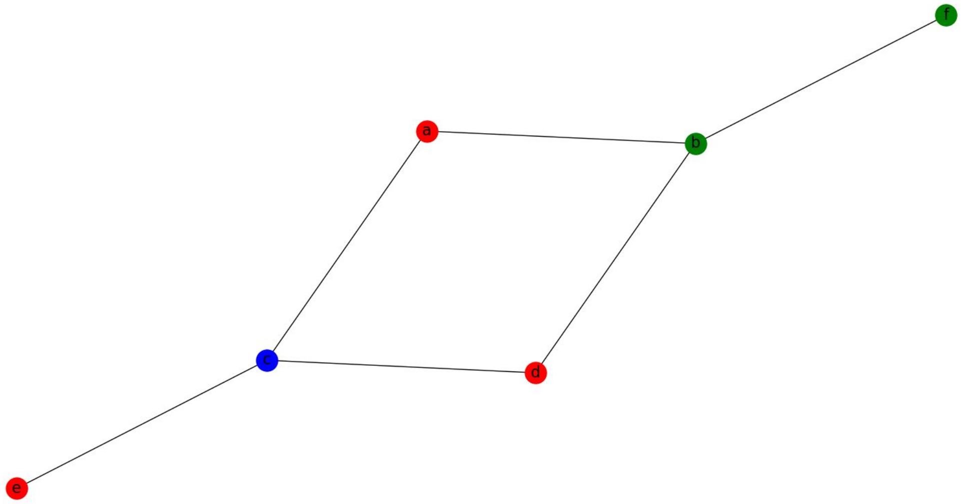
Displaying the Function

- **networkx** and **matplotlib.pyplot** libraries

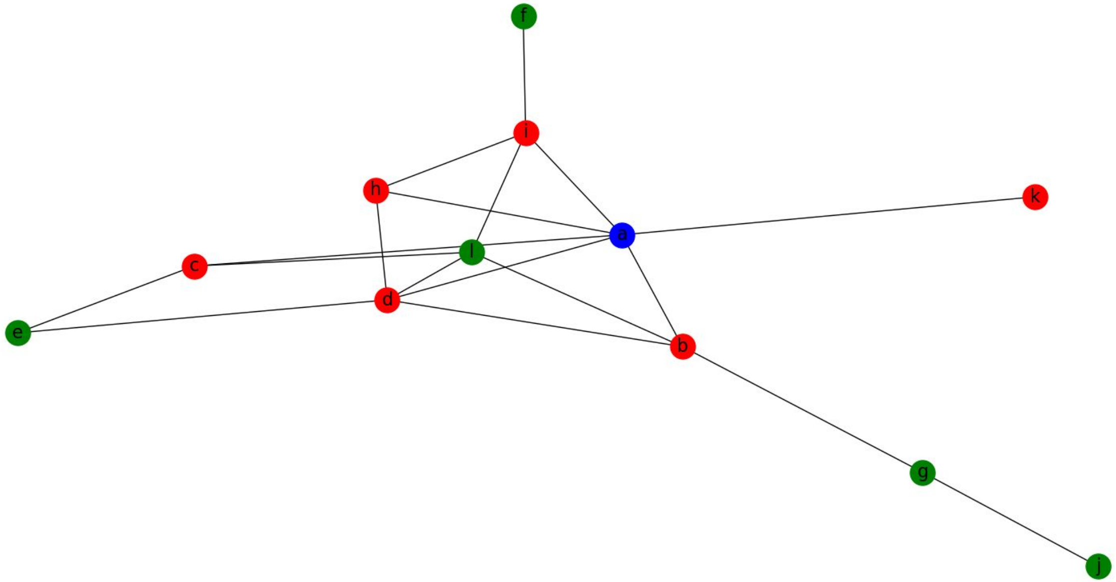
```
pos = nx.spring_layout(G_vis)
nx.draw_networkx_nodes(G_vis, pos, result[0], 500, 'r')
nx.draw_networkx_nodes(G_vis, pos, result[1], 500, 'g')
nx.draw_networkx_nodes(G_vis, pos, result[2], 500, 'b')

nx.draw_networkx_labels(G_vis, pos, labels, font_size = 16)

nx.draw_networkx_edges(G_vis, pos, width = 1.0)

plt.axis('off')
plt.show()
```

Some Results

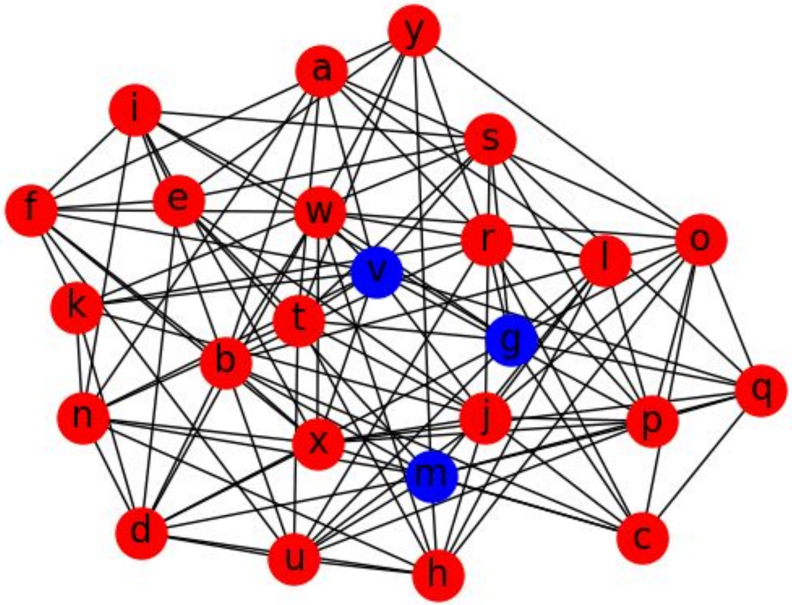
- Red = 0, Green = 1, Blue = 2



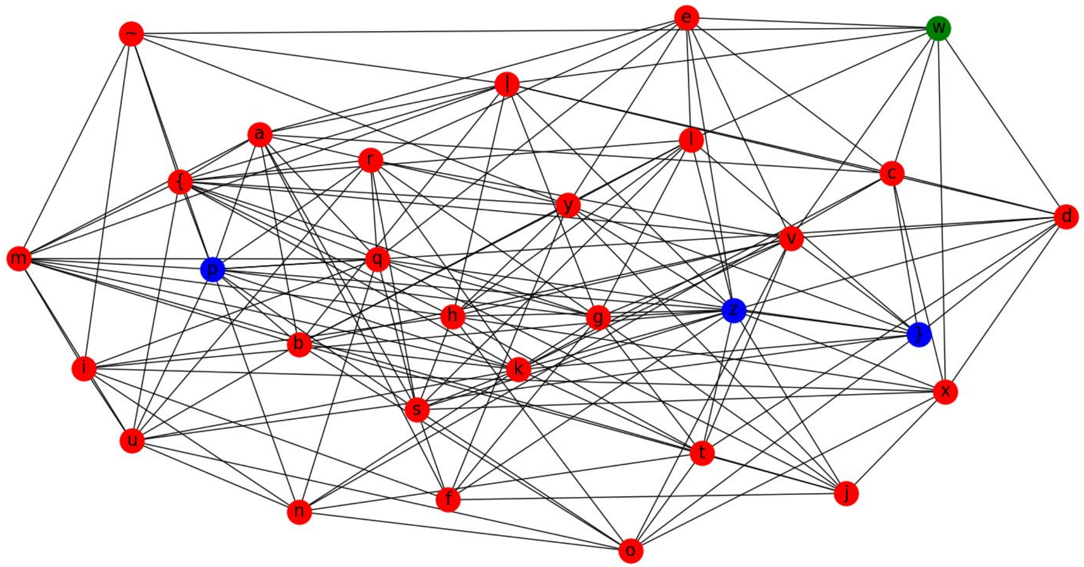
Some Results



Some Results

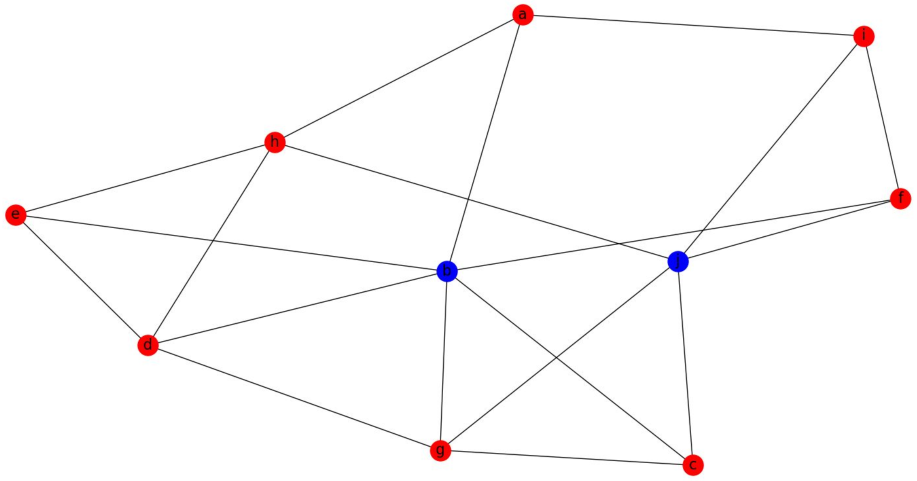


Some Results



Live Example!

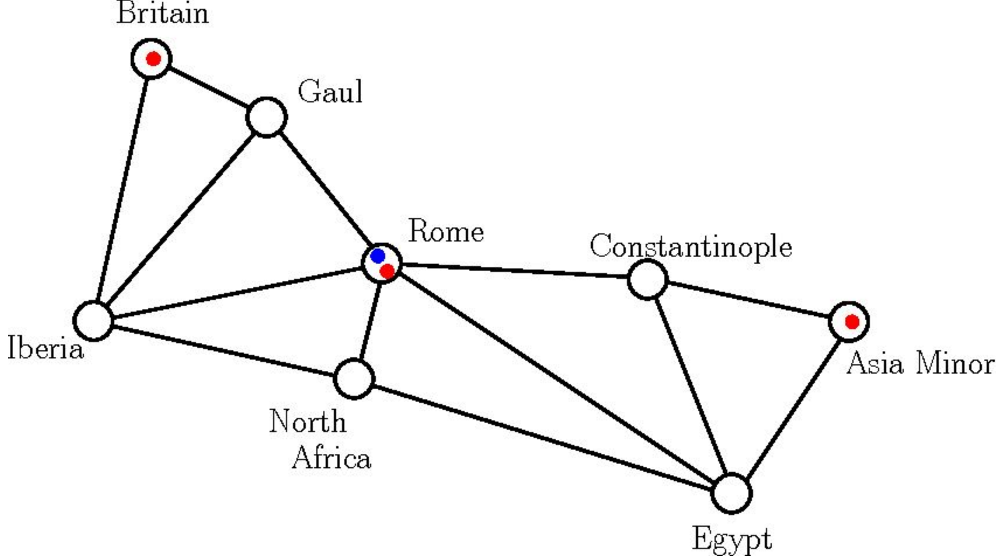
```
Order: 10
Graph generated, now finding min. Roman Domination
0s (red): a c d e f g h i
1s (green):
2s (blue): b j
Roman Domination Number:  4
```

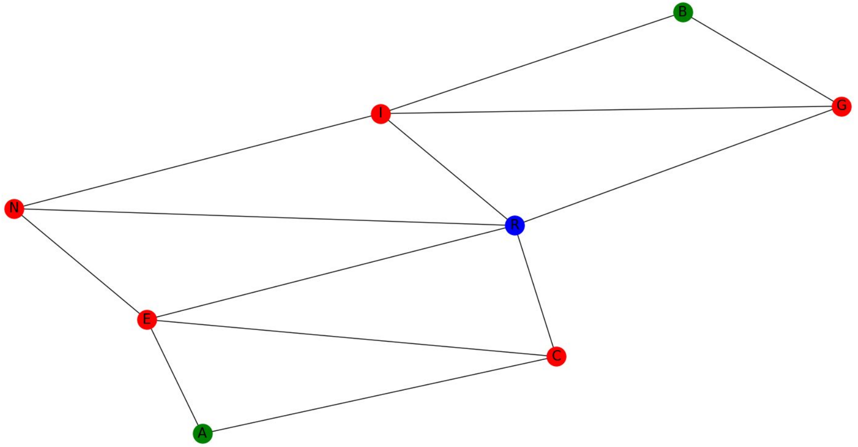


Improvements for the Future

- Tighten the restrictions on possible RD functions further
  - $\gamma(G) \leq \gamma_R(G) \leq 2\gamma(G)$
  - No 1-vertex and 2-vertex are connected
  - All 0-vertex are adjacent to at most two 1-vertices
  - $2n/(\Delta + 1) \leq \gamma_R(G)$
- Possibly more efficient partitioning procedure
- Expand program to show **all** minimum RD functions of a graph
- Still, even with improvements due to the sheer number of combinations, the problem of finding the minimum Roman domination of an arbitrary graph remains NP-Hard
- Make program generic

Conclusion


Britain
Gaul
Rome
Constantinople
Iberia
Asia Minor
North Africa
Egypt


B
I
G
N
R
E
C
A