CS 170 Project Reflection

Team

1 Input generation

We designed our solutions with Geogebra. We generated our inputs by first packing non-overlapping circles of penalty radius as tightly as possible, and then placing points in these circles. We designed our solutions this way because each circle had to cover exactly the points we placed, as otherwise one would need more circles, which would incur a greater penalty. It was easy to find the set of points each circle had to cover, but because there were many options for each tower center to cover its given set of points, as because the optimal solution was a very tight packing, we thought this would make our problem more difficult to solve. We did the same thing for small, medium, and large. To save time on medium and large, we designed a small section, and repeated the structure across the map.

Looking back, many teams chose to use greedy solutions, which could find the optimal solution for our input after shifting circles out of range of each other. Another possible design that could do better is one for which a greedy algorithm outputs fewer circles than the optimal number in the lowest penalty solution.

2 Approaches

2.1 Main approaches

Our main approach consisted of two parts. In part one, we used a greedy solution to try to find the minimum number of towers needed to cover all cities without consideration for the increase in penalty of overlapping towers. In part two, we performed annealing on the greedy solution.

The main inspiration of our greedy algorithm was from the 1-D analogy of the problem. In that problem, a greedy solution of putting an interval with its left endpoint on the left-most point yielded the optimal solution. We noticed that the lattice points covered by a circle of radius 3 covers form an almost 5-by-5 square, with one protrusion on each side. Using this intuition, our greedy solution looked at the topmost city (ties broken from left to right), and tries all tower locations that will cover it in a (generally) greedy fashion.

We made the greedy solution non-deterministic because the protrusion makes it such that a strict greedy algorithm may fail to find the minimum cover. Furthermore, randomness in greedy means that the starting points for annealing can be more varied. To create randomness, we pick the tower according to a distribution weighted by how many cities it covers. We have a tolerance parameter. When it is at 0, it chooses according to a strict greedy; as it increases, it allows for more suboptimal covers at

that particular step. Our algorithm fluctuates this tolerance parameter in some way after placing down each circle. We then ran into the problem of the algorithm giving us solutions with substantially more circles than optimal. To combat this, we utilized memoization. We keep track of the subproblem as a boolean array of length N, where a FALSE value corresponds to an uncovered city. We also keep track of the optimal number of towers placed to reach this subproblem. This way, if we notice that we arrived at a subproblem that we have seen before and our current tower count is higher than what we have stored, then there is no need to proceed. With this, we can capture cases where the minimum number of towers is achieved by some not-necessarily greedy tower at the beginning. After running 10000 iterations of greedy and taking the best one, we almost always obtain a set of solutions with the minimum number of towers. Note that to further increase randomness in the greedy process, at each step, we randomly chose the top-most or the bottom-most city as our target.

Next comes the annealing stage. At each step, we randomly picked a tower, a shift in the x coordinate, and a shift in the y coordinate. If the shift caused some cities to become uncovered, we generated new random x and y coordinate shifts until the shift was valid. If the valid shift reduced total penalty, we always made the shift. If not, we made the shift with probability $e^{-\delta T}$, where δ is the amount our penalty would increase by if we performed the shift, and T is a "temperature" variable that decreases exponentially by a factor of 0.999 after each shift we perform. Our annealing algorithm would move cities around until it found a local minimum. We ran annealing on the same greedy solution many times to try to find a better local minimum, or the global minimum.

We chose this 2-part strategy because we thought that having fewer towers that covered the same number of points would generally lead to lower total penalty, as each new tower incurred a penalty, and more towers would generally lead to more overlap. Then, annealing would take that as a starting point and reduce penalty as much as possible.

For inputs that our algorithm performed badly on, we ran more iterations of annealing to have a higher chance of reaching a lower local minimum penalty.

2.2 Other approaches

Our initial greedy algorithm was different, as it simply placed the tower that would cover the greatest number of cities. This performed badly on many inputs, as it would often leave a few cities far apart that would take many towers to cover; it would not give the minimum number of towers needed.

To improve on this strategy, we tried greedy with n tower lookahead, where it would place the tower that maximized the sum of number of cities covered by the next n towers combined. This performed better than our initial greedy, but took too much computation time, so realistically, we could only do one tower lookahead.

We solved large 089 and small 127 by hand, as these inputs were highly structured with one optimal solution.

3 Computational resources

We did most of our computation on three laptops and one desktop. We ran small and medium inputs on local machines, where the number of problems we allocated to each machine was proportional to its CPU count. For large inputs, we additionally used several free-tier AWS EC2 virtual machines over three accounts. For performance, we utilized NumPy arrays and parallelism provided by `solve_all.py`.