An Analysis of Circle Packing Algorithms

Leo J Sipowicz
University of Colorado
Computer Science Undergrad
Boulder, US
lesi8860@colorado.edu

*A. **Abstract*—The purpose of this project is to apply the knowledge gained from ECEN 2703 (Discrete Math Compt Engineers) to better understand and solve a problem. I used my knowledge from the class to build two different circle packing algorithms. I compared and contrasted these algorithms using big(O) notation and by testing run times and function calls.**

II. Problem Statement

Create a circle packing program in Python. The algorithm will be passed an area in x by y dimensions corresponding to pixels and an array of circles with different radiuses. It will fit as many circles from the array into the area as possible and return a 2d array with the radius of the circle and the x y coordinate of where to position it to maximize space.

III. Minor functions

A. Main()-figure 1

The main() function is the heart of the program, and understanding its layout is critical for understanding the algorithms used to pack the circle. The main() function has five main steps (1) the desired window size is chosen. This is done with two integer variables, one for the x dimension and one for the y, in units pixels. This will affect the algorithms later by determining how much space there is to fit circles on the page. (2) The next step is to create an array of the desired size depending on how many circles need to be packed; this is done by creating a NumPy array filled with zeros to start. (3) This array is then looped through and filled with desired circle radiuses. There are a few ways of doing this each of which will affect the algorithms differently. Possible options are constant sizes and radiuses of ascending and descending order. However, random numbers between 4 and 15 were used to assign radiuses to my circles for the tests because it most closely resembles an actual use scenario for the algorithms. This will be explored in detail later about which kinds of arrays each function handles best. (4) The fourth step in the main() function is the critical step of passing the array of circle radiuses along with the window size to my algorithm pack() and being returned a new 2d array with the radius and packed x and y locations within the window size. (5) The last step is to pass the newly created packed array to a function called display(), where each circle is plotted with its radius at the x and y coordinates attached to it. This is useful for verifying the correctness of the algorithms.

Figure 1

```
if __name__ == '__main__':
    windowSizeX = 500
    windowSizeY = 500

    circleArray = np.empty(500, dtype=int)
    for t in range(circleArray.size):
        circleArray[t] = random.randint(4, 15)
    FinishedArray = randomGuessCirclePack(windowSizeX, windowSizeY, circleArray)
    print("Calls to circleLocValid: " + str(circleLocationCounter))
    display(windowSizeX, windowSizeY, FinishedArray)
```

B. Display()-figures 2 and 3

The display function is a simple little function that displays the circles that have been packed into the screen. The function takes an int X window size, int Y window size, and FinishedArray[][] structured as (figure 3). The display function relies on the graphics.py file created by John Zelle to create a window and display the circles. It works by creating a window of the size display that was passed and then looping through the finishedArray, creating a point at the x and y coordinates then drawing a circle of radius r. It then displays the circles to the window and closes on mouse click.

Figure 2

```
def display(windowSizeX, windowSizeY, FinishedArray):
    win = GraphWin("display", windowSizeX, windowSizeY)
    t = 0
    for x in FinishedArray:
        t = t + 1
        pt = Point(x[1], x[2])
        cir = Circle(pt, x[0])
        cir.draw(win)
    win.getMouse()
    win.close()
```

Figure 3

Example of FinishedArray[][]

(int)Radius	(int)xPosition	(int)yPosition
10	230	280
5	10	80

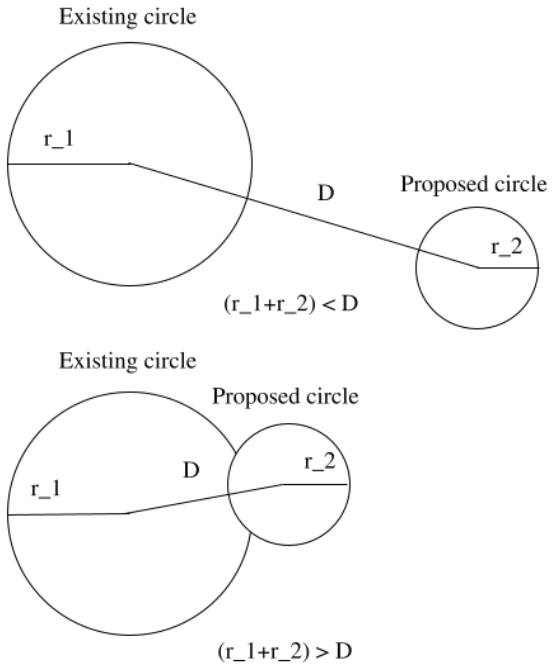
C. *circleLocationValid()-figure 4*

This is an essential function for the program's overall efficiency because it is called before every circle is placed to make sure that it is not intersecting an edge or another circle. The algorithm works by first checking if coordinates + radius are larger than x and y dimensions or if the coordinates - radius is smaller than 0, indicating that the circle is out of bounds. If the circle is out of bounds, the function returns false. After that's checked, the algorithm loops through all the circles already created and checks if the distance between the already created circle is less than the radius of the already completed circle plus the new circle indicating that the borders overlap (see *figure 4.1*). Suppose the circles overlap between the new circle and any circle created before the function returns false. If all of the tests are passed, it returns true.

Figure 4

```
def circleLocationValid(windowSizeX, windowSizeY, xLocation, yLocation, radius, finishedArray, x):
    for x in range(x):
        distance = math.dist([xLocation, yLocation], [finishedArray[x][1], finishedArray[x][2]])
        if xLocation + radius > windowSizeX or xLocation - radius < 0
        or yLocation + radius > windowSizeY or yLocation - radius < 0:
            return False
        elif distance < radius + finishedArray[x][0]:
            return False
    return True
```

Figure 4.1



Math used to determine if two circles overlap in circleLocationValid()

IV. Circle Packing Algorithms

A. *randomGuessCirclePack()(iteration-1)-figure-5*

This algorithm is a straightforward circle packing algorithm. All it does is loop through every circle in the initial circle array and guess a random x and y coordinate for it to go. It then checks if this location overlaps with another circle using the circleLocationValid() function, and if it is valid, it is added to the finishedArrayList. If the location is not valid (overlapping another circle or hitting edge), the loop is set back, and the algorithm tries again. This algorithm is helpful in its simpleness but does have some issues dealing with complexity. The guess check method works very well initially but can slow down a lot with more extensive arrays; especially when looking for the last few holes that would perfectly fit the given circle, it can take a long time. Because of this, there is a feature of the function where if spots were tried and failed 100 times the initial length of the array, the function would give up and return what it had made. The other large issue of the algorithm is that it is much harder to find a spot for a bigger circle to fit at the end of the loop than at the start. Because by the end, most spaces are taken and not always used efficiently.

1) *Test Results:*

For the given test input of a size 500 array of circles ranging from size 4 to 15, this algorithm averaged on 10 attempts successfully packing 347 of the total 500 circles and used 50348 average calls to circleLocationValid().

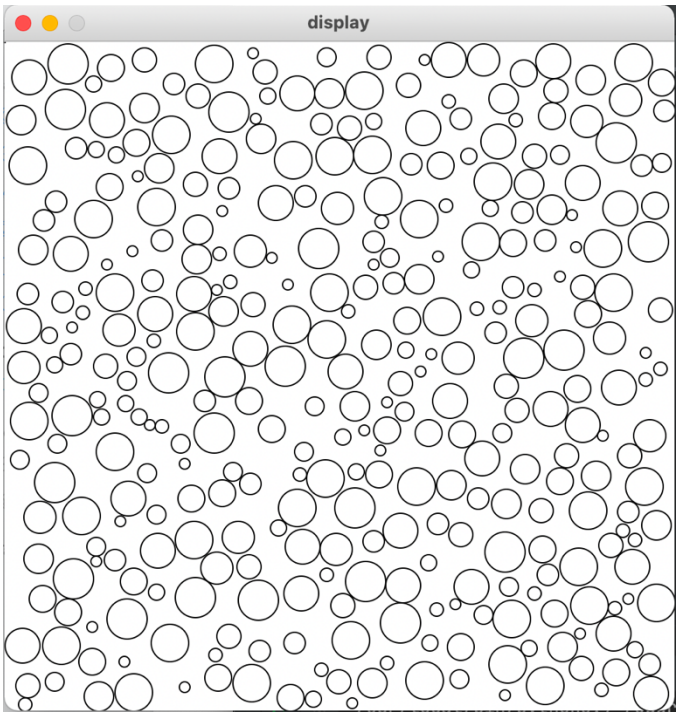
An example of the output can be seen in *figure 5.1*

Figure 5

```
def randomGuessCirclePack(windowSizeX, windowSizeY, circleArray):
    global circleLocationCounter
    finishedArray = np.zeros([circleArray.size, 3])
    x = 0
    failCounter = 0
    while x < circleArray.size:
        xLocation = random.randint(0, windowSizeX)
        yLocation = random.randint(0, windowSizeY)
        radius = circleArray[x]

        circleLocationCounter = circleLocationCounter + 1
        if circleLocationValid(windowSizeX, windowSizeY, xLocation, yLocation, radius, finishedArray, x):
            finishedArray[x] = [circleArray[x], xLocation, yLocation]
        else:
            x = x - 1
            failCounter = failCounter + 1
            if (failCounter == circleArray.size * 100):
                return finishedArray
        x = x + 1
    return finishedArray
```

Figure 5.1



This is the result of running an array of 500 circles of random size radius (between 4-15) though the randomGuessCirclePack()(iteration-1) algorithm. It was able to pack only 338 circles.

B. *randomGuessCirclePack()(iteration-2)-figure-6*

This algorithm is the same as the first circle packing algorithm(figure 5), with the addition of a few lines of code that sort the list into decreasing order at the start of the algorithm. This has advantages and disadvantages; the advantages are that if there is enough space to pack all the circles on the page, it is much more effective than the first iteration of the algorithm and will always produce a final array with more of the initial circles being able to fit on the screen. The issues arise when there is not enough space on the screen for all the circles, or the algorithm is unlucky with placement. If this happens, the algorithm will only print the largest ones, sacrificing the number of circles that will fit and the correctness of the programming assuming a use case where the order of the circles to be packed is essential. If this is the case, this algorithm is incorrect because it will try only the largest circles first.

1) *Test Results:*

For the given test input of a size 500 array of circles ranging from size 4 to 15, this algorithm averaged on 10 attempts to pack 472.6 circles and used 26459.2 average calls to circleLocationValid(). Although it should be noted that if attempted on a larger array, that was not possible to fit, this algorithm would prove incorrect based on the problem statement, and it is only correct when there is enough space for all the circles. There was enough space for all circles 8-10 tries when this algorithm was tested, resulting in a perfect score of 500 circles being packed.

An example of the output can be seen in *figure 6.1*

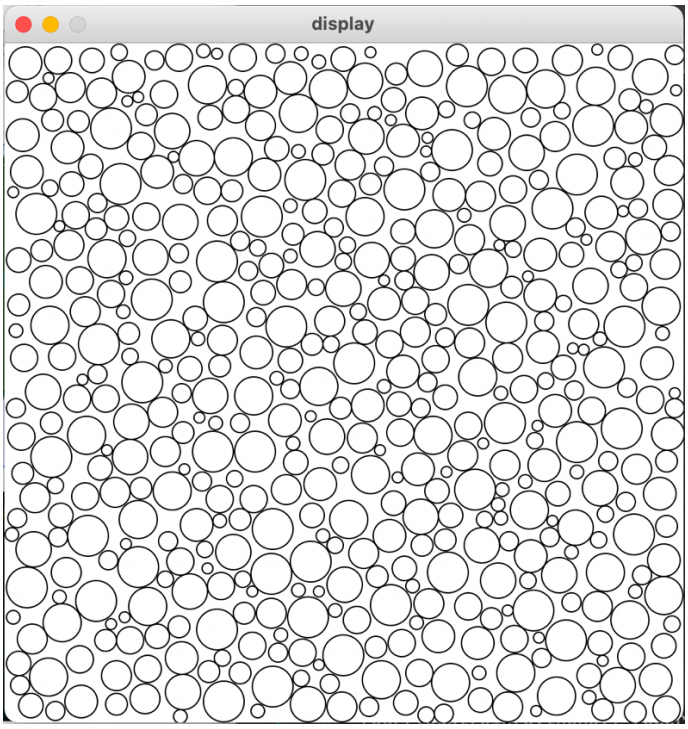
Figure 6

```
def randomGuessCirclePack(windowSizeX, windowSizeY, circleArray):
    global circleLocationCounter
    circleArray = np.sort(circleArray)
    circleArray = circleArray[::-1]
    finishedArray = np.zeros([circleArray.size, 3])
    x = 0
    failCounter = 0
    while x < circleArray.size:
        xLocation = random.randint(0, windowSizeX)
        yLocation = random.randint(0, windowSizeY)
        radius = circleArray[x]

        circleLocationCounter = circleLocationCounter + 1
        if circleLocationValid(windowSizeX, windowSizeY,
                               xLocation,yLocation, radius, finishedArray, x):

            finishedArray[x] = [circleArray[x], xLocation, yLocation]
        else:
            x = x - 1
            failCounter = failCounter + 1
            if (failCounter == circleArray.size * 100):
                return finishedArray
        x = x + 1
    return finishedArray
```

Figure 6.1



This is the result of running an array of 500 circles of random size radius (between 4-15) though the randomGuessCirclePack()(iteration-2) algorithm. It was able to pack all 500 circles.

C. *circularPack() figure-7*

The circular pack algorithm works by first creating a point in the middle of the screen of the given radius of the first circle in the array. After this circle is placed in the middle of the screen and added to the FinishedArray, a loop begins looping through each finished circle and filling its perimeter with circles of the given size. This is done until there are no more circles that can be filled with circles. The circle perimeters are filled with the function fillWithCircs() (figure 8). The fillWithCircs() function works by looping 360 degrees around the passed circle and checking using circleLocationValid() to see if the position is allowed. If this position is allowed, the circle is added to FinishedArray, and the loop continues until all 360 degrees are tested and the allowed circles are added. This function also relies on the constant amountOfPack that determines how large the increments in the loop should be. This determines the packing density by choosing how often around the 360 degrees a circle is tested. If this number is too high, it can cause significant gaps in the final pack, and if it is too low, the program will run very slowly.

Because the circular pack algorithm always places circles on a border with another circle, it maximizes space in a way that randomGuessCirclePack() cannot. Because every circle borders another, if a very low amountOfPack value is used, it will always be able to pack in more circles in the correct order than randomGuessCirclePack().

Another advantage of this algorithm is that without a special function that counts up failed attempts, this function can effectively max pack a given square with the given array when passed a very large number. Because this algorithm stops when all circles are tested for 360 degrees of places to add circles, when this is finished, the algorithm stops without wasting extra resources. An additional feature of this algorithm could be a second loop to find missed spots that circles could fit in the first loop. Because of the nature of packing random size circles, there are occasions when a large circle can not be packed in many places, but if that circle was retested with a smaller circle in the array, a spot might be found.

1) *Test Results:*

An amountOfPack value of 10 was used for the tests because it seems to be a good compromise between speed and pack density. For the given test input of a size 500 array of circles ranging from size 4 to 15, this algorithm averaged on 10 attempts to pack 500 circles and used 16942.2 total calls to circleLocationValid(). This means that the algorithm was able to pack 500 circles every time, unlike the previous functions.

An example of the output can be seen in *figure 7.1*

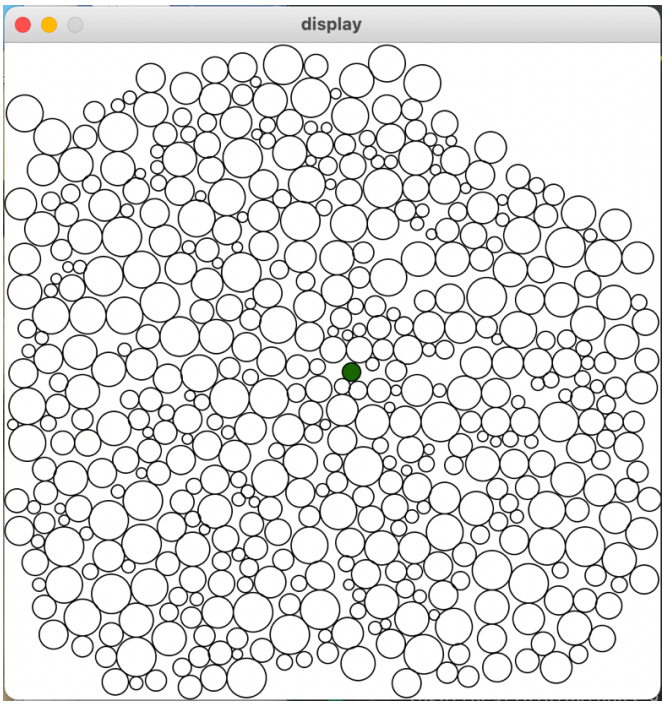
Figure 7

```
def circularPack(windowSizeX, windowSizeY, circleArray):
    global circlesUsed
    finishedArray = np.zeros([circleArray.size, 3])

    # start in middle of square
    radius = circleArray[0]
    xPos = windowSizeX / 2
    yPos = windowSizeY / 2
    finishedArray[0] = [radius, xPos, yPos]
    for t in range(inputCircleSize):
        if finishedArray[t][0] != 0:
            fillWithCircs(finishedArray[t], circleArray, finishedArray)

    return finishedArray
```

Figure 7.1



This is the result of running an array of 500 circles of random size radius (between 4-15) though the circularPack()algorithm. It was able to pack all 500 circles. Here the green circle indicates the first one made and therefore the first one packed.

Figure 8

```
def fillWithCircles(inputCircle, circleArray, finishedArray):
    global circlesUsed
    global circleLocationCounter
    inputRadius = inputCircle[0]
    inputXpos = inputCircle[1]
    inputYpos = inputCircle[2]

    x = 0
    while x < 360:
        if circlesUsed < inputCircleSize:
            newRadius = circleArray[circlesUsed]
            distanceBetween = newRadius + inputRadius
            proposedXpos = (distanceBetween * math.cos(math.radians(x))) + inputXpos
            proposedYpos = (distanceBetween * math.sin(math.radians(x))) + inputYpos

            circleLocationCounter = circleLocationCounter + 1
            if circleLocationValid(windowSizeX, windowSizeY, proposedXpos, proposedYpos, newRadius, finishedArray):
                finishedArray[circlesUsed] = [newRadius, proposedXpos, proposedYpos]
                circlesUsed = circlesUsed + 1
        x = x + amountOfPack
```

V. Time Complexity of randomGuessCirclePack() circularPack()

A. *Time complexity of circleLocationValid()-figure 9*

To find the Big O of randomGuessCirclePack() and circularPack(), we first need to find the big O of circleLocatioValid(), which each function calls in their execution. circleLocatioValid()'s time complexity is . This is because the amount of time it takes to execute is determined by the number of circles that have already been completed that it must loop through and check if they interfere with the proposed circle.

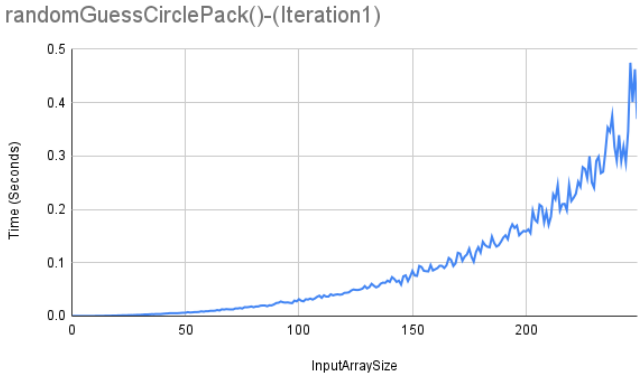
B. *Time complexity of randomGuessCirclePack()-figure 10*

RandomGuessCirclePack() has a time complexity of $O(n^2)$. This is because the programs loops for each circle in the initial array, and for each circle in the initial array, it must call circleLocatioValid(), which we know has a time complexity of O(n). These two loops compound, causing a total time complexity of $O(n^2)$. While sorting the array as seen in iteration two does make the program run faster, it does not make the time complexity smaller it can only be seen as a best-case scenario for n because more of the ifs will not be triggered.

C. *Time complexity of circularPack()-figure 11*

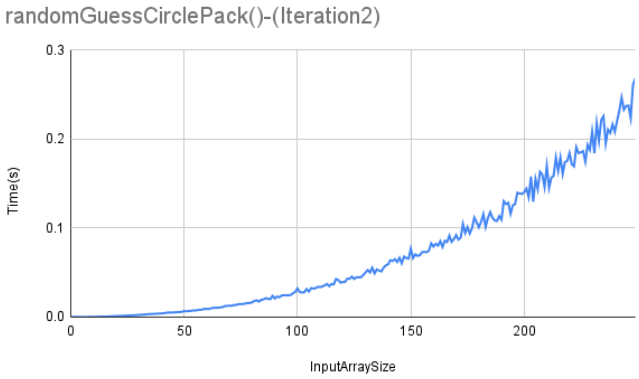
CircularPack() has a time complexity of $O(n^3)$. This is because of the three embedded loops used to pack the circles. The first loop is iterating through each circle created; this calls another loop which iterates through 360 degrees of the circle which then calls randomGuessCirclePack(), which we know has a time complexity of O(n). Even though variables like the amountOfPack can be changed to make the algorithm faster, they can not decrease the time complexity and only decrease the n.

figure 9



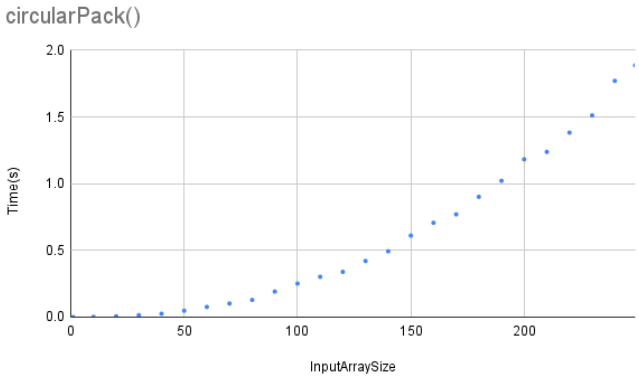
This graph shows the time the randomGuessCirclePack() (iteration 1) algorithm took to pack depending on the number of circles in the passed array. The circles were generated in the same way as before. This graph has clear exponential growth. The graph is a bit sporadic but this is due to the random nature of the algorithm. Sometimes its placement is lucky and doesn't need to try as many circles and sometimes it is not lucky and can not find a place for the circles.

Figure 10



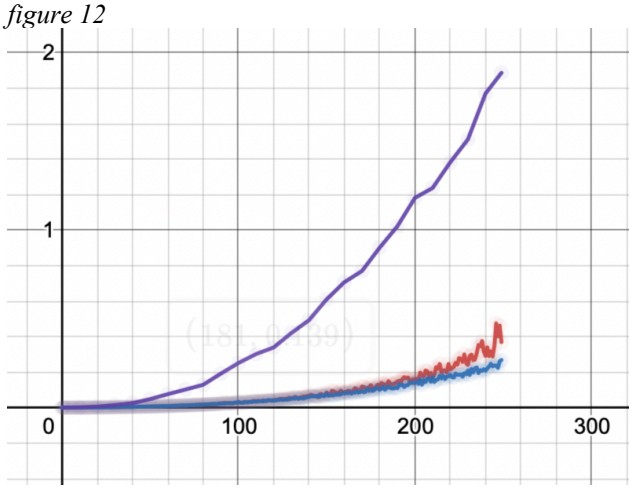
This graph shows the time the randomGuessCirclePack() (iteration 2) algorithm took to pack depending on the number of circles in the passed array. This algorithm is faster for packing the square for all input under 250 as compared to iteration 1. This is the because of the ideal nature of sorting circles by radius before they are packed. Sorting the array first also makes the graph less sporadic because its slightly less random if a space is occupied. Although arrays of all sizes less than 250 are faster to pack the growth rate is the same.

figure 11



This graph shows the time the circularPack() algorithm took to pack depending on the number of circles in the passed array. This algorithm clearly has a larger growth rate than the randomGuessCirclePack() algorithms as it starts with similar times but grows much quicker as the input array size increases, figure 12 shows this as well if not immediately clear.

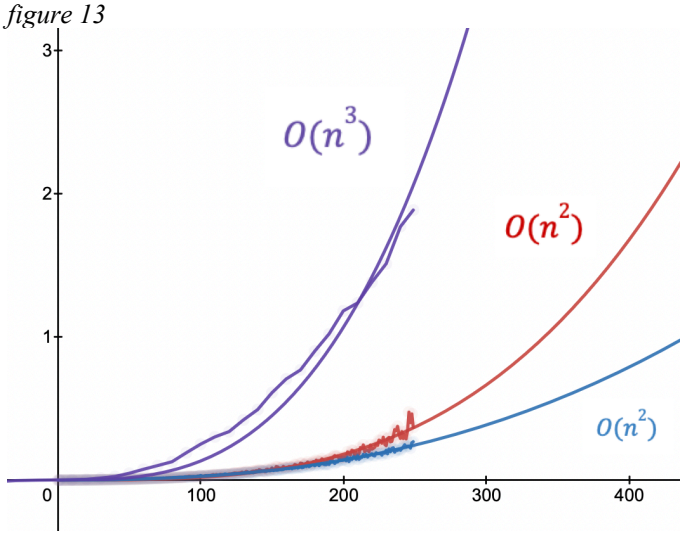
figure 12



This graph shows the same data set as before but graphed together so each functions growth rates can be easily compared.

- Red is randomGuessCirclePack() (iteration 1)
- Blue is randomGuessCirclePack() (iteration 2)
- Purple is circularPack()

figure 13



This graph shows the same graph as *figure 12* but with a exponential regression done for each scatter plot and extrapolated out. This graph makes it very clear that not only is circularPack() slower for any input larger than 100 units but it is also growing at a faster rate and will always be slower than the randomGuessCirclePack() algorithms. For this particular test the lines of best fit were as followed.

- Red/randomGuessCirclePack() (iteration 1)
$$f(x) = (4.474 * 10^{-6})x^2$$

- Blue /randomGuessCirclePack() (iteration 2)
$$f(x) = (3.3551 * 10^{-6})x^2$$

- Purple/circularPack()
$$f(x) = (1.335 * 10^{-7})x^3$$

VI. State-of-the-art circle packing algorithms

The state of the art and industry used circle packing algorithms are all linearized algorithms involving creating triangles out of possible combinations of circles and picking the most optimal one. An example of a industry used circle packing function is GOpack() which is linearized algorithm capable of quickly computing optimal circle packing involving hundreds of thousands of circles. This is far beyond the capabilities of the algorithms talked about in this paper. This is the advantage of linearizing the algorithm and not using a guess and check technique like was done for randomGuessCirclePack() and circularPack(). Because when linearized and running at O(n) time the algorithm is much more useful especially for more complex data sets and objects that need to be packed with circles. With a linearization of a circle packing algorithm like GOpack it is even possible pack circles over 3d shapes using the geometry the algorithm is based on.

VII. Conclusion

Each algorithm has a very specific use case in which they preform best. Because randomGuessCirclePack() has $O(n^2)$ complexity it is an efficient solution for very large arrays particularly when there is plenty of room to fit all circles on the screen. The $O(n^3)$ complexity of the circularPack() makes it slower with very large numbers but it does a better job of packing the circles close together. It is also a strong algorithm to use if the array has much more circles than can be fit on the screen. This is because the algorithm will fit as many as it can and then stop unlike the arbitrary stopping point of randomGuessCirclePack(). Although both algorithms discussed do have possible use cases for small scale circle packing for things like image and font creation, on any large project a linearized circle packing algorithm would be best. For matLab the best option would be GOpack and for python circlify would be recommended.

[1] Gerald L. Orick, Kenneth Stephenson, Charles Collins, A linearized circle packing algorithm, Computational Geometry,Volume 64,2017,Pages 13-29, ISSN 0925-7721, https://doi.org/10.1016/j.comgeo.2017.03.002. (https://www.sciencedirect.com/science/article/pii/S0925772117300172)

[2] Kenneth Stephenson (2021). kensmath/GOPack (https://github.com/kensmath/GOPack), GitHub. Retrieved December 5, 2021.

[3] Elmotec (2021). pypi/circlify/ (https://pypi.org/project/circlify/), pypi. Retrieved December 6, 2021.