5-6-2021

Sampling Compactness Scores to Detect Gerrymandering in Squaretopia

Joshua Mariz
jmariz@lion.lmu.edu

Loyola Marymount University
University Honors Program

Sampling Compactness Scores to Detect Gerrymandering in Squaretopia

A thesis submitted in partial satisfaction

of the requirements of the University Honors Program

of Loyola Marymount University

by

Joshua Mariz

May 5, 2021

Sampling Compactness Scores to Detect Gerrymandering in Squaretopia

A thesis submitted to
Loyola Marymount University
The Mathematics Department
in partial fulfillment of the requirements
for Graduation with the Bachelor of Science Degree

by

Joshua Mariz

May 2021

Sampling Compactness Scores to Detect Gerrymandering in Squaretopia

Senior Thesis by

Joshua Mariz

Robert Rovetti, Thesis Director

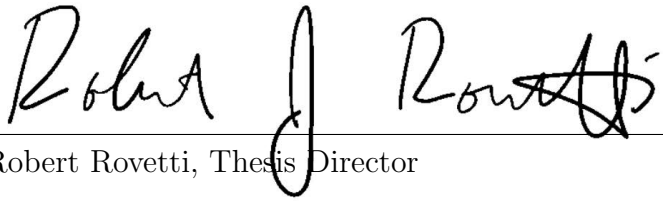
Abstract

In electoral politics, gerrymandering is the phenomenon of creating electoral district partitionings that are often not geographically compact for the unfair benefit of one political party over another. Researchers have proposed several methods to quantify compactness, but identifying gerrymandering using these measures is an open problem. We analyze the possible distributions of compactness scores by exploring "Squaretopia," a square n x n grid that we must partition into n equally-sized contiguous districts that each contain n cells. However, even in this simplified model, the number of possible partitions of a Squaretopia of size $n = 9$ exceeds 700 trillion, rendering the generation of all possible partitions a computationally expensive task and leading us to consider sampling. We develop Partitioner, a recursive algorithm written in the Java programming language, to randomly generate samples of 10,000 partitions by choosing an unoccupied cell and then randomly adding contiguous neighbor cells until the district contains n cells—a process that is repeated until the grid is completely partitioned. The first version of this algorithm produced samples that overrepresented high compactness scores; we reduced this bias by adding a weighting factor to the district-creation process to increase the probability that Partitioner generates straighter districts, thus shifting the compactness score distribution as expected. The optimal weighting factor minimizes the difference between the populations and their samples for the Reock and Length-Width scores. Using these weightings, we generate samples of larger Squaretopias and demonstrate how we can detect likely gerrymandering by statistically identifying unreasonably small compactness scores.

Thesis written by

Joshua Mariz

Approved by



06 May 2021

Robert Rovetti, Thesis Director — Date

05/06/2021

Patrick Shanahan, Mathematics Department Chair — Date

Contents

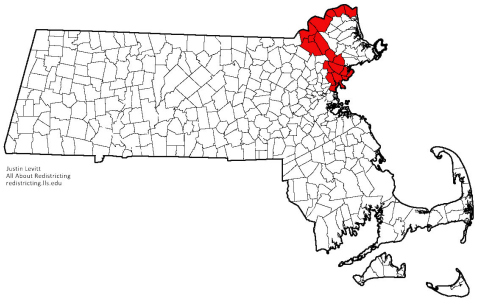
Figures

Tables

Introduction

In the United States, the House of Representatives makes and passes federal laws. The citizens in each of the 435 congressional districts elect one representative to the House of Representatives. Every decade, states redraw districts after the census. It is state legislatures and commissions who propose new districts. Unfortunately, gerrymandering may sometimes occur in these proposals. Gerrymandering is the phenomenon of creating electoral district partitionings that are often not geographically compact, to unfairly benefit one political party over another. The term "Gerry-mander" originated in 1812 when Gerry, the governor of Massachusetts, approved a redistricting plan containing a senate district that resembled a salamander. Figure 1(a) is a map showing the location of this district [Levitt, 2020], while Figure 1(b) is a photograph of the *Boston Gazette*'s 1812 depiction of the district's resemblance to a salamander [Davis, 2017]. In Figures 2-5, we see more recent examples of questionable districts. These districts, from the 2010 redistricting cycle, are Illinois's 4th, Maryland's 3rd, Ohio's 9th, and Texas's 2nd. To render Figures 2-5, the Environmental Systems Research Institute (Esri) analyzed GIS shapefile data created by the United States Department of the Interior [Esri, 2013].

Rick Gillman points out three important features of congressional districts: population equity, contiguity, and compactness [Gillman, 2002]. To maintain population equity, each congressional district should represent approximately an equal number of citizens. We can conceptualize contiguity as the ability to travel to every point in a district without entering another district. Because bodies of water can disconnect a district's land, we say "without entering another district," instead of "without leaving the district." Lastly, compactness indicates how compact or spread out a district is, with greater compactness being preferable. Figures 1-5 are examples of incompact districts.



(a) The Problematic District

(b) *Boston Gazette* Illustration

Figure 1: The Salamander-like District Approved by Governor Gerry

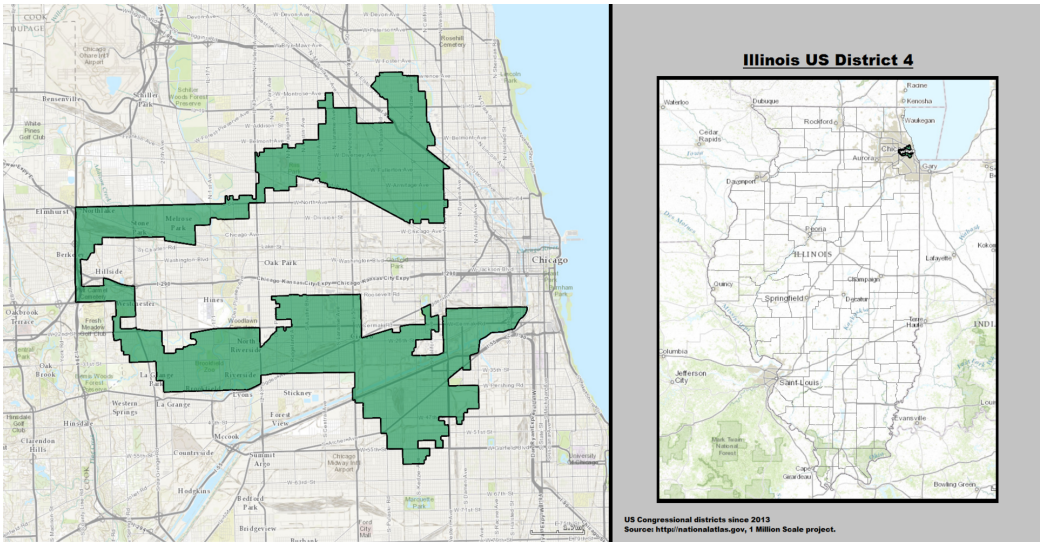


Figure 2: Illinois's 4th Congressional District

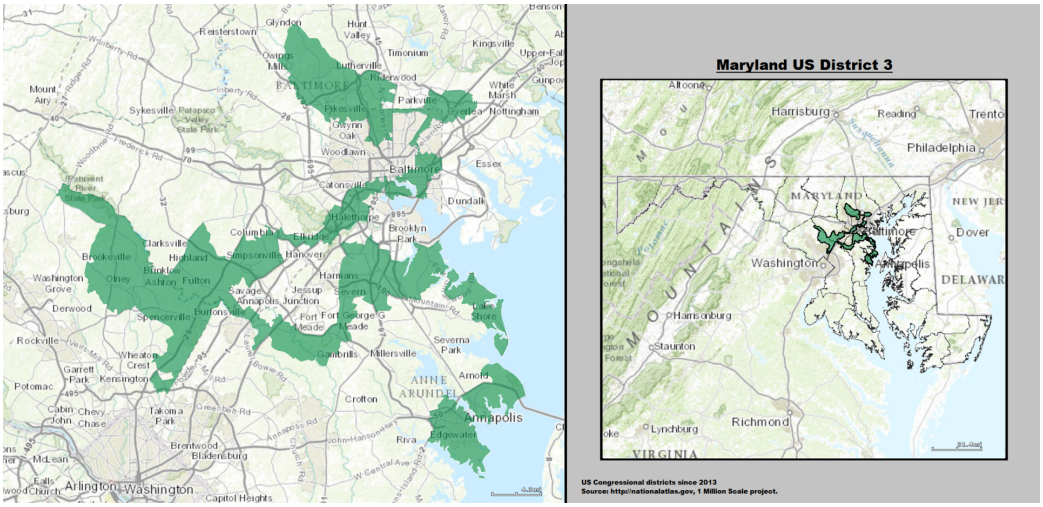


Figure 3: Maryland's 3rd Congressional District

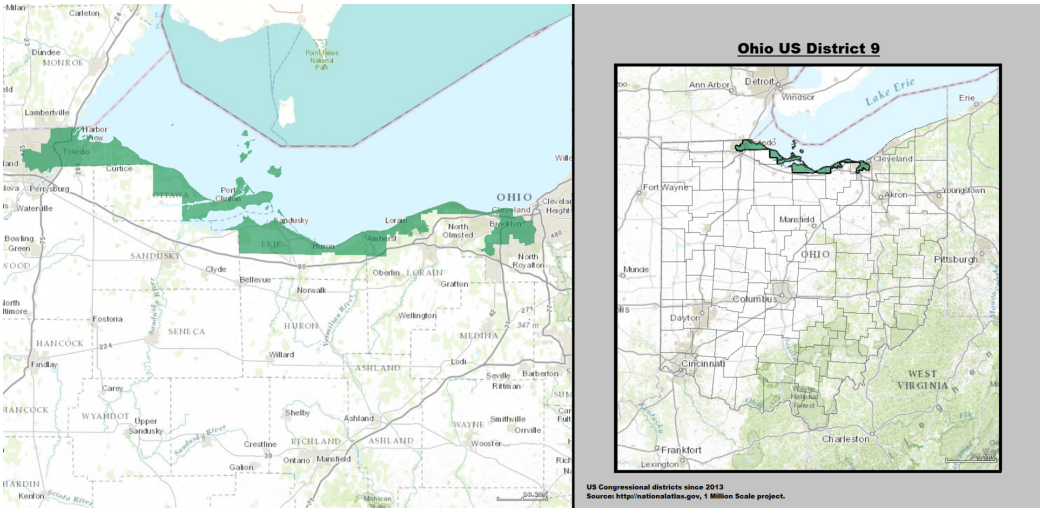


Figure 4: Ohio's 9th Congressional District

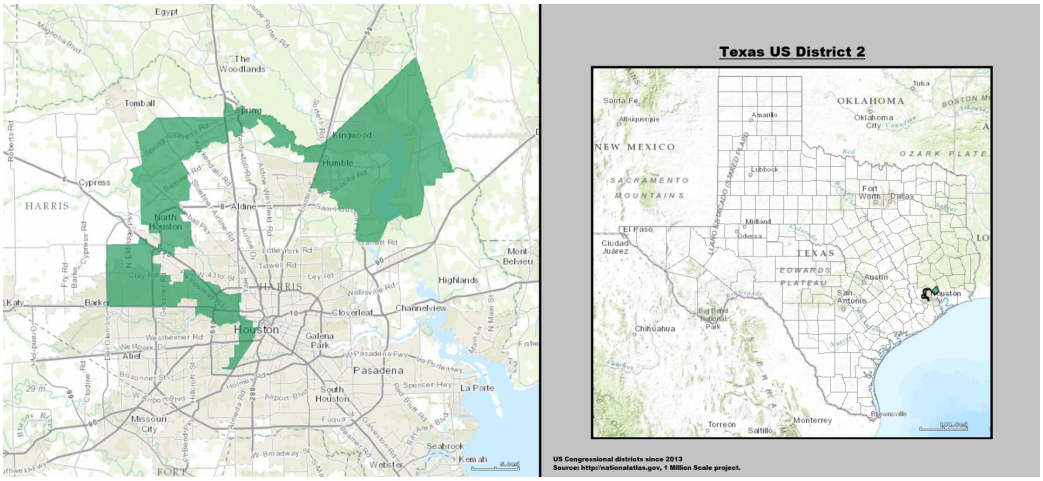


Figure 5: Texas's 2nd Congressional District

To measure a district's compactness, we calculate the district's compactness score. Gillman suggests that a compactness score should be dimensionless and on a scale of 0 to 1, where 1 describes a highly compact region. Researchers have proposed many methods for quantifying compactness. In this paper, we use four scoring methods: Length-Width, Reock, Schwartzberg, and Polsby-Popper.

To simplify many of the complexities involved with redistricting, we redistrict in Squaretopia, a square n x n grid containing n^2 cells that we must partition into n contiguous districts that each contain n cells. We assume that each cell contains the same number of Squaretopia citizens. Given these conditions, we guarantee population equity and district contiguity in our partitions. However, we make no such claim for compactness. In this paper, we develop an algorithm that redistricts Squaretopia and scores partitions. We provide a basic introduction to polyominoes and motivate our approach before discussing our algorithm.

D. Hugh Redelmeier defines a polyomino as a collection of equal-sized squares in a plane, touching each other along complete edges [Redelmeier, 1981]. Every square, also known as a cell, need not touch every other one, but a polyomino must be connected. Free polyominoes are distinct when none is a rigid transformation of another. In Figure 6, we have eight pentominoes. A pentomino is a five-celled polyomino. The red pentominoes are reflections of each other, and the blue pentominoes are rotations of each other. Thus, we only have one free pentomino in the figure.

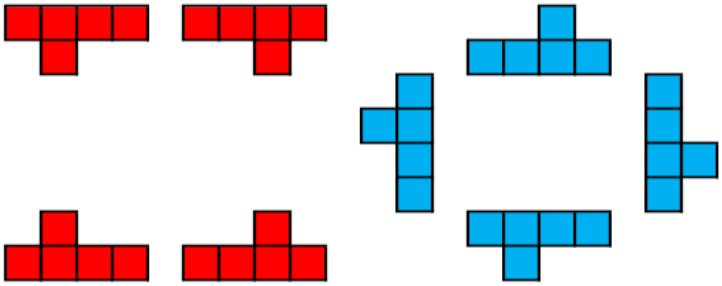


Figure 6: Reflections and Rotations of the Same Pentomino

To create a valid partition of a size n Squaretopia, we find a set of n free n-ominoes that fully cover the n x n grid. For instance, a partition of a size 3 Squaretopia must be composed of three trominoes. However, three trominoes don't necessarily compose a partition. A partition can contain the same free polyomino multiple times. Figure 7 shows the ten partitions of a size 3 Squaretopia. We see that there are only two free trominoes. Although we observe rotational and reflection symmetry, each partition is unique. This is because we assume that each cell contains the same number of citizens, not the same citizens. We call the ten partitions in Figure 7 the population of the partitions of a size 3 Squaretopia.

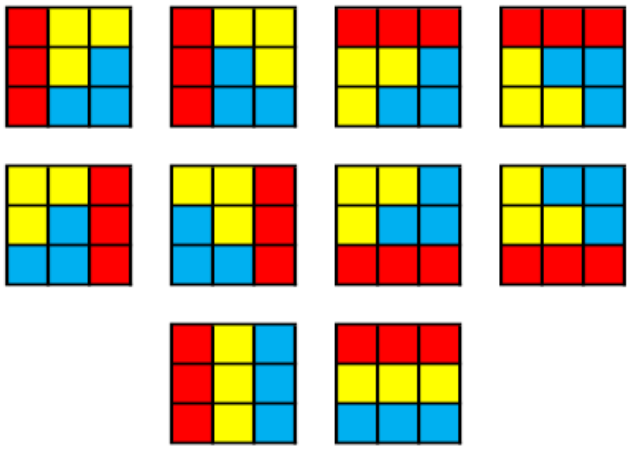


Figure 7: Population of Partitions of a Size 3 Squaretopia

The Metric Geometry and Gerrymandering Group, led by Moon Duchin, has found the sizes of the populations of partitions of Squaretopias of sizes up to and including size 9 [Duchin, 2019]. As the size of Squaretopia increases, so does the complexity of partitions. Table 1 shows this increasing complexity.

Squaretopia Size (n)	Free n-ominoes	Unique Partitions
2	1	2
3	2	10
4	5	117
5	12	4,006
6	35	451,206
7	108	158,753,814
8	369	187,497,290,034
9	1,285	706,152,947,468,301
10	4,655	Unknown
11	17,073	Unknown

Table 1: Increasing Complexity of Squaretopia Partitions

Generating all the partitions for Squaretopias of increasing sizes becomes more and more computationally expensive. The task requires increasing quantities of time and memory. Such a problem warrants a sampling approach. Instead of generating the population of partitions of a Squaretopia of size $n > 7$, we should generate a representative sample.

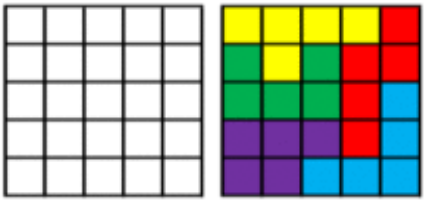
Duchin utilizes a Markov chain Monte Carlo (MCMC) method to explore the population of Squaretopia partitions. Duchin's MCMC method begins with a partitioned Squaretopia, and then swaps a pair of adjacent cells at each step of the random walk. Only a swap that results in a valid partition is performed and considered a step. According to Duchin, ergodic theory guarantees that sufficiently long random walks generate samples that properly represent the population.

Can a recursive method also generate representative samples? Can a recursive method help us determine if a given partition is significantly not compact, i.e., highly unlikely to have been drawn by chance? We develop Partitioner, a recursive algorithm written in the Java programming language, to generate samples of random partitions.

Methods I

Generating a Partition

A valid partition of a size n Squaretopia must contain n contiguous districts. A set of n n-ominos forms a partition if the n-ominos cover all Squaretopia cells without overlapping. In Figure 8 we see an example of one possible partition of a size 5 Squaretopia. A size 5 Squaretopia refers to a square 5 x 5 grid.



(a) Empty (b) Partitioned

Figure 8: Example of One Partition of a Size 5 Squaretopia

Partitioner, a recursive algorithm written in the Java programming language, generates samples of random partitions. Partitioner also calculates the Length-Width, Reock, Schwartzberg, and Polsby-Popper compactness scores of each partition. The code for this project has been made public at `https://github.com/jmariz/SeniorThesis`. Appendix F contains this project's most important files.

In Methods I, we demonstrate how the recursive algorithm generates one partition. Figures 9 - 11 show the step-by-step generation of the partition seen in Figure 8(b). We start with an empty Squaretopia. The empty Squaretopia in Figure 9(a) has $n^2 = 5^2 = 25$ free cells. A free cell is a cell that does not belong to a district. The set of all free cells is

$\{c_{1,1}, c_{1,2}, c_{1,3}, c_{1,4}, c_{1,5}, c_{2,1}, \ldots, c_{5,5}\}$. Partitioner selects a cell uniformly at random from this set. In Figure 9(b), we see that Partitioner chose $c_{4,4}$ as the first cell in the first district. Now, consider the set of all free cells adjacent to $c_{4,4}$. These cells are the neighbors of our incomplete first district. The set of neighboring cells is $\{c_{3,4}, c_{4,3}, c_{4,5}, c_{5,4}\}$. Partitioner selects a cell uniformly at random from this set. In 9(c), we see that Partitioner chose $c_{3,4}$ as the first district's second cell. Partitioner removes $c_{3,4}$ from the set of neighboring cells, and adds $c_{2,4}, c_{3,3}$, and $c_{3,5}$. The set of neighboring cells is now $\{c_{4,3}, c_{4,5}, c_{5,4}, c_{2,4}, c_{3,3}, c_{3,5}\}$. Partitioner chooses uniformly at random from the set of neighboring cells, removes the chosen cell, and updates the set until the first district has $n = 5$ cells. Figure 9 shows how Partitioner generates the first district of the partition seen in 8(b).

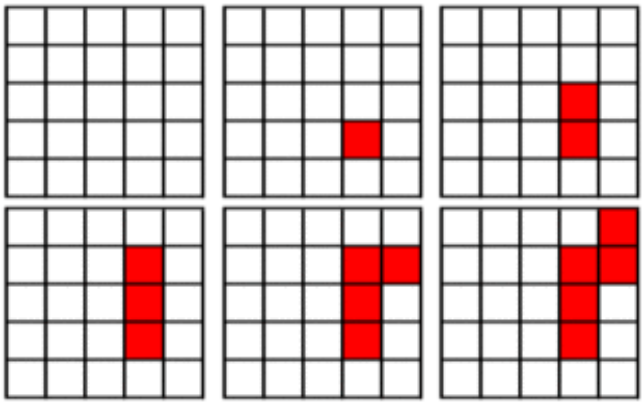


Figure 9: Partitioner Generates the First District

Partitioner begins generating the second district in the same manner. Partitioner first selects a free cell uniformly at random. In Figure 10(a) we see that $c_{1,4}$ is the first cell in the second district. The set of neighboring cells now only includes $c_{1,3}$. Partitioner chooses $c_{1,3}$ as the second cell, removes the cell from the set, and adds the new neighboring cells to the set. Figure 10(c)-(e) continue this process.

Partitioner encounters a problem in (e); we cannot assign $c_{1,1}$ to a contiguous district of size 5. Partitioner recurses, removing the cell most recently added to the district. Figure 10(f) shows the removal of $c_{2,1}$ from the second district. The set of neighboring cells is now $\{c_{2,3}, c_{1,1}, c_{3,2}\}$. Note that we also remove $c_{2,1}$ from this set because choosing $c_{2,1}$ as the next cell won't lead to a valid partition. In (g), Partitioner chooses $c_{1,1}$ as the last cell of

the second district. During the generation of the third district, Partitioner encounters the same problem in (l). Cell $c_{2,3}$ is left isolated. In (m)-(r), we see that Partitioner recurses a few times before finding a fifth cell that forms a valid third district. If none of the possible fifth cells had lead to a valid partition, then Partitioner would have recursed at an additional layer of depth and removed the fourth cell. Figure 11 shows the generation of the last two districts.

We identify an optimization made to the recursive algorithm. When selecting the first cell of a district, Partitioner will prioritize choosing a cell with only one free neighboring cell. This optimization increases the probability that Partitioner generates valid partitions and reduces the need for recursion. Partitioner prioritizes $c_{1,4}$ in Figure 9(f), $c_{2,1}$ in Figure 10(g), $c_{3,5}$ in Figure 10(r), and $c_{4,3}$ in Figure 11(e), as these cells have only one free neighboring cell.

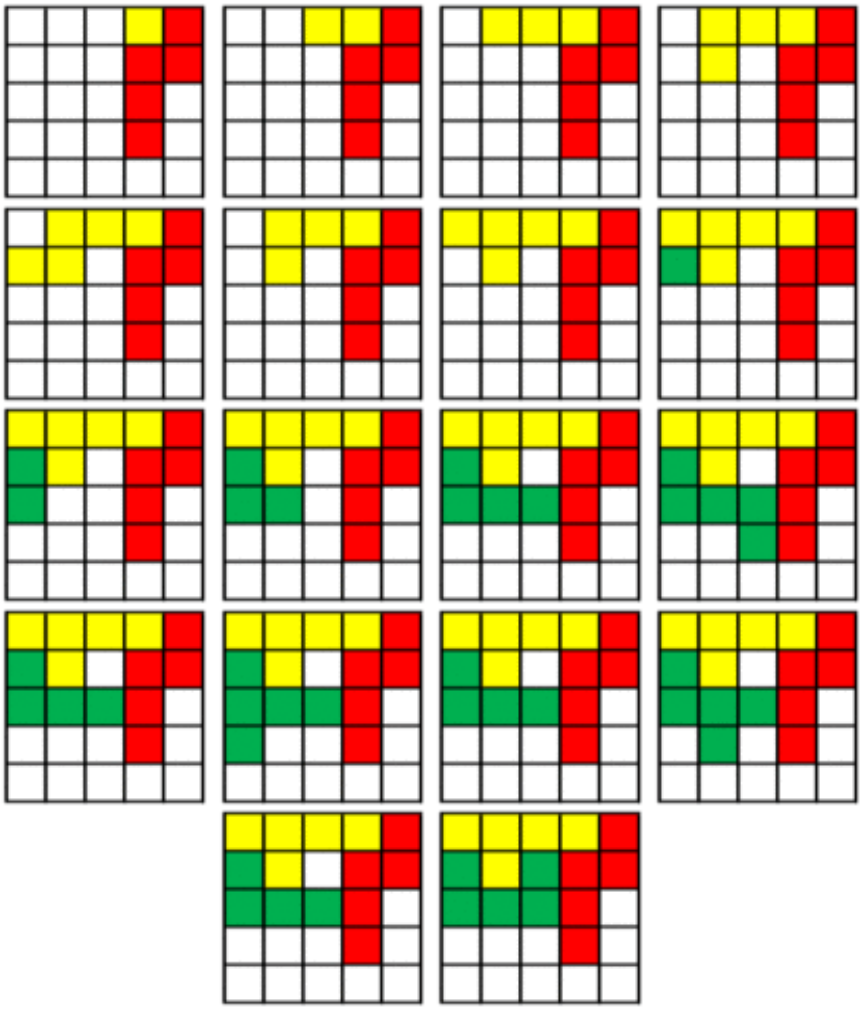


Figure 10: Partitioner Uses Recursion to Generate a Valid Partition

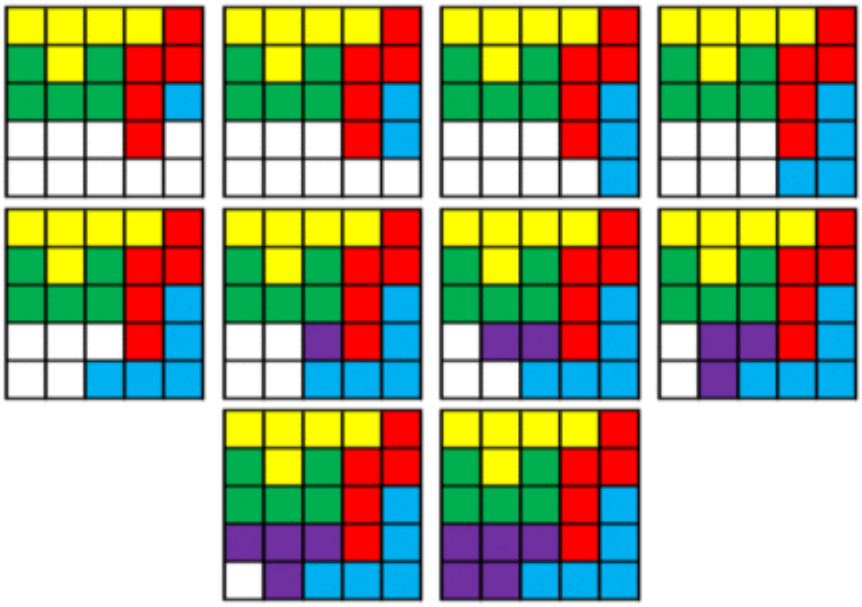


Figure 11: Partitioner Generates the Last Districts

Compactness Scores

We consider four scoring methods in our investigation of compactness in Squaretopia. As Gilman recommends, the four scoring methods are on a scale of 0 to 1, where 1 describes a highly compact region. Consider the red district from the partition shown in Figure 11(j). Let A_{red} and P_{red} be the red district's area and perimeter, respectively. We define a district's length and width as the length and width of the district's minimum bounding rectangle. Let L_{red} and W_{red} be the length and width, respectively, of the red district. Figure 12 shows the red district's minimum bounding rectangle.

Curtis states that one formulation of compactness is dividing a district's length by its width [Curtis C., 1995]. We divide width by length to ensure that the score is less than or equal to 1. The Length-Width score of the red district is $\frac{W_{red}}{L_{red}} = \frac{2}{4} = 0.5$.

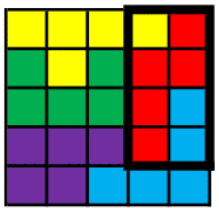


Figure 12: Red District's Minimum Bounding Rectangle

Definition 1 (Length-Width score)**.** The Length-Width (LW) score of district i is $\frac{W_i}{L_i}$ where W_i and L_i are the district's width and length, respectively. The Length-Width score of a partition of Squaretopia is the average Length-Width score of the districts in the partition.

Reock states that dividing a district's area by the area of the smallest possible circumscribing circle serves as a good measure of compactness [Reock, 1961]. In Squaretopia, the analogous definition of a district's Reock score is the district's area divided by the area of the minimum bounding square. The Reock score of the red district is $\frac{A_{red}}{(L_{red})^2} = \frac{5}{4^2} = 0.31$. Note that we round compactness scores to the nearest hundredth.

Definition 2 (Reock score)**.** The Reock (RE) score of district i is $\frac{A_i}{(L_i)^2}$ where A_i and L_i are the district's area and length, respectively. The Reock score of a partition of Squaretopia is the average Reock score of the districts in the partition.

Schwartzberg states that we can determine the compactness of a district by dividing the circumference of a circle of equal area by the perimeter of the district [Schwartzberg, 1966]. In Squaretopia, the analogous definition of a district's Schwartzberg score is the quotient of the perimeter of a square of equal area and the district's perimeter. The Schwartzberg score of the red district is $\frac{4\sqrt{A_{red}}}{P_{red}} = \frac{4\sqrt{5}}{12} = 0.75$.

Definition 3 (Schwartzberg score)**.** The Schwartzberg (SB) score of district i is $\frac{4\sqrt{A_i}}{P_i}$ where A_i and P_i are the district's area and perimeter, respectively. The Schwartzberg score of a partition is the average Schwartzberg score of the districts in the partition.

Polsby and Popper state that we can measure a district's compactness by dividing the district's area by the area of a circle with a circumference of length equal to the district's perimeter [Polsby and Popper, 1991].

In Squaretopia, the analogous definition of a district's Polsby-Popper score is the district's area divided by the area of a square with a perimeter equal to the district's perimeter. The Polsby-Popper score of the red district is $\frac{A_{red}}{(\frac{P_{red}}{4})^2} = \frac{5}{(\frac{12}{4})^2} = 0.56$.

Definition 4 (Polsby-Popper score)**.** The Polsby-Popper (PP) score of district i is $\frac{A_i}{(\frac{P_i}{4})^2}$ where A_i and P_i are the district's area and perimeter, respectively. The Polsby-Popper score of a partition is the average Polsby-Popper score of the districts in the partition.

We can use Partitioner to generate a sample of independent partitions. For example, we can generate a sample of 10,000 partitions of a size 5 Squaretopia. For each partition, Partitioner will output the Length-Width, Reock, Schwartzberg, and Polsby-Popper scores. Partitioner's output resembles Table 2. Recall that we round all compactness scores to the nearest hundredth. Partitioner multiplies each rounded compactness score by 100.

Partition\Score	LW	RE	SB	PP
1	80	46	75	56
2	77	51	78	60
3	63	51	86	75
⋮	⋮	⋮	⋮	⋮
9,999	73	56	83	70
10,000	51	39	80	65

Table 2: Sample Output for Partitioner's Generation of 10,000 Partitions of a Size 5 Squaretopia

Then, we can calculate the relative frequencies of each score. Table 3 shows the relative frequencies of compactness scores by scoring method.

Score	LW	RE	SB	PP
0	0	0	0	0
1	0	0	0	0
⋮	⋮	⋮	⋮	⋮
44	0.0041	0.0768	0	0
⋮	⋮	⋮	⋮	⋮
75	0	0	0.1394	0.1117
⋮	⋮	⋮	⋮	⋮
100	0	0	0	0
Total	1	1	1	1

Table 3: Sample Relative Frequencies of Compactness Scores

Using the table containing the relative frequencies of compactness scores for a sample of partitions, we can create the cumulative relative frequency table. Table 4 contains the cumulative relative frequencies, and the cumulative relative frequency distribution of the sample of 10,000 Length-Width scores is plotted in Figure 13. Although we depict the cumulative distribution as a continuous function, a step function more accurately represents the cumulative distribution, since the set of compactness scores only consists of nonnegative integers less than or equal to 100.

Score	LW	RE	SB	PP
0	0	0	0	0
1	0	0	0	0
⋮	⋮	⋮	⋮	⋮
44	0.0055	0.3679	0	0
⋮	⋮	⋮	⋮	⋮
75	0.8024	1	0.1394	1
⋮	⋮	⋮	⋮	⋮
100	1	1	1	1

Table 4: Sample Cumulative Relative Frequencies of Compactness Scores

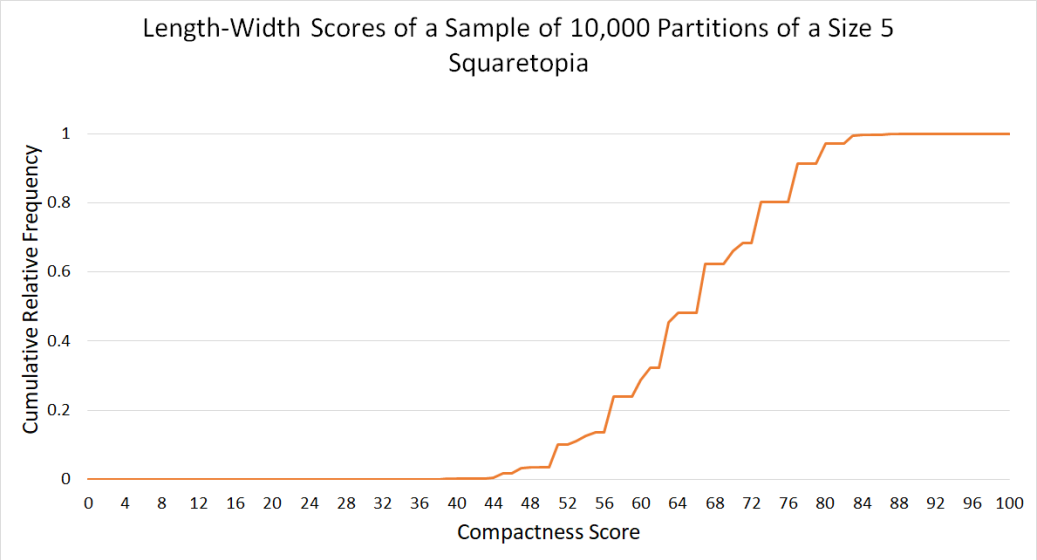


Figure 13: Sample Cumulative Relative Frequency Distribution Graph

Results I

Determining a Sufficient Sample Size

Do different sample sizes produce significantly different cumulative distributions? To investigate this question, we focused on the size 6 and size 7 Squaretopias, as the number of possible partitions for these are known and exceed a few thousand. We used Partitioner to generate 14 independent samples. For a Squaretopia of size 6, we generated samples of five hundred, one thousand, three thousand, five thousand, seven thousand, nine thousand, and ten thousand partitions. We did the same for a Squaretopia of size 7. Figure 14 shows the plotted cumulative relative frequency distributions of the Length-Width scores of six samples of size 6 Squaretopia partitions. "LW 500" refers to the LW score cumulative distribution of the sample of five hundred partitions. We see almost no variation in the cumulative distributions. Only the sample of five hundred partitions stands out, and not substantially so. The other graphs exhibit similar characteristics, so we include them in Appendix A.

We define the max difference measure to compare two cumulative distributions. To calculate the max difference between two cumulative distributions, we calculate the absolute value of the difference between the two cumulative distribution's frequencies at each score, and take the maximum value.

Definition 5 (Max Difference Measure)**.** The max difference between the cumulative frequency distribution of a sample of size n and the cumulative frequency distribution of a sample of size m is
$max\{|F_s^n - F_s^m| : s \in \mathbb{Z} \wedge 0 \leq s \leq 100\}$ where F_s^k is the relative frequency of the compactness score, s, calculated from the sample of size k.

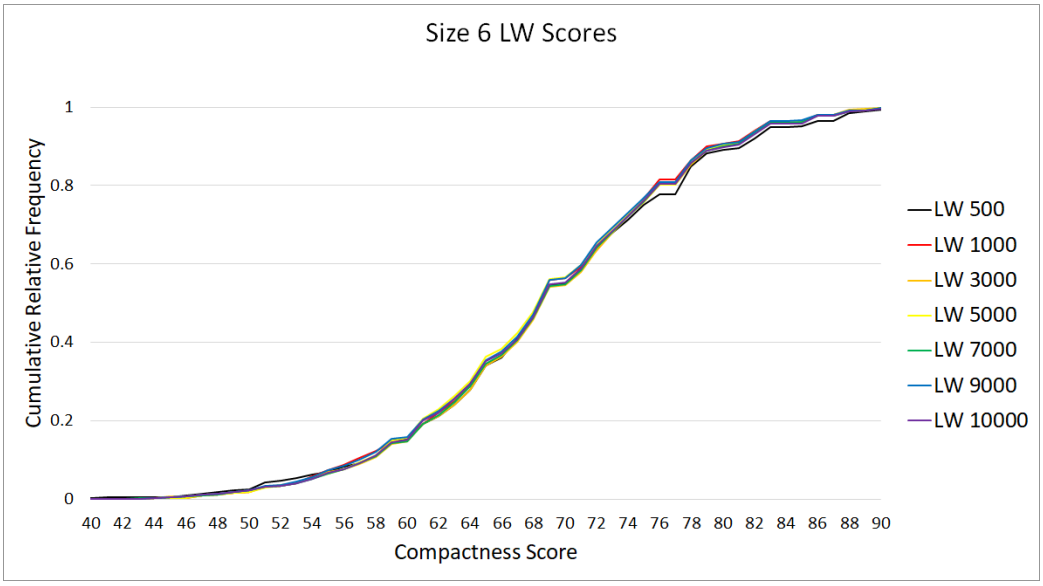


Figure 14: Size 6 Squaretopia Cumul. Rel. Freq. Dist. of LW Scores of Multi-Sized Samples

We use the sample of ten thousand partitions as our baseline. Figures 15 and 16 graph the max difference vs. the sample size. We compare the samples of sizes five hundred, one thousand, three thousand, five thousand, seven thousand, and nine thousand to the two samples of ten thousand partitions. In both figures, we see that the sample of size five hundred has the greatest max difference for all four scoring methods. In both figures, the max difference remains below 0.02 for samples of size greater than one thousand. We also note that the data for the Polsby-Popper score closely follows that of the Schwartzberg score. We double-checked the data for these two scores and confirm that this was not an error. We note that the Schwartzberg score is the square root of the Polsby-Popper score, but we do not explore this relationship further. Based on this data, we expect that the cumulative relative frequency distribution of a sample of size ten thousand will not substantially differ from that of a sample with a few thousand less partitions. Samples appear to converge to the same distribution. Thus, we believe that a sample of ten thousand partitions will be representative of the partitions that Partitioner generates.

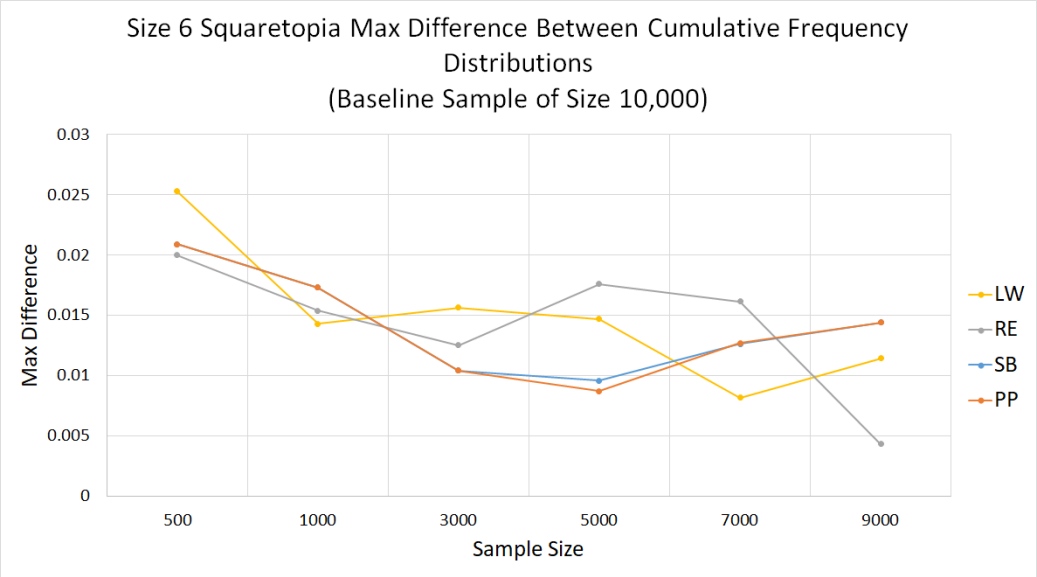


Figure 15: Size 6 Squaretopia Max Diff. Between Cumul. Rel. Freq. Dist. (Baseline Sample of Size 1000)

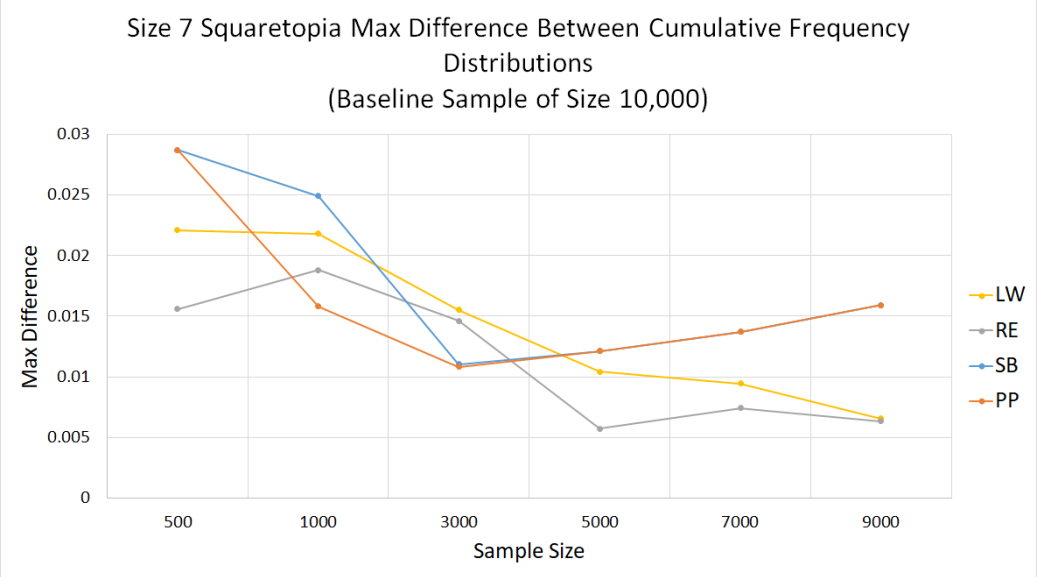


Figure 16: Size 7 Squaretopia Max Diff. Between Cumul. Rel. Freq. Dist. (Baseline Sample of Size 1000)

Comparison to the Known Population Distributions

Now that we have identified a good sample size, we generate four samples of ten thousand partitions of Squaretopias of sizes 4, 5, 6, and 7. We have ten thousand partitions for each Squaretopia size, and each partition has four scores. Dr. Robert Rovetti has scored the populations of partitions of Squaretopias of these sizes. We graph the cumulative relative frequency distributions of the scores of the populations and of the generated samples. Figure 17 shows the cumulative distributions of the LW scores of the size 7 Squaretopia population and sample. All 16 graphs are included in Appendix B.

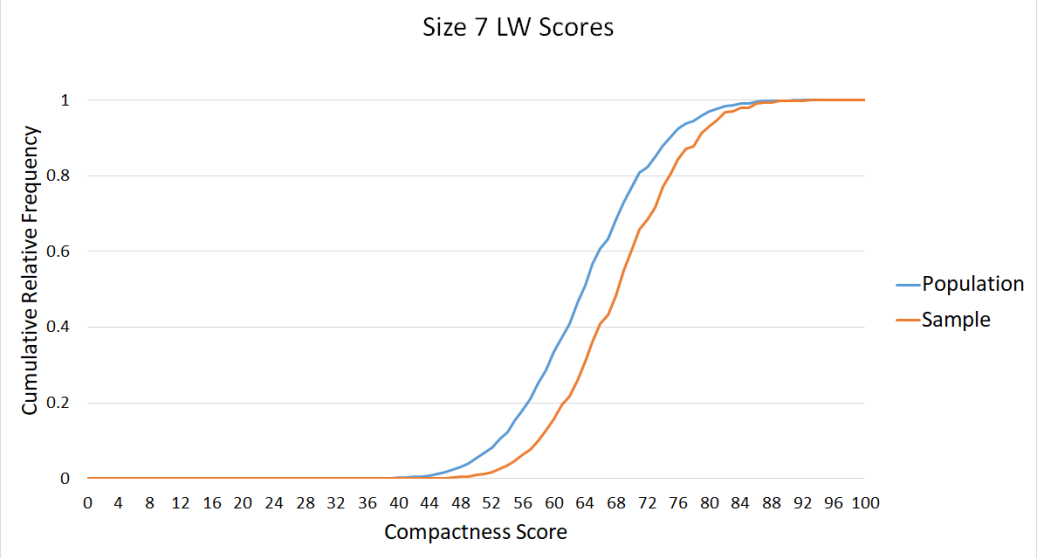


Figure 17: Size 7 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

Comparing Partitioner's samples of compactness scores to Dr. Rovetti's populations of scores reveals that Partitioner overrepresents highly compact partitions. In all 16 graphs, we observe that the cumulative distribution of the sample's compactness scores lies to the right of the population's cumulative distribution. Partitioner is biased towards more compact partitions. Partitioner may be generating compact districts too frequently.

To investigate this pattern, we use Partitioner to generate more samples. However, we stop Partitioner after it has created the first district. Our samples contain incomplete partitions that each contain only a single district. We also unbound Partitioner. Instead of generating a district of size n in an n x n grid, we allow Partitioner to generate single districts of size n in a grid of size $2n-1$ by $2n-1$. We force Partitioner to choose cell $c_{n,n}$ as the first cell, but we make no further changes to the algorithm.

Figure 18 classifies the 12 free pentominoes. Figure 19(a) shows the distribution of the 12 pentomino classes for a sample of 10,000 pentominoes generated in a 5 x 5 grid. Figure 19(b) shows the distribution of the 12 pentomino classes for a sample of 10,000 pentominoes generated in a 9 x 9 grid. For the pentominoes generated in a 9 x 9 grid, we force cell $c_{5,5}$ to be the first cell of all pentominoes. Figure 19(c) shows the actual distribution of the 12 pentomino classes in the population of all 4,006 size 5 Squaretopia partitions. In (c), we can see that the relatively compact pentomino class 4 occurs most frequently in the population of partitions. Class 4 is also the most frequent in (a), but class 4's frequency exceeds all others by a much greater magnitude than in (c). Figure 19(a) also shows that Partitioner fails to generate an adequate number of the incompact classes 1 and 2. For these reasons, we believe that Partitioner overrepresents compact districts and underrepresents incompact districts.

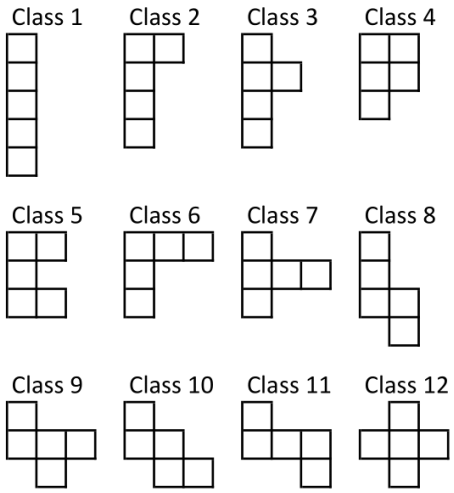
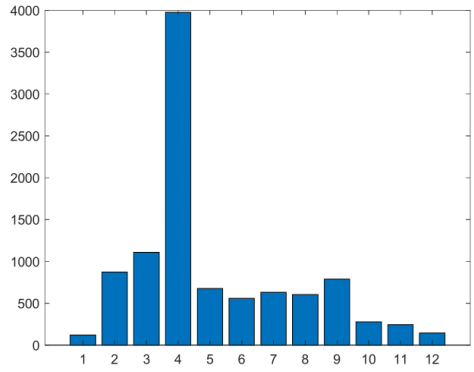
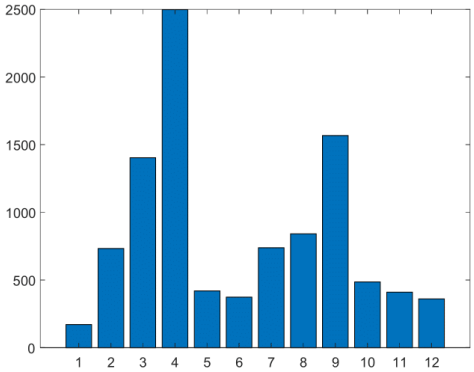


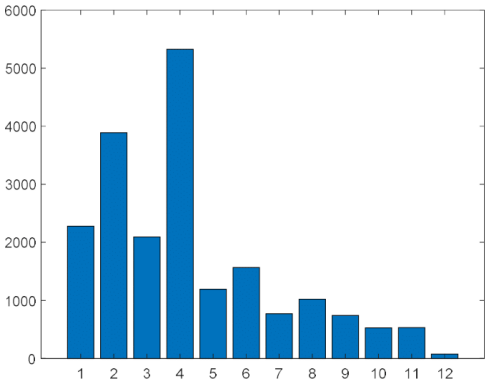
Figure 18: Pentomino Classes



(a) Districts of Size 5 in a 5 x 5 Grid



(b) Districts of Size 5 in a 9 x 9 Grid



(c) Districts of Size 5 in the Population of
All 4,006 Size 5 Squaretopia Partitions

Figure 19: Distribution of Single Districts by Class as Defined in Figure 18

Methods II

Weighted Partitioner

We modify the Partitioner algorithm to more accurately reproduce the known population distributions of compactness scores. We add a weighting factor to Partitioner, and name this modified algorithm "Weighted Partitioner." Greater values for the weighting factor, w, where $0 \leq w \leq 100$, increase the probability that Weighted Partitioner generates straighter, less compact districts. We hypothesize that this modification will allow us to generate samples that better represent the known population distributions.

We reproduce Figure 9 below in Figure 20. As in Partitioner, we start with an empty Squaretopia. In 20(a), the set of all free cells is $\{c_{1,1}, c_{1,2}, c_{1,3}, c_{1,4}, c_{1,5}, c_{2,1}, \ldots, c_{5,5}\}$. Weighted Partitioner also selects a cell uniformly at random from this set. In Figure 20(b), we see that Weighted Partitioner chose $c_{4,4}$ as the first cell in the first district. Now, consider the set of all free cells adjacent to $c_{4,4}$. The set of the incomplete district's neighboring cells is $\{c_{3,4}, c_{4,3}, c_{4,5}, c_{5,4}\}$. Weighted Partitioner also selects a cell uniformly at random from this set. In 20(c), we see that Partitioner chose $c_{3,4}$ as the first district's second cell. Partitioner removes $c_{3,4}$ from the set of neighboring cells, and adds $c_{2,4}, c_{3,3}$, and $c_{3,5}$. The set of neighboring cells is now $\{c_{4,3}, c_{4,5}, c_{5,4}, c_{2,4}, c_{3,3}, c_{3,5}\}$.

There is one difference between Partitioner and Weighted Partitioner. Weighted Partitioner does not choose uniformly at random from the set of neighboring cells. Weighted Partitioner "established a direction" when it chose $c_{3,4}$. The $c_{3,4}$ cell is above the first cell, $c_{4,4}$, so Weighted Partitioner will choose the cell above $c_{3,4}$ with the weighting factor probability of $w\%$. The remaining $(100 - w)\%$ is split equally among the other cells in the set. In Figure 20(c), the set of neighboring cells is $\{c_{4,3}, c_{4,5}, c_{5,4}, c_{2,4}, c_{3,3}, c_{3,5}\}$. Weighted Partitioner will choose $c_{2,4}$ with a $w\%$ probability, and any one of

the other 5 cells in the set will be chosen with a $\frac{100-w}{5}\%$ probability.

In Figure 20(d), we see that Weighted Partitioner chose $c_{2,4}$ as the next cell. The set of neighboring cells is now $\{c_{4,3}, c_{4,5}, c_{5,4}, c_{3,3}, c_{3,5}, c_{1,4}, c_{2,3}, c_{2,5}\}$. The third cell is above the second cell, so Weighted Partitioner has a $w\%$ probability of choosing the cell above the third cell. In other words, Weighted Partitioner will choose $c_{1,4}$ with a $w\%$ probability, and any one of the other 7 cells in the set will be chosen with a $\frac{100-w}{7}\%$ probability.

In Figure 20(e), we see that Weighted Partitioner chose $c_{2,5}$, not the weighted $c_{1,4}$ cell. The set of neighboring cells is now $\{c_{4,3}, c_{4,5}, c_{5,4}, c_{3,3}, c_{3,5}, c_{1,4}, c_{2,3}, c_{1,5}\}$. The fourth cell, $c_{2,5}$, is to the right of $c_{2,4}$, so Weighted Partitioner would like to weigh more heavily the cell to the right of $c_{2,5}$. However, the set of neighboring cells does not contain $c_{2,6}$, so Weighted Partitioner will instead select a cell uniformly at random from the set of neighboring cells. In (f), we see that Weighted Partitioner chose $c_{1,5}$. If the district needed six cells, the cell above $c_{1,5}$ would be the next weighted cell.

In summary, once a direction has been established, Weighted Partitioner will select the next cell in that direction with a weighting factor probability of $w\%$. Each of the other cells in the set of neighboring cells has a $\frac{100-w}{k-1}\%$ probability of being chosen, where k is the number of cells in the set of neighbors. If we have $k = 1$, then the one cell in the set of neighbors is chosen with certainty. If the set of neighboring cells does not contain the next sequential cell, then Weighted Partitioner chooses a cell uniformly at random. Besides this difference, Partitioner and Weighted Partitioner are the same.

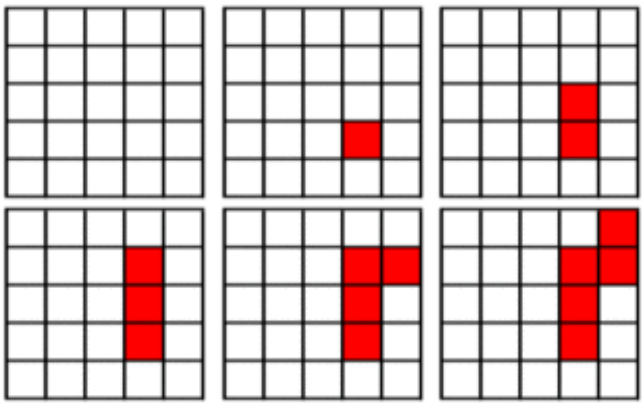


Figure 20: Weighted Partitioner Generates the First District

The Chi-Square Measure

Lastly, we define the chi-square (χ^2) measure to quantify how well the distribution of a sample's compactness scores approximates that of the population. For each compactness score, we first subtract the expected number of partitions from the observed number of partitions with that score, square the difference, and then divide by the expected number of partitions. The χ^2 measure equals the sum of all these values. A lower χ^2 value means that a sample's distribution of compactness scores better approximates the population's distribution of compactness scores.

Definition 6 (χ^2 Measure)**.** The χ^2 measure quantifies the discrepancy between the distribution of a sample's compactness scores and that of the population. We define $\chi^2 = \sum_{i=0}^{100} \frac{(O_i - E_i)^2}{E_i}$ where O_i is the number of partitions with a compactness score of i in a sample of size n and $E_i = n(\frac{\text{number of partitions with a score of } i \text{ in the population}}{\text{number of partitions in the population}})$. If the number of partitions with a score of i in the population is 0, i.e., $E_i = 0$, we let $\frac{(O_i - E_i)^2}{E_i} = 0$.

For example, there are 4,006 possible partitions of a size 5 Squaretopia. Out of these 4,006 partitions, 580 have a Length-Width score of 67. If we generate a sample of 10,000 partitions, we hope to generate $10,000(\frac{580}{4,006}) \approx 1,448$ partitions with a Length-Width score of 67. However, in a sample of 10,000 partitions, we may generate 1,400 partitions with a Length-Width score of 67. In this case, we calculate a value of $\frac{(1400-1448)^2}{1448} \approx 1.59$ for the compactness score of 67. To calculate the χ^2 value, we perform a similar calculation for all 101 possible compactness scores and then find the sum of all these calculations.

To quantify how well the distribution of a sample's compactness scores approximates that of the population, we concentrate our analysis on the χ^2 measure instead of the max difference measure. The χ^2 measure quantifies the difference at each possible compactness score, while the max difference measure only captures the largest difference at any one compactness score. Our goal is to generate distributions that best approximate those of populations, so we want to incorporate the information available at each compactness score.

Results II

Weighted Samples

We use Weighted Partitioner and vary the weighting factor to generate samples of partitions of size 4, 5, 6, and 7 Squaretopias. In Figure 21, we graph the cumulative relative frequency distributions of the Length-Width scores of the size 7 Squaretopia partitions in the weighted samples that Weighted Partitioner generates with weighting factor values of 0, 25, 50, 75, and 100. The figure also includes the cumulative distribution of the population and the cumulative distribution of the sample from Figure 17. These two cumulative distributions are labeled as "Population" and "Unweighted," respectively, in Figure 21. Using the same five weighted samples of a size 7 Squaretopia and the unweighted samples from Results I, we create similar graphs for the Reock, Schwartzberg, and Polsby-Popper scores. Figures 22 - 24 show these graphs. These four graphs and equivalent sets of graphs for Squaretopias of sizes 4, 5, and 6 are included in Appendix C.

In general, as we hypothesized, we find that a weighting factor of 100 generates a sample whose cumulative relative frequency distributions lies to the left of both that of the population and that of the unweighted sample we generated in Results I. We expected that a high weighting factor would generate incompact partitions with increased probability. We observe the opposite for a low weighting factor. A weighting factor of 0 generates a sample whose cumulative relative frequency distribution lies to the right of both that of the population and that of the unweighted sample we generated in Results I. The cumulative distribution of Weighted Partitioner's sample shifts to the left as we increase the weighting factor.

We note the subtle distinction between a weighting factor of 0 and the term, "unweighted." Partitioner always selects cells from the set of neighbors uniformly at random, whereas a weighting factor of 0 prevents Weighted

Partitioner from choosing the next sequential state from the set of neighbors.

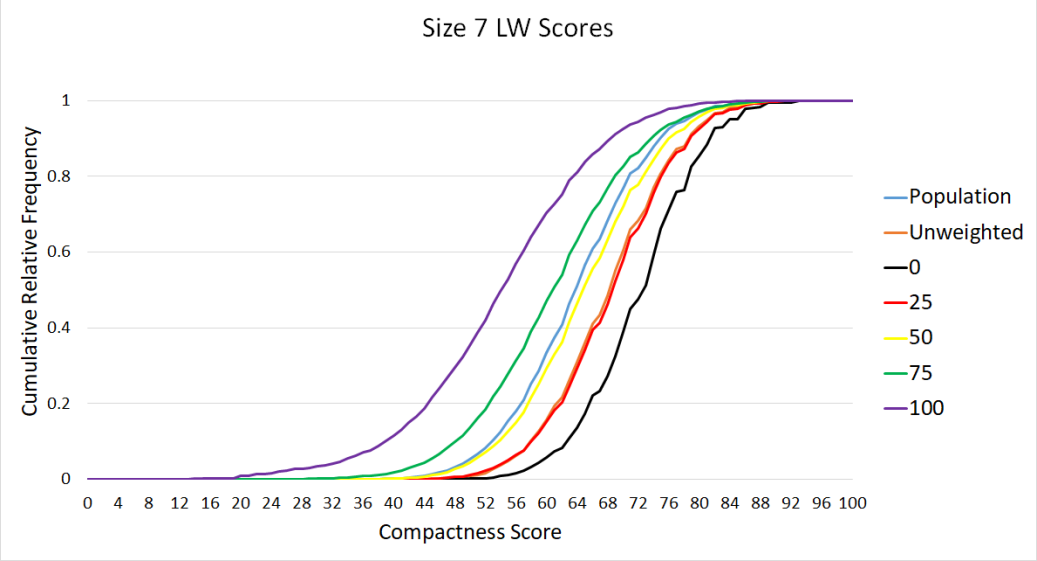


Figure 21: Size 7 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

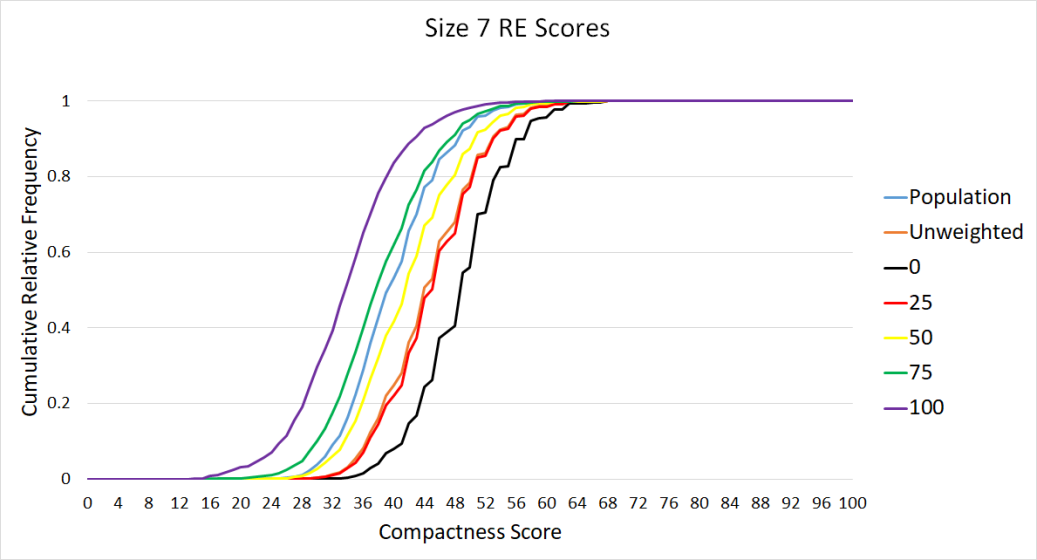


Figure 22: Size 7 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

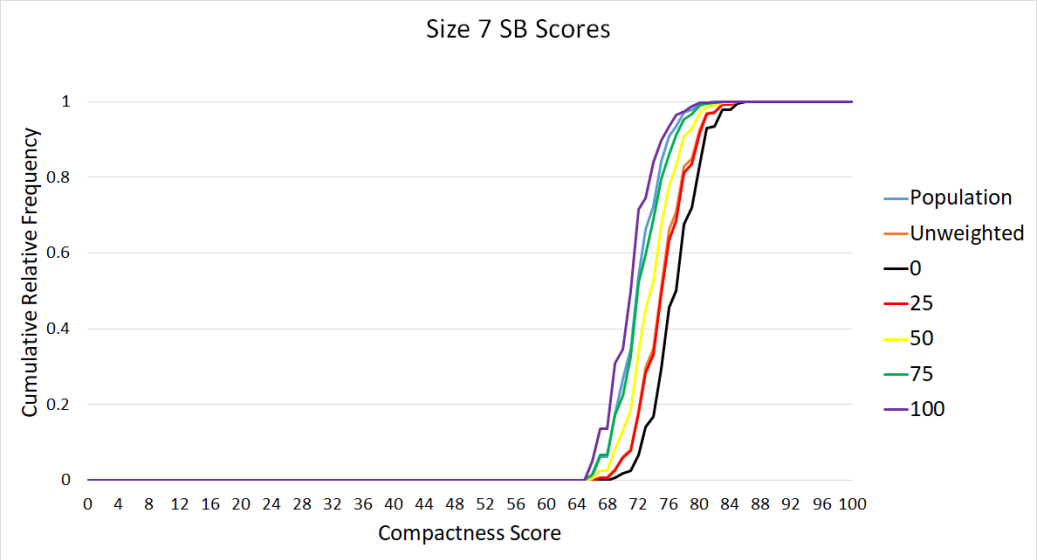


Figure 23: Size 7 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

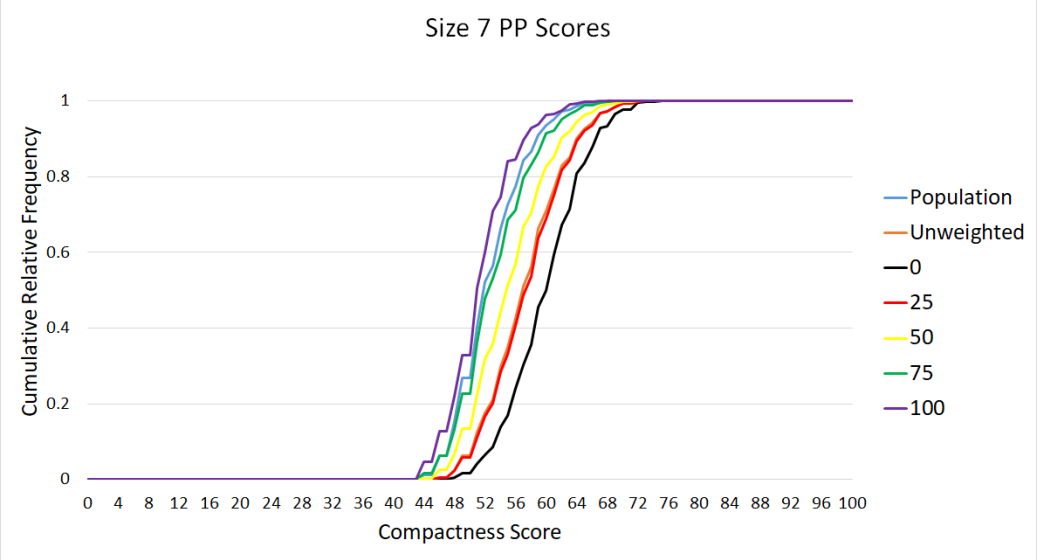


Figure 24: Size 7 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

We generate more samples with different weights and calculate the χ^2

value of each sample. In Figures 25 - 28, we plot the χ^2 value vs. the weighting factor for the four scoring methods. Each figure shows how the χ^2 value varies for the four Squaretopia sizes.

We note the parabolic shape of the graphs in the figures for the Length-Width and Reock scores. In the Length-Width figure, the minimum χ^2 values of all four "parabolas" occur between weighting factors of 45 and 60. Weighting factors within this interval will best approximate the populations' distributions of Length-Width compactness scores. In the Reock figure, the minimum χ^2 values of all four "parabolas" occur between weighting factors of 50 and 60. Weighting factors within this interval will best approximate the populations' distributions of Reock compactness scores. The Schwartzberg and Polsby-Popper figures do not display such a clear relationship between weighting factors and χ^2 values. Using the max difference measure instead of the χ^2 measure results in a similar analysis. The max difference vs. weighting factor graphs for the same samples are included in Appendix D.

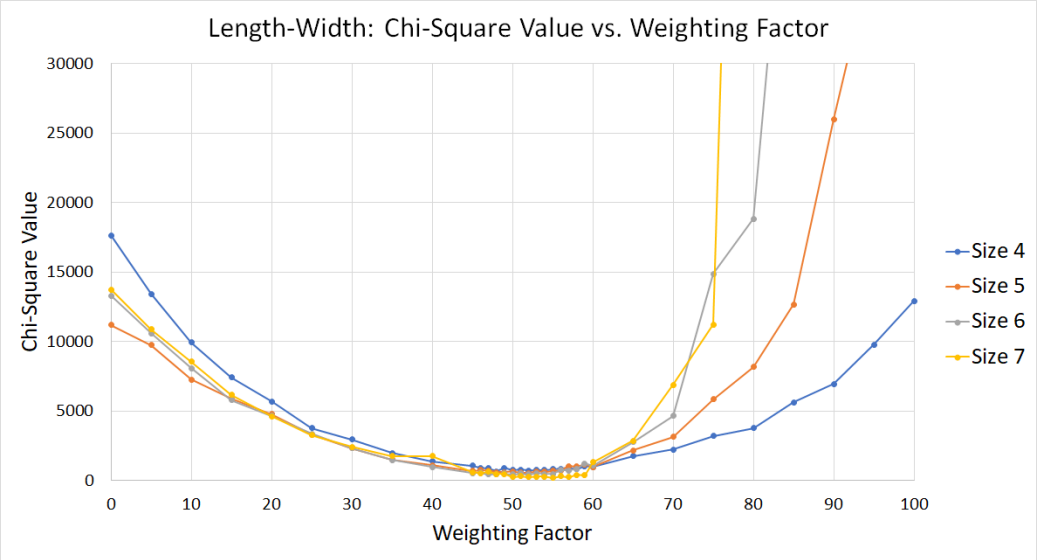


Figure 25: LW Chi-Square Value vs. Weighting Factor (Calculated from Samples of 10,000 Partitions of Size 4, 5, 6, and 7 Squaretopias Generated From Various Weighting Factors)

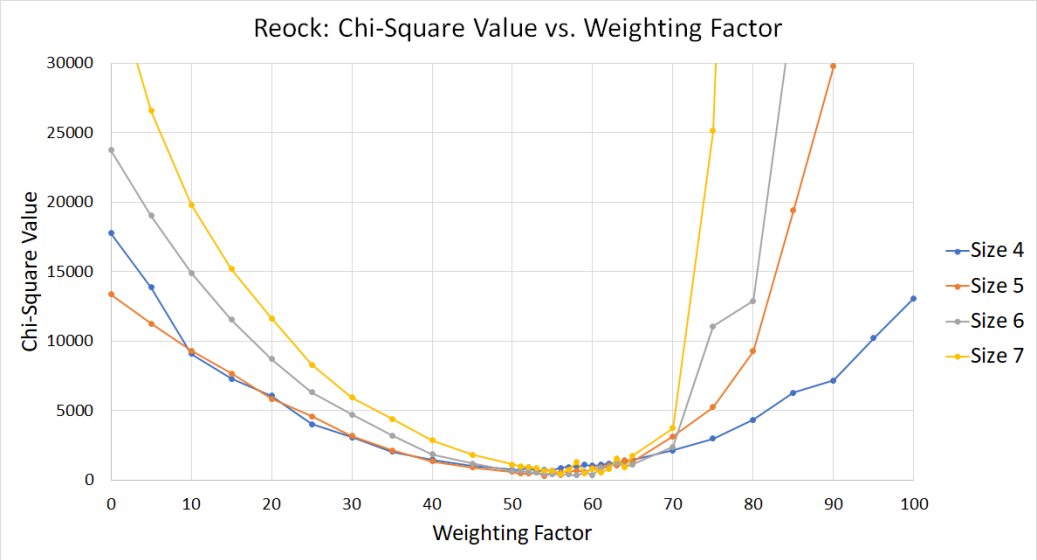


Figure 26: RE Chi-Square Value vs. Weighting Factor (Calculated from Samples of 10,000 Partitions of Size 4, 5, 6, and 7 Squaretopias Generated From Various Weighting Factors)

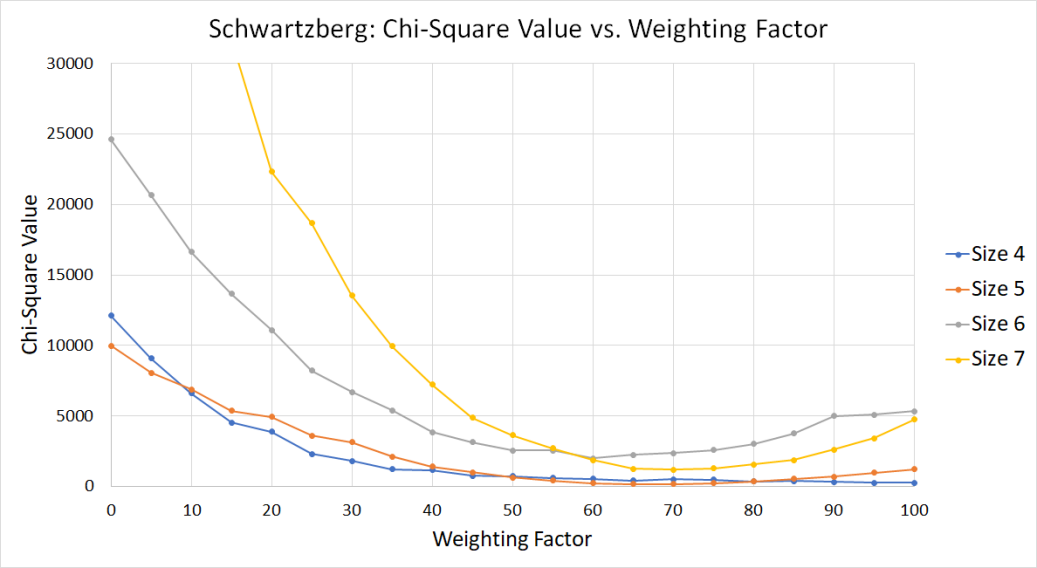


Figure 27: SB Chi-Square Value vs. Weighting Factor (Calculated from Samples of 10,000 Partitions of Size 4, 5, 6, and 7 Squaretopias Generated From Various Weighting Factors)

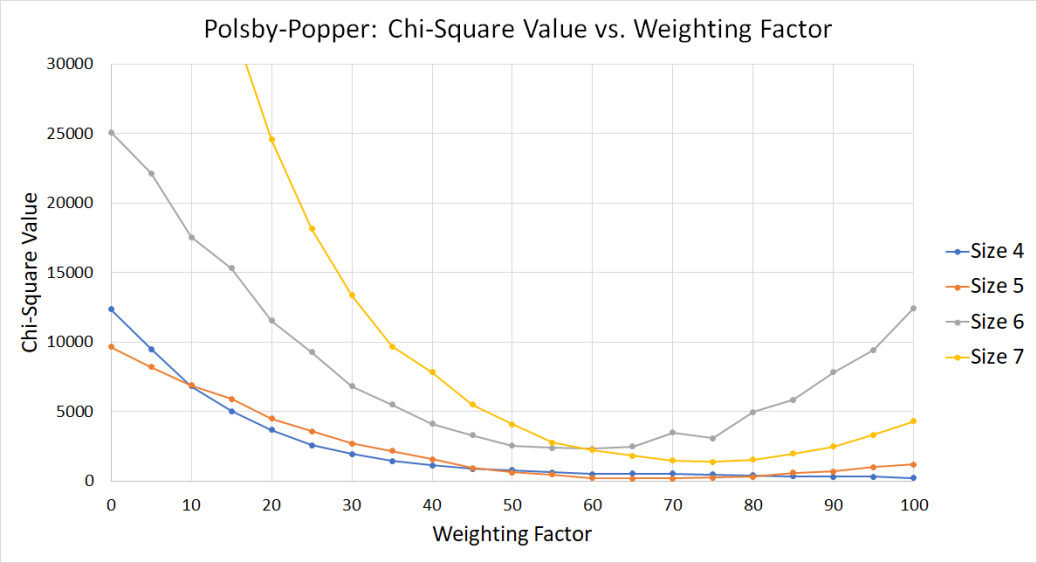


Figure 28: PP Chi-Square Value vs. Weighting Factor (Calculated from Samples of 10,000 Partitions of Size 4, 5, 6, and 7 Squaretopias Generated From Various Weighting Factors)

Extrapolation

Having established that there is an interval which contains weighting factors that generate samples that closely approximate the known populations of Length-Width and Reock scores, we assume that this will also be true for larger Squaretopias. Using weighting factors that lie in the χ^2-minimizing intervals, we approximate the distribution of the Length-Width and Reock compactness scores of Squaretopias of sizes 8, 9, 10, and 11. Although statistical noise prevents us from identifying the exact value of the best weighting factor, using any weighting factor that lies in the χ^2-minimizing interval should generate a good approximation of the known population distributions of Length-Width and Reock compactness scores. For the rest of our analysis, we use a weighting factor of 52 for the Length-Width scoring method and a weighing factor of 56 for the Reock scoring method. In the Length-Width and Reock graphs (Figures 76 and 77) of Appendix D, we see that these weighting factors also minimize the max difference for the four Squaretopia sizes. The graphs in Appendix E show the cumulative distributions of Weighted Partitioner's samples generated by using our chosen weighting

factors. These samples are for Squaretopias of sizes 4, 5, 6, and 7. We also include the cumulative distributions of populations and of the unweighted samples seen in Appendix B's graphs.

In Figure 29, we use a weighting factor of 52 to generate samples of 10,000 partitions of Squaretopias of sizes 8, 9, 10, and 11. In Figure 30, we use a weighting factor of 56 to generate samples of 10,000 partitions of Squaretopias of sizes 8, 9, 10, and 11. In each graph, we see the cumulative relative frequency distributions of the four samples' Length-Width or Reock scores for that weighting factor.

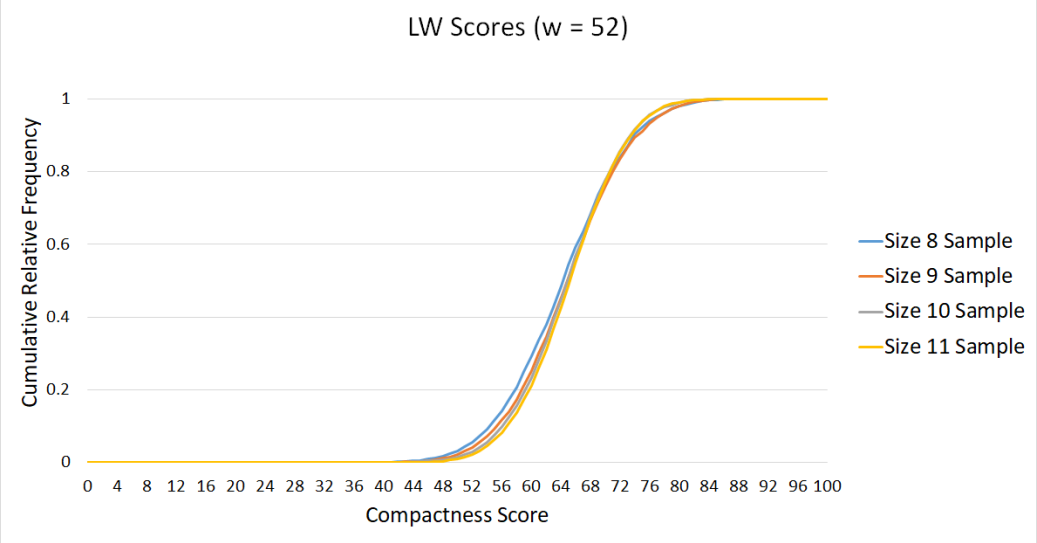


Figure 29: Extrapolated Cumul. Rel. Freq. Dist. of the LW Scores of Size 8, 9, 10, and 11 Squaretopias Using a Weighing Factor of 52

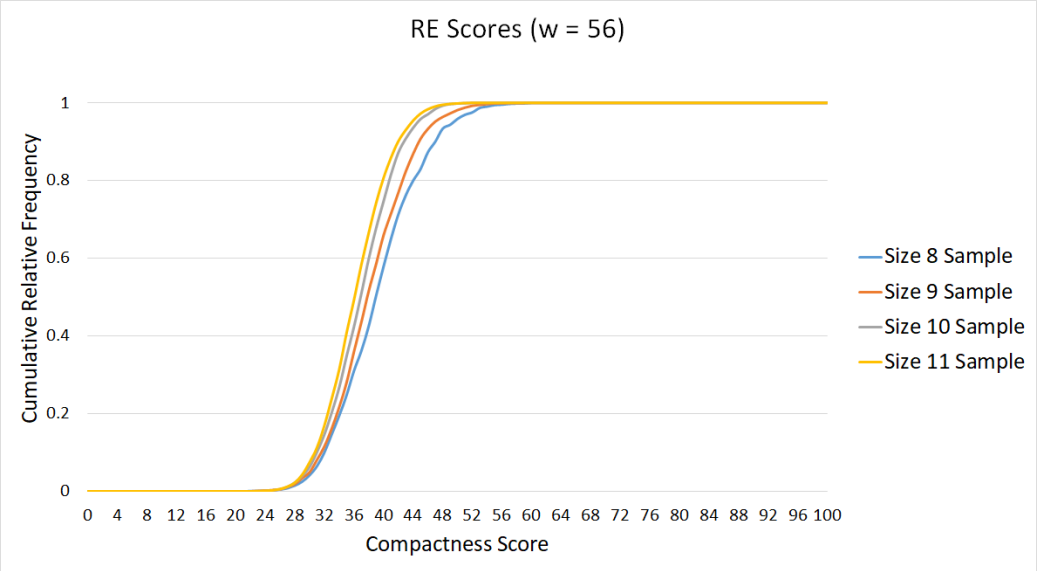


Figure 30: Extrapolated Cumul. Rel. Freq. Dist. of the RE Scores of Size 8, 9, 10, and 11 Squaretopias Using a Weighing Factor of 56

Detecting a Gerrymandered Partition

Suppose that a state legislature or commission proposes a size 11 Squaretopia partition such as the one seen in Figure 31. The partition has a Length-Width score of 54. Is the partition sufficiently compact?

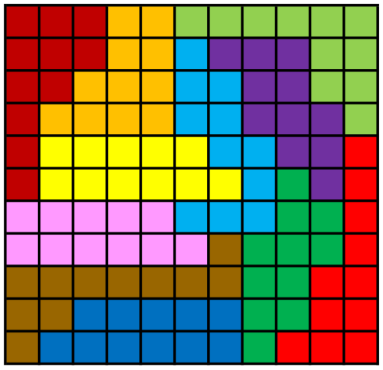


Figure 31: A Proposed Partition of a Size 11 Squaretopia

We use Weighted Partitioner and a weighting factor of 52 to generate a

sample of 10,000 size 11 Squaretopia partitions. Figure 32 shows the frequency of each compactness score in the sample.

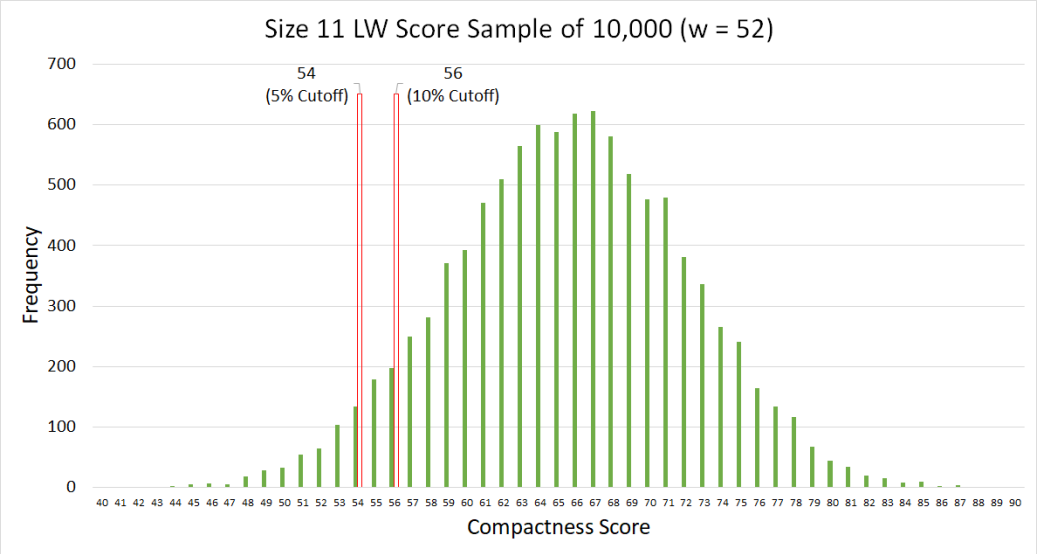


Figure 32: Absolute Frequencies of LW Compactness Scores in a 52-Weighted Sample of 10,000 Partitions of a Size 11 Squaretopia

Figure 32 also shows the 5% and 10% cutoffs. Any scores to the left of a cutoff would be considered significantly incompact and wouldn't be acceptable. We find that a score of 54 lies in the group of 4.56% least compact scores. This means that out of all possible ways to partition Squaretopia, it is unlikely to find such a small compactness score by chance. This unlikeliness would lend evidence to the claim that the districts were purposefully drawn to be incompact. It's important to realize that a low compactness score doesn't always prove that there is intentional gerrymandering, but it does flag a proposed partition as highly suspect. While we cannot use Weighted Partitioner's samples to determine whether a given partition is gerrymandered, sampling compactness scores does give us an idea of how compact we should expect most partitions to be.

Unfortunately, data on the Reock scores of U.S. congressional districts differ from our estimated distributions of the compactness scores of the larger Squaretopias. Ansolabehere and Palmer analyze shapefile data to score every U.S. congressional district from the first Congress to the 2010 redistricting cycle, excluding single-district states [Ansolabehere and Palmer, 2016]. They

use the Reock scoring method but not the Length-Width. Table 5 contains the descriptive statistics that Ansolabehere and Palmer provide for the Reock scoring method. Using these statistics and assuming a normal distribution for the Reock scores of the 34,996 congressional districts, we estimate the cumulative relative frequency distribution of the Reock compactness scores of all U.S. congressional districts. Figure 33 shows our estimate and the extrapolated cumulative distributions from Figure 30. U.S. congressional districts have Reock scores over a wider interval and have a distribution with heavier tails. Figure 34 compares the scores of U.S. congressional districts with the scores of single districts in these Squaretopias. We see that U.S. congressional districts have a distribution of Reock scores that is relatively more compact than those of the single districts in these Squaretopias.

		Percentile				
Mean	Standard Deviation	10%	25%	50%	75%	90%
0.405	0.110	0.260	0.326	0.408	0.481	0.547

Table 5: Descriptive Statistics for the Reock Scores of U.S. Congressional Districts

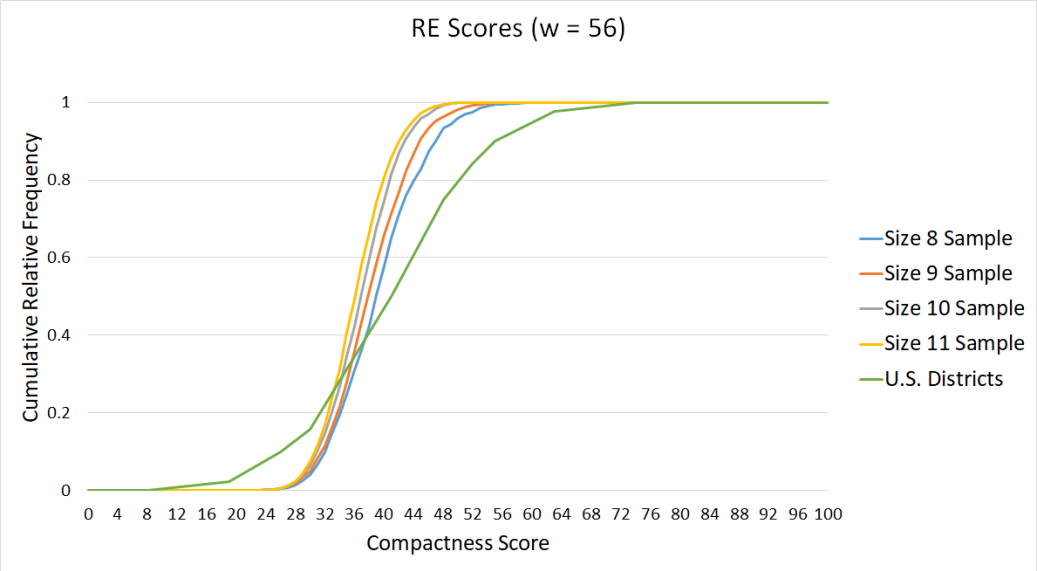


Figure 33: Cumul. Rel. Freq. Dist. of the RE Scores of U.S. Congressional Districts and Extrapolated Cumul. Rel. Freq. Dist. of the RE Scores of Size 8, 9, 10, and 11 Squaretopias Using a Weighing Factor of 56

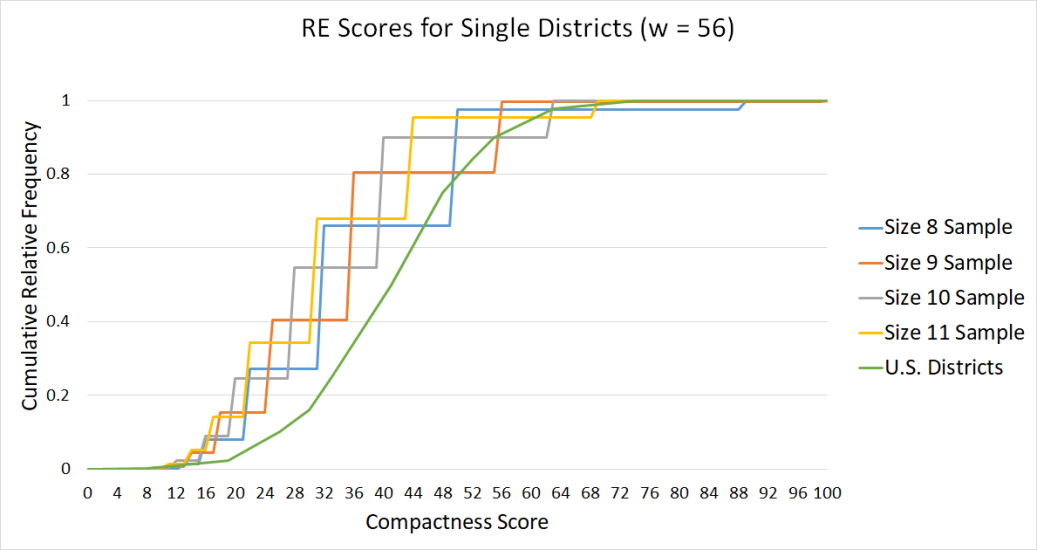


Figure 34: Cumul. Rel. Freq. Dist. of the RE Scores of U.S. Congressional Districts and Extrapolated Cumul. Rel. Freq. Dist. of the RE Scores of Single Districts in Size 8, 9, 10, and 11 Squaretopias Using a Weighing Factor of 56

Discussion

The inability to generate the population of partitions of a size n Squaretopia is not the only weakness of our approach. As the size of Squaretopia increases, a recursive method requires more time to find a valid partition. Generating 10,000 partitions of a size 7 Squaretopia takes us less than a minute, while generating 10,000 partitions of a size 11 Squaretopia takes us about an hour. For large enough Squaretopias, our recursive algorithm may fail to produce a valid partition because the recursion tree eventually becomes so massive that our computers run out of memory. Our recursive algorithm becomes intractable for Squaretopias larger than size 15. In addition, no compactness scoring method is flawless. Compactness scores may inaccurately convey the quality of districts. For example, despite being incompact, the red 17-omino in Figure 35 has a perfect Length-Width score of 1. Although the blue 49-omino meanders like a snake, the 49-omino has a relatively compact Reock score of about 0.60.

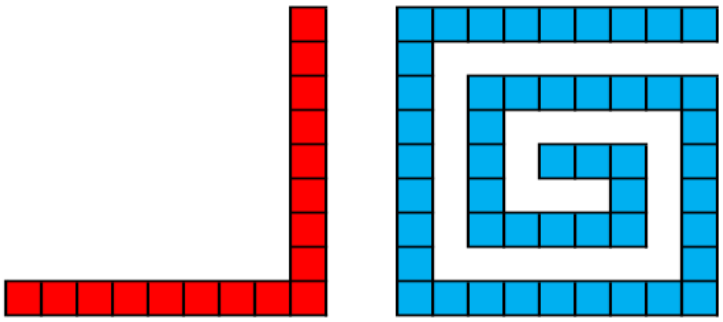


Figure 35: A 17-omino and a 49-omino

Moreover, by defining the compactness score of a partition as the average compactness score of the partition's districts, we have given state legislatures

and commissions the opportunity to hide a few incompact districts among mostly compact districts. One might propose defining the compactness score of a partition as the lowest compactness score of the partition's districts. While we cannot think of a reason for prohibiting this definition, we see that measuring compactness offers much room for debate.

Young concludes that "compactness is such a hazy and ill-defined concept that it seems impossible to apply it, in any rigorous sense, to matters of law" [Young, 1988]. Young states that a reliance on formulas has the semblance, but not the substance, of justice. A focus on formulas allows computers to design gerrymandered redistricting plans that satisfy mathematical criteria. Thus, we must not only prioritize compactness if we hope to prevent gerrymandering.

Further Research

We used four scoring methods in this paper. Implementing other scoring methods such as the Convex Hull, X-Symmetry, and the Perimeter Test can produce an informative extension of our research. Removing one or a combination of the conditions of our Squaretopia partitioning problem will also yield new extensions. For instance, could we partition an m x n grid where m and n are two positive integers such that $m \neq n$? Or, what if each Squaretopia cell no longer contains an equal number of Squaretopia citizens? Could we include bodies of water in Squaretopia? Cells representing bodies of water would not be assigned to any district. What weighted and unweighted distributions of compactness scores will these conditions produce? Lastly, we invite the reader to improve the code and algorithms for Partitioner and Weighted Partitioner.

Acknowledgements

I would like to thank Dr. Robert Rovetti of the Loyola Marymount University Mathematics Department. This project would not have been possible without his constant encouragement and invaluable guidance. I would also like to thank Dr. Andrew Forney of the Loyola Marymount University Computer Science Department. Dr. Forney's Algorithms lectures helped me develop the programming skills necessary for this project.

Bibliography

[Ansolabehere and Palmer, 2016] Ansolabehere, S. and Palmer, M. (2016). A Two-Hundred Year Statistical History of the Gerrymander. *Ohio State Law Journal*, 77(4):741–762.

[Curtis C., 1995] Curtis C., Harris, J. (1995). A Scientific Method of Districting. *Behavioral science*, 40(4):9.

[Davis, 2017] Davis, J. (2017). The Gerrymander: a New Species of Monster. *Library of Congress Newspaper*.

[Duchin, 2019] Duchin, M. (2019). Geometry v. Gerrymandering. *The Best Writing on Mathematics 2019*, pages 1–11.

[Esri, 2013] Esri (2013).

[Gillman, 2002] Gillman, R. (2002). Geometry and Gerrymandering. *Math Horizons*, 10(1):10–22.

[Levitt, 2020] Levitt, J. (2020). *All About Redistricting*.

[Polsby and Popper, 1991] Polsby, D. D. and Popper, R. D. (1991). The Third Criterion: Compactness as a Procedural Safeguard against Partisan Gerrymandering. *Yale Law & Policy Review*, 9(2):301–353.

[Redelmeier, 1981] Redelmeier, D. (1981). Counting Polyominoes: Yet Another Attack. *Discrete Mathematics*, 36(2):191–203.

[Reock, 1961] Reock, E. C. (1961). A Note: Measuring Compactness as a Requirement of Legislative Apportionment. *Midwest Journal of Political Science*, 5(1):70–74.

[Schwartzberg, 1966] Schwartzberg, J. E. (1966). Reapportionment, Gerrymanders, and the Notion of Compactness. *Minnesota Law Review*, 50:443.

[Young, 1988] Young, H. P. (1988). Measuring the Compactness of Legislative Districts. *Legislative Studies Quarterly*, 13(1):105–115.

Appendix A

We used Partitioner to generate 14 independent samples. For a Squaretopia of size 6, we generated samples of five hundred, one thousand, three thousand, five thousand, seven thousand, nine thousand, and ten thousand partitions. We did the same for a Squaretopia of size 7. Figure 36 shows the plotted cumulative relative frequency distributions of the Length-Width scores of six samples of size 6 Squaretopia partitions. "LW 500" refers to the LW score cumulative distribution of the sample of five hundred partitions. Note that the ranges of the figures' horizontal axes differ.

Figure 36: Size 6 Squaretopia Cumul. Rel. Freq. Dist. of LW Scores of Multi-Sized Samples

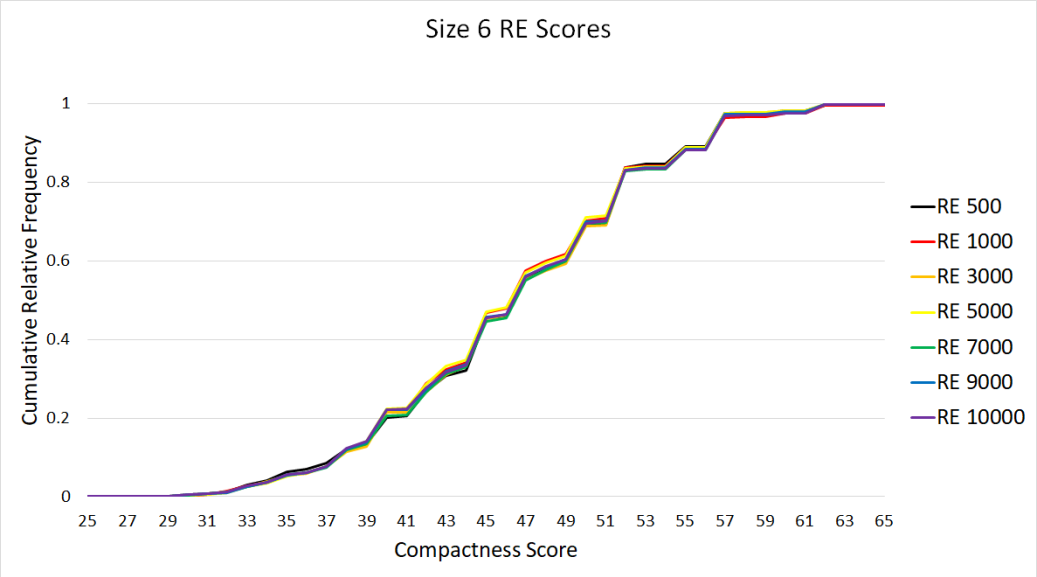


Figure 37: Size 6 Squaretopia Cumul. Rel. Freq. Dist. of RE Scores of Multi-Sized Samples

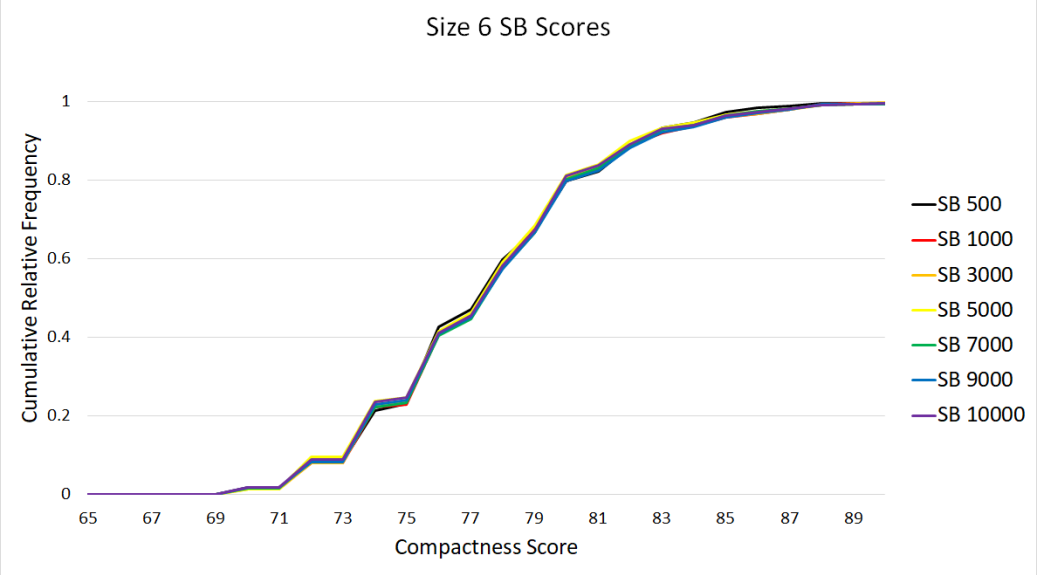


Figure 38: Size 6 Squaretopia Cumul. Rel. Freq. Dist. of SB Scores of Multi-Sized Samples

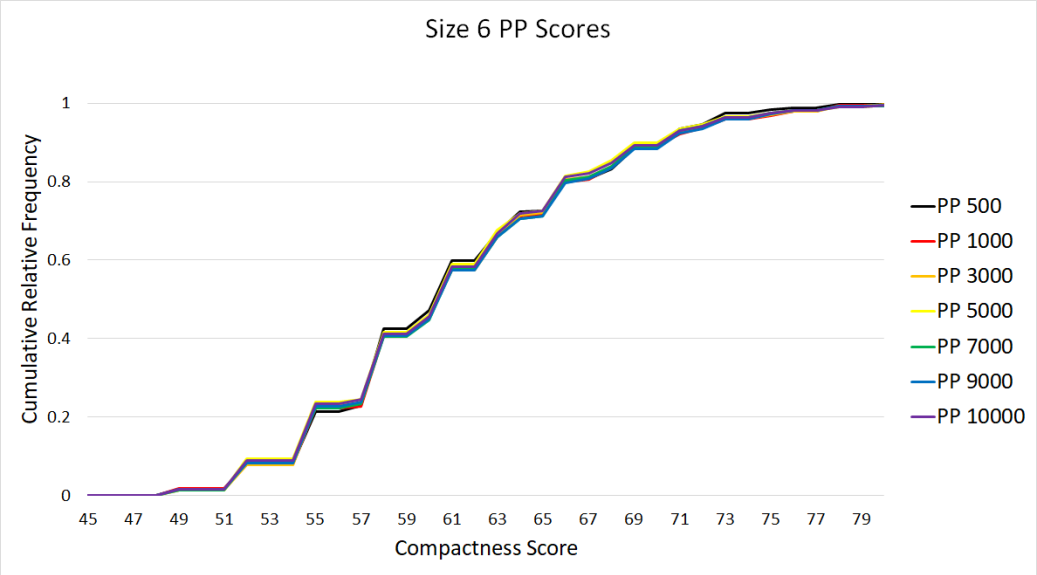


Figure 39: Size 6 Squaretopia Cumul. Rel. Freq. Dist. of PP Scores of Multi-Sized Samples

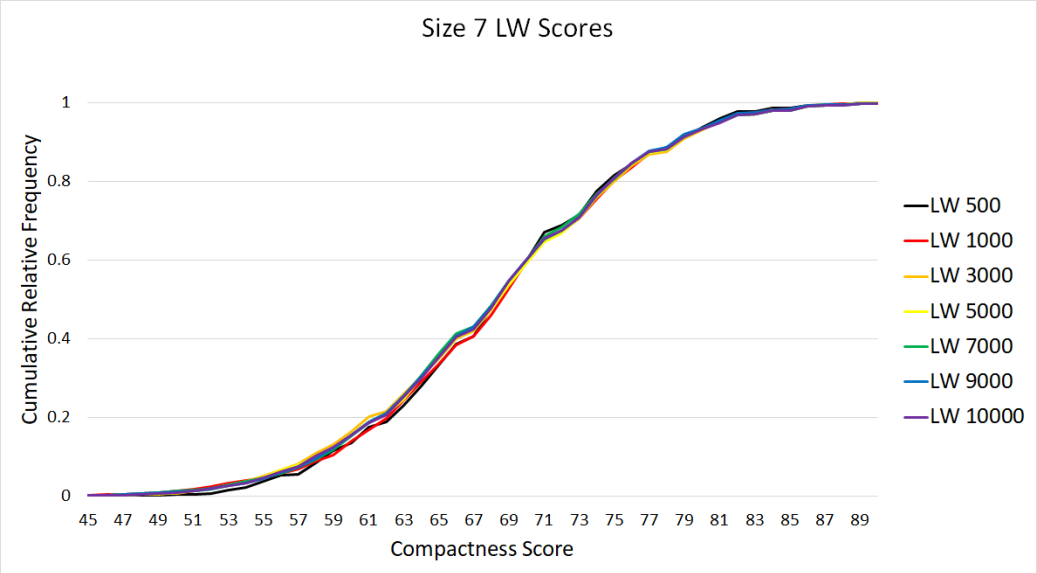


Figure 40: Size 7 Squaretopia Cumul. Rel. Freq. Dist. of LW Scores of Multi-Sized Samples

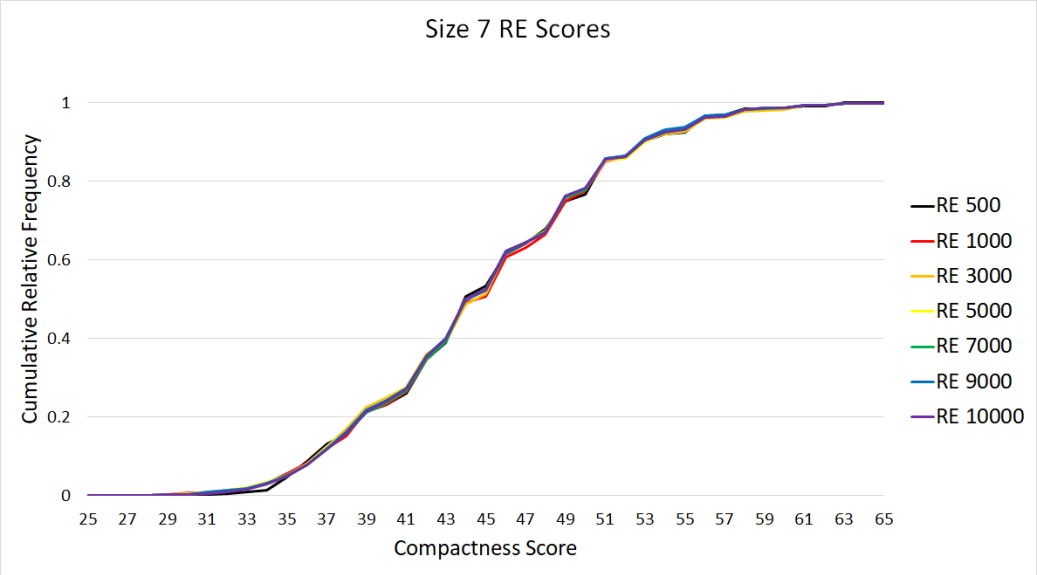


Figure 41: Size 7 Squaretopia Cumul. Rel. Freq. Dist. of RE Scores of Multi-Sized Samples

Figure 42: Size 7 Squaretopia Cumul. Rel. Freq. Dist. of SB Scores of Multi-Sized Samples

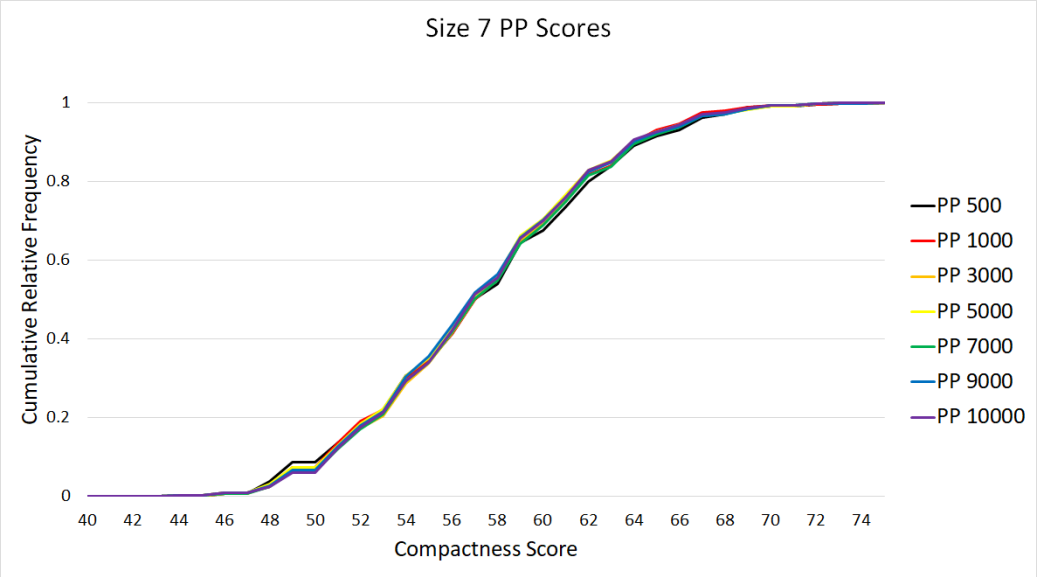


Figure 43: Size 7 Squaretopia Cumul. Rel. Freq. Dist. of PP Scores of Multi-Sized Samples

Appendix B

We generate four samples of ten thousand partitions of Squaretopias of sizes 4, 5, 6, and 7. We have ten thousand partitions for each Squaretopia size, and each partition has four scores. Dr. Robert Rovetti has scored the populations of partitions of Squaretopias of these sizes. We graph the cumulative relative frequency distributions of the scores of the populations and of the generated samples. All horizontal axes range from 0 to 100.

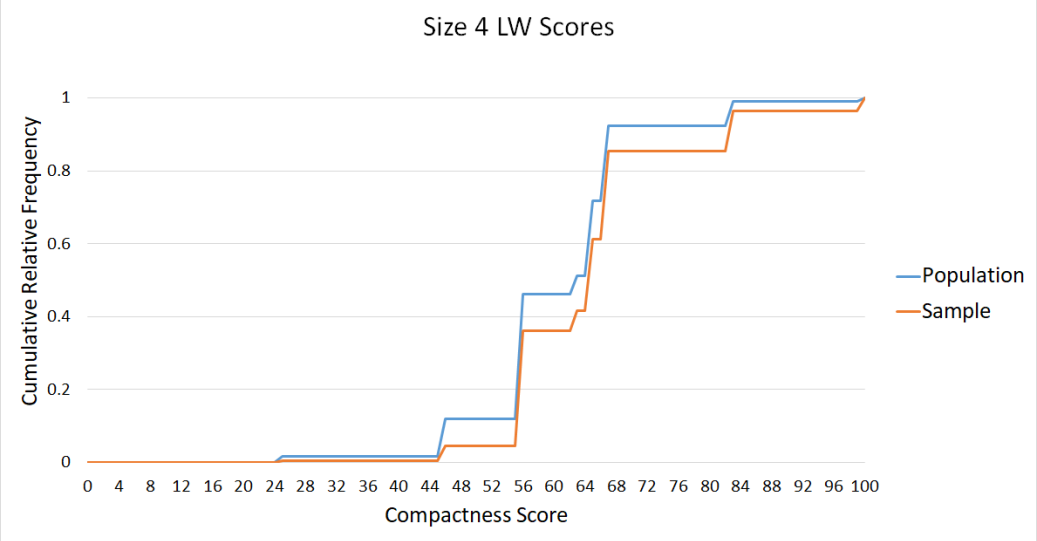


Figure 44: Size 4 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

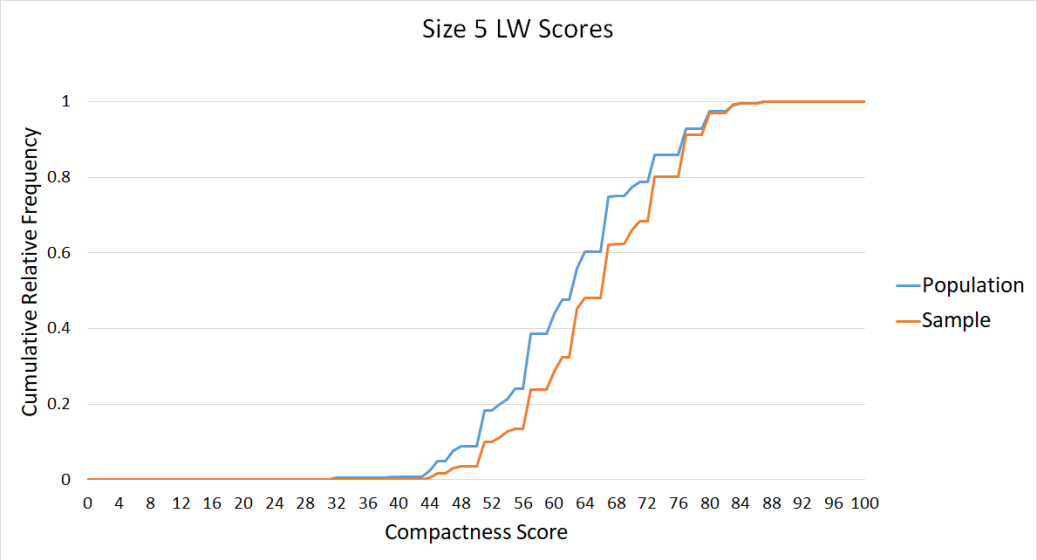


Figure 45: Size 5 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

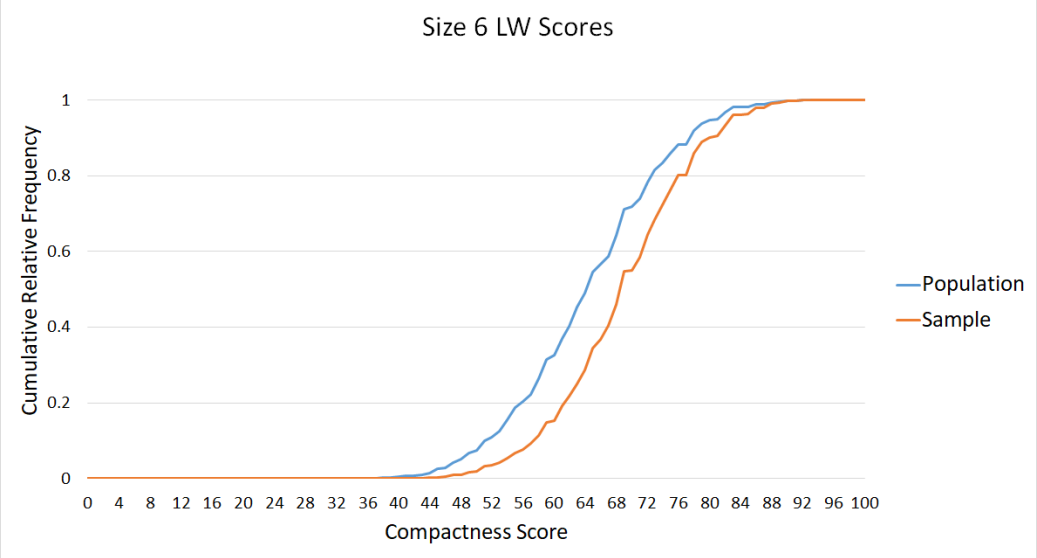


Figure 46: Size 6 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

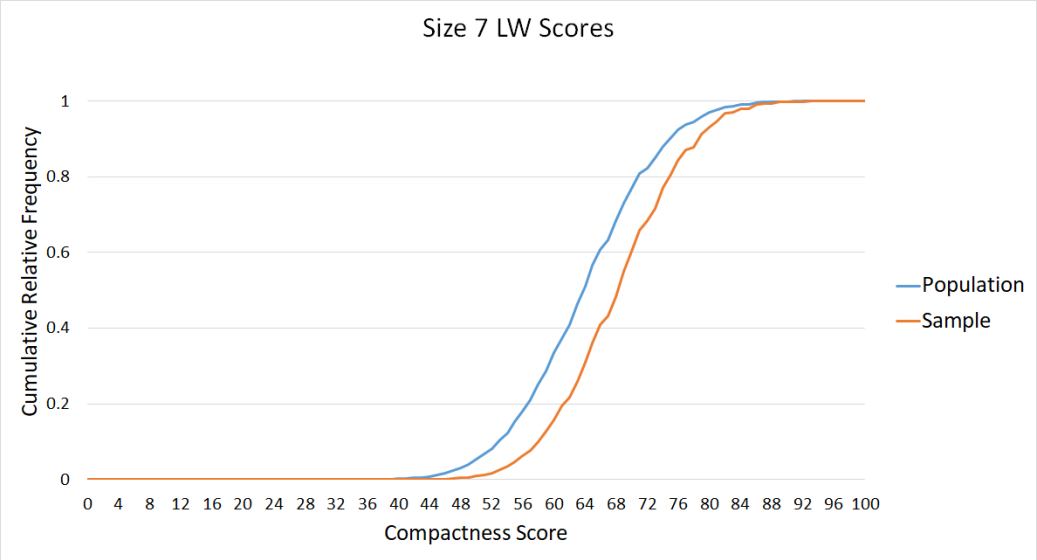


Figure 47: Size 7 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

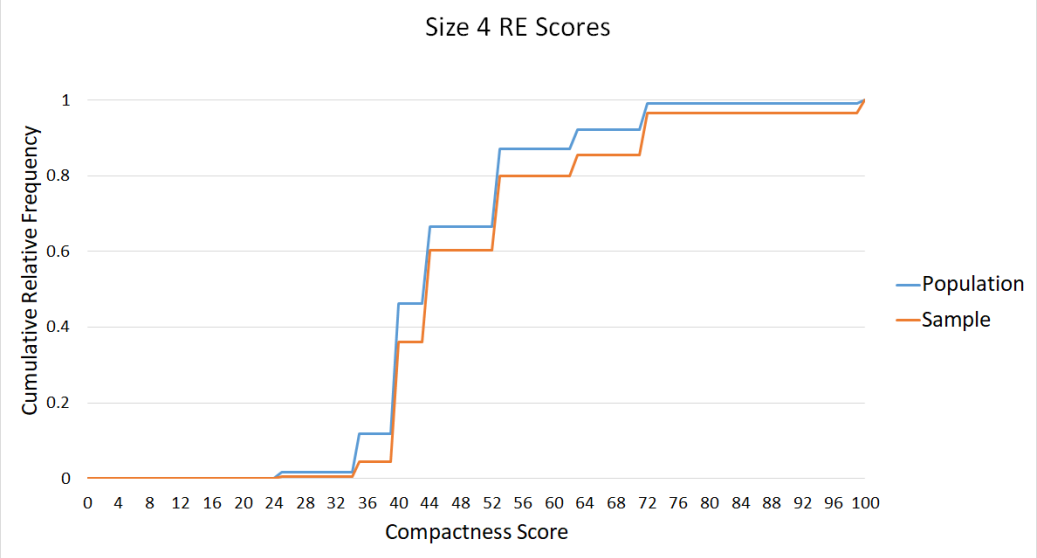


Figure 48: Size 4 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

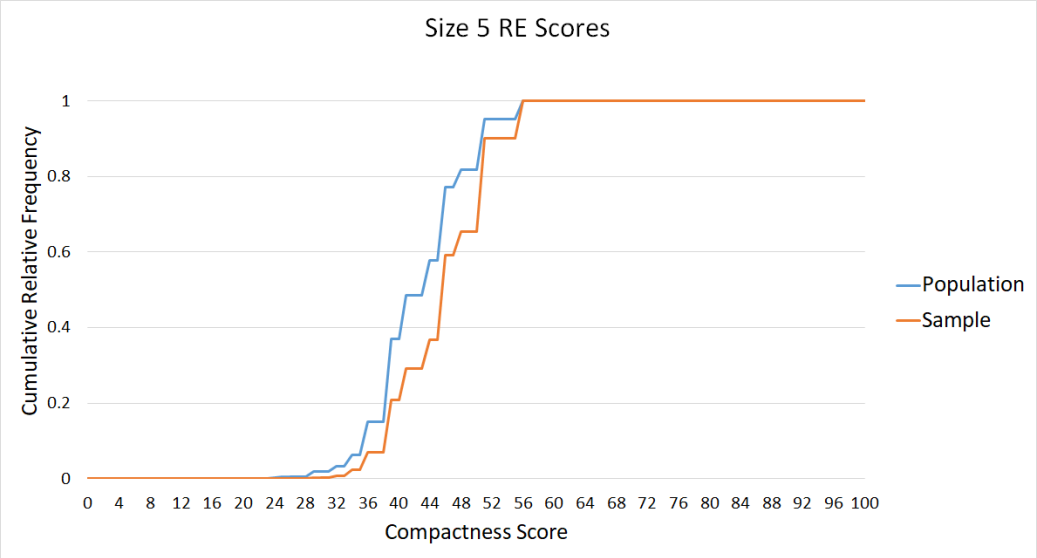


Figure 49: Size 5 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

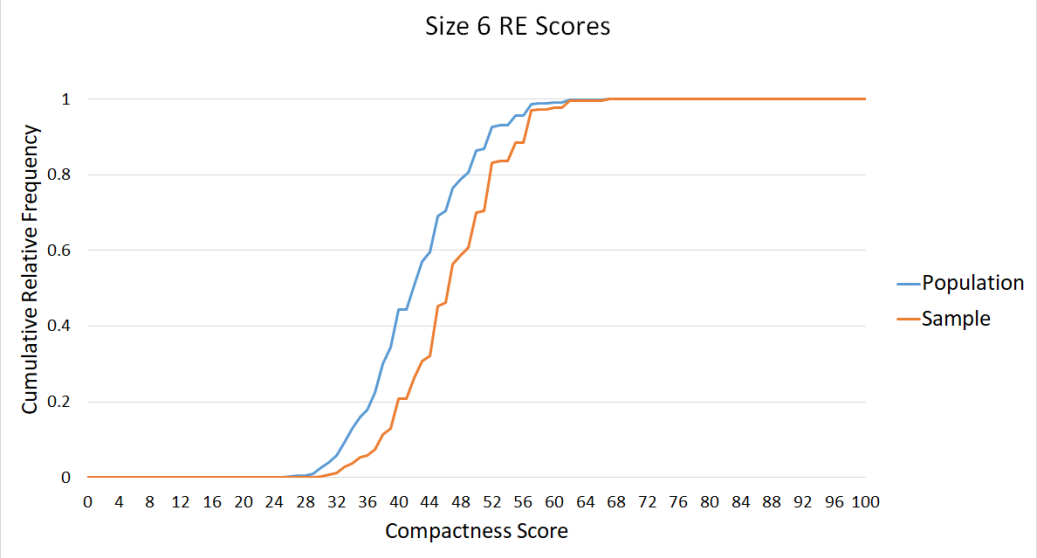


Figure 50: Size 6 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

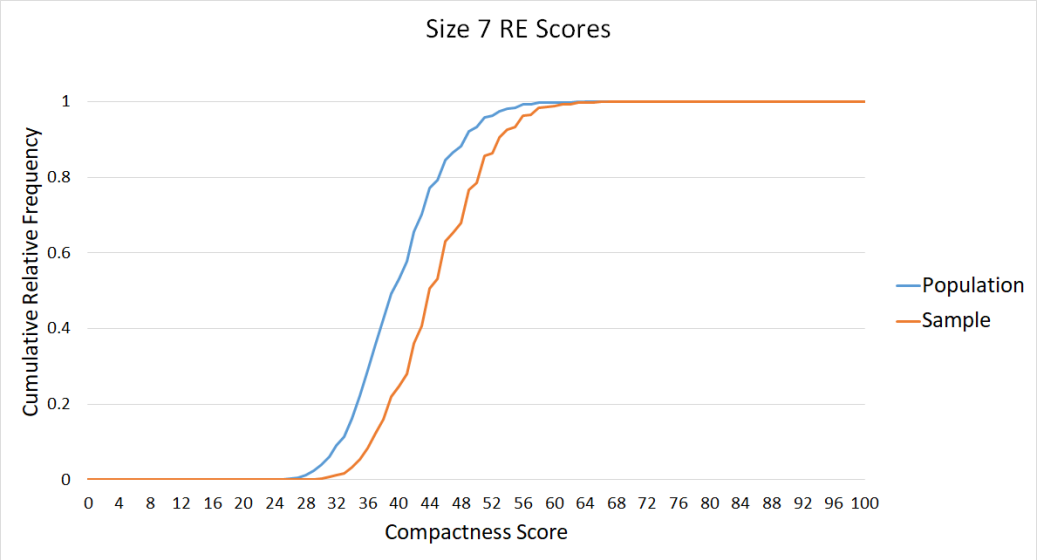


Figure 51: Size 7 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

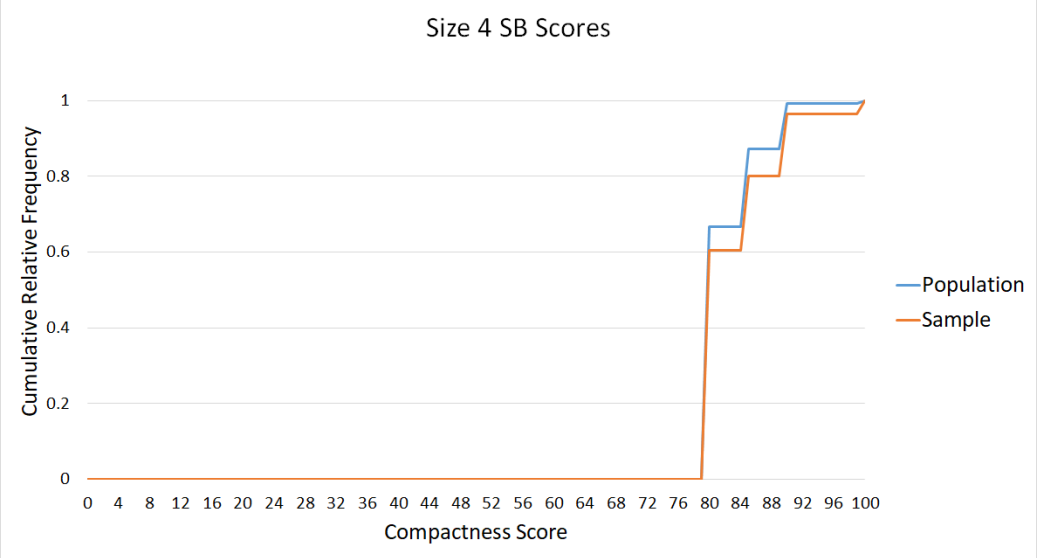


Figure 52: Size 4 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

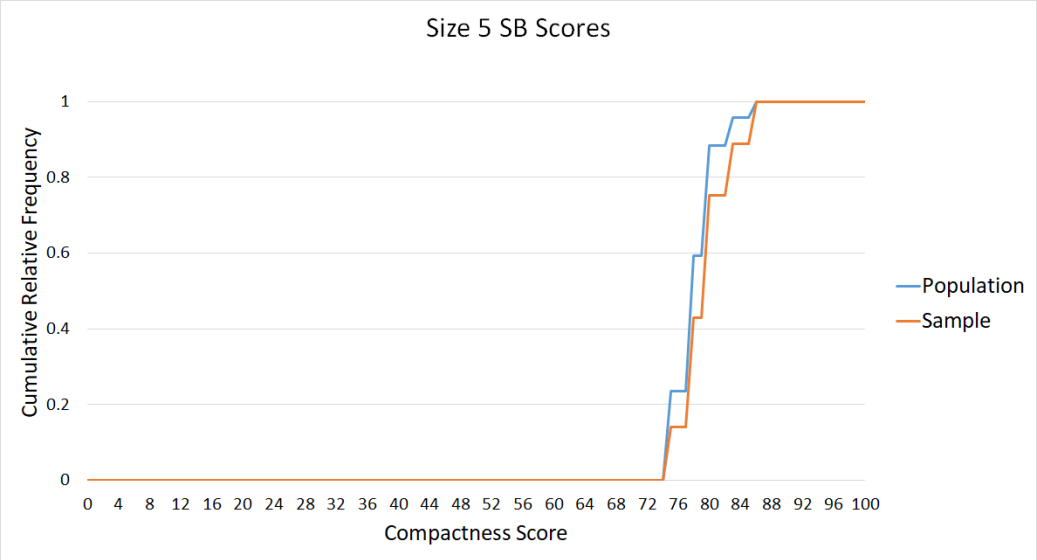


Figure 53: Size 5 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

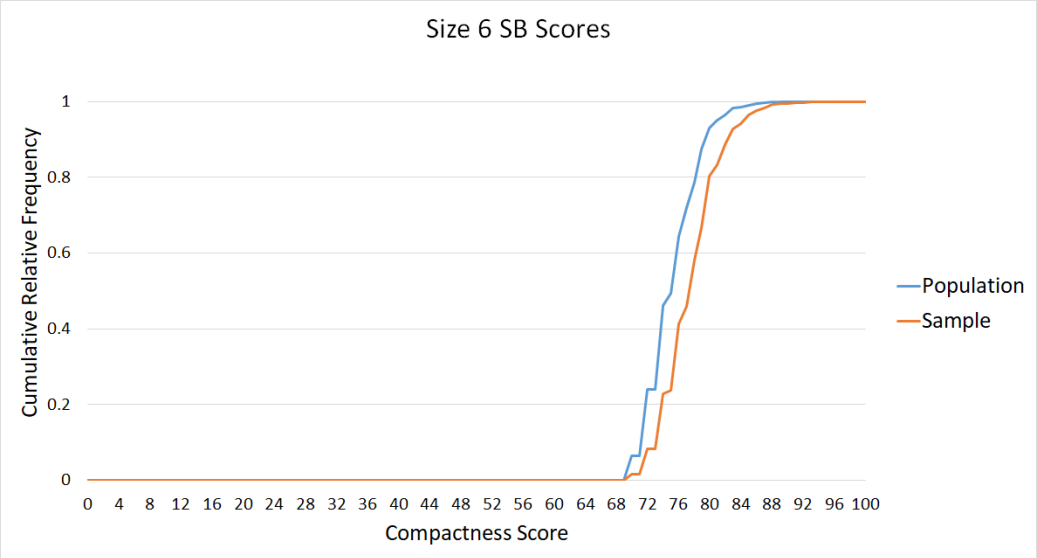


Figure 54: Size 6 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

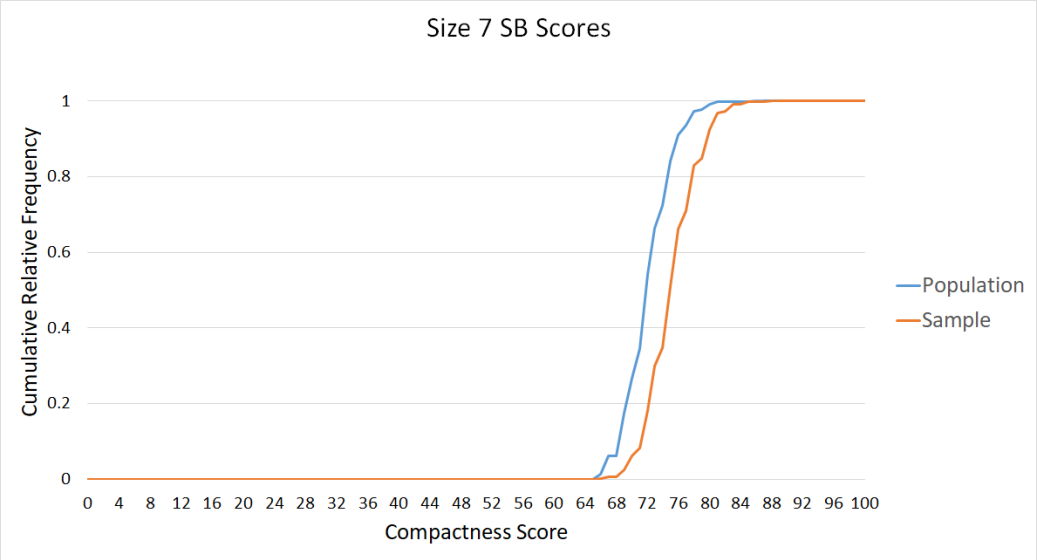


Figure 55: Size 7 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

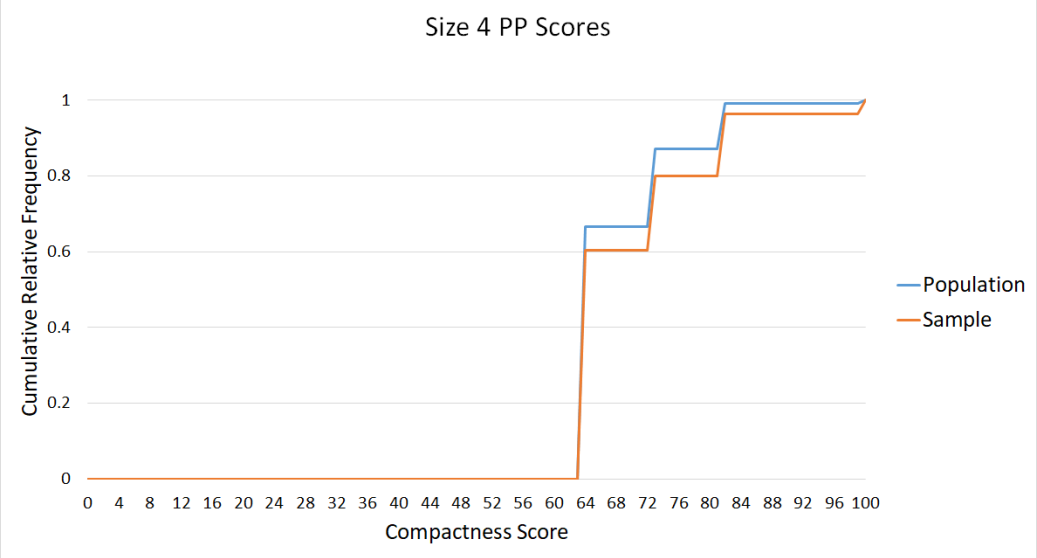


Figure 56: Size 4 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

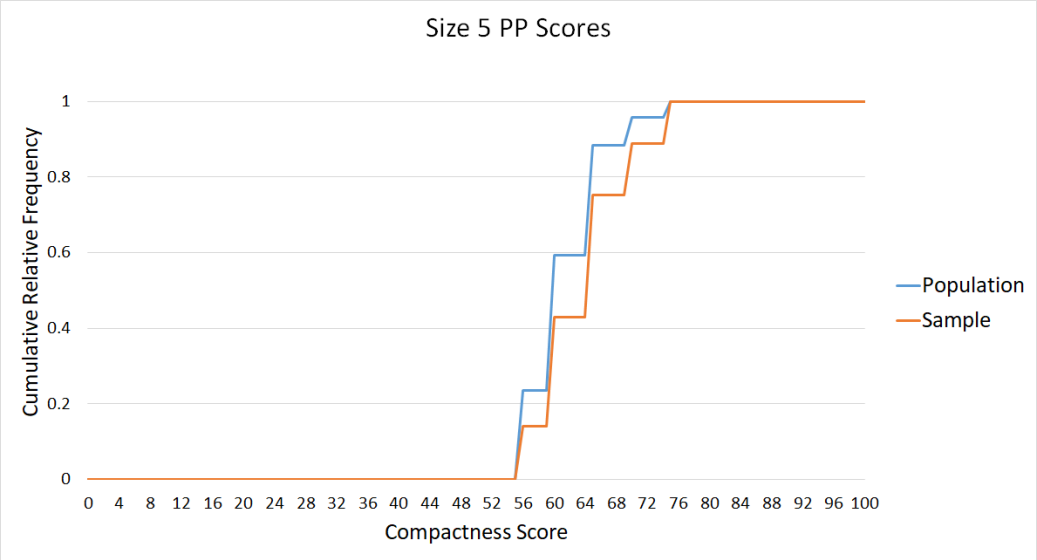


Figure 57: Size 5 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

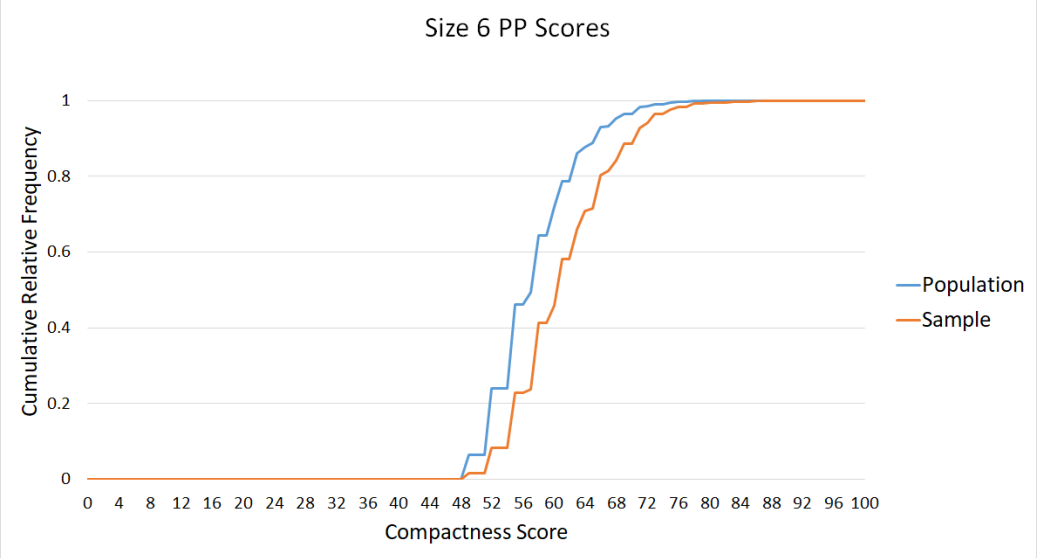


Figure 58: Size 6 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

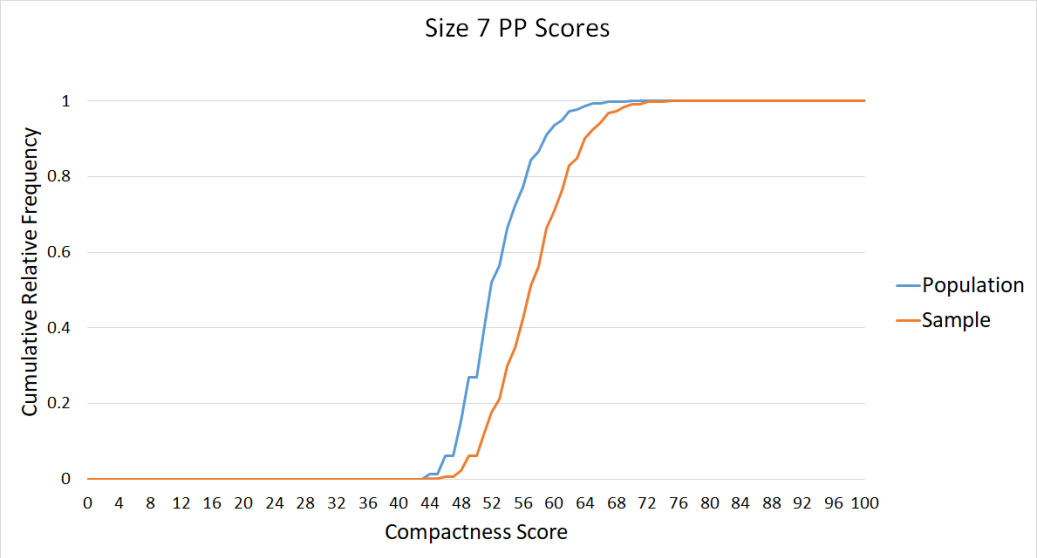


Figure 59: Size 7 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population and a Sample of 10,000 Partitions

Appendix C

We use Weighted Partitioner and vary the weighting factor to generate samples of partitions of size 4, 5, 6, and 7 Squaretopias. We graph the cumulative relative frequency distributions of the four types of compactness scores of the size 4, 5, 6, and 7 Squaretopia partitions in the weighted samples that Weighted Partitioner generates with weighting factor values of 0, 25, 50, 75, and 100. The figures also include the cumulative distributions of populations and the cumulative distributions of the samples from Appendix B. On each graph, these two cumulative distributions are labeled as "Population" and "Unweighted." All horizontal axes range from 0 to 100.

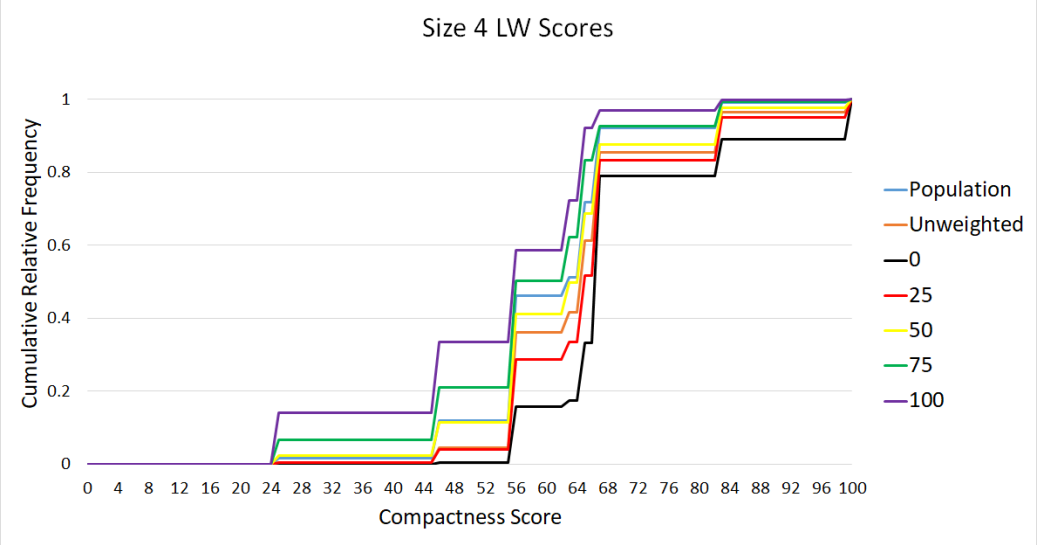


Figure 60: Size 4 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

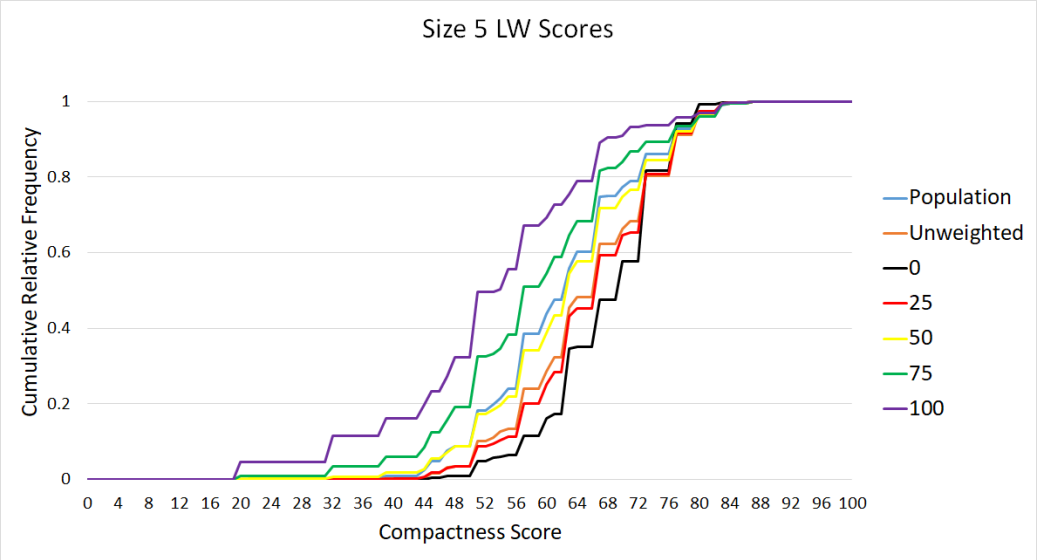


Figure 61: Size 5 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

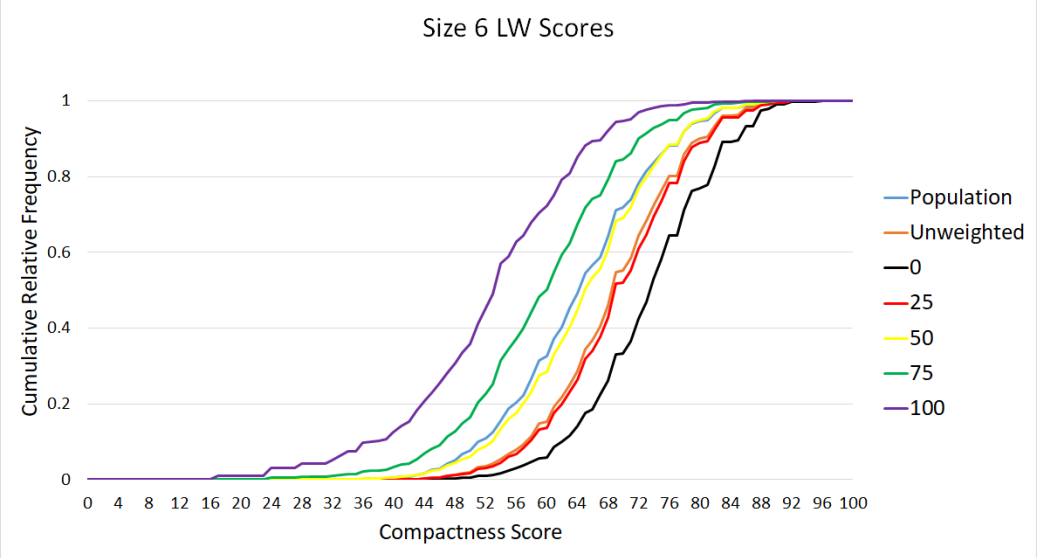


Figure 62: Size 6 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

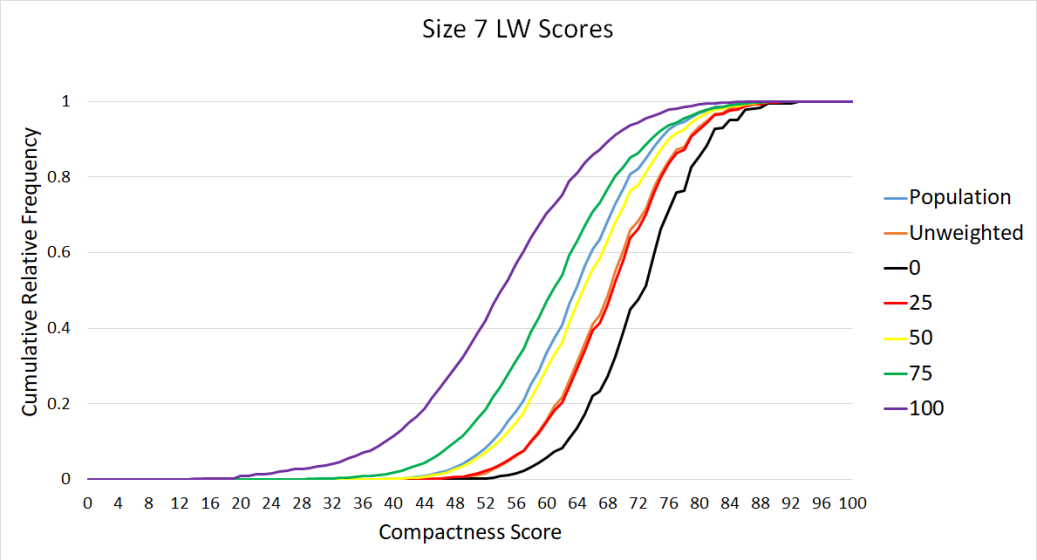


Figure 63: Size 7 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

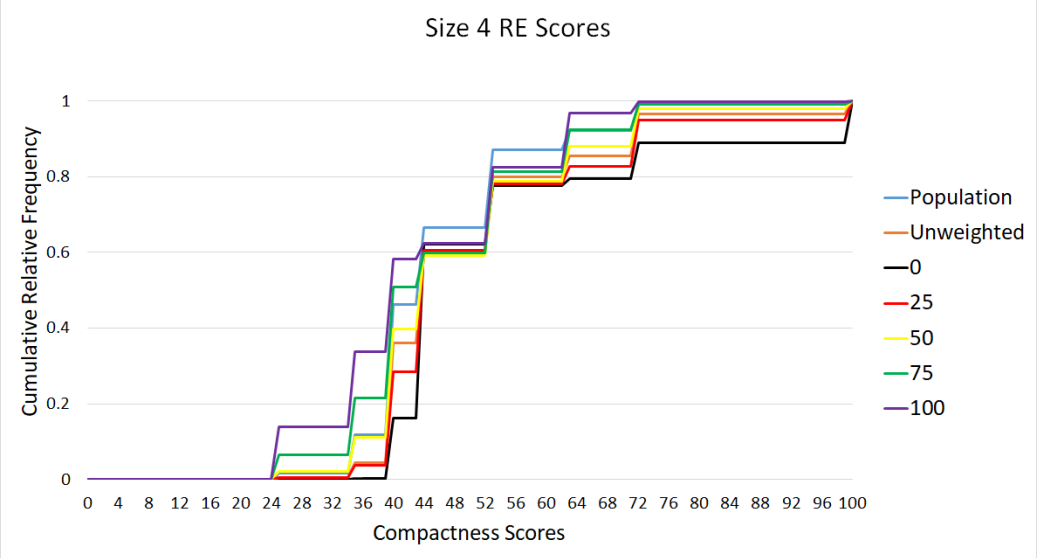


Figure 64: Size 4 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

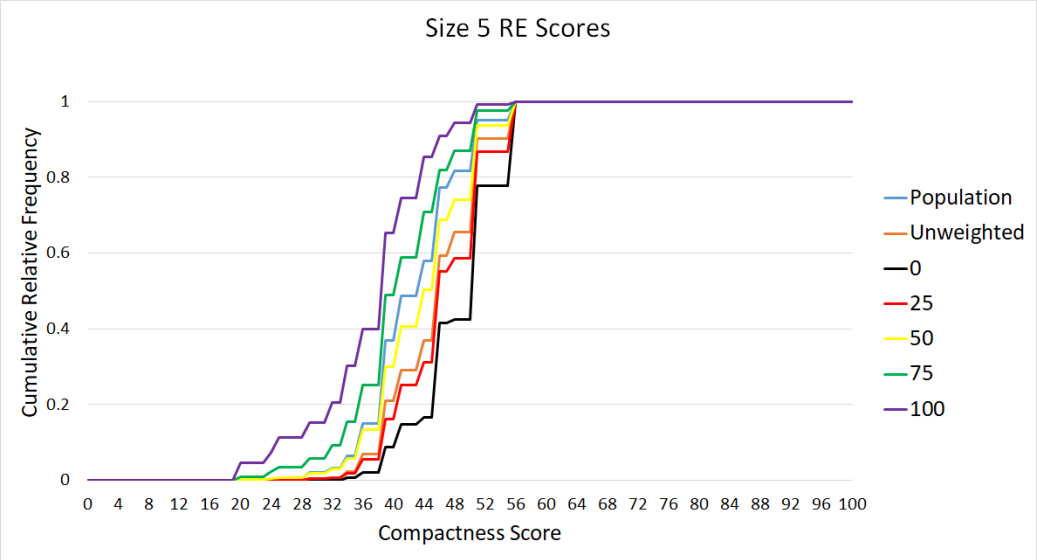


Figure 65: Size 5 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

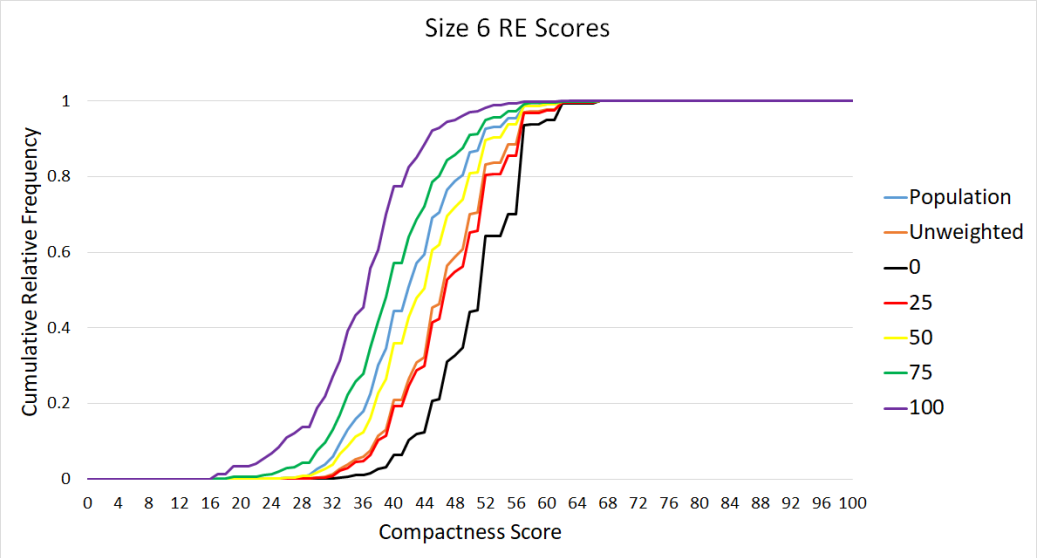


Figure 66: Size 6 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

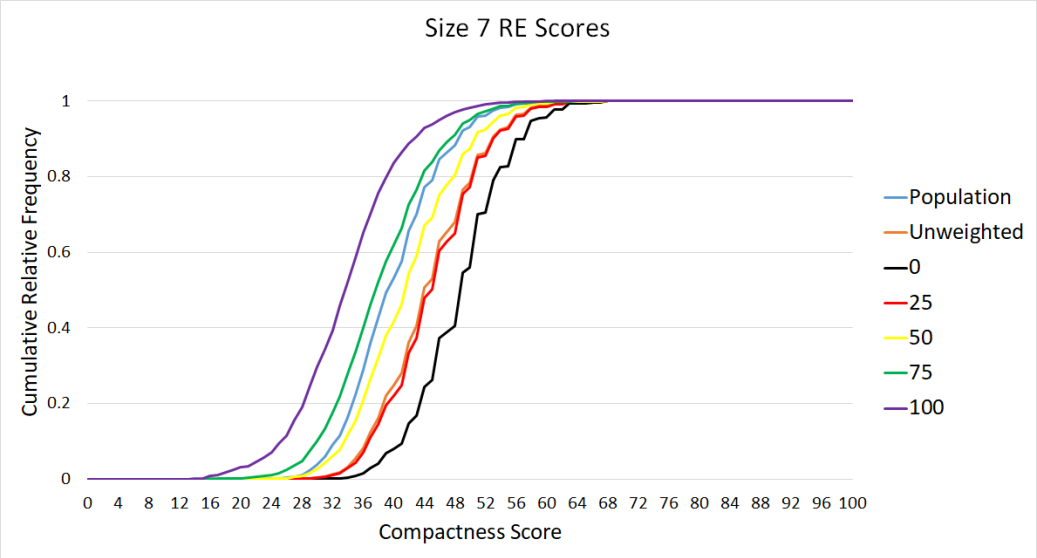


Figure 67: Size 7 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

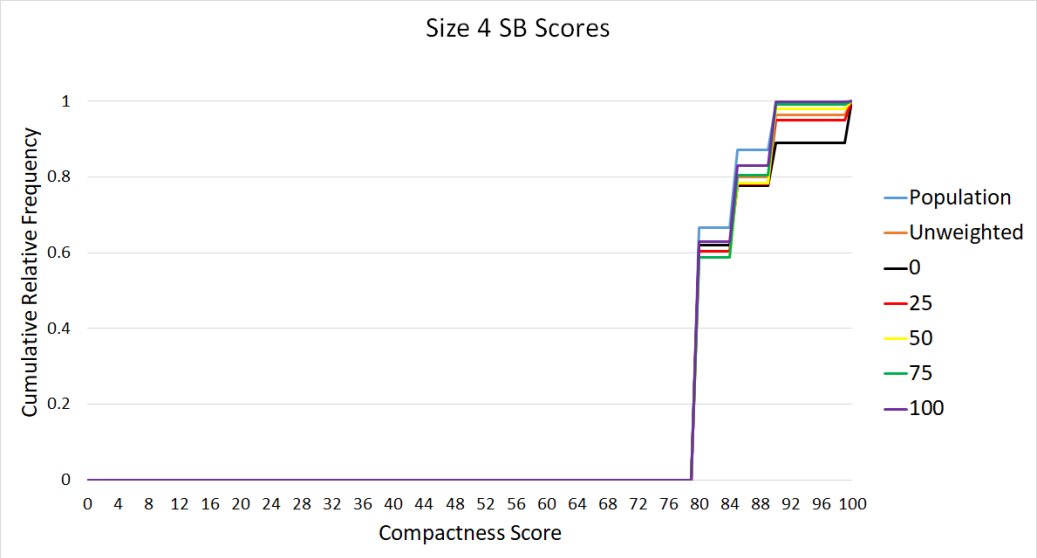


Figure 68: Size 4 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

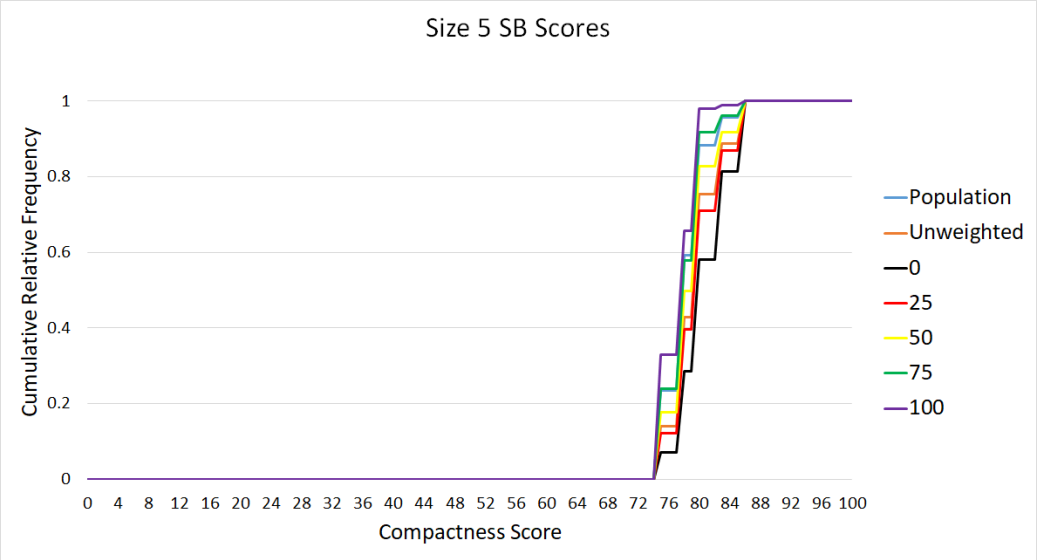


Figure 69: Size 5 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

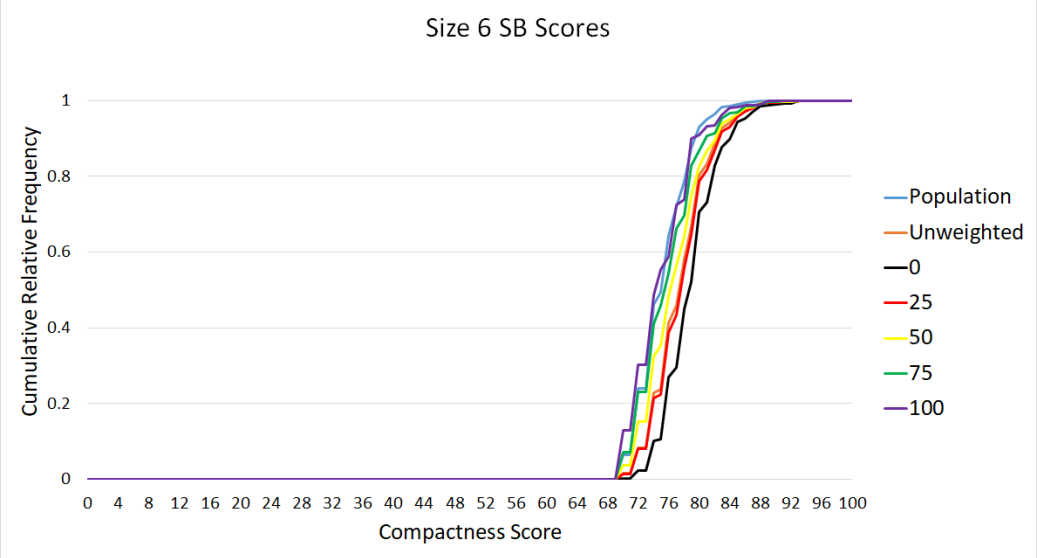


Figure 70: Size 6 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

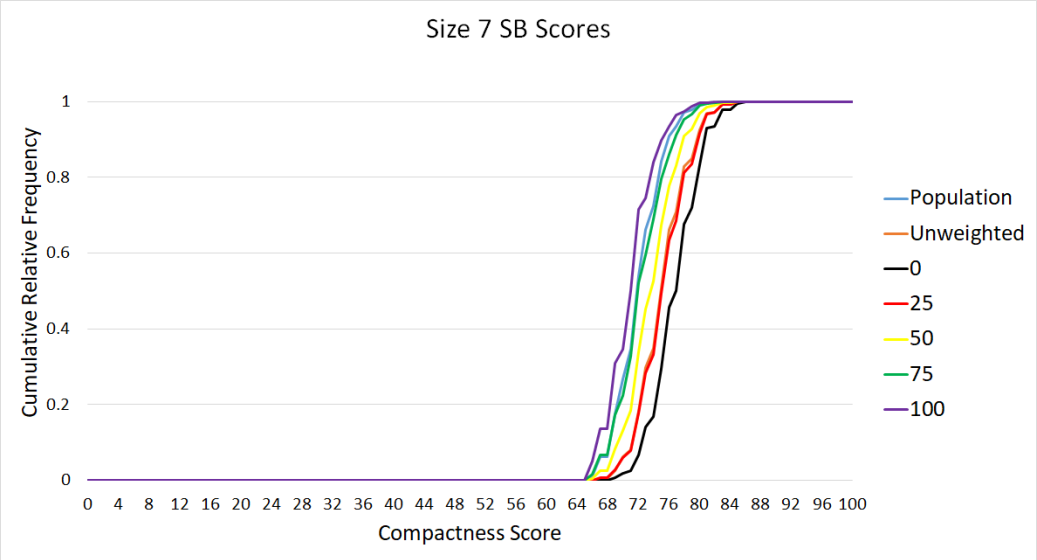


Figure 71: Size 7 Squaretopia SB Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

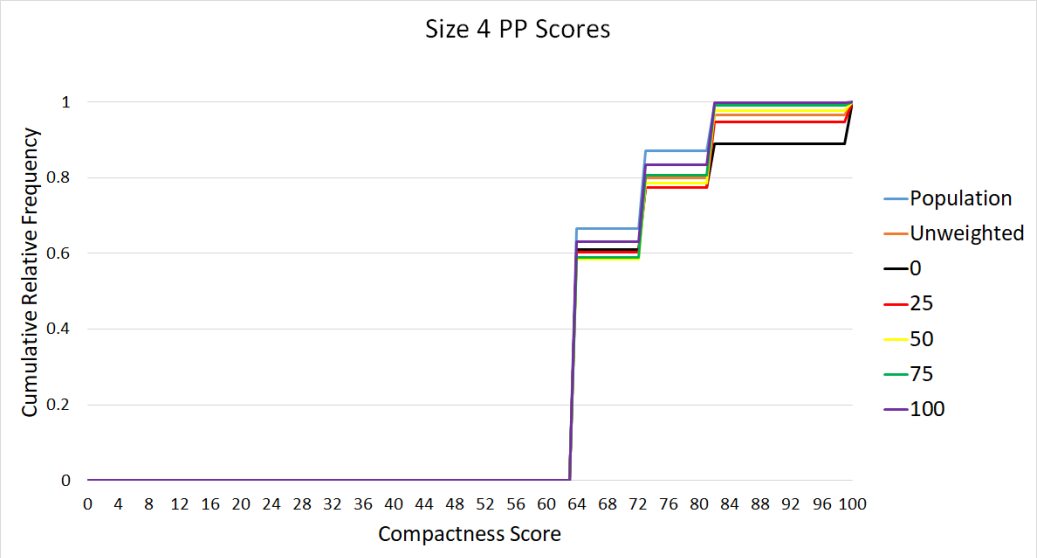


Figure 72: Size 4 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

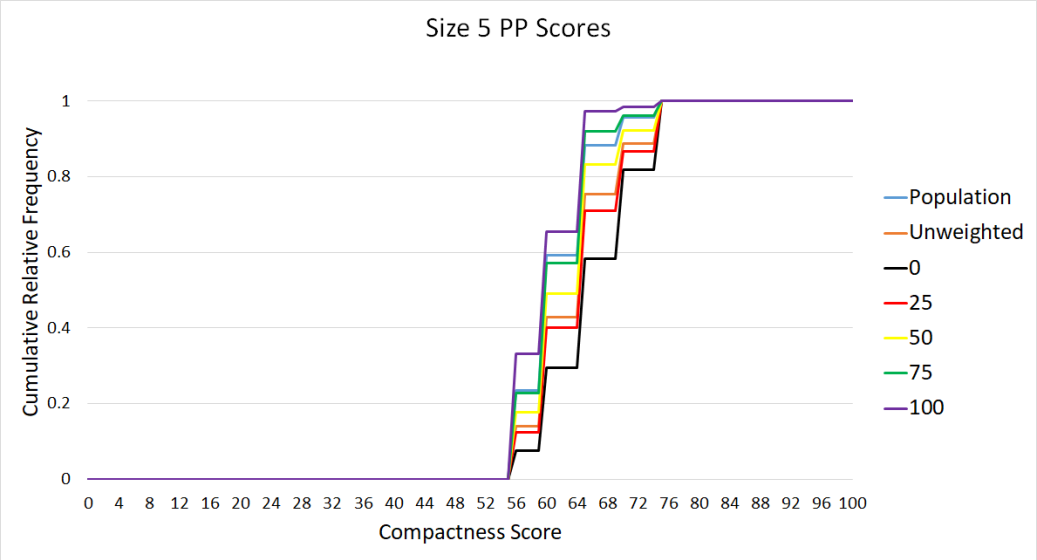


Figure 73: Size 5 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

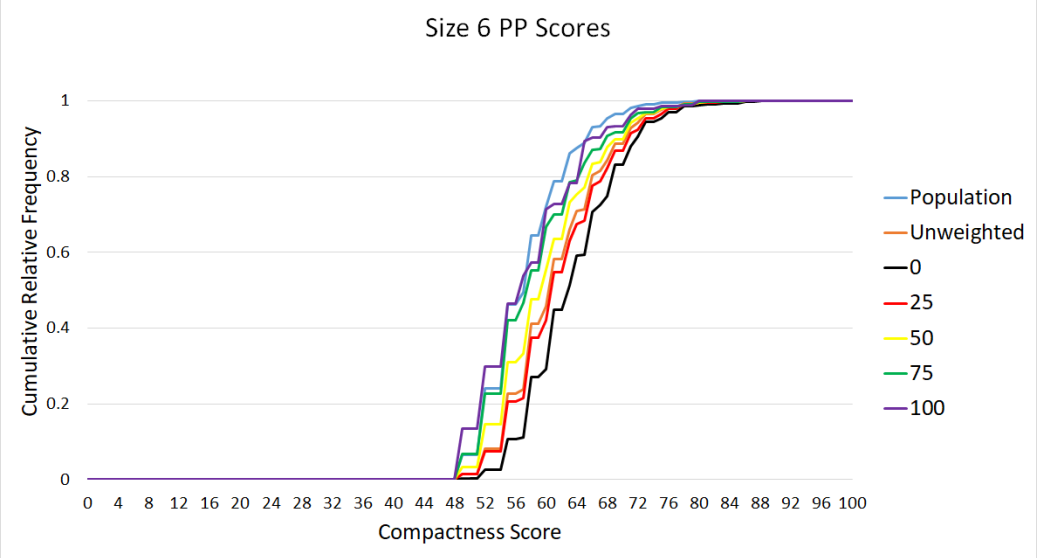


Figure 74: Size 6 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

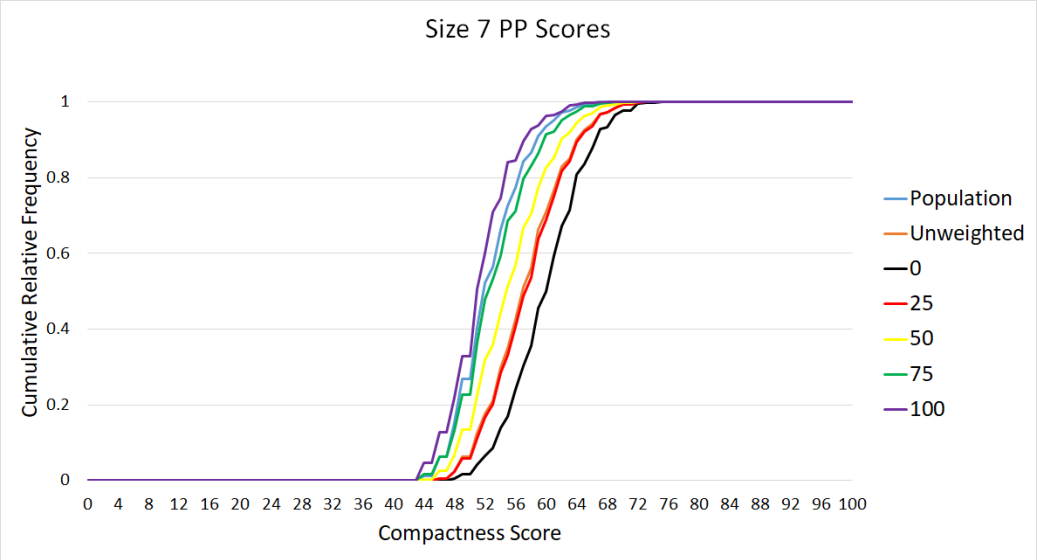


Figure 75: Size 7 Squaretopia PP Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and Five Weighted Samples of 10,000 Partitions

Appendix D

Using the same samples from our discussion of the χ^2 value vs. the weighting factor in Results II, we plot the max difference value vs. the weighting factor for the four scoring methods. Each figure shows how the max difference value varies for the four Squaretopia sizes. All horizontal axes range from 0 to 100.

Figure 76: LW Max Difference vs. Weighting Factor (Calculated from Samples of 10,000 Partitions of Size 4, 5, 6, and 7 Squaretopias Generated From Various Weighting Factors)

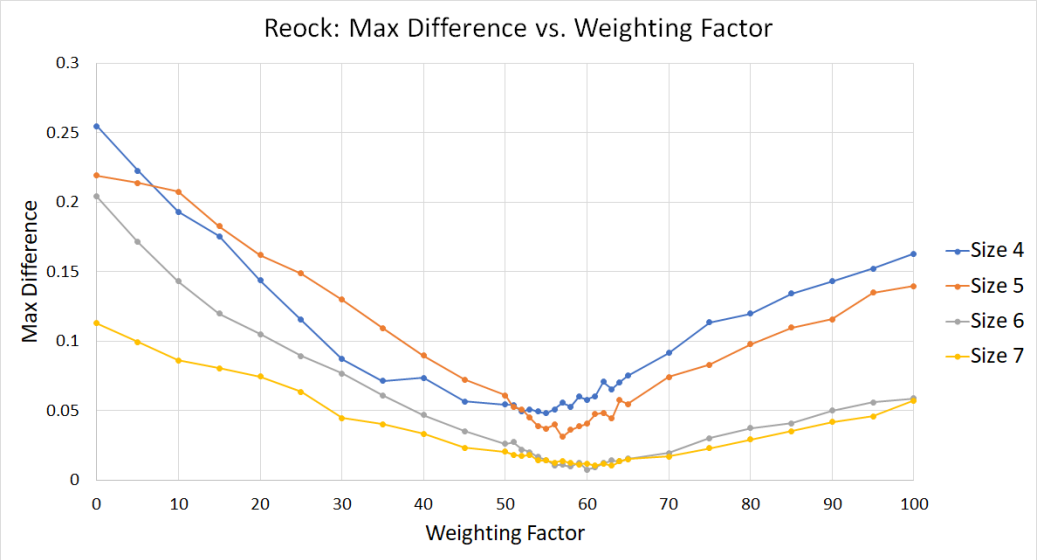


Figure 77: RE Max Difference vs. Weighting Factor (Calculated from Samples of 10,000 Partitions of Size 4, 5, 6, and 7 Squaretopias Generated From Various Weighting Factors)

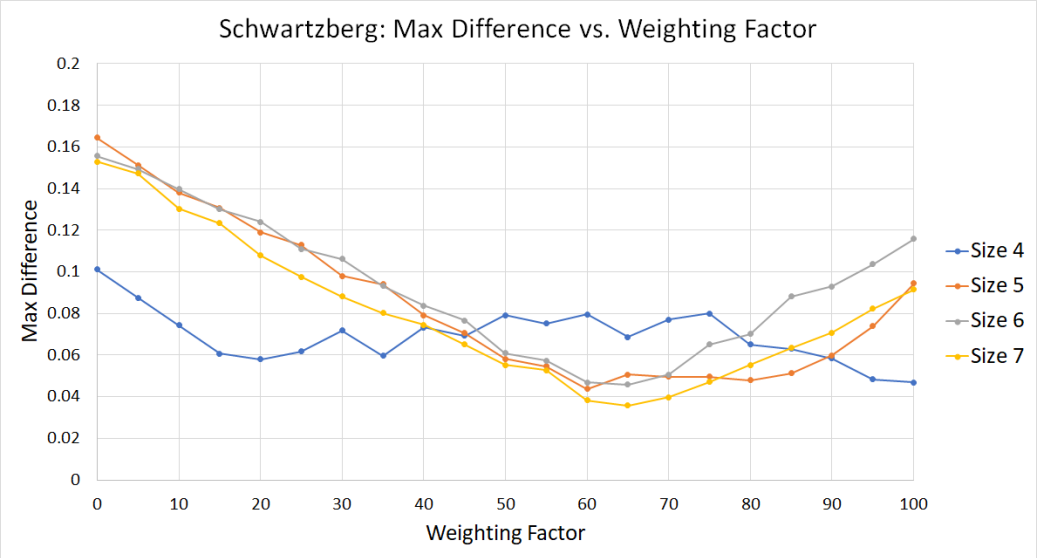


Figure 78: SB Max Difference vs. Weighting Factor (Calculated from Samples of 10,000 Partitions of Size 4, 5, 6, and 7 Squaretopias Generated From Various Weighting Factors)

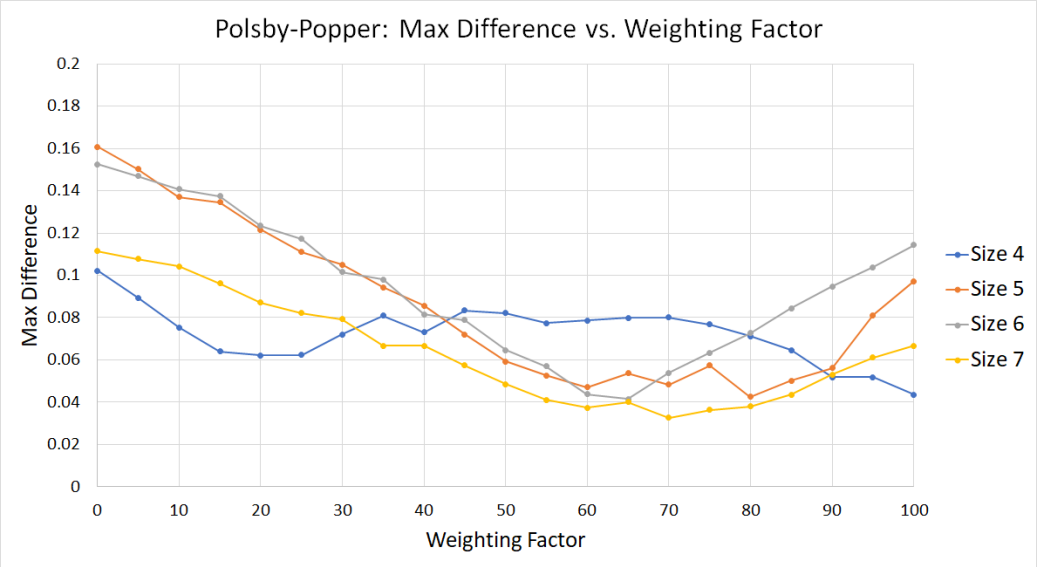


Figure 79: PP Max Difference vs. Weighting Factor (Calculated from Samples of 10,000 Partitions of Size 4, 5, 6, and 7 Squaretopias Generated From Various Weighting Factors)

Appendix E

We generate four samples of ten thousand partitions of Squaretopias of sizes 4, 5, 6, and 7 using a weighting factor of 52. We have four samples of ten thousand partitions for each Squaretopia size, and each partition has a Length-Width score. We graph the cumulative relative frequency distributions of the Length-Width scores of the populations, the unweighted samples from Appendix B, and a 52-weighted sample. We repeat this process for the Reock scoring method, but we use a weighting factor of 56. All horizontal axes range from 0 to 100.

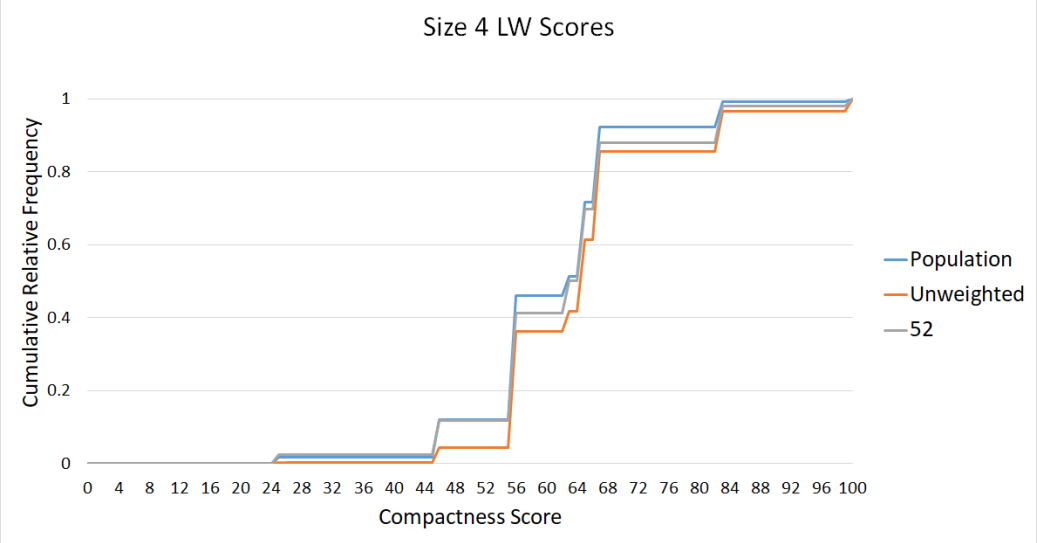


Figure 80: Size 4 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and a 52-Weighted Sample of 10,000 Partitions

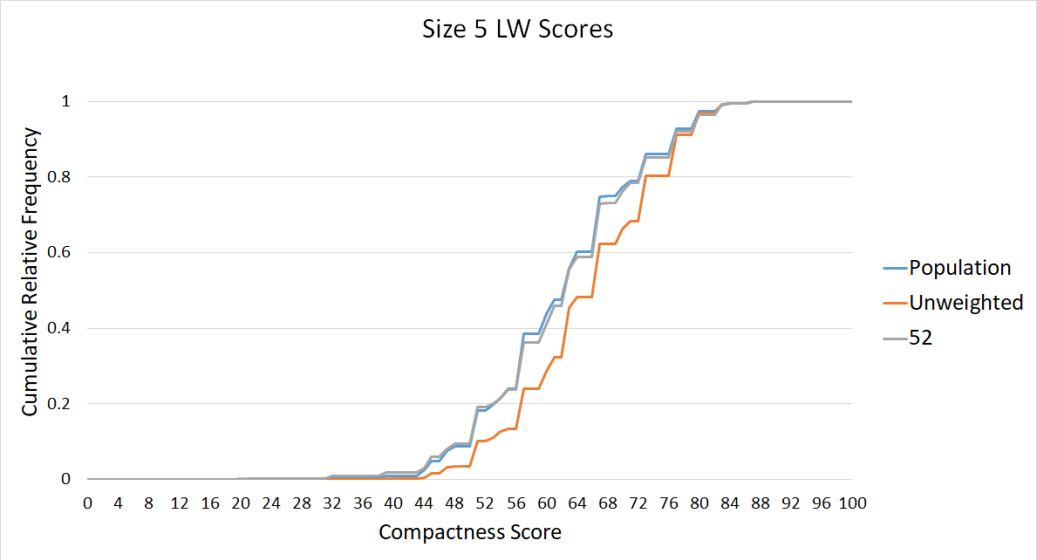


Figure 81: Size 5 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and a 52-Weighted Sample of 10,000 Partitions

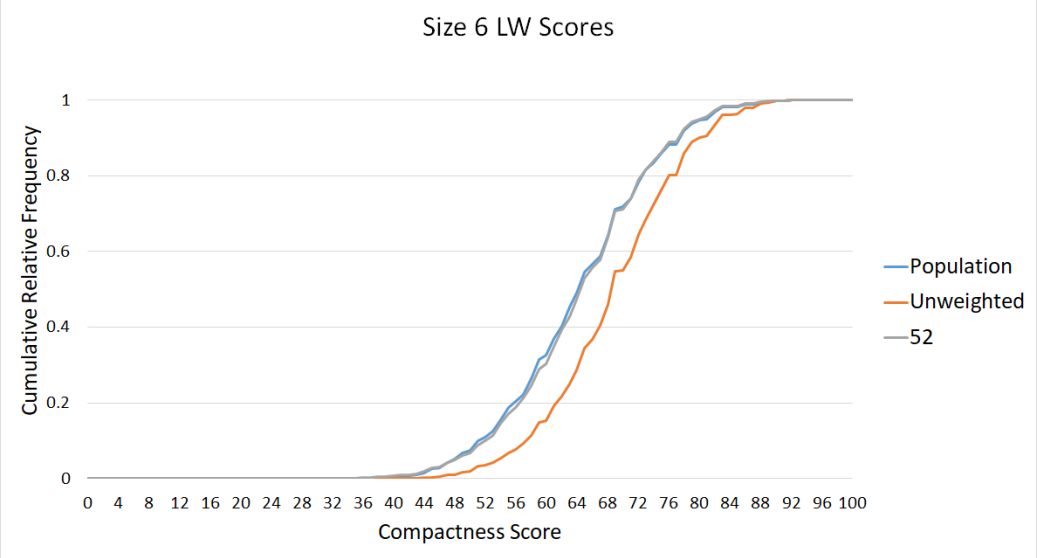


Figure 82: Size 6 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and a 52-Weighted Sample of 10,000 Partitions

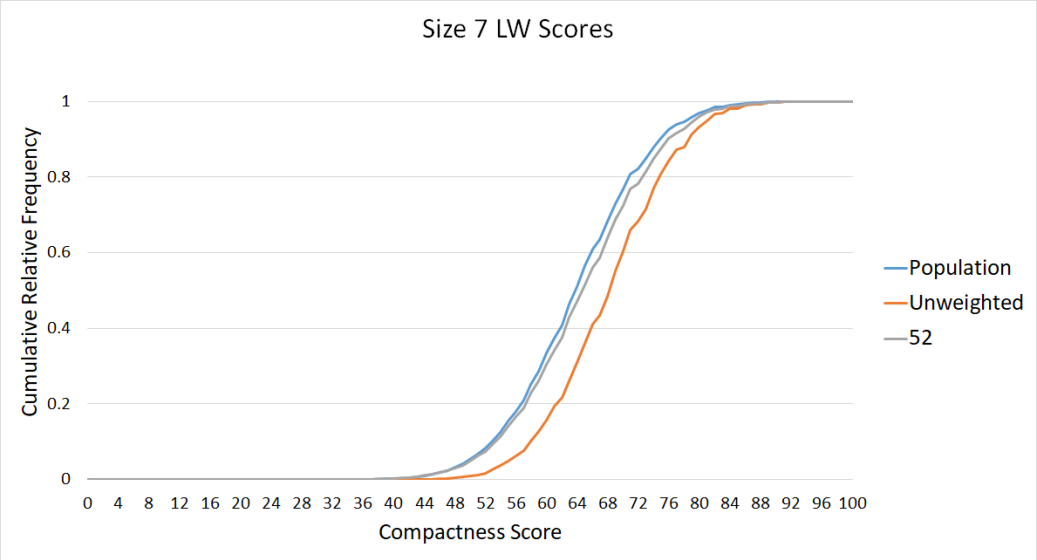


Figure 83: Size 7 Squaretopia LW Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and a 52-Weighted Sample of 10,000 Partitions

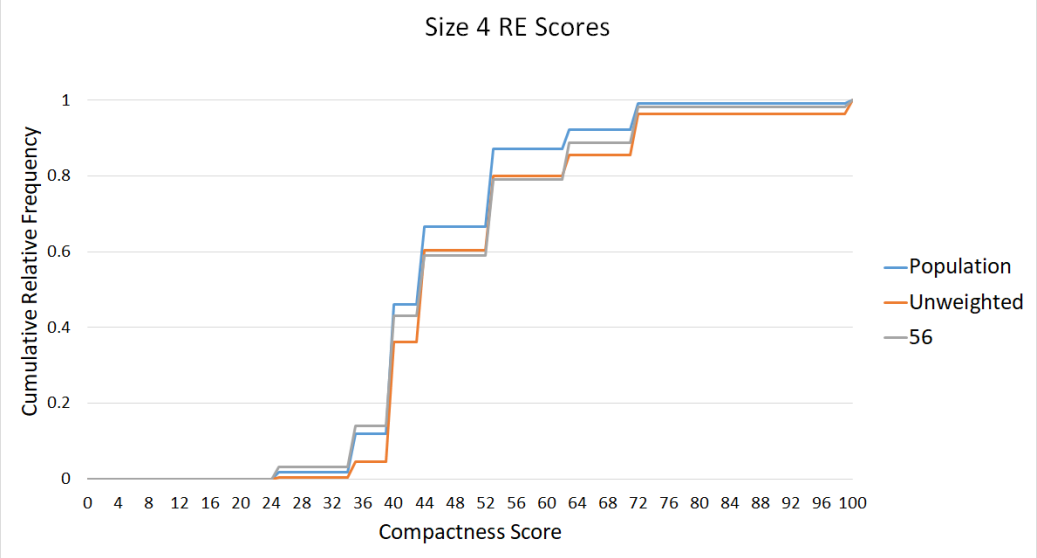


Figure 84: Size 4 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and a 56-Weighted Sample of 10,000 Partitions

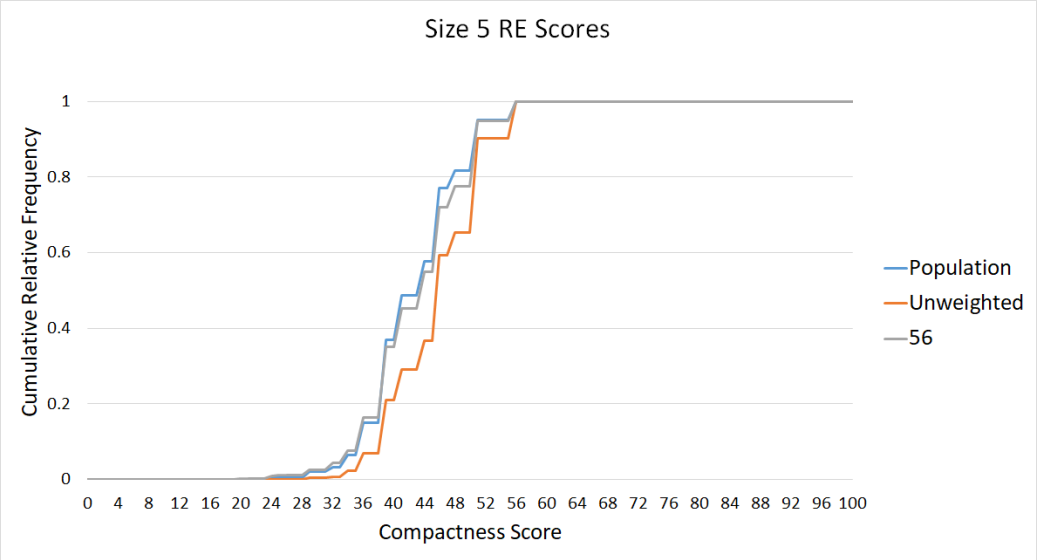


Figure 85: Size 5 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and a 56-Weighted Sample of 10,000 Partitions

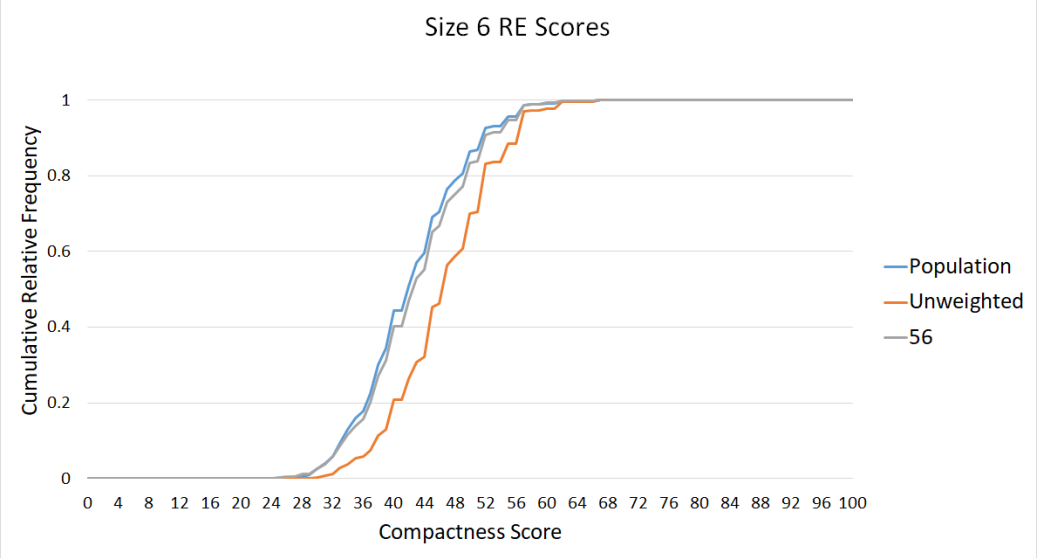


Figure 86: Size 6 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and a 56-Weighted Sample of 10,000 Partitions

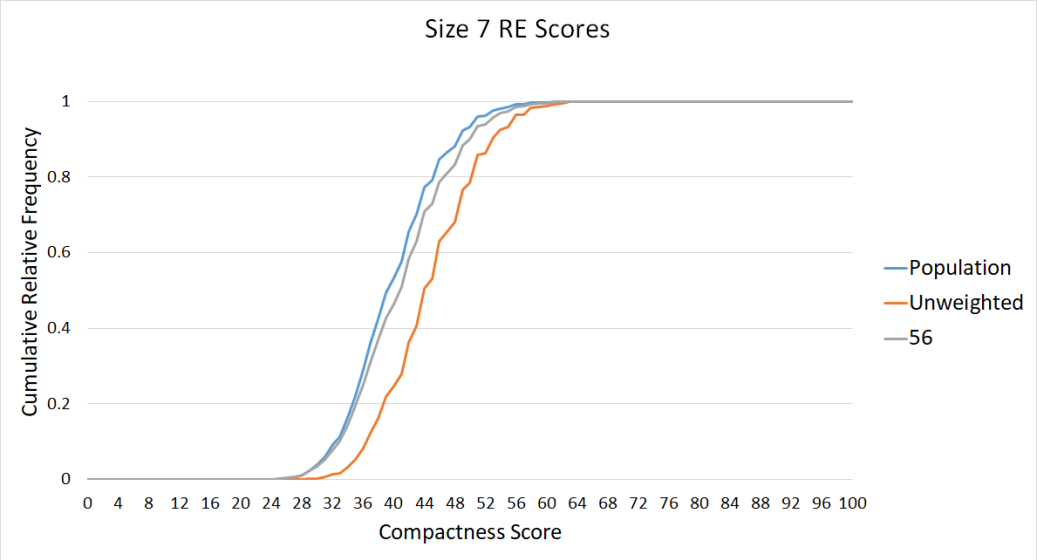


Figure 87: Size 7 Squaretopia RE Scores Cumul. Rel. Freq. Dist. of the Population, an Unweighted Sample of 10,000 Partitions, and a 56-Weighted Sample of 10,000 Partitions

Appendix F

In Appendix F, we include this project's most important files:

- SquaretopiaCell.java defines our Squaretopia cell representation of a given cell in an n by n square grid
- SquaretopiaMatrix.java defines our Squaretopia representation using an array of SquaretopiaCell arrays
- Partitioner.java defines Partitioner, which uniformly at random partitions an n by n Squaretopia into n equally sized contiguous districts
- WeightedPartitioner.java defines WeightedPartitioner which utilizes a weighting factor probability to partition an n by n Squaretopia into n equally sized contiguous districts
- PartitionTests.java allows us to run the following class files:
  - Partitioner.java
  - SingleDistricter.java
  - UnboundedSingleDistricter.java
  - WeightedPartitioner.java
  - WeightedSingleDistricter.java
  - WeightedUnboundedSingleDistricter.java

All files (including the SingleDistricter classes) are available at `https://github.com/jmariz/SeniorThesis`

```
// SqauretopiaCell.java
package partition;


/**
 * Joshua Mariz 05/05/2021
 * Squaretopia cell representation of a given cell in an n by n square grid
 */
public class SquaretopiaCell {


    public int row;
    public int col;
    public int districtNumber;
    public boolean checked; // used when checking for the validity of a district assignment
    public int direction; // relative to parent, this cell is in this direction
                          // where -1 is not yet defined, 1 is above, 2 is right, 3 is below, 4 is left
    public double lowerBoundForProbability; // used in weighted partitioning
    public double upperBoundForProbability; // used in weighted partitioning


    /**
     * Constructs a new SquaretopiaCell, which tracks the given row and column that it
     * represents in the Squaretopia.
     * @param row Integer row number of this cell (the top row is the first row)
     * @param col Integer column number of this cell (the leftmost column is the first column)
     */
    SquaretopiaCell (int row, int col) {
        this.row = row;
        this.col = col;
        this.districtNumber = 0;
        this.checked = false;
        this.direction = -1;


    }


    @Override
```

```
    public boolean equals (Object other) {
        return other instanceof SquaretopiaCell
            ? this.row == ((SquaretopiaCell) other).row && this.col ==
((SquaretopiaCell) other).col
            : false;
    }

    @Override
    public int hashCode () {
        return row * col;
    }

    @Override
    public String toString () {
        return "(" + row + ", " + col + ")";
    }

}
```

```
// SquaretopiaMatrix.java
package partition;

import java.util.HashSet;
import java.util.Set;

/**
 * Joshua Mariz 05/05/2021
 * Squaretopia representation using an array of SquaretopiaCell arrays
 */
final public class SquaretopiaMatrix {

    private final int M;             // number of rows
    private final int N;             // number of columns
    public final SquaretopiaCell[][] data;   // M-by-N SquaretopiaCell array

    /**
     * Sets up an M by N representation of Squaretopia using an array of
SquaretopiaCell arrays.
     * NOTE: This method initializes the outer padding layer of the Squaretopia matrix
to -1. This is so that each inner cell has four adjacent neighbors.
     * @param M Integer number of rows in this Squaretopia
     * @param N Integer number of columns in this Squaretopia
     * @return void
     */
    public SquaretopiaMatrix(int M, int N) {
        this.M = M;
        this.N = N;
        data = new SquaretopiaCell[M][N];
        for (int i = 0; i < M; i++) {
            for (int j = 0; j < N; j++) {
                data[i][j] = new SquaretopiaCell(i, j);
                if(i == 0 || i == M - 1 || j == 0 || j == N - 1) {
                    data[i][j].districtNumber = -1;
                    data[i][j].checked = true;
                }
```

```
            }
        }
    }

    /**
     * Prints a String representation of this Squaretopia. Prints a matrix where each number corresponds to the district number of that Squaretopia cell.
     * NOTE: This method does not print the outer padding layer of -1s.
     * @return void
     */
    public void show() {
        for (int i = 1; i < M - 1; i++) { // exclude outer rows of -1s
            for (int j = 1; j < N - 1; j++) { // exclude outer columns of -1s
                int districtNum = data[i][j].districtNumber;
                if(districtNum >= 0 && districtNum <= 9) {
                    System.out.print("   " + data[i][j].districtNumber);
                }
                else {
                    System.out.print("  " + data[i][j].districtNumber);
                }
            }
            System.out.print("");
            System.out.println("");
        }
    }

    /**
     * Creates a set containing all the inner Squaretopia cells.
     * NOTE: This method assumes that all inner cells are unclaimed, so it should only be called after the initialization of this Squaretopia.
     * @return Set<SquaretopiaCell> containing all the inner cells of this Squaretopia matrix
     */
    public Set<SquaretopiaCell> generateSetOfFreeCells() {
        int matrixLength = data.length; // assumes square matrix
        Set<SquaretopiaCell> setOfCells = new HashSet<>();
```

```
        for(int i = 1; i < matrixLength - 1; i++) { // does not add perimeter padding
cells to the set
            for(int j = 1; j < matrixLength - 1; j++) {
                setOfCells.add(new SquaretopiaCell(i, j));
            }
        }
        return setOfCells;
    }

    /**
     * Helpful method for debugging. Prints a matrix containing the
SquaretopiaCell.checked values for each cell in this Squaretopia.
     * @return void
     */
    public void printCheckedValues() {
        for (int i = 1; i < M - 1; i++) {
            for (int j = 1; j < N - 1; j++) {
                System.out.print(" " + data[i][j].checked);
            }
            System.out.print("");
            System.out.println("");
        }
    }

    /**
     * Determines if this Squaretopia matrix is valid. An invalid matrix contains at
least one isolated group of k Squaretopia cells,
     * where n, the number of cells in each district, doesn't divide k.
     * NOTE: This method assumes that this Squaretopia matrix is a square.
     * @param matrix SquaretopiaMatrix whose validity we will check
     * @return Boolean value for the statement: This matrix is valid.
     */
    public Boolean validMap() {
        int matrixLength = data.length;
        SquaretopiaMatrix matrixCopy = duplicateMatrix();
        for(int i = 1; i < matrixLength - 1; i++) {
            for (int j = 1; j < matrixLength - 1; j++) {
```

```
                if (matrixCopy.data[i][j].checked == false) {
                    // the set below contains all the unchecked neighbors that a rook
chess piece can reach when starting from the i, j location
                    Set<SquaretopiaCell> neighborCells =
findContiguousNeighbors(matrixCopy, matrixCopy.data[i][j]);
                        if(neighborCells.size() % (matrixLength - 2) !=  0) {
                            return false;
                        }
                }
            }
        }
        return true;
    }

    /**
     * Creates a SquaretopiaMatrix duplicate of this Squaretopia matrix.
     * NOTE: This method assumes that this Squaretopia matrix is a square.
     * @param matrix SquaretopiaMatrix that we will duplicate
     * @return void
     */
    public SquaretopiaMatrix duplicateMatrix() {
        int matrixLength = data.length; // assumes matrix is a square
        SquaretopiaMatrix matrixDuplicate = new SquaretopiaMatrix(matrixLength,
matrixLength);
        for(int i = 0; i < matrixLength; i++) {
            for(int j = 0; j < matrixLength; j++) {
                // we only need the checked field to determine if a completed district
is valid
                matrixDuplicate.data[i][j].checked = data[i][j].checked;
            }
        }
        return matrixDuplicate;
    }

    /**
     * Creates a set containing all the unchecked neighbors that a rook chess piece
can reach when starting from a given location in this Squaretopia.
```

```
     * @param matrix SquaretopiaMatrix that contains Squaretopia cells and some
district assignments

     * @param currentLocation SquaretopiaCell whose contiguous unchecked neighbors we
will find

     * @return Set<SquaretopiaCell> containing all the unchecked neighbors that a rook
chess piece can reach when starting from currentLocation

     */

    public Set<SquaretopiaCell> findContiguousNeighbors (SquaretopiaMatrix matrix,
SquaretopiaCell currentLocation) {

        int curLocRow = currentLocation.row;

        int curLocCol = currentLocation.col;

        Set<SquaretopiaCell> contiguousCells = new HashSet<>();

        contiguousCells.add(currentLocation);

        currentLocation.checked = true;

        if(matrix.data[curLocRow - 1][curLocCol].checked == false) { // check
availability of the cell above

            contiguousCells.addAll(findContiguousNeighbors(matrix,
matrix.data[curLocRow - 1][curLocCol]));

        }

        if(matrix.data[curLocRow + 1][curLocCol].checked == false) { // check
availability of the cell below

            contiguousCells.addAll(findContiguousNeighbors(matrix,
matrix.data[curLocRow + 1][curLocCol]));

        }

        if(matrix.data[curLocRow][curLocCol - 1].checked == false) { // check
availability of the cell to the left

            contiguousCells.addAll(findContiguousNeighbors(matrix,
matrix.data[curLocRow][curLocCol - 1]));

        }

        if(matrix.data[curLocRow][curLocCol + 1].checked == false) { // check
availability of the cell to the right

            contiguousCells.addAll(findContiguousNeighbors(matrix,
matrix.data[curLocRow][curLocCol + 1]));

        }

        return contiguousCells;

    }




    /**

     * Calculates the perimeter of each district in a finished Squaretopia partition.

     * @param matrix SquaretopiaMatrix that has been fully partitioned
```

```
     * @return int[] containing the perimeters of each district in the partition: with
district 1's perimeter at index 0, district 2's perimeter at index 1, etc.
     */
    public int[] calculatePerimeters() {
        int numOfDistricts = data.length - 2;
        int[] perimetersArray = new int[numOfDistricts];
        for(int i = 0; i < numOfDistricts; i++) {
            perimetersArray[i] = 4 * numOfDistricts;
        }
        for(int i = 1; i < data.length - 1; i++) {
            for(int j = 1; j < data.length - 1; j++) {
                int districtNumOfCurrentCell = data[i][j].districtNumber;
                if(districtNumOfCurrentCell == data[i - 1][j].districtNumber) {
                    perimetersArray[districtNumOfCurrentCell - 1]--;
                }
                if(districtNumOfCurrentCell == data[i + 1][j].districtNumber) {
                    perimetersArray[districtNumOfCurrentCell - 1]--;
                }
                if(districtNumOfCurrentCell == data[i][j - 1].districtNumber) {
                    perimetersArray[districtNumOfCurrentCell - 1]--;
                }
                if(districtNumOfCurrentCell == data[i][j + 1].districtNumber) {
                    perimetersArray[districtNumOfCurrentCell - 1]--;
                }
            }
        }
      return perimetersArray;
    }


    /**
     * Calculates the Length-Width score of each district in a Squaretopia partition,
then finds the Length-Width
     * score of the partition by taking the average of the districts' Length-Width
scores
     * @return double Length-Width compactness score (unrounded) of the partition
     */
    public double LengthWidth() {
```

```
        // locate and save extrema cells
        double scoreTotal = 0;
        double numOfDistricts = data.length - 2;
        SquaretopiaCell[][] extrema = new SquaretopiaCell[(int) numOfDistricts][4]; //
each district has four cells in this order min length, max length, min width, max
width
        for (int i = 1; i < data.length - 1; i++) {
            for (int j = 1; j < data.length - 1; j++) {
                int districtNumberOfThisCell = data[i][j].districtNumber;
                SquaretopiaCell thisCell = data[i][j];
                if(extrema[districtNumberOfThisCell - 1][0] == null) {
                    extrema[districtNumberOfThisCell - 1][0] = thisCell;
                    extrema[districtNumberOfThisCell - 1][1] = thisCell;
                    extrema[districtNumberOfThisCell - 1][2] = thisCell;
                    extrema[districtNumberOfThisCell - 1][3] = thisCell;
                }
                if(thisCell.row < extrema[districtNumberOfThisCell - 1][0].row) {
                    extrema[districtNumberOfThisCell - 1][0] = thisCell;
                }
                if(thisCell.row > extrema[districtNumberOfThisCell - 1][1].row) {
                    extrema[districtNumberOfThisCell - 1][1] = thisCell;
                }
                if(thisCell.col < extrema[districtNumberOfThisCell - 1][2].col) {
                    extrema[districtNumberOfThisCell - 1][2] = thisCell;
                }
                if(thisCell.col > extrema[districtNumberOfThisCell - 1][3].col) {
                    extrema[districtNumberOfThisCell - 1][3] = thisCell;
                }
            }
        }
        double length;
        double width;
        for (int i = 0; i < extrema.length; i++) {
            length = Math.abs(extrema[i][0].row - extrema[i][1].row) + 1; // add one
to count properly
            width = Math.abs(extrema[i][2].col - extrema[i][3].col) + 1;
            if (length < width) {
```

```
                scoreTotal += length / width;
            } else {
                scoreTotal += width / length;
            }
        }
        double averageScore = scoreTotal / numOfDdistricts;
        return averageScore;
    }

    /**
     * Calculates the Reock score of each district in a Squaretopia partition, then
finds the Reock
     * score of the partition by taking the average of the districts' Reock scores
     * @return double Reock compactness score (unrounded) of the partition
     */
    public double Reock() {
        // locate and save extrema cells
        double scoreTotal = 0;
        double numOfDistricts = data.length - 2;
        SquaretopiaCell[][] extrema = new SquaretopiaCell[(int) numOfDistricts][4]; //
each district has four cells in this order min length, max length, min width, max
width
        for (int i = 1; i < data.length - 1; i++) {
            for (int j = 1; j < data.length - 1; j++) {
                int districtNumberOfThisCell = data[i][j].districtNumber;
                SquaretopiaCell thisCell = data[i][j];
                if(extrema[districtNumberOfThisCell - 1][0] == null) {
                    extrema[districtNumberOfThisCell - 1][0] = thisCell;
                    extrema[districtNumberOfThisCell - 1][1] = thisCell;
                    extrema[districtNumberOfThisCell - 1][2] = thisCell;
                    extrema[districtNumberOfThisCell - 1][3] = thisCell;
                }
                if(thisCell.row < extrema[districtNumberOfThisCell - 1][0].row) {
                    extrema[districtNumberOfThisCell - 1][0] = thisCell;
                }
                if(thisCell.row > extrema[districtNumberOfThisCell - 1][1].row) {
                    extrema[districtNumberOfThisCell - 1][1] = thisCell;
```

```
                }
                if(thisCell.col < extrema[districtNumberOfThisCell - 1][2].col) {
                    extrema[districtNumberOfThisCell - 1][2] = thisCell;
                }
                if(thisCell.col > extrema[districtNumberOfThisCell - 1][3].col) {
                    extrema[districtNumberOfThisCell - 1][3] = thisCell;
                }
            }
        }
        double length;
        double width;
        for (int i = 0; i < extrema.length; i++) {
            length = Math.abs(extrema[i][0].row - extrema[i][1].row) + 1; // add one
to count properly
            width = Math.abs(extrema[i][2].col - extrema[i][3].col) + 1;
            if (length < width) {
                scoreTotal += numOfDdistricts / (width * width);
            } else {
                scoreTotal += numOfDdistricts / (length * length);
            }
        }
        double averageScore = scoreTotal / numOfDistricts;
        return averageScore;
    }


    /**
     * Calculates the Polsby-Popper score of each district in a Squaretopia partition,
then finds the Polsby-Popper
     * score of the partition by taking the average of the districts' Polsby-Popper
scores
     * @return double Polsby-Popper compactness score (unrounded) of the partition
     */
    public double PolsbyPopper() {
        int[] perimetersArray = this.calculatePerimeters();
        double scoreTotal = 0;
        int numOfDistricts = data.length - 2;
        double areaOfADistrict = data.length - 2;
```

```
        for(int i = 0; i < perimetersArray.length; i++) {
            scoreTotal += areaOfADistrict /
Math.pow(Double.valueOf(perimetersArray[i]) / 4, 2);
        }
        double averageScore = scoreTotal / numOfDistricts;
        return averageScore;
    }

    /**
     * Calculates the Schwartzberg score of each district in a Squaretopia partition,
then finds the Schwartzberg
     * score of the partition by taking the average of the districts' Schwartzberg
scores
     * @return double Schwartzberg compactness score (unrounded) of the partition
     */
    public double Schwartzberg() {
        int[] perimetersArray = this.calculatePerimeters();
        double scoreTotal = 0;
        int numOfDistricts = data.length - 2;
        double areaOfADistrict = data.length - 2;
        for(int i = 0; i < perimetersArray.length; i++) {
            scoreTotal += (4 * Math.sqrt(Double.valueOf(areaOfADistrict))) /
Double.valueOf(perimetersArray[i]);
        }
        double averageScore = scoreTotal / numOfDistricts;
        return averageScore;
    }

    /**
     * Calculates the Reock score of the FIRST AND ONLY district in an INCOMPLETE
Squaretopia partition
     * @return double Reock compactness score (unrounded) of the only district in the
incomplete partition
     */
    public double singleReock() {
        // locate and save extrema cells
        double scoreTotal = 0;
        double numOfDistricts = 1;
```

```
        SquaretopiaCell[][] extrema = new SquaretopiaCell[(int) numOfDistricts][4]; // each district has four cells in this order min length, max length, min width, max width
        for (int i = 1; i < data.length - 1; i++) {
            for (int j = 1; j < data.length - 1; j++) {
                int districtNumberOfThisCell = data[i][j].districtNumber;
                if(districtNumberOfThisCell != 0) {
                    SquaretopiaCell thisCell = data[i][j];
                    if(extrema[districtNumberOfThisCell - 1][0] == null) {
                        extrema[districtNumberOfThisCell - 1][0] = thisCell;
                        extrema[districtNumberOfThisCell - 1][1] = thisCell;
                        extrema[districtNumberOfThisCell - 1][2] = thisCell;
                        extrema[districtNumberOfThisCell - 1][3] = thisCell;
                    }
                    if(thisCell.row < extrema[districtNumberOfThisCell - 1][0].row) {
                        extrema[districtNumberOfThisCell - 1][0] = thisCell;
                    }
                    if(thisCell.row > extrema[districtNumberOfThisCell - 1][1].row) {
                        extrema[districtNumberOfThisCell - 1][1] = thisCell;
                    }
                    if(thisCell.col < extrema[districtNumberOfThisCell - 1][2].col) {
                        extrema[districtNumberOfThisCell - 1][2] = thisCell;
                    }
                    if(thisCell.col > extrema[districtNumberOfThisCell - 1][3].col) {
                        extrema[districtNumberOfThisCell - 1][3] = thisCell;
                    }
                }
            }
        }
        double length;
        double width;
        for (int i = 0; i < extrema.length; i++) {
            length = Math.abs(extrema[i][0].row - extrema[i][1].row) + 1; // add one to count properly
            width = Math.abs(extrema[i][2].col - extrema[i][3].col) + 1;
            if (length < width) {
                scoreTotal += (data.length - 2) / (width * width);
```

```
            } else {
                scoreTotal += (data.length - 2) / (length * length);
            }
        }
        double averageScore = scoreTotal / numOfDdistricts;
        return averageScore;
    }

}
```

```
// Partitioner.java
package partition;


import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;


/**
 * Joshua Mariz 05/05/2021
 * Partitioner.java
 * Partitioner uniformly at random partitions an n by n Squaretopia into n equally
sized contiguous districts
 */
public class Partitioner {


    /**
     * Prepares the Squaretopia partitioning process.
     * @param size Integer number of the Squaretopia's size
     * @param numOfTrials Integer number for the number of Squaretopias we will
partition
     * @return void
     */
    public static void Partition (int size, int numOfTrials) {
        int adjustedSize = size + 2; // includes the outer layer of padding cells
        int trialsConducted = 0;
        Boolean isValidPartition;
        while(trialsConducted < numOfTrials) {
            isValidPartition = true;
            SquaretopiaMatrix Squaretopia = new SquaretopiaMatrix(adjustedSize,
adjustedSize);
            Set<SquaretopiaCell> freeCells = Squaretopia.generateSetOfFreeCells();
            Set<SquaretopiaCell> currentDistrict = new HashSet<>();
            Set<SquaretopiaCell> currentDistrictFreeNeighbors = new HashSet<>();
            Set<SquaretopiaCell> recentlyAddedTransitions = new HashSet<>();
            SquaretopiaCell claimedCell;
            while(freeCells.size() > 0) {
                claimedCell = deadEnd(Squaretopia, freeCells);
```

```
                if(claimedCell == null) {
                    claimedCell = randomCell(freeCells);
                }
                claimer(Squaretopia, freeCells, currentDistrict, claimedCell);
                currentDistrictFreeNeighbors.addAll(getTransitions(Squaretopia,
claimedCell));
                recentlyAddedTransitions.addAll(getTransitions(Squaretopia,
claimedCell));
                if (recursiveDistricter(Squaretopia, freeCells, currentDistrict,
currentDistrictFreeNeighbors, recentlyAddedTransitions) == null) {
                    isValidPartition = false;
                    break;
                }
                currentDistrict.clear();
                currentDistrictFreeNeighbors.clear();
            }
            if(isValidPartition) { // output partition and rounded compactness scores
(LW RE SB PP)
                // COMMENT OUT THE LINE BELOW TO NOT OUTPUT THE PARTITIONED
SQUARETOPIA
                Squaretopia.show();
                // COMMENT OUT THE LINE BELOW TO NOT OUTPUT THE SCORES OF THE
PARTITIONED SQUARETOPIA
                System.out.println(Math.round(Squaretopia.LengthWidth()*100) + " " +
Math.round(Squaretopia.Reock()*100) + " " + Math.round(Squaretopia.Schwartzberg()*100)
+ " " + Math.round(Squaretopia.PolsbyPopper()*100));
                System.out.println("");
                trialsConducted++;
            }
        }


    }


    // recursive algorithm
    public static SquaretopiaMatrix recursiveDistricter (SquaretopiaMatrix matrix,
Set<SquaretopiaCell> freeCells, Set<SquaretopiaCell> currentDistrict,
Set<SquaretopiaCell> allPossibleTransitions, Set<SquaretopiaCell>
recentlyAddedTransitions) {
          if(currentDistrict.size() == (matrix.data.length - 2)) { // assumes matrix
is a square!
              if(matrix.validMap()) {
```

```
                recentlyAddedTransitions.clear();
                return matrix;
            } else {
                allPossibleTransitions.removeAll(recentlyAddedTransitions);
                return null;
            }
        }

        Set<SquaretopiaCell> newAllPossibleNeighbors = new HashSet<>();
        newAllPossibleNeighbors.addAll(allPossibleTransitions);
        Set<SquaretopiaCell> newRecentlyAddedTransitions = new HashSet<>();

        while(newAllPossibleNeighbors.size() != 0) {
            SquaretopiaCell nextCell = isolatedCell(matrix,
newAllPossibleNeighbors);
            if(nextCell == null) {
                nextCell = randomCell(newAllPossibleNeighbors);
            }
            claimer(matrix, freeCells, currentDistrict, nextCell);
            newAllPossibleNeighbors.remove(nextCell);
            newRecentlyAddedTransitions =
recentlyAddedTransitions(newAllPossibleNeighbors, getTransitions(matrix, nextCell));
            newAllPossibleNeighbors.addAll(getTransitions(matrix, nextCell));
            if(recursiveDistricter(matrix, freeCells, currentDistrict,
newAllPossibleNeighbors, newRecentlyAddedTransitions) == null) {
                newAllPossibleNeighbors.remove(nextCell);
                returner(matrix, freeCells, currentDistrict, nextCell);
                newAllPossibleNeighbors.removeAll(newRecentlyAddedTransitions);
            } else {
                return matrix;
            }
        }
        return null;
    }

    // chooses a cell uniformly at random from the set of free cells
    public static SquaretopiaCell randomCell(Set<SquaretopiaCell> set) {
```

```
        SquaretopiaCell chosenCell = new SquaretopiaCell(-1, -1);
        int numOfCells = set.size();
        int chosenCellIndex = (int) Math.ceil((Math.random() * numOfCells));
        Iterator<SquaretopiaCell> it = set.iterator();
        for (int i = 0; i < chosenCellIndex; i++) {
            chosenCell = it.next();
        }
        return chosenCell;
    }


    // determines which district we are constructing
    public static int getCurrentDistrictNumber(SquaretopiaMatrix matrix,
Set<SquaretopiaCell> freeCells) {
        int n = matrix.data.length - 2; // assumes matrix is a square and subtract 2
because of outer layer
        int currentDistrictNumber = (int) Math.ceil( Double.valueOf(n * n -
freeCells.size()) / n );
        return currentDistrictNumber;
    }


    // gets a specified cell's free neighbors
    public static Set<SquaretopiaCell> getTransitions (SquaretopiaMatrix matrix,
SquaretopiaCell currentLocation) {
        int curLocRow = currentLocation.row;
        int curLocCol = currentLocation.col;
        Set<SquaretopiaCell> possibleTransitions = new HashSet<>();
        if(matrix.data[curLocRow - 1][curLocCol].districtNumber == 0) { // check
availability of the cell above
            possibleTransitions.add(new SquaretopiaCell(curLocRow - 1, curLocCol));
        }
        if(matrix.data[curLocRow + 1][curLocCol].districtNumber == 0) { // check
availability of the cell below
            possibleTransitions.add(new SquaretopiaCell(curLocRow + 1, curLocCol));
        }
        if(matrix.data[curLocRow][curLocCol - 1].districtNumber == 0) { // check
availability of the cell to the left
            possibleTransitions.add(new SquaretopiaCell(curLocRow, curLocCol - 1));
        }
```

```
        if(matrix.data[curLocRow][curLocCol + 1].districtNumber == 0) { // check
availability of the cell to the right

            possibleTransitions.add(new SquaretopiaCell(curLocRow, curLocCol + 1));

        }

        return possibleTransitions;

    }


    // updates the matrix

    public static SquaretopiaMatrix updateMatrix(SquaretopiaMatrix matrix,
Set<SquaretopiaCell> freeCells, SquaretopiaCell claimedCell) {

        int claimedCellDistrict =
matrix.data[claimedCell.row][claimedCell.col].districtNumber;

        if(claimedCellDistrict == 0) {

            matrix.data[claimedCell.row][claimedCell.col].districtNumber =
getCurrentDistrictNumber(matrix, freeCells);

            matrix.data[claimedCell.row][claimedCell.col].checked = true;

        } else {

            matrix.data[claimedCell.row][claimedCell.col].districtNumber = 0;

            matrix.data[claimedCell.row][claimedCell.col].checked = false;

        }

        return matrix;

    }


    // returns a SquaretopiaCell with one neighbor if it exists

    public static SquaretopiaCell deadEnd (SquaretopiaMatrix matrix,
Set<SquaretopiaCell> allPossibleTransitions) {

        for(SquaretopiaCell cell : allPossibleTransitions) {

            if(getTransitions(matrix, cell).size() == 1) {

                return cell;

            }

        }

        return null;

    }


    // returns a SquaretopiaCell with one neighbor if it exists

    public static SquaretopiaCell isolatedCell (SquaretopiaMatrix matrix,
Set<SquaretopiaCell> allPossibleTransitions) {

        for(SquaretopiaCell cell : allPossibleTransitions) {
```

```
            if(getTransitions(matrix, cell).size() == 0) {
                return cell;
            }
        }
        return null;
    }

    // takes care of everything when claiming a free cell
    public static void claimer (SquaretopiaMatrix matrix, Set<SquaretopiaCell>
freeCells, Set<SquaretopiaCell> currentDistrict, SquaretopiaCell claimedCell) {
        freeCells.remove(claimedCell);
        updateMatrix(matrix, freeCells, claimedCell);
        currentDistrict.add(claimedCell);
    }

    // takes care of everything when returning an already claimed cell
    public static void returner (SquaretopiaMatrix matrix, Set<SquaretopiaCell>
freeCells, Set<SquaretopiaCell> currentDistrict, SquaretopiaCell returnedCell) {
        freeCells.add(returnedCell);
        currentDistrict.remove(returnedCell);
        updateMatrix(matrix, freeCells, returnedCell);
    }

    // determines the recently added transitions
    public static Set<SquaretopiaCell> recentlyAddedTransitions (Set<SquaretopiaCell>
freeCells, Set<SquaretopiaCell> transitions) {
        Set<SquaretopiaCell> recentlyAddedTransitions = new HashSet<>();
        for(SquaretopiaCell cell : transitions) {
            if(freeCells.contains(cell) == false) {
                recentlyAddedTransitions.add(cell);
            }
        }
        return recentlyAddedTransitions;
    }

}
```

```
// WeightedPartitioner.java

package partition;


import java.util.HashSet;

import java.util.Iterator;

import java.util.Set;


/**

 * Joshua Mariz 05/05/2021

 * WeightedPartitioner.java

 * WeightedPartitioner utilizes a weighting factor probability to partition an n by n
Squaretopia into n equally sized contiguous districts

 */

public class WeightedPartitioner {


    /**

     * Prepares the Squaretopia weighted partitioning process.

     * NOTE: The weighting factor probability argument must be greater than or equal
to 0 AND less than or equal to 100.

     * @param size Integer number of the Squaretopia's size

     * @param numOfTrials Integer number for the number of Squaretopias we will
partition

     * @param probability Double value for our weighting factor probability

     * @return void

     */

    public static void Partition (int size, int numOfTrials, double probability) {

        int adjustedSize = size + 2; // includes the outer layer of padding cells

        int trialsConducted = 0;

        Boolean isValidPartition;

        while(trialsConducted < numOfTrials) {

            isValidPartition = true;

            SquaretopiaMatrix Squaretopia = new SquaretopiaMatrix(adjustedSize,
adjustedSize);

            Set<SquaretopiaCell> freeCell = Squaretopia.generateSetOfFreeCells();

            Set<SquaretopiaCell> currentDistrict = new HashSet<>();

            Set<SquaretopiaCell> currentDistrictFreeNeighbors = new HashSet<>();

            Set<SquaretopiaCell> recentlyAddedTransitions = new HashSet<>();
```

```
            SquaretopiaCell claimedCell;
            while(freeCell.size() > 0) {
                claimedCell = deadEnd(Squaretopia, freeCell);
                if(claimedCell == null) {
                    claimedCell = randomCell(freeCell, null, probability);
                }
                claimer(Squaretopia, freeCell, currentDistrict, claimedCell);
                currentDistrictFreeNeighbors.addAll(getTransitions(Squaretopia,
claimedCell));
                recentlyAddedTransitions.addAll(getTransitions(Squaretopia,
claimedCell));
                if (recursiveDistricter(Squaretopia, freeCell, currentDistrict,
currentDistrictFreeNeighbors, recentlyAddedTransitions, claimedCell, probability) ==
null) {
                    isValidPartition = false;
                    break;
                }
                currentDistrict.clear();
                currentDistrictFreeNeighbors.clear();
            }
            if(isValidPartition) { // output partition and rounded compactness scores
(LW RE SB PP)
                // COMMENT OUT THE LINE BELOW TO NOT OUTPUT THE PARTITIONED
SQUARETOPIA
                Squaretopia.show();
                // COMMENT OUT THE LINE BELOW TO NOT OUTPUT THE SCORES OF THE
PARTITIONED SQUARETOPIA
                System.out.println(Math.round(Squaretopia.LengthWidth()*100) + " " +
Math.round(Squaretopia.Reock()*100) + " " + Math.round(Squaretopia.Schwartzberg()*100)
+ " " + Math.round(Squaretopia.PolsbyPopper()*100));
                System.out.println("");
                trialsConducted++;
            }
        }


    }


    // recursive algorithm
    public static SquaretopiaMatrix recursiveDistricter (SquaretopiaMatrix matrix,
Set<SquaretopiaCell> freeCells, Set<SquaretopiaCell> currentDistrict,
```

```
Set<SquaretopiaCell> allPossibleTransitions, Set<SquaretopiaCell>
recentlyAddedTransitions, SquaretopiaCell recentlyAddedCell, double probability) {

          if(currentDistrict.size() == (matrix.data.length - 2)) { // assumes matrix
is a square!

              if(matrix.validMap()) {

                  recentlyAddedTransitions.clear();

                  return matrix;

              } else {

                  allPossibleTransitions.removeAll(recentlyAddedTransitions);

                  return null;

              }

          }


          Set<SquaretopiaCell> newAllPossibleNeighbors = new HashSet<>();

          newAllPossibleNeighbors.addAll(allPossibleTransitions);

          Set<SquaretopiaCell> newRecentlyAddedTransitions = new HashSet<>();


          while(newAllPossibleNeighbors.size() != 0) {

              SquaretopiaCell nextCell = isolatedCell(matrix,
newAllPossibleNeighbors);

              if(nextCell == null) {

                  nextCell = randomCell(newAllPossibleNeighbors, recentlyAddedCell,
probability);

              }

              claimer(matrix, freeCells, currentDistrict, nextCell);

              newAllPossibleNeighbors.remove(nextCell);

              newRecentlyAddedTransitions =
recentlyAddedTransitions(newAllPossibleNeighbors, getTransitions(matrix, nextCell));

              newAllPossibleNeighbors.addAll(getTransitions(matrix, nextCell));

              if(recursiveDistricter(matrix, freeCells, currentDistrict,
newAllPossibleNeighbors, newRecentlyAddedTransitions, nextCell, probability) == null)
{

                  newAllPossibleNeighbors.remove(nextCell);

                  returner(matrix, freeCells, currentDistrict, nextCell);

                  newAllPossibleNeighbors.removeAll(newRecentlyAddedTransitions);

              } else {

                  return matrix;

              }

          }
```

```
            return null;
        }


        // chooses a cell at random from the set of free cells (this method includes the
weighting factor)
        // w, where 0 <= w <= 100, is the minimum probability of selecting the next
sequential cell (the cell that follows if we continue in the same direction)
        public static SquaretopiaCell randomCell(Set<SquaretopiaCell> set, SquaretopiaCell
recentlyClaimedCell, double w) {
            w = w / 100;
            SquaretopiaCell chosenCell = new SquaretopiaCell(-1, -1);
            SquaretopiaCell targetCell = nextSequentialCell(recentlyClaimedCell); // cell
that will be assigned a w% probability of being chosen
            int indexOfTargetCellInSet = findIndex(set, targetCell);
            if(indexOfTargetCellInSet == -1) { // each cell in the set has an equal
probability of being chosen if the targetCell is not in the set
                int numOfCells = set.size();
                int chosenCellIndex = (int) Math.ceil((Math.random() * numOfCells));
                Iterator<SquaretopiaCell> it = set.iterator();
                for (int i = 0; i < chosenCellIndex; i++) {
                    chosenCell = it.next();
                }
                return chosenCell;
            } else { // weigh targetCell with a w% probability
                double numOfCells = set.size();
                double remainingProbability = 1 - w;
                double nonTargetCellProbability;
                if(set.size() == 1) {
                    Iterator<SquaretopiaCell> it = set.iterator();
                    return it.next();
                } else {
                    nonTargetCellProbability = remainingProbability / (numOfCells - 1);
                    double previousUpperBound = 0;
                    for(SquaretopiaCell cell : set) {
                        cell.lowerBoundForProbability = previousUpperBound;
                        if(cell.equals(targetCell)) {
                            cell.upperBoundForProbability = previousUpperBound + w;
                        } else {
```

```
                        cell.upperBoundForProbability = previousUpperBound +
nonTargetCellProbability;
                    }
                    previousUpperBound = cell.upperBoundForProbability;
                }
                double randomNumber = Math.random();
                for(SquaretopiaCell cell : set) {
                    if((cell.lowerBoundForProbability <= randomNumber) &&
(randomNumber < cell.upperBoundForProbability)) {
                        return cell;
                    }
                }
            }
            System.out.println("Something went wrong");
            return null; // Something went wrong
        }
    }


    // determines the index of a given cell in some given set (returns -1 if the cell
is not in the set)
    public static int findIndex(Set<SquaretopiaCell> someSet, SquaretopiaCell
desiredCell) {
        if(desiredCell != null) {
            Iterator<SquaretopiaCell> it = someSet.iterator();
            int indexOfDesiredCell = 0;
            int maxIterations = someSet.size();
            SquaretopiaCell currentCell;
            for (int i = 0; i < maxIterations; i++) {
                currentCell = it.next();
                if(currentCell.equals(desiredCell)) {
                    return indexOfDesiredCell;
                }
                indexOfDesiredCell++;
            }
        }
        return -1;
    }
```

```
    // determines the next cell in some direction given a current cell and the current
cell's direction

    public static SquaretopiaCell nextSequentialCell(SquaretopiaCell cell) {

        SquaretopiaCell sequentialCell = null;

        if(cell != null) {

            if(cell.direction == 1) {

                sequentialCell = new SquaretopiaCell(cell.row - 1, cell.col);

            } else if (cell.direction == 2) {

                sequentialCell = new SquaretopiaCell(cell.row, cell.col + 1);

            } else if (cell.direction == 3) {

                sequentialCell = new SquaretopiaCell(cell.row + 1, cell.col);

            } else if (cell.direction == 4) {

                sequentialCell = new SquaretopiaCell(cell.row, cell.col - 1);

            }

        }

        return sequentialCell;

    }


    // determines which district we are constructing

    public static int getCurrentDistrictNumber(SquaretopiaMatrix matrix,
Set<SquaretopiaCell> freeCells) {

        int n = matrix.data.length - 2; // assumes matrix is a square and subtract 2
because of outer layer

        int currentDistrictNumber = (int) Math.ceil( Double.valueOf(n * n -
freeCells.size()) / n );

        return currentDistrictNumber;

    }


    // gets a specified cell's free neighbors

    public static Set<SquaretopiaCell> getTransitions (SquaretopiaMatrix matrix,
SquaretopiaCell currentLocation) {

        int curLocRow = currentLocation.row;

        int curLocCol = currentLocation.col;

        Set<SquaretopiaCell> possibleTransitions = new HashSet<>();

        if(matrix.data[curLocRow - 1][curLocCol].districtNumber == 0) { // check
availability of the cell above

            SquaretopiaCell possibleTransitionAbove = new SquaretopiaCell(curLocRow -
1, curLocCol);
```

```
            possibleTransitionAbove.direction = 1;

            possibleTransitions.add(possibleTransitionAbove);

        }

        if(matrix.data[curLocRow + 1][curLocCol].districtNumber == 0) { // check
availability of the cell below

            SquaretopiaCell possibleTransitionBelow = new SquaretopiaCell(curLocRow +
1, curLocCol);

            possibleTransitionBelow.direction = 3;

            possibleTransitions.add(possibleTransitionBelow);

        }

        if(matrix.data[curLocRow][curLocCol - 1].districtNumber == 0) { // check
availability of the cell to the left

            SquaretopiaCell possibleTransitionLeft = new SquaretopiaCell(curLocRow,
curLocCol - 1);

            possibleTransitionLeft.direction = 4;

            possibleTransitions.add(possibleTransitionLeft);

        }

        if(matrix.data[curLocRow][curLocCol + 1].districtNumber == 0) { // check
availability of the cell to the right

            SquaretopiaCell possibleTransitionRight = new SquaretopiaCell(curLocRow,
curLocCol + 1);

            possibleTransitionRight.direction = 2;

            possibleTransitions.add(possibleTransitionRight);

        }

        return possibleTransitions;

    }



    // updates the matrix

    public static SquaretopiaMatrix updateMatrix(SquaretopiaMatrix matrix,
Set<SquaretopiaCell> freeCells, SquaretopiaCell claimedCell) {

        int claimedCellDistrict =
matrix.data[claimedCell.row][claimedCell.col].districtNumber;

        if(claimedCellDistrict == 0) {

            matrix.data[claimedCell.row][claimedCell.col].districtNumber =
getCurrentDistrictNumber(matrix, freeCells);

            matrix.data[claimedCell.row][claimedCell.col].checked = true;

        } else {

            matrix.data[claimedCell.row][claimedCell.col].districtNumber = 0;

            matrix.data[claimedCell.row][claimedCell.col].checked = false;

        }
```

```
        return matrix;
    }

    // returns a SquaretopiaCell with one neighbor if it exists
    public static SquaretopiaCell deadEnd (SquaretopiaMatrix matrix,
Set<SquaretopiaCell> allPossibleTransitions) {
        for(SquaretopiaCell cell : allPossibleTransitions) {
            if(getTransitions(matrix, cell).size() == 1) {
                return cell;
            }
        }
        return null;
    }

    // returns a SquaretopiaCell with one neighbor if it exists
    public static SquaretopiaCell isolatedCell (SquaretopiaMatrix matrix,
Set<SquaretopiaCell> allPossibleTransitions) {
        for(SquaretopiaCell cell : allPossibleTransitions) {
            if(getTransitions(matrix, cell).size() == 0) {
                return cell;
            }
        }
        return null;
    }

    // takes care of everything when claiming a free cell
    public static void claimer (SquaretopiaMatrix matrix, Set<SquaretopiaCell>
freeCells, Set<SquaretopiaCell> currentDistrict, SquaretopiaCell claimedCell) {
        freeCells.remove(claimedCell);
        updateMatrix(matrix, freeCells, claimedCell);
        currentDistrict.add(claimedCell);
    }

    // takes care of everything when returning an already claimed cell
    public static void returner (SquaretopiaMatrix matrix, Set<SquaretopiaCell>
freeCells, Set<SquaretopiaCell> currentDistrict, SquaretopiaCell returnedCell) {
        freeCells.add(returnedCell);
```

```
        currentDistrict.remove(returnedCell);

        updateMatrix(matrix, freeCells, returnedCell);

    }


    // determines the recently added transitions

    public static Set<SquaretopiaCell> recentlyAddedTransitions (Set<SquaretopiaCell>
freeCells, Set<SquaretopiaCell> transitions) {

        Set<SquaretopiaCell> recentlyAddedTransitions = new HashSet<>();

        for(SquaretopiaCell cell : transitions) {

            if(freeCells.contains(cell) == false) {

                recentlyAddedTransitions.add(cell);

            }

        }

        return recentlyAddedTransitions;

    }


}
```

```
// PartitionTests.java
package partition;


import org.junit.Test;


/**
 * Joshua Mariz 05/05/2021
 * PartitionTests.java
 * PartitionTests allows us to run the following classes
 * - Partitioner.java
 * - SingleDistricter.java
 * - UnboundedSingleDistricter.java
 * - WeightedPartitioner.java
 * - WeightedSingleDistricter.java
 * - WeightedUnboundedSingleDistricter.java
 * NOTE: The weighting factor probability argument must be greater than or equal to 0
AND less than or equal to 100.
 */


public class PartitionTests {


    @Test
    public void Partition_t0() {
        // UNCOMMENT OUT THE FILE YOU WANT TO RUN; TRY DIFFERENT ARGUMENTS
           Partitioner.Partition(5, 2);                              //
Partitioner.Partition(Squaretopia Size, Number of Partitions);
        // SingleDistricter.Partition(5, 2);                         //
SingleDistricter.Partition(Squaretopia Size, Number of Single Districts);
        // UnboundedSingleDistricter.Partition(5, 2);                //
UnboundedSingleDistricter.Partition(Squaretopia Size, Number of Single Districts);
        // WeightedPartitioner.Partition(5, 2, 50);                  //
WeightedPartitioner.Partition(Squaretopia Size, Number of Partitions, Weighting Factor
Probability);
        // WeightedSingleDistricter.Partition(5, 2, 50);             //
WeightedSingleDistricter.Partition(Squaretopia Size, Number of Single Districts,
Weighting Factor Probability);
        // WeightedUnboundedSingleDistricter.Partition(5, 2, 50);    //
WeightedUnboundedSingleDistricter.Partition(Squaretopia Size, Number of Single
Districts, Weighting Factor Probability);
    }
```

```
}
```