CMIMC Circle Covers

Stephen Huan, Luke Thistlethwaite, Maxwell Bai
lmoa try 2

February 11, 2021

See this GitHub repository for source code and the source for this document.

1 Introduction

The problem Circle Covers is as follows: given a list of points, find the best places to put circles of varying radii in order to cover as many points as possible. Of course, if two circles cover the same point, then that only counts as one point covered.

The overarching strategy is going to be greedy, that is, we will consider the best position to put a single circle, and repeat for all the circles. It is best to consider the circles in order of *decreasing* radii since larger circles are more important (a suboptimal positioning will lose more points compared to a smaller circle). We therefore start with the largest circle, find the single best place to put it (measured by number of points covered), remove the points it touches from consideration (since overlaps don't count) and then continue to the second largest circle, and so on. Greedy algorithms generally perform unexpectedly well but will generally not be optimal. In this case greedy is not optimal because of the overlap—if overlaps counted then greedy would be optimal.

2 Strategies

Given this overarching greedy framework, we must consider how to actually find the best center point for a particular circle. This is where the difficulty lies and the following strategies attempt to solve this problem differently, with the exception of the last strategy which is able to compute an optimal solution if the input has certain properties.

2.1 *k*-means (`code/greedy_kmeans.py`)

k-means is a clustering algorithm that attempts to group points into k clusters, where each cluster is determined by a center point. Points are assigned to the closest center point, and the goal is to place the center points in such a way that minimizes the overall sum of distances from a point to its nearest center point. For more information, see my

lecture on k-means and my lecture on the paper *Accelerating Exact k-means Algorithms with Geometric Reasoning*. Long story short, we can accelerate k-means with a data structure called the ***kd-tree***, to the point that it is reasonable for the largest input cases.

In particular, we run k-means to get M center points, where M is the number of circles, and find the best center for the circle in each cluster by trying each point in the cluster. We assign bigger radii to bigger clusters, where the "size" of a cluster is its sum of the distances from each point in the cluster to the center point.

Since our candidates are the points in each cluster, it turns out we don't have sufficient resolution, i.e. the input points have integer values while the optimal center point may be in between grid lines. While we could use the following strategies on each cluster locally, it would introduce a lot of complexity for what is in my opinion not much value.

2.2 Grid-based (`code/greedy_grid.py`)

In the next strategy, our center candidates for a particular circle is just going to be points on an evenly spaced grid. Because the largest difference in x and largest difference in y is not too large (both at most 2,000, e.g. every point is in a 2000x2000 box), this is reasonable if our "resolution" is not too fine. If the width is W and height is H, and our resolution, i.e. the spacing of the horizontal and lines is d, then we will have $\frac{W}{d}$ vertical lines and $\frac{H}{d}$ horizontal lines, so $\frac{WH}{d^2}$ points to consider. Thus, finer and finer resolutions will usually give more circles (since the center point position is more accurate) but will pay a heavy penalty in terms of runtime.

To efficiently test each candidate center point we need to determine how many points it covers. We could either naively loop over the points (which would be slow) or use some sort of "bucketing" strategy (where we bucket each point into its corresponding box and simply sum over the boxes we include) but this is prone to error and not flexible. The simplest (but still efficient) strategy is to use a kd-tree to do a generalized nearest neighbor query, e.g. find the closest point to the center point repeatedly until we run out of points within a radius of r, in which case we stop. For more information, see the CMU slides on kd-trees and searching with kd-trees.

2.2.1 Iterative Refinement (`code/greedy_update.py`)

Because of the high cost of increasing resolution, we must consider how to efficiently improve resolution. Intuitively, our grid strategy is very wasteful since if there are large empty patches without many points, we still consider just as many points in those areas as places where the points are dense and therefore imperative that the positioning is as accurate as possible.

Instead of doing a single pass, we first do a coarse pass and successively refine the resolution with successive passes. To avoid the blowup in time with increasing resolution, we also reduce the search area. We stop searching when "convergence" happens, i.e. when the movement of the center point between iterations is reasonably small. In essence, we are "zooming in" to a center point.

I halve d each iteration while halving the width and the height. Since $\frac{(\frac{W}{2})(\frac{H}{2})}{(\frac{d}{2})^2} = \frac{WH}{d^2}$, each iteration takes the same amount of time so the overall time will be $\frac{WH}{d^2}$ times the number of iterations, $O(\log \varepsilon)$ where ε is the distance cutoff (since we halve d each iteration, there will be a point where d is so small that the movement cannot be greater than ε, in which case we stop the loop).

If we instead reduced d by a factor of $\sqrt{2}$, then each iteration takes half the time the previous iteration, and because $x + \frac{1}{2}x + \frac{1}{4}x + \ldots$ converges to $2x$, it would take at most twice as long as one pass no matter the number of iterations (so we could go to a theoretically infinite resolution).

I haven't tried that convergence schedule, but in practice higher and higher resolutions become less important in successive passes (since the change is so miniscule), and what is more important is reasonably higher resolution in the first few passes (to find the highest density regions).

This strategy is what I used for all the testcases except for testcase 1, which will be discussed at the end (this strategy can also compute the optimal solution to testcase 1, however it can't show it's optimal).

2.3 Circle Pairs (`code/greedy_finite.py`)

Our grid strategy is intrinsically flawed because it tries to consider infinite points— since it can't, it just considers points as close as possible. There are, however, a finite number of possible candidates. Suppose we have a circle encompassing some points. Clearly, we can move the circle so that it touches some point without losing points. From here, the circle is still unconstrained so we can still move it around until it hits a second point. Once it touches two points, it's essentially fixed. Thus, there are a finite number of possible center candidates, namely, each circle is given by a pair of points on the circle.

Suppose we have two points (x_0, y_0) and (x_1, y_1) on the circle and we wish to find the location of the center of the circle (x_c, y_c) with radius r. By the definition of a circle, we have the equations:

$$(x_0 - x_c)^2 + (y_0 - y_c)^2 = r^2$$
$$(x_1 - x_c)^2 + (y_1 - y_c)^2 = r^2$$

Expanding both equations,

$$[x_0^2 - 2x_0x_c + x_c^2] + [y_0^2 - 2y_0y_c + y_c^2] = r^2$$
$$[x_1^2 - 2x_1x_c + x_c^2] + [y_1^2 - 2y_1y_c + y_c^2] = r^2$$

Subtracting the bottom equation from the top equation,

$$x_0^2 - x_1^2 + 2x_c(x_1 - x_0) + y_0^2 - y_1^2 + 2y_c(y_1 - y_0) = 0$$

This is a complicated line. Putting in slope-intercept form,

$$y_c = \frac{x_0 - x_1}{y_1 - y_0}x_c + \frac{x_0^2 - x_1^2 + y_0^2 - y_1^2}{2(y_0 - y_1)}$$

Calling the slope m and the intercept b,

$$y_c = mx_c + b$$

We now plug this form into the first equation:

$$(x_0 - x_c)^2 + (y_0 - (mx_c + b))^2 = r^2$$
$$[x_0^2 - 2x_0x_c + x_c^2] + [(y_0 - b)^2 - 2(y_0 - b)mx_c + (mx_c)^2] = r^2$$
$$(m^2 + 1)x_c^2 - 2(m(y_0 - b) + x_0)x_c + x_0^2 + (y_0 - b)^2 - r^2 = 0$$

We end up with a quadratic which can be solved for x_c and y_c derived from the line. We therefore have *two* center candidates per pair of circles. An unfortunate edge case is if $y_1 - y_0 = 0$, in which case the line is vertical and thus ill-defined. In this case x_c is a constant value and y_c can be found with a similar quadratic.

An important implementation detail is to account for numerical error. Since we are placing the circle center so that it just covers these two points, it is particularly sensitive to numerical error. We can "nudge" the center in the direction of the two points which will make it safer and will likely not lose any points. If $\vec{p}_0$ is our first point and $\vec{p}_1$ is out second point, $\vec{m} = \frac{\vec{p}_0 + \vec{p}_1}{2}$ is the midpoint, and if $\vec{c}$ is our center point, the vector pointing from the center to the midpoint is $\vec{m} - \vec{c}$. We therefore update $\vec{c}$ to $\vec{c} + \varepsilon(\vec{m} - \vec{c})$. If $\varepsilon = 0$ then $\vec{c}$ stays in its original position and if $\varepsilon = 1$ then $\vec{c}$ is moved to the midpoint of the two points. I set $\varepsilon = 10^{-6}$ which is small enough to be unlikely that points are lost while large enough to guarantee the two points are within r of the center point.

Unfortunately this strategy cannot be used for most testcases since the number of circle candidates is proportional to the number of *pairs* of points, which is $O(n^2)$. We could get around this by picking a subset of pairs or by using the optimal solver in the grid refinement strategy when the number of points in the bounds of the grid is small enough, but in practice the optimal circle placement doesn't necessarily improve points covered—remember that our overall strategy is greedy, so while the optimal circle placement will be optimal for a single circle, there's no guarantee that it's optimal after the placement of *all* circles.

2.4 Integer Linear Programming (`code/setcover.py`)

The finite number of circle candidates is useful for a completely different approach than greedy, however. The first testcase is the only testcase where the radii are all the same. Can we take advantage of this? If we consider each circle candidate as generating a set of points the corresponding circle covers (with fixed radius) then the question is "given a list of sets, pick M such that the size of the union of the sets is maximized". This is the maximum set cover problem, and is discussed here, on page 35.

In particular, it can be shown that a greedy approach is at worst $1 - \frac{1}{e} \approx 0.63$ times the size of the optimal solution. We can, however, compute the optimal strategy directly with *integer linear programming*, or an optimization problem where the objective is linear and the constraints are linear, and the variables can only take integer values.

If x_i is whether the ith circle is picked and y_i is whether the ith point is covered, then we can phrase the problem in terms of the following program:

$$\text{maximize } \sum y_i \qquad \text{(number of points)} \qquad \text{(set cover program)}$$

subject to:

$$\sum x_i \leq M \qquad \text{(can only pick } M \text{ circles)}$$

$$\text{for each } y_i\text{: } \sum_{j|y_i \in x_j} x_j \geq y_i \qquad \text{(if } y_i \text{ is covered, at least one set covers it)}$$

$$x_i, y_j \in \{0, 1\} \qquad (x_i \text{ and } y_j \text{ are binary variables})$$

We can solve this linear program with the python-mip mixed-integer linear program solver to arrive at the provably optimal solution of 61 points for the first testcase.

The problem with the linear program (other than the $O(n^2)$ number of candidate sets) is the fact that the radii are different, so we will have different possible programs given by the number of ways to assign radii to center candidates, which grows extremely large extremely quickly.

3 Conclusion

Greedy is very strong, and is therefore very hard to beat. Greedy is however, nontrivial to optimize—there are still decisions to be made.