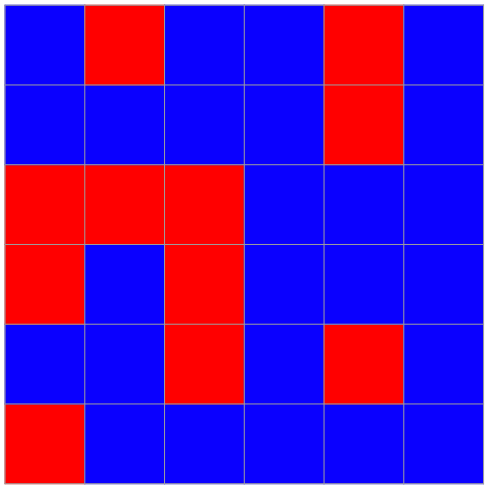
Initialization

Set the voting pattern of census blocks in North Squarolina:

In[1]:=
```
votes = Flatten@{{1, 0, 1, 1, 0, 1}, {1, 1, 1, 1, 0, 1},
    {0, 0, 0, 1, 1, 1}, {0, 1, 0, 1, 1, 1}, {1, 1, 0, 1, 0, 1}, {0, 1, 1, 1, 1, 1}};
```

In[•]:=
```
ArrayPlot[Partition[votes, 6],
 ColorRules → {0 → RGBColor[1, 0, 0], 1 → RGBColor[10 / 255, 0, 1]}, Mesh → All]
```

Out[•]=



Random Districtings

Completely Random

The next kind of random districtings we consider are those made by randomly assigning each census block to a district. We represent some districting by a list of lists, where each inner list represents a district and contains the indices of the census blocks that it includes:

In[•]:=
```
completelyRandomDistricting := Values@PositionIndex@RandomInteger[{1, 4}, 36]
```

In[•]:=
```
completelyRandomDistricting
```

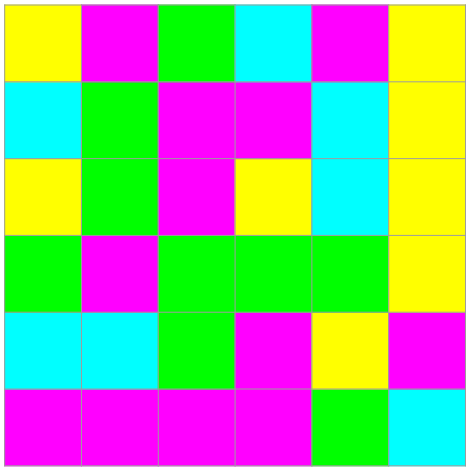
Out[•]=
```
{{1, 3, 6, 10, 11, 24, 26, 30, 31}, {2, 4, 5, 15, 21, 23, 27, 36},
 {7, 8, 12, 20, 22, 25, 29, 32, 34}, {9, 13, 14, 16, 17, 18, 19, 28, 33, 35}}
```

Next, we can create a function for visualizing districtings:

In[2]:=
```
plotDistricting[d_, w_: 6] := ArrayPlot[Partition[
    Normal@SparseArray[Catenate@MapIndexed[Thread[#1 → First[#2]] &, d]], w],
  ColorRules → {1 → Yellow, 2 → Magenta, 3 → Green, 4 → Cyan},
  Mesh → All, ImageSize → 250]
```

```
In[ ]:= plotDistricting[completelyRandomDistricting]
```

Out[ ]=

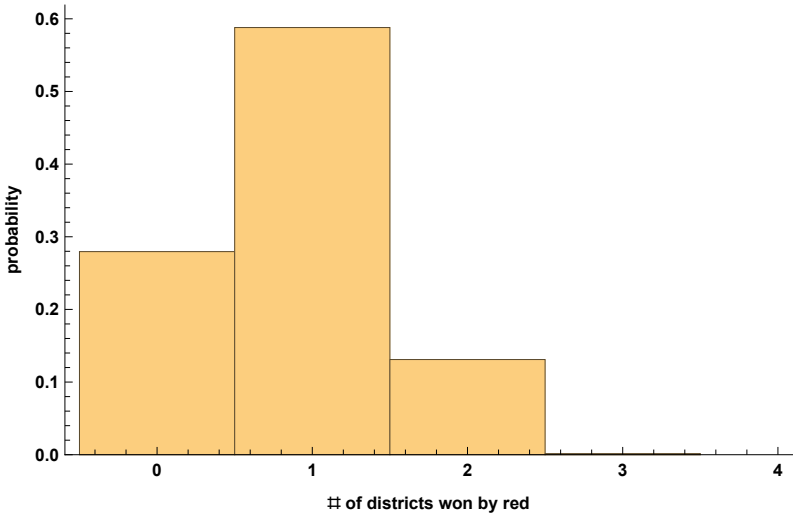


Finally, we can calculate the number of districts that each part wins on average with these noise districtings in North Squarolina:

```
In[ ]:= completelyRandomDistrictingSimulation = Table[
          Total@Boole[Total[votes[[#]]] < 5 & /@ completelyRandomDistricting], 100 000];
```

```
In[ ]:= Histogram[completelyRandomDistrictingSimulation,
         {1}, "Probability", Frame → {{True, False}, {True, False}},
         FrameLabel → {"# of districts won by red", "probability"},
         PlotRange → {{-0.5, 4}, All}]
```

Out[ ]=



However, this simulation is not terribly accurate because it allows for districts that are wildly different size. It also allows for non-contiguous districts. In fact, we can see that out of 100 000 completely random districtings, none of them contain only contiguous districts:

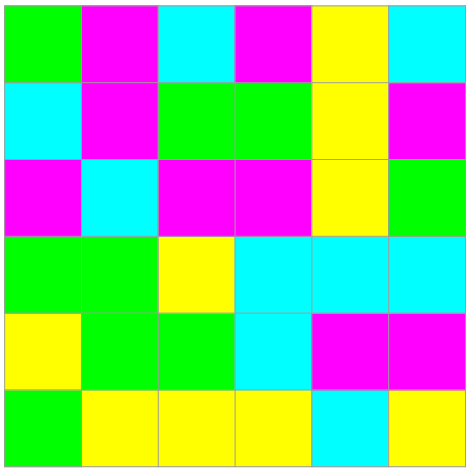
```
In[ ]:= Select[Table[Image@RandomInteger[{1, 4}, {6, 6}], 100 000],
          (i ↦ ! AnyTrue[Range[4], Max@MorphologicalComponents[
                  Binarize[i, {#, #}], CornerNeighbors → False] > 1 &])]
Out[ ]= {}
```

Random Equal-Sized

The next kind of random districtings we consider are those made by randomly selecting 9 blocks to be in district 1, another 9 in district 2, and so on. These are like the completely random ones, except all districts are the same size:

```
In[15]:= randomDistricting := Partition[RandomSample[Range[36]], 9]
In[ ]:= randomDistricting
Out[ ]= {{5, 11, 7, 20, 19, 12, 8, 35, 16}, {25, 23, 4, 27, 31, 17, 1, 15, 33},
         {32, 36, 28, 30, 9, 2, 21, 29, 24}, {26, 22, 10, 3, 13, 14, 34, 18, 6}}
In[ ]:= plotDistricting[randomDistricting]
```

Out[ ]=

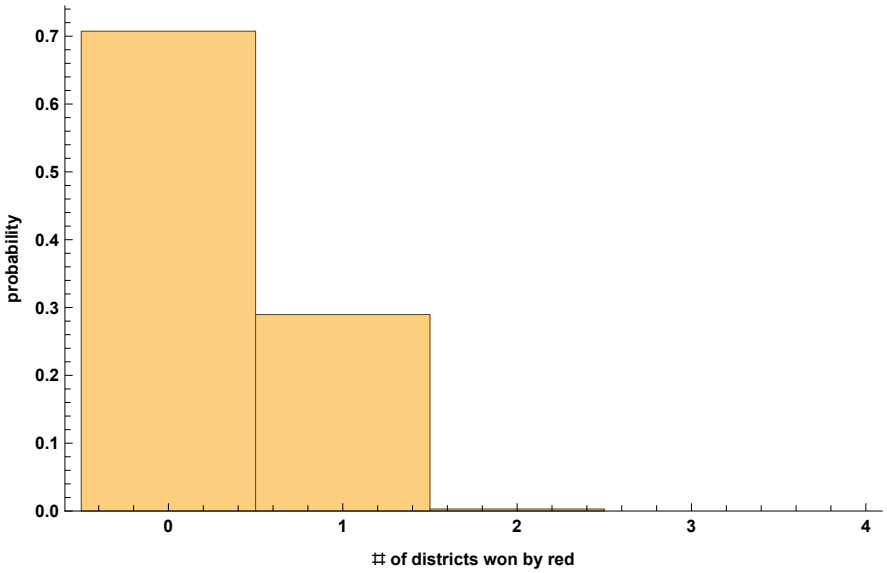


Finally, we can calculate the number of districts that each part wins on average with these noise districtings in North Squarolina:

```
In[ ]:= randomDistrictingSimulation =
          Table[Total@Boole[Total[votes[[#]]] < 5 & /@ randomDistricting], 100 000];
```

```
In[•]:= Histogram[randomDdistrictingSimulation, {1},
         "Probability", Frame → {{True, False}, {True, False}},
         FrameLabel → {"# of districts won by red", "probability"},
         PlotRange → {{-0.5, 4}, All}]
```

Out[•]=



Out of the 100 000 districtings we generated, only a few hundred managed to win 2 districts for red:

```
In[•]:= KeySort@Counts[randomDistrictingSimulation]
```

Out[•]= ⟨| 0 → 70 738, 1 → 28 966, 2 → 296 |⟩

We can also calculate the expected value of the number of districts won by red:

```
In[•]:= N@Mean[randomDistrictingSimulation]
```

Out[•]= 0.29558

We can see that this is much lower than the directly proportional number of seats we would expect for red:

```
In[•]:= N@4 * (36 - Total[votes]) / 36
```

Out[•]= 1.22222

Markov Chain Monte Carlo (MCMC)

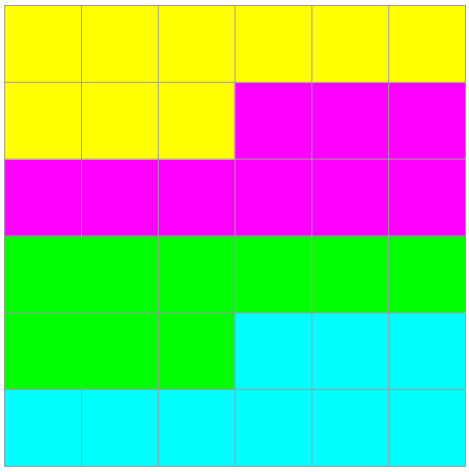
We define a function for randomly mutating districts. It takes partitions of the census blocks (a districting) and a graph that shows the adjacency of the census blocks, and mutates it a single step, preserving contiguity and approximately preserving area:

In[3]:=
```
randomlyMutateDistricts[partitionsI_, g_] :=
 Block[{partitions = partitionsI, partition, otherPartition, vertex},
  partition = RandomChoice[Position[#, Min[#]] &[Length /@ partitions]][[1]];
  vertex = RandomChoice[Complement[
     AdjacencyList[g, partitions[[partition]]], partitions[[partition]]]];
  otherPartition = Position[partitions, _?(MemberQ[vertex]), {1}][[1, 1]];
  If[Length[partitions[[otherPartition]]] > 1 && ConnectedGraphQ[
     Subgraph[g, DeleteCases[partitions[[otherPartition]], vertex]]],
   partitions = MapAt[DeleteCases[#, vertex] &, partitions, otherPartition];
   AppendTo[partitions[[partition]], vertex];
  ];
  partitions
 ]
```

As an example, here is some arbitrary districting:

In[ ]:=
```
plotDistricting@Partition[Range[36], 9]
```

Out[ ]=

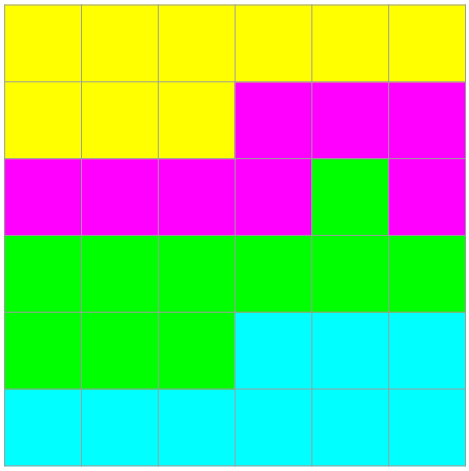


And here we randomly mutate it:

In[ ]:=
```
g = GridGraph[{6, 6}];
```

In[ ]:=
```
plotDistricting@randomlyMutateDdistricts[Partition[Range[36], 9], g]
```

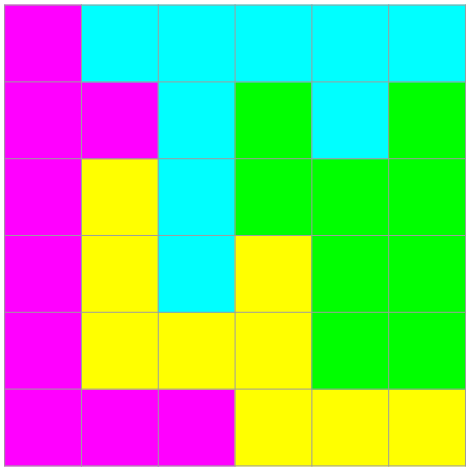
Out[ ]=



Each application of this mutation is equivalent to one step in a Markov chain. Through repeated application, we get a more random districting:

In[ ]:=
```
plotDistricting@
 Nest[randomlyMutateDistricts[#, g] &, Partition[Range[36], 9], 10 000]
```

Out[ ]=

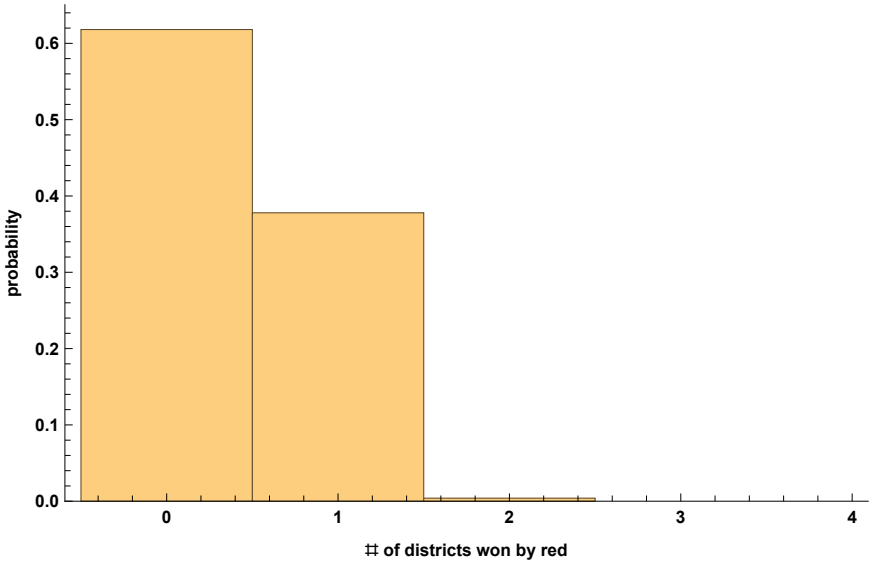


For each random districting we generate, we can calculate the number of districts that red will win:

In[13]:=
```
mcmcSimulation = Total /@ Boole@Map[Total[votes[[#]]] < 5 &,
      NestList[randomlyMutateDistricts[#, g] &, Nest[randomlyMutateDistricts[#, g] &,
        Partition[Range[36], 9], 100 000], 100 000], {2}];
```

```
In[14]:= Histogram[mcmcSimulation, {1}, "Probability", Frame → {{True, False}, {True, False}},
          FrameLabel → {"# of districts won by red", "probability"},
          PlotRange → {{-0.5, 4}, All}]
```

Out[14]=



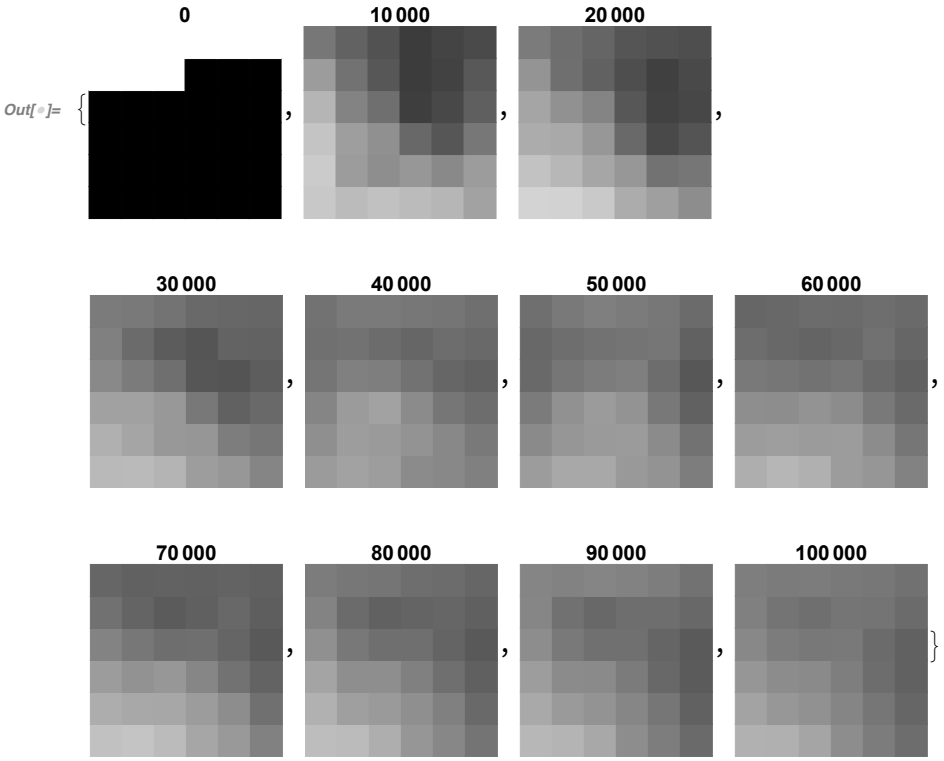

```
In[ ]:= KeySort@Counts[mcmcSimulation]
Out[ ]= ⟨| 0 → 62 264, 1 → 37 548, 2 → 189 |⟩

In[ ]:= N@Mean[mcmcSimulation]
Out[ ]= 0.379256
```

With this model, we can see that red wins more on average, but it also was less likely to win 2 districts.

Here, we started with a non-random initial condition, and then we mutated for 100 000 steps to get our second initial condition. Once we have our second initial condition, we continue mutating for another 100 000 steps, collecting data. However, we need to confirm that the initial 100 000 steps we mutated for to get the initial condition is enough that there will not be too much of a lingering influence from the non-random initial condition. Here, we start with the non-random arrangement, and then we plot the frequency with which that each cell is in district 1 as we mutate:

```
In[•]:= imgs = (Image@SparseArray[Thread[QuotientRemainder[#[[1]] - 1, 6] + 1 → 1], 6] & /@
      NestList[randomlyMutateDistricts[#, g] &,
       Partition[Range[6^2], 6^2/4], 100 001]);
    accImgs = FoldList[ImageAdd, imgs];
    Table[Show[
      Image@Rescale[#/Total[Flatten[#]] &@ImageData[accImgs[[n]]], {0, 2/36 // N}],
      PlotLabel → n - 1, ImageSize → 100], {n, 1, 100 001, 10 000}]
```

Out[•]=



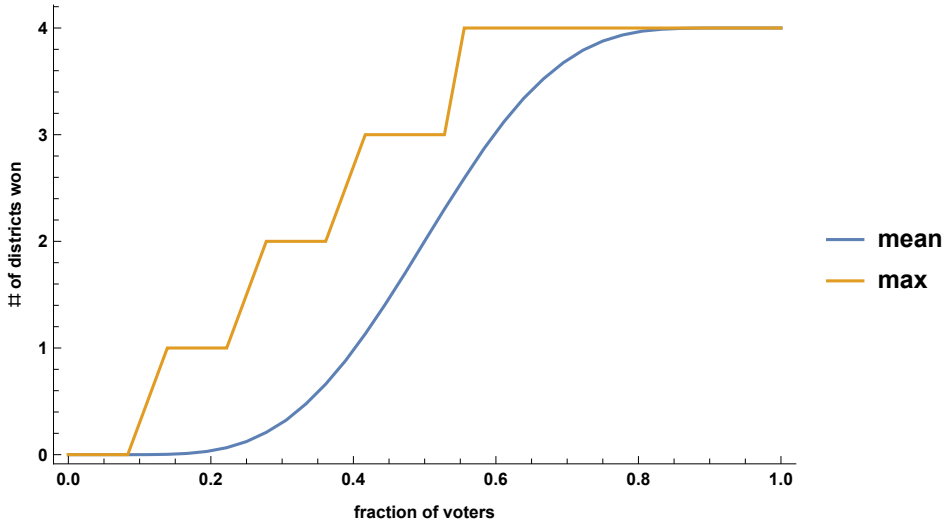
We can see that after 100 000 all configurations are almost equally likely, suggesting that this method for generating random districts works.

We can also use this model to create districts of with random voter arrangements. Here, we create a random voter arrangements with *n* voters, and then compute the average and maximum number of districts that go to each party as a function of *n*:

In[156]:=
```
LaunchKernels[11];
simulationData =
  With[{size = 6},
   ParallelTable[
    {N[n/size^2],
     Module[{createVotes, votes, g, partitions, allDistrictings, outcomes},
      createVotes :=
       RandomSample[Join[ConstantArray[1, n], ConstantArray[0, size^2 - n]]];
      g = GridGraph[{size, size}];
      partitions = Nest[randomlyMutateDistricts[#, g] &,
        Partition[Range[size^2], size^2/4], 100 000];
      allDistrictings = NestList[randomlyMutateDistricts[#, g] &,
        partitions, 100 000];
      outcomes = Total /@ Map[(votes = createVotes;
           (If[#1 === #2, 0.5, If[#1 > #2, 0, 1]] &@@ Lookup[
                Counts@votes[[#]], {0, 1}, 0]) &/@#) &, allDistrictings];
      N@{Mean[outcomes], Max[outcomes]}
     ]
    },
    {n, 0, size^2}]
  ];
CloseKernels[];
```

In[159]:=
```
ListLinePlot[Transpose[Thread /@ simulationData],
 Frame → {{True, False}, {True, False}},
 FrameLabel → {"fraction of voters", "# of districts won"},
 PlotLegends → {"mean", "max"}]
```

Out[159]=



The shape of this function shows us why Gerrymandering is effective. A small difference in the fraction of voters can quickly change the fraction of districts won. We can also see how the best district can do

significantly better than the average district, showing how far Gerrymandering can take you.

We can also see where the proposed districts fall compared to this curve. Here, we import the proposed districts from the data file:

In[193]:=
```
proposedDdistricts =
  Table[Values@KeySort@PositionIndex@Flatten@Reverse@Transpose@Partition[
          Round@Transpose[Rest@Import[FileNameJoin[{rawDataDirectory,
                  "mixon_mcm_data.xlsx"}]][[1]]][[n]], 6], {n, 4, 6}];
```

Then, we calculate the number of districts that each proposal will win for the reds:

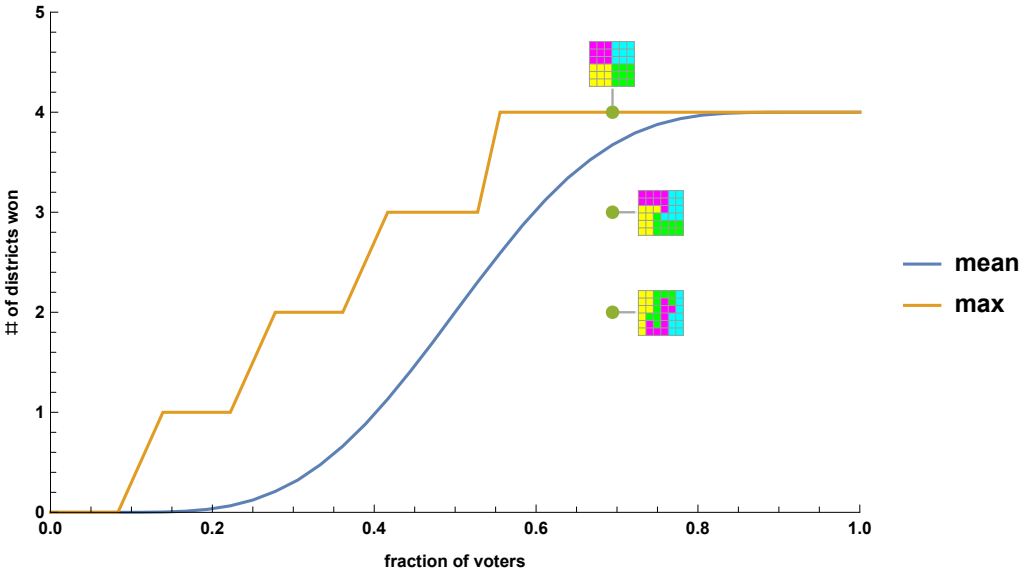
In[202]:=
```
Total /@
 Map[If[#1 === #2, 0.5, If[#1 > #2, 0, 1]] &@@ Lookup[Counts@votes[[#]], {0, 1}, 0] &,
  proposedDistricts, {2}]
```

Out[202]=
```
{4, 3, 2}
```

We can place these on the plot from before to see how these compare with averages for districts with the same proportion of red and blue voters:

In[738]:=
```
ListLinePlot[Append[Transpose[Thread /@ simulationData],
  {Callout[{Total[votes] / Length[votes], 2},
    plotDistricting[proposedDistricts[[3]]], Right],
   Callout[{Total[votes] / Length[votes], 3},
    plotDistricting[proposedDistricts[[2]]], Right],
   Callout[{Total[votes] / Length[votes], 4},
    plotDistricting[proposedDistricts[[1]]], Top]}],
 Frame → {{True, False}, {True, False}}, FrameLabel →
  {"fraction of voters", "# of districts won"}, Joined → {True, True, False},
 PlotLegends → {"mean", "max"}, PlotRange → {{0, 1}, {0, 5}}]
```

Out[738]=



We can see that proposal A yields the maximum number of districts for blue, while C yields fewer.

We can also compare with the data from our simulation on North Squarolina's voting arrangement to see how likely it would be to draw districts to reach each of these election outcomes:

```
In[231]:= 100 * N@Counts[mcmcSimulation] / Length[mcmcSimulation]
Out[231]= ⟨| 0 → 61.8074, 1 → 37.7936, 2 → 0.398996 |⟩
```

We can see that A gets the same result as about 62% of random districts, B gets the same result as about 38%, and C gets the same result as a tiny 0.4%.

Random Seeding

Previously, we created our initial conditions for the MCMC case by running our mutation algorithm on some non-random initial condition 100 000 times. We showed that this produces a relatively uniform distribution of districts. However, if we want an even more uniform distribution, or if we want to generate initial conditions on much larger graphs, we need to randomly general initial conditions.

Our mutation algorithm always brings the number of blocks in each district closer together, so we can even just start with a random seed where the districts are unequal in size. All that is important is that they are contiguous. To do so, we take the grid graph that we used to represent adjacency, we remove all of the edges, and then we repeatedly add edges back to the smallest connected components. (Some seeds will lead to states that our MCMC algorithm cannot mutate into equal-area districts, and so we ignore those seeds.)

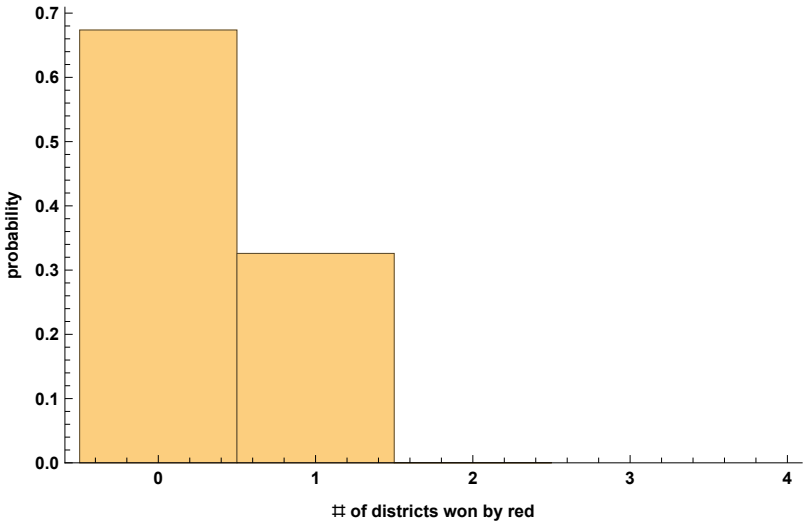
```
In[575]:= seededDistricting[g_, td_: 4, steps_: 1000, maxDist_: 0] := Module[{r, first = True},
  While[first || ! Abs[Subtract @@ MinMax[Length /@ r]] ≤ maxDist,
   first = False;
   r = NestWhile[randomlyMutateDdistricts[#, g] &,
     ConnectedComponents@NestWhile[
       ng ↦ EdgeAdd[ng, RandomChoice@Complement[IncidenceList[g, RandomChoice@
              MinimalBy[Select[ConnectedComponents[ng], Complement[IncidenceList[
                    g, #], EdgeList[ng]] =!= {} &], Length]], EdgeList[ng]]],
       Graph[VertexList[g], {}], Length[ConnectedComponents[#]] > td &],
     ! Abs[Subtract @@ MinMax[Length /@ #]] ≤ maxDist &, 1, steps]
  ];
  r]
```

Like before, we can try running this many times and seeing the distribution of election results we get:

```
In[472]:= LaunchKernels[11];
seededDistrictingSimulation = ParallelTable[
   Total@Boole[Total[votes[[#]]] < 5 & /@ seededDistricting[g]], 100 000];
CloseKernels[];
```

In[477]:=
```
Histogram[seededDistrictingSimulation, {1},
 "Probability", Frame → {{True, False}, {True, False}},
 FrameLabel → {"# of districts won by red", "probability"},
 PlotRange → {{-0.5, 4}, All}]
```

Out[477]=



In[479]:=
```
KeySort@Counts[seededDdistrictingSimulation]
```

Out[479]=
```
⟨|0 → 67 374, 1 → 32 598, 2 → 28|⟩
```

This gives only slightly different results from our earlier algorithm with MCMC.

Real Data

Using real voting data from OpenElections, we can try running our algorithms on real data. First, we import the data from OpenElections. We will be using North Carolina as an example.

We use precinct level data but then generate county level data for alignment reasons:

In[627]:=
```
rawData = Import[
   FileNameJoin[{rawDataDirectory, "20121106__nc__general__precinct__raw.csv"}]];
```

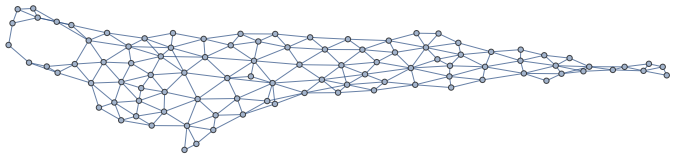
In[628]:=
```
countyMap =
  Association[# → Interpreter["USCounty"][ToLowerCase[#] <> " county NC"] & /@
    Union@rawData[[2 ;;, 16]]];
```

In[630]:=
```
countyVotes = DeleteCases[Query[All, Query[GroupBy[First → Last]] /* Map[Total] /*
      (Boole[Lookup[#, "DEM", 0] > Lookup[#, "REP", 0]] &), {15, 19}]@
   KeyMap[countyMap, GroupBy[Select[Rest[rawData],
      #[[8]] === "US HOUSE OF REPRESENTATIVES" &], #[[16]] &]], {0, 0}];
```

Next, we generate the adjacency graph for counties in North Carolina:

In[634]:=
```
ncg = VertexDelete[Graph[Keys[countyVotes],
    DeleteDuplicatesBy[Catenate@KeyValueMap[Thread@*UndirectedEdge,
       EntityValue[Keys[countyVotes], "BorderingCounties", "EntityAssociation"]],
     Sort]], Except[Alternatives @@ Keys[countyVotes]]]
```

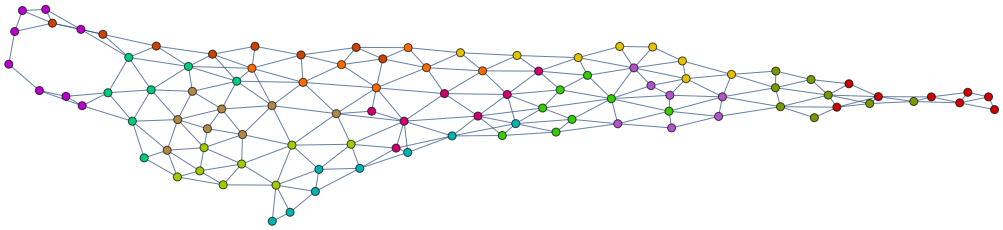
Out[634]=



Finally, we can just use our seeded MCMC districting to generate random districts:

In[636]:=
```
HighlightGraph[ncg, seededDdistricting[ncg, 13, 1000, 1]]
```

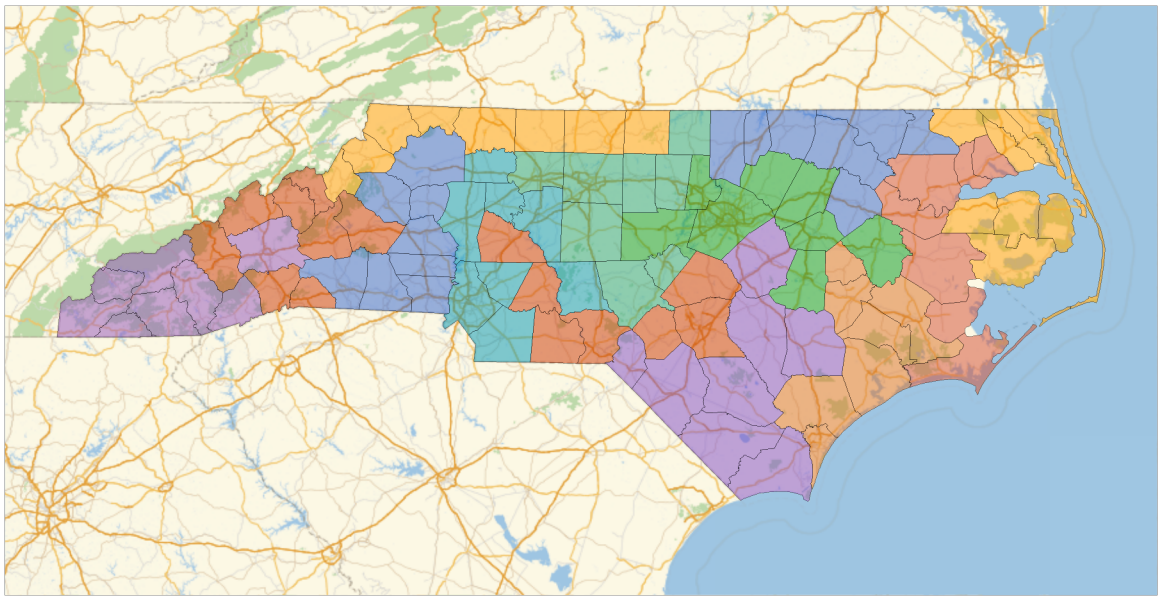
Out[636]=



We can also put them on a map:

In[644]:=
```
GeoListPlot[seededDistricting[ncg, 13, 1000, 1], PlotLegends → None, ImageSize → 600]
```

Out[644]=



Our algorithm is fairly efficient, partitioning NC in about a third of a second on a laptop:

In[646]:=
```
RepeatedTiming[seededDistricting[ncg, 13, 1000, 1];]
```

Out[646]=
```
{0.33, Null}
```

We can then do our same statistical analysis by generating 10 000 random districts and measuring what fraction of them are won by what party:

In[740]:=
```
LaunchKernels[11];
ncDistrinctingSimulation = ParallelTable[Total@Boole[Mean[#] > 0.5 & /@
      Map[countyVotes, seededDistricting[ncg, 13, 1000, 1], {2}]], 10 000];
CloseKernels[];
```

We can see the distribution of election outcomes that we would expect from random districts:

In[743]:=
```
Histogram[13 - ncDistrinctingSimulation, {1},
 "Probability", Frame → {{True, False}, {True, False}},
 FrameLabel → {"# of districts won by red", "probability"},
 PlotRange → {{-0.5, 13}, All}]
```

Out[743]=