

YOUR PROJECT TITLE AND ACRONYM REPLACE THIS

YOUR NAME REPLACES THIS

Assignment 3

7/2/20

For this assignment, you will describe and implement release 1 of your term project. You will incorporate *sorting* for a particular kind of circumstance, and *multithreading*. Try to build out your existing project—and paste it at the top of this document, heading and all—so that you have something substantial to show others at the end of the term, otherwise start a brand-new project.

If you lack the programming skills to implement multithreading, use pseudocode.

Submit this completed Word document, replacing all parts “Replace this …”, observing and retaining the gray text. Your materials—in black 12-point Times New Roman—should not exceed 5 pages excluding references, figures, and appendices. Use the Appendix sections for additional material if you need to. These will be read only on an as-needed basis.

3.1 SUMMARY DESCRIPTION

One- or two-paragraph overall description of your proposed term project—half-page (12-point Times New Roman) limit. By the end, term projects will incorporate most of the topics discussed in the course—so read ahead get a feel for what’s to come. To do this, you may need to alter the direction of your project or introduce an additional project. You may alter this or even replace it as the semester progresses. You will probably find it useful to use your project acronym.

The goal of the project is to create an amalgam of algorithms that can process a series of arbitrary tilings and tiling rule sets. The main algorithm that is run for this term project is an algorithm that I am working on independently of the class, and is called a PenroseAutomata. The PA algorithm is split up into two extremely computationally complex parts. The first part, is a projective algorithm that takes a dim dimensional mother lattice whose hyperplanes are offset by a shift vector shiftVect, and projects it using a function such that it can create any unitary 2D tiling. Once the tiling is constructed, it is evaluated such that neighborhoods are generated for each tiling, along with a series of statistics that are quickly measured. Then the most intensive part of the algorithm takes place which is the Automata portion of the PenroseAutomata. The tiling is automated with an arbitrary set of state change functions, which act as a generalization of Conway’s game of life but with states chosen from a state continuum .

This however is not the function of the algorithms to be created in class. The first algorithm completed for Assignment 2, was a quadTree generating algorithm. Which takes a 4 dimensionally projected tiling (a 2D grid) and converts it into a canonicalized quad tree g generations after an initial condition is set.

3.2 PROJECTED I/O EXAMPLE FROM PROJECTED COMPLETED PROJECT

Provide an example of projected concrete output for designated input. You will not be held to fulfilling exactly this—it is just explanatory at this point, to indicate where your project is going. We recognize that project direction and details will change as the term progress. This section refers to the project as a whole, not just to what you will produce this week, so we can gain an idea of what you have in mind overall.

<https://github.com/jcanedo279/PenroseAutomata>

One can look at the above repo to have a rough representation of what the algorithm does. Taking in a series of parameters that represent a tiling and an automata, the algorithm spits out statistics of the tiling automata and a gif of the automata itself. If you are to scroll to the bottom of the readme, there are a series of gifs that represent several methods of creating these and how certain parameters affect the results. Below the gifs is a better explanation of what each of the parameters inputted into the algorithm does.

Because of the extremely large nature of the output, in the form of several individually large frames, and the also large input, I will refrain from putting a direct example. But rather will explain approximately what is inputted and outputted. Furthermore because a genetic algorithm is used to optimize the emergent complexity of the automata, the example input will be even more simplified.

Inputs:

Tiling:

- Dimension and size of mother lattice
- Initial condition of lattice hyperplanes

State Space:

- Number of colors and number of states

Automata:

- Color sequence (numColors length), and color sequence generation method
- State transition matrices (three)
- Generation info. (min, max, fit, print)
- Calculate using minimum set of boundaries
- Use a boundary approximation technique
- OG Conway's Game of Life mode
- Tile outline and display options
- Information needed to create a boundary
- Information needed to create an invalid set of tiles
- Directory paths information
- Genetic algorithm information

Outputs:

Within outputData/ directory and within the poolMultigridData/ directory (for a genetic algorithms) or unfitMultigridData/ directory (for non genetic algorithms), the output is stored. Specifically, the output produces as follows:

- An animation of the Penrose automata
- detailedInfo.json and paramSafetyCheck.json, used for storing the genetic info of the automata and for storing the parameter check system output
- Color composition (magnitude and normalized percentage) vs. generation plot
- Average change vs. generation plot
- Value statistics vs. generation plot
- Tile stability vs. generation plot
- Boundary statistics vs. generation plot (if using boundaries)
- Tile fitness vs. generation plot

More concrete examples of the actual files are displayed in different levels of depth in the main README displayed at the root of the GitHub repository linked above.

In the future, one could hopefully use my term project to analyze more information on the tiling automata. Such as converting the tiling into a quad tree or performing Dijkstra's greedy search on the tiling.

3.3 REQUIREMENTS IMPLEMENTED IN THIS RELEASE

Supply functional requirements statement that you accomplished for this assignment, together with input where applicable, and output. Please try to state requirement in declarative form, as illustrated in the examples, because here we want to know the functionality intended (*what*, not *how*). The example material should be deleted. Keep in mind that the implementation of these requirements will incorporate *divide and conquer*, and that will probably influence the requirements you choose to implement in this assignment.

3.3.1 Multigrid redundant sort

Req1: Given four input parameters (x_i , x_f , y_i , y_f) (defaulted to the whole grid) representing the boundaries of a sub matrix of a larger given matrix, the sub matrix is copied from the larger given matrix and stored.

Req2: The now stored sub matrix is then sorted left to right top to bottom such that if one was to read the rows of the matrix as one would a sentence in a book, each subsequent element will be larger or equal to the last. (this is made sure of no need to check)

Req3: The sub matrix is sorted in a way such that it efficiently deals with repeated elements

Req4: The sorted sub matrix is displayed on the console

3.3.2 Display tiling multiprocessing

Req1: Given one of three settings, displayTiling() creates and/or returns an axis of tiles by multithreading, multiprocessing, or regular list comprehension.

3.4 ILLUSTRATIVE OUTPUT

Provide illustrative output from your application showing that the requirements have been met. Explain what class.method(s) produce it.

Req 3.3.1:

Input non-parameterized multigridSort() one and console output one:

The screenshot shows a Jupyter Notebook workspace titled "QuadTree.py — Untitled (Workspace)". The code cell contains Python code for initializing a QuadTree and performing a multigrid sort. The output cell shows the original tile population and the completed algorithm execution time. The bottom cell displays the resulting array of sorted coordinates.

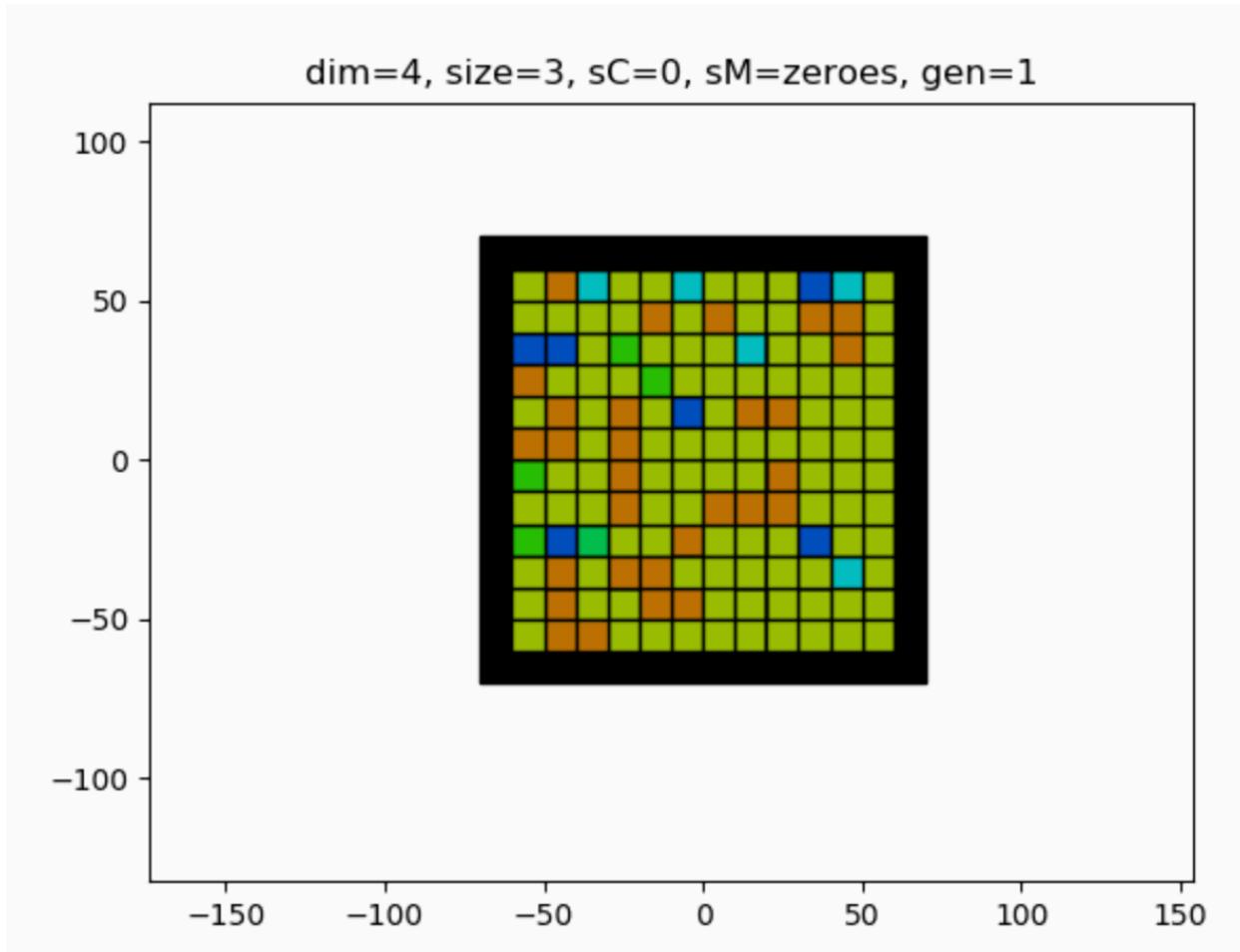
```
QuadTree.py — Untitled (Workspace)
GeneticAutomata.py MultigridList.py Multigrid.py QuadTree.py X README.md hr4.py main.py

PenroseAutomata > src > QuadTree.py > main
293
294     def main():
295         ## Minimum size is 3 excluding 4
296         size = 3
297
298         sC = 0
299
300         ## These must both be greater than 4
301         numStates = 1000
302         numColors = 10
303
304         ## Do not change these
305         initVal = (True, False, False, False)
306
307         ## How many generations into the animation we start the quadTree algorithm
308         minGen, maxGen, fitGen, printGen = 0, 0, 11, 0
309
310         tileOutline = True
311
312         borderSet, borderColor, borderVal, dispBorder = {0,1,2,3,4,5,6}, 'black', -1, True
313
314
315         quadTree = QuadTree[size, sC,
316                             numStates, numColors,
317                             initVal,
318                             minGen, maxGen, fitGen, printGen,
319                             tileOutline,
320                             borderSet=borderSet, borderColor=borderColor, borderVal=borderVal, dispBorder=dispBorder]
321
322         #quadTree.printQuadTreeNodes(quadTree.quadRoot, dispid=True)
323
324         quadTree.multigridSort()

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
Original tile populated
Grid(s) 0-2 Generated succesfully
Grid(s) 1-2 Displayed and analyzed succesfully
Algorithm Completed
Executed in 3.0493900775909424 seconds

array([[ 38.4916   ,  40.802   ,  46.5976   ,  47.017   ,
       49.2575   ,  50.2244  ,  52.11875  ,  52.3882  ,
       56.27466667,  59.38266667,  59.52175  ,
       60.69644444,  61.63325 ,  67.31875  ,
       67.6255   ,  70.289   ,  88.40466667,  89.0852  ,
       90.65525  ,  91.0982  ,  95.531   ,
       95.99625  , 101.61175 , 105.79225 , 107.072   ,
      108.55125 , 110.39425 , 112.93075 ,
      [113.29309091, 114.255  , 115.6735  , 115.842  ,
      116.9365  , 120.331  , 120.50022222],
      [125.81825 , 126.581  , 133.358  , 134.53375 ,
      143.6232  , 144.8032 , 152.49890909],
      [156.67766667, 195.619  , 215.9472 , 220.2074 ,
      348.11725 , 431.209  , 455.93125 ]])
(base) Jorges-MacBook-Pro:PenroseAutomata jorgecaneado$
```

Output displayed tiling one:



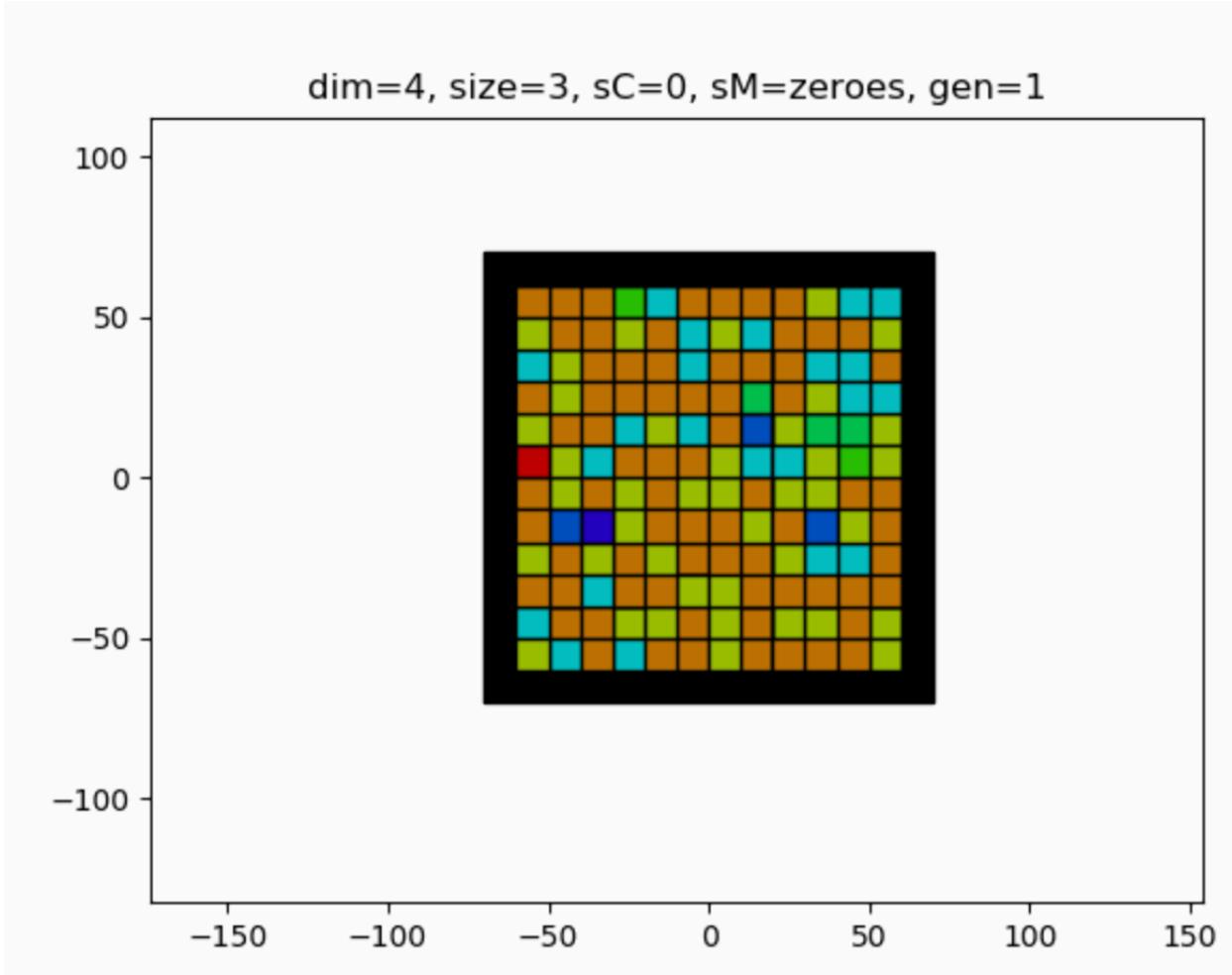
Input non-parameterized multigridSort() two and console output two:

The screenshot shows a Jupyter Notebook interface with the following details:

- File Explorer:** Shows files: GeneticAutomata.py, MultigridList.py, Multigrid.py, QuadTree.py (active), README.md, hr4.py, main.py.
- Code Cell:** Displays Python code for the `main()` function of `QuadTree.py`. The code initializes parameters like size, numStates, numColors, and generation counts, then creates a `quadTree` object and performs a multigrid sort.
- Console Output:**
 - Shows validation messages: "All Parameters Validated" and "Parameter Safety Check Passed".
 - Logs the process: "No flaws were caught on the input condition, parameterization successful, attempting build".
 - Details about tile generation: "Tile vertices generated", "Tile neighbourhood generated", "Original tile populated", "Grid(s) 0-2 Generated successfully", "Grid(s) 1-2 Displayed and analyzed successfully".
 - Completion message: "Algorithm Completed".
 - Timing information: "Executed in 3.2500200271606445 seconds".
 - A printed array of coordinates:

```
array([[ 55.028125 ,  74.6175    ,  88.8725    , 189.0125    ],
       [213.88125 , 253.8575    , 257.79    , 269.475    ],
       [273.483125 , 311.265    , 316.23333333, 360.10055556],
       [416.128125 , 534.4975    , 616.096875    , 637.49375 ]])
```
- Bottom Status Bar:** Shows "se: conda" and "Ln 32".

Output displayed tiling two:



Req 3.3.2:

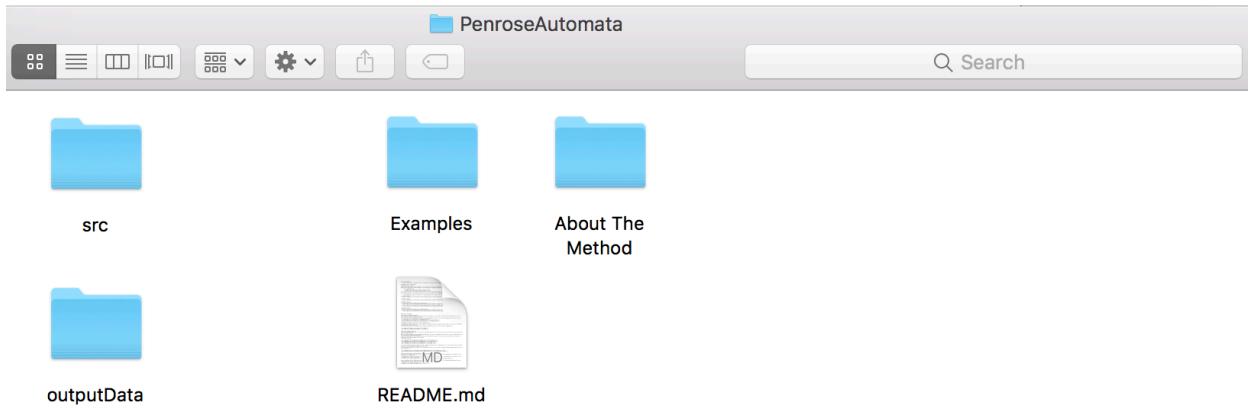
The above displayed tilings one and two are calculated via multithreading and multiprocessing respectively

3.5 YOUR DIRECTORY

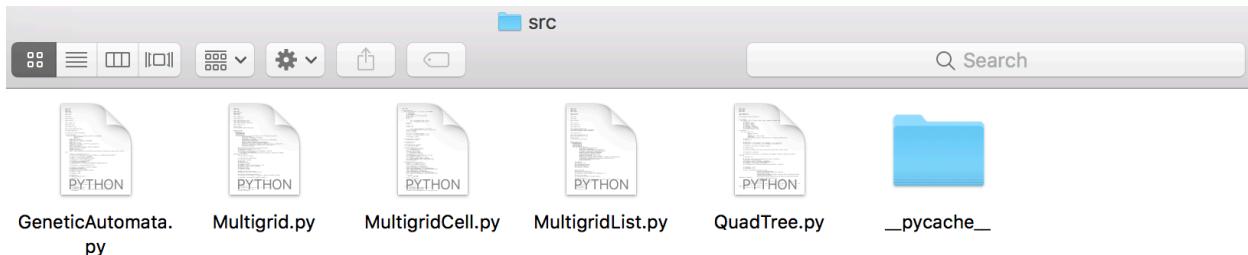
Show a screenshot of your directory, including methods, even if it consists of only one class.

The directory is a tree-like structure where the root folder comprises four sub-folders and a README.

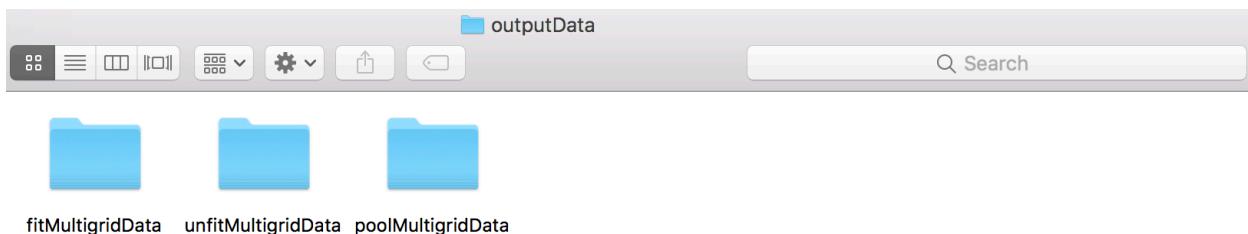
Root folder in the directory ‘PenroseAutomata/‘



The ‘/src/’ sub-directory containing all the executable scripts



The ‘/outputData/‘ sub-directory containing all the output animations/figures/statistics



‘/outputData/poolMultigridData’ comprises ‘fittest.json’ containing the genes of the fittest tilings, ‘/outputData/poolMultigridData/gaStats/’ (containing a figure of the average genetic algorithm population fitness) and folders comprising genetic algorithm outputs which are too nuanced to be discussed here.

Furthermore, ‘/outputData/fitMultigridData’ comprises all the folders of individual animations and their respective statistics where each animation survives to more than fitGen generations

'/outputData/unfitMultigridData' comprises all the folders of all remaining animations and their respective statistics

Finally, '/Examples/' and '/AboutTheMethod/' comprise examples to be used for Github and the mathematical description/ assignment documents (ie assignment photos and submission) respectively.

3.6.1 SPECIALIZED SORTING IMPLEMENTED

Your implementation should include *specialized sorting* in a manner that is useful to your application. Explain where and how you applied these, using the headings below.

The first implementation of this release is a sorting algorithm that is specific to sorting 2D rasterized arrays with lots of repeated elements. This is because, as the automata progresses more generations into the future, it becomes more and more likely for the automata to reach a potential sink, the point at which all tiles on the tiling are in their lowest potential state and unable to react in any way. Furthermore, the sorting method multigridSort() takes in four optional parameters which index a sub matrix of a given larger matrix. One day, it would be fairly easy to sort subsections of the quadTree, ie sorting specific nodes where each node contains a sub matrix.

The grid is to be sorted left to right top to bottom as if reading the rows of the matrix as one would lines in a book would be reading in strictly increasing order

A matrix:	can be sorted into	An output matrix:
[[9, 8, 7]		[[0, 0, 0]
[6, 0, 0]		[0, 0, 6]
[0, 0, 0]]		[7, 8, 9]]

To do this we can create a dictionary of the unique values in the grid along with their respective number of occurrences, this can be done in Theta($n^{**}2$) time complexity where $n=2^k$ is the side length of the square sub matrix. We then sort the keys natively by quick sort in $O(u^{**}2)$ time where u is the number of unique elements in the grid. Finally the sorted values are re-placed into the grid in Theta($n^{**}2$) time. Overall this algorithm performs sorting a matrix in $\Theta(n^{**}2)+O(u^{**}2)$.

One day, it would be fairly easy to sort subsections of the quadTree, ie sorting specific nodes where each node contains a sub matrix.

3.6.2 SORTING AND MULTITHREADING IMPLEMENTED

Your implementation should include *concurrency* in a manner that is useful to your application. Explain where and how you applied these, using the headings below.

Using cProfiler to profile the code, we can see how much time is spent in each of the methods in order to find out which methods are most worth multithreading. We must also add some restrictions to this profiling method, that being that whatever we multithread cannot be sequential. That is, multithreading cannot be applied to an output that builds off of itself, ie I cannot multithread popIter (the method that calculates the state of the grid based off of the previous state) since the next state of the tiling is dependent on the previous state.

However one method does stick out, a particularly costly method. This method is displayTiling() (or the analogous displayBoundaries() which displays the tiling of a bounded tiling). displayTiling() is an ideal method because it does not build upon previous work and because it is so time consumptive.

displayTiling() works by iterating over each tiling and creating and properly coloring a patch object (containing the vertices of a polygon) which it adds to a list of patch objects. Ideally, several threads could compile this faster than a single thread, but with Python's Global Interpreter Lock which limits global python interpretation to one per process, Python is unable to take advantage of parallel threading in any way that I have been able to find. After several attempted implementations, two have stuck out. Using multithreading pools and multiprocessing pools. While multiprocessing pools do execute in parallel, the time complexity due to overhead of transferring the pools around is slightly larger than the time complexity of calculating the result, making it ineffective. And while multithreading cannot execute concurrently in python, this approach to multithreading would work in many other languages (ie Java or Cpp), yet multithreading will still execute in the same amount of time as not multithreading so no harm no fowl.

3.6.1 Class model and Sequence Diagram

Identify where you included *sorting* and *multithreading*. To do this use tools (e.g., Visio), PowerPoint, or a combine models as in [this example](#) (which you are free to cut and paste from). Insert indications in red (as in [this example](#)) to show where the three features below apply.

Relation	Class	Main Functions	
implements	Genetic Automaton	genPop() evaluatePop() genPool() crossPool()	genGenes() mutateGene() genPopFromSave()
overloaded contains	Quadtree	genValMap() makeQuadTree() makeQuadNodes()	multigridSort() printQuadTreeNodes()
overloaded contains	Quadt Node	N/A	
overloaded contains	Multigrid List	parameterSafetyCheck() updateAnimations() genBounds() genDirectoryPaths()	saveAndExit()
overloaded contains	Multigrid	genShiftVector() genTilingVerts() genTilingNeighbourhoods() genTiling()	genNextNaturalGridState() genNextTile() genBoundaryList() displayTiles()
overloaded contains	Multigrid Cell	setVertices() setVal() setStability() setHyperplaneEqn()	getPointCoord() genNormalVert() setColor()

3.1
specialized
sort
implemented

3.2
multithreading
implemented

3.6.2 Code showing sorting.

Show the relevant code (only) and explain why specialized sorting is appropriate here. It should be clear where the code is located (class and method).

```
QuadTree.py — Untitled (Workspace)
① GeneticAutomata.py ② MultigridList.py ③ Multigrid.py ④ QuadTree.py X ⑤ README.md ⑥ hr4.py ⑦ main.py •

PenroseAutomata > src > QuadTree.py > main
230     if currNode.topRight != None:
231         self.printQuadTreeNodes(currNode.topRight, allNodes=allNodes, dispid=dispid)
232
233     def multigridSort(self, xi=0, xf=-1, yi=0, yf=-1):
234         ## Axiom0: All functions and data structures such as the QuadTree and QuadNode classes are implemented correctly
235         # This means that all states (and there are many) that this algorithm accumulates on is taken axiomatically as correct
236         # even if that is not so (as it is).
237
238         ## Intent: The intent of this function is to sort a sub matrix (defined by the indices xi,xf,yi,yf) of self.valMap, in such a way that
239         # repeated elements are efficiently sorted from left to right, then top to bottom as if reading lines from a book where the lines are in
240         # increasing order. Finally this sorted sub matrix is displayed on the console
241
242         ## Prec0: self.valMap contains a perfect squared matrix (side lengths of n=2^k) with values in the closed interior [0, numStates]
243         ## Prec1: xi, xf, yi, yf are valid integer indices of self.valMap in the two grid directors.
244         ## Prec2: xf>xi and yf>yi, additionally xf-xi==yf-yi
245
246         ## Post0: self.valMap is unaltered
247         ## Post1: outMap contains a square matrix with values sorted as described in the intent
248         ## Post2: outMap is neatly displayed on the console
249         ## Post3: xf>xi and yf>yi, additionally xf-xi==yf-yi
250
251         ## Invar0: self.valMap is unaltered
252         ## Invar1: xf>xi and yf>yi, additionally xf-xi==yf-yi
253
254         ## State 1: The final x and y coordinates of the submatrix are set to the maximum dimmension of the matrix if they are -1
255         if xf == -1:
256             xf = len(self.valMap)-1
257         if yf == -1:
258             yf = len(self.valMap)-1
259
260         ## State 2: The matrix is copied to ensure no information is lost, furthermore we only use the submatrix bounded by the input parameters
261         valMap = np.array(self.valMap)[np.ix_([xi,xf],[yi,yf])]
262
263         ## State 3: An empty array is used to store the sorted grid
264         outArray = [0] * (len(valMap))**2
265
266         ## State 4: For each element of the matrix, the item and its respective number of occurrences is added to a dictionary val0ccurrences representing
267         # the number of currences of each unique value in the sub matrix
268         val0ccurrences = {}
269         for row in valMap:
270             for item in row:
271                 if item in val0ccurrences:
272                     val0ccurrences[item] = val0ccurrences[item] + 1
273                 else:
274                     val0ccurrences[item] = 1
275
276         ## State 5: The keys of the dictionary (a list of unique values), are sorted (done natively by quicksort)
277         sortedKeys = sorted(val0ccurrences.keys())
278
279         ## State 6: The sorted keys are added to the output arrary outArray in their respective number of occurrences
280         i = 0
281         outArray = [0] * (len(valMap))**2
282
283         for value in sortedKeys:
284             ## State 6.valueInd: For each value in the sorted keys list, the value is added val0ccurrences[value] times,
285             # where valueInd is the index of value in sortedKeys
286             o = 0
287             while(o<val0ccurrences[value]):
288                 outArray[i] = value
289                 i += 1
290                 o += 1
291
292             ## State 7: outArray is reshaped using numpy from a 1D list back into a 2D non-rasterized matrix
293             outMat = np.array(outArray).reshape(len(valMap), len(valMap))
294
295             ## State 8: The output matrix outMat is displayed on the console, preferably nicely formatted too
296             #print('\n'.join([''.join(['{round(item, 3)}' for item in row]) for row in outMat]))
297             pprint(outMat)

se: conda) ⑧ 0 △ 1
```

3.6.3 Code showing multithreading

Show the relevant code (only) and explain why multithreading is appropriate here. It should be clear where the code is located (class and method).

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** GeneticAutomata.py, MultigridList.py, Multigrid.py (highlighted), QuadTree.py, README.md, hr4.py, main.py.
- Code Content:** The Multigrid.py file contains Python code for multithreading. It includes comments explaining the intent of the code, such as displaying tiles, creating patches, and handling different multiprocessor states (multithread or multiprocessing). The code uses loops to iterate over tile dimensions and a ThreadPool to parallelize patch creation.
- Bottom Status Bar:** Shows the current environment (e: conda), line count (Ln 767), column count (Col 36), spaces used (Spaces: 4), encoding (UTF-8), line feed (LF), and the language (Python).

```
## Display all tiles
def displayTiling(self, animating=True):
    ## Axiom0: All functions and data structures such as the QuadTree and QuadNode classes are implemented correctly
    # This means that all states (and there are many) that this algorithm accumulates on is taken axiomatically as correct
    # even if that is not so (as it is).
    ## Axiom1: If multip is 'multithread', there are at least 6 available threads
    ## Axiom2: If multip is 'multiprocess', there are enough available cpus such that os.cpu_count()-1 is greater than or equal to 1

    ## Intent: The intent of display tiling is to iterate over all tiles in the tiling and add the proper patch corresponding to a tiling to the figure.
    # Furthermore, we can do this in one of three ways, by creating a pool of threads, by creating a pool of processes, and by traditionally iterating linearly
    # over the loop with a single process and thread. Because of Python's GIL (Global Interpreter Lock), we are unable to take advantage of multithreading in any
    # way that I could find, though this would work in other languages.

    ## Pre0: animating is an input parameter, either True or False
    ## Pre1: createPatch is a function that takes in a tuple of four integers (r,a,s,b) st: 0<=r<=s<self.dim and -self.size<=a=b<=self.size
    ## Pre2: createPatch is implemented properly (being outside the scope of this project) and returns a patch that can be easily plotted in the axis self.ax

    ## Post0: self.patches is a list containing all the patches in the tiling, where each patch is a plottable object comprising vertices, colors, opacities, etc..
    ## Post1: self.ax contains the patches in self.patches
    ## Post2: If the PA is not automated, we save the tiling frame in a folder containing all the tiling frames in the PA animation
    ## Post3: If the tiling is animated and the original tiling, the axis self.ax is manually returned

    ## State0: multip in {'multithread','multiprocess',all other inputs}, and the figure is handled by an auxiliary helper method
    self.setFigExtras()
    multip = False
    ## State1: we conditionally map {'multithread','multiprocess',all other inputs}-->(State1.0.0, State1.1.0, State1.2.0)
if multip == 'multithread':
    ## State1.0.0: Import all necessary multithreading modules
    from multiprocessing.dummy import Pool as ThreadPool
    ## State1.0.1: For each of the tiles, the input parameter to the createPatch function is added to a list
    inputs = []
    for r in range(self.dim):
        for a in range(-self.size, self.size+1):
            for s in range(r+1, self.dim):
                for b in range(-self.size, self.size+1):
                    ## State1.0.1.0: A representative createPatch(inputCoords) input is created
                    inputC = (r, a, s, b)
                    ## State1.0.1.1: The representative input is added to the list of inputs
                    inputs.append(inputC)
    ## State1.0.2: A ThreadPool of six thread workers is created
    pool = ThreadPool(6)
    ## State1.0.3: The ThreadPool maps the createPatch function onto all the input parameters
    patches = pool imap(self.createPatch, inputs)
    pool.close()
    pool.join()
    ## State1.0.4: All the patches are added to the figure for display
    self.patches = []
    for patch in patches:
        self.ax.add_patch(patch)
        self.patches.append(patch)
elif multip == 'multiprocess':
    ## State1.1.0: All necessary multithreading modules
```

Multigrid.py — Untitled (Workspace)

```

PenroseAutomata > src > Multigrid.py > Multigrid > displayTiling
162     ## State1.0.3: The threadpool maps the createPatch function onto all the input parameters
783     patches = pool imap(self.createPatch, inputs)
784     pool.close()
785     pool.join()
786     ## State1.0.4: All the patches are added to the figure for display
787     self.patches = []
788     for patch in patches:
789         self.ax.add_patch(patch)
790         self.patches.append(patch)
791     elif multip == 'multiprocess':
792         ## State1.1.0: All necessary multithreading modules
793         from multiprocessing import get_context
794         ## State1.1.1: For each of the tiles, the input parameter to the createPatch function is added to a list
795         inputs = []
796         for r in range(self.dim):
797             for a in range(-self.size, self.size+1):
798                 for s in range(r+1, self.dim):
799                     for b in range(-self.size, self.size+1):
800                         ## State1.1.0: A representative createPatch(inputCoords) input is created
801                         inputC = (r, a, s, b)
802                         ## State1.1.1: The representative input is added to the list of inputs
803                         inputs.append(inputC)
804         ## State1.1.2: A MultiProcessPool along with a context_switch, enable multiprocessing in Python3 (applies on dispPool after with:)
805         with get_context("spawn").Pool(os.cpu_count()-1) as dispPool:
806             ## State1.1.2.0: A Multiprocess Pool maps the createPatch function onto all the input parameters
807             patches = [item for item in dispPool imap(self.createPatch, inputs)]
808             dispPool.close()
809             dispPool.join()
810         ## State1.1.3: All the patches are added to the figure for display
811         self.patches = []
812         for patch in patches:
813             self.ax.add_patch(patch)
814             self.patches.append(patch)
815     else:
816         ## State1.2.0: All necessary multithreading modules
817         self.patches = []
818         for r in range(self.dim):
819             for a in range(-self.size, self.size+1):
820                 for s in range(r+1, self.dim):
821                     for b in range(-self.size, self.size+1):
822                         ## State1.2.0.0: For All input parameters, the patch is constructed and added to the list of patches and figure linearly
823                         patch = self.createPatch([r,a,s,b])
824                         self.ax.add_patch(patch)
825                         self.patches.append(patch)
826         ## State2: If the pt is the original, the first figure is returned manually
827         if animating and self.ptIndex==0:
828             return self.ax
829         ## State3: If the PA is not animated, the first figure is saved manually to the IO filestruct
830         if not animating:
831             self.savefig()
832         ## State4: If we get this far, the figure is automatically saved by an external function
833
834

```

: conda) ⊗ 0 △ 1 Ln 767, Col 36 Spaces: 4 UTF-8

3.7 YOUR CODE

Unless your facilitator requests another method, copy your Eclipse project to your file system, zip it, and attach it. Please contact your facilitator in advance if you want to request an alternative means.

<https://github.com/jcanedo279/PenroseAutomata> is a repository comprising a master branch and a branch for each assignment, for assignment three, see the branch of assignment three to see the code representative of the images and descriptions above.

3.8 Instructor's Evaluation

Criterion	D	C	B	A	Letter Grade	%
Technical Correctness	No justification of correctness	Technically mostly correct; tested	Implementation correctness well justified; well specified and tested	Implementation correctness thoroughly justified throughout by precise block- and line-specifications and tests.		0.0
Clarity in Presentation	Unclear	Somewhat clear; some comments	Clear with a few exceptions. Well commented.	Entirely clear throughout; very well commented at high and low levels.		0.0
Depth and Thoroughness of Coverage	Shallow or superficial coverage of most topics	Satisfactory depth and thoroughness	Evidence of depth and thoroughness in covering most topics	Evidence of depth and thoroughness in covering all topics		0.0
					Assignment Grade:	0.0
The resulting grade is the average of these, using A+=97, A=95, A-=90, B+=87, B=85, B-=80 etc. The maximum is 100.						
To obtain an A grade for the course, your weighted average should be >93. A->=90. B+>=87. B:>83. B->=80 etc.						

Appendix 3.1 (if needed; should be referenced above, and will be read as-needed only)

Appendix 3.2 (if needed; should be referenced above, and will be read as-needed only)