Daniel de las Heras Garcia
Dr. Venkatesh
Introduction to Simulations and Stochastic Processes

Project 2

For this project, we implemented the proposed algorithm to design redistricting plans by uploading an initial plan using an adjacency matrix and switching one precinct to another district at a time.

The algorithm follows the following steps:

Step 1: Identifying all the nodes that are at the boundary of a district. Let this set be B=$\{i \in N : \exists (i,j) \in E \; and \; (i,j) \notin E(p)\}$ and randomly select one with equal probability .
Step 2: Identifying all the districts bordering the selected edges and select one randomly. Hence from $D_i = \{\forall P_k \in p : \exists (i,j) \in E, i \in P_l, j \in P_k\}$ choose one $d \in D_i$ with probability $\frac{1}{|D_i|}$.
Step 3: Propose the swap to the district selected. In other words if we select $d \in D_i$ then $i \in P^*_{k,t} \; and \; j \in P^*_{l,t}$, then we propose the swap to $i \in P^*{}_{k,t}$ and $i \notin P^*{}_{k,t}$.

The probability of acceptance of this proposal is given by:

$$\alpha(p_{t-1} \rightarrow p_t, i) = min(1, \frac{|B^*|}{|B|})$$

and we implemented the following methods:

For this two classes were implemented in Python, one class Redistricter, and the other one Plan. The plan creates an object which stores the information of a redistricting plan and gives several useful methods which we will use to implement the algorithm above. These methods are:

```
# get district name of current node
    def get_district(self, node)

# move node to new district and update all edge dicts and lists
    def move_node(self, node, district)

# get neighbors of current node, return dict {district : node_list}
    def get_neighbors(self, node)
```

To implement the algorithm we used three attributes from the Redistricter class:

self.initial_plan: gives the initial plan as a Plan object
self.current_plan: gives the current plan as a Plan object

and we implemented the following methods:

find_boundary_nodes(plan): This method uses the method implemented in the class Plan “plan.cut_edges” and loops over all the boundary edges and stores the nodes in an array and the method returns this array.

propose_transition(plan): this method chooses with the same probability one of the nodes from **find_boundary_nodes(plan)** and then it checks all the boundary districts to this node, and it chooses one of these districts with an equal chance and uses the method in **Plan move_node()** to move the node to the chosen district. The method then returns a plan object.

compute_transition_probability(plan1, plan2): This method computes the transition probability from plan1 to plan 2 by using the Metropolis Criterion from step 3.

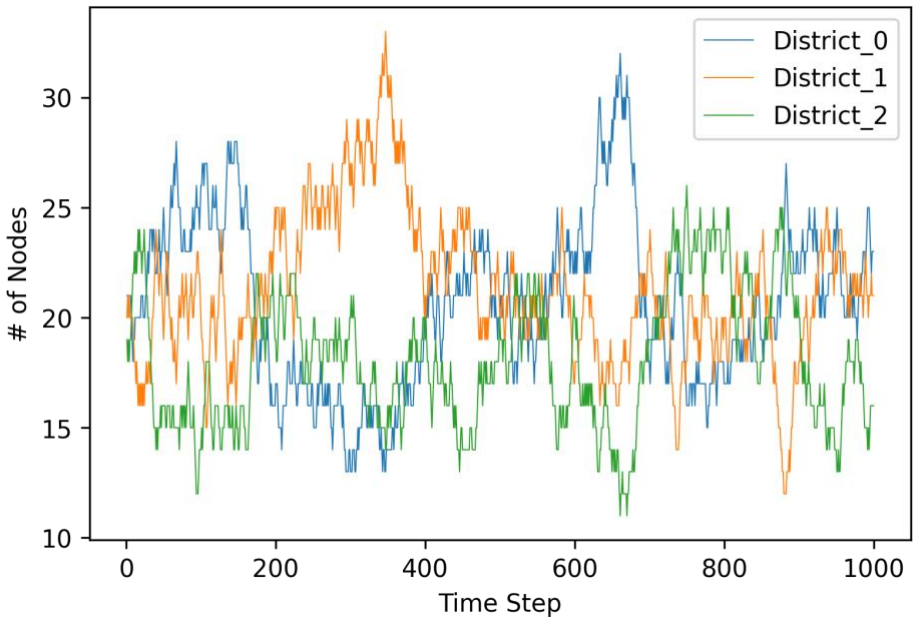
consider_transition(plan1, plan2): this method returns True with the probability computed by **compute_transition_probability** or False otherwise

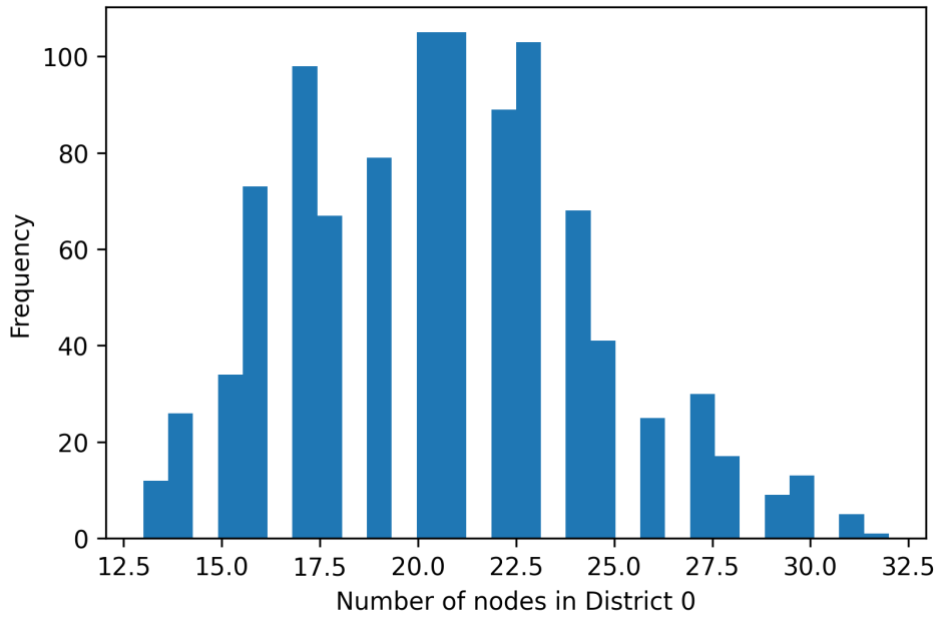
perform_transition(): This method uses all the previous methods to perform a transition and update or keep the proposed transition saving it to **current_plan** depending on the Metropolis Criterion.

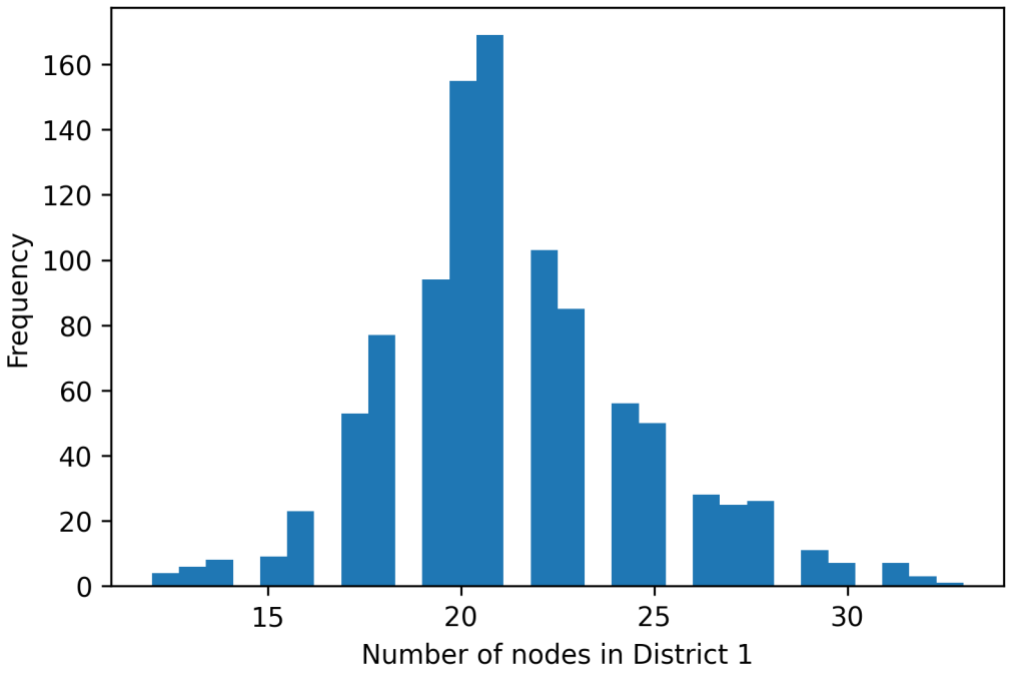
Once both classes were finished and were working correctly, I implemented a code to run 1000 transitions of the plan and recorded the number of precincts per district after every transition. The initial plan has 20 precincts per district (3 districts) and after 1000 transitions I got the results which are displayed at the end of the paper.

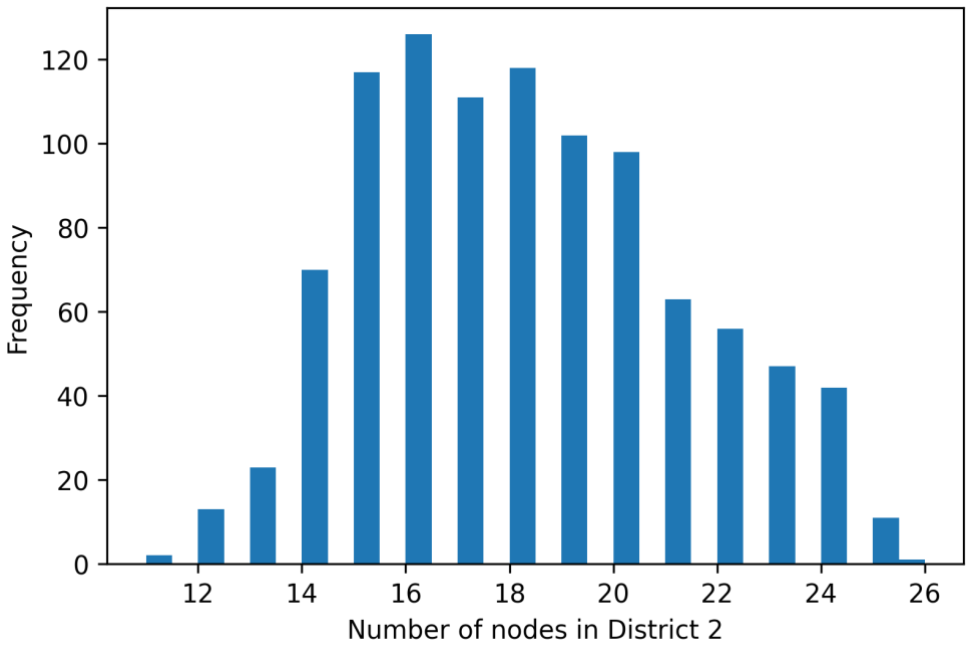
From these visualizations, we can see that the mean of every representation lies around starting point 20 and the distribution of the number of nodes per district is the normal distribution. We can interpret this same thing from the first visualization. Close to 68% of the data lies within 1 standard deviation from the mean.

Overall, even though the code was difficult to implement at first, I managed to make all the methods work successfully and perform the transition of redistricting plans. This is a key step to successfully complete the final project. The visualizations give great insights into how the Markov chain we implemented works, but this will change slightly once different constraints like population are implemented in the final project.


District_0
District_1
District_2
# of Nodes
Time Step
10
15
20
25
30
0
200
400
600
800
1000


Frequency
Number of nodes in District 0
0
20
40
60
80
100
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0
32.5


Frequency
0
20
40
60
80
100
120
140
160
15
20
25
30
Number of nodes in District 1


Frequency
0
20
40
60
80
100
120
12
14
16
18
20
22
24
26
Number of nodes in District 2