CS 6301.008.18S Lecture—March 22, 2017

Main topics are #approximation_algorithms and #k-center_clustering.

k-Center Clustering

- This semester, we have looked at a couple *optimization problems*. These are problems for which there are several feasible solutions. Each solution has a value, and your goal is to compute the solution of maximum or minimum value.
    - In linear programming, all points within the feasible polytope were feasible. We wanted to find the point whose dot product with the objective vector was maximized.
    - In robot motion planning, any path through the free space is feasible, but for one of the lectures we wanted the shortest path through the free space.
    - In Fréchet distance, any pair of reparameterizations describing the person and dog's walk could be considered feasible, but we wanted the pair that minimized the necessary leash length.
- In each instance, we found a polynomial time algorithm that found the best or *optimal* solution to the problem.
- Today, I want to discuss another geometric optimization problem to show that things aren't always so nice.
- The problem is called *k-center clustering*.
- We're given a set of points P = {p_1, p_2, …, p_n} in the plane and an integer k.
- The goal is to find a collection of k circles that enclose every point of P where the largest circle radius is as small as possible.
- In other words, we want to find k center points C = {c_1, c_2, …, c_k}.
- We want to minimize the value or cost of the chosen centers which is denoted cost(C) = max_i min_j ||p_i c_j||. Try to make sure every point is close to a center.
- The points closest to center c_j form its *cluster*. The *radius* of the cluster is the largest center to point distance.
- The fastest known algorithm for this problem runs in n^O(sqrt(k)) time [Hwang, Lee, Chan '93]. This algorithm does *not* run in polynomial time when k is part of the input.
- And it's not really surprising that there isn't a polynomial time algorithm. k-center clustering is an example of an NP-hard problem.
- NP-hard means a polynomial time algorithm for k-center clustering can be used to get a polynomial time algorithm for any problem in the complexity class NP.
- And NP contains some very hard problems.
    - Is there any assignment of variables to satisfy a given boolean formula?
    - Can I implement this function using only 10 registers?
    - Is there a group of more than 1000 mutual friends on Facebook?

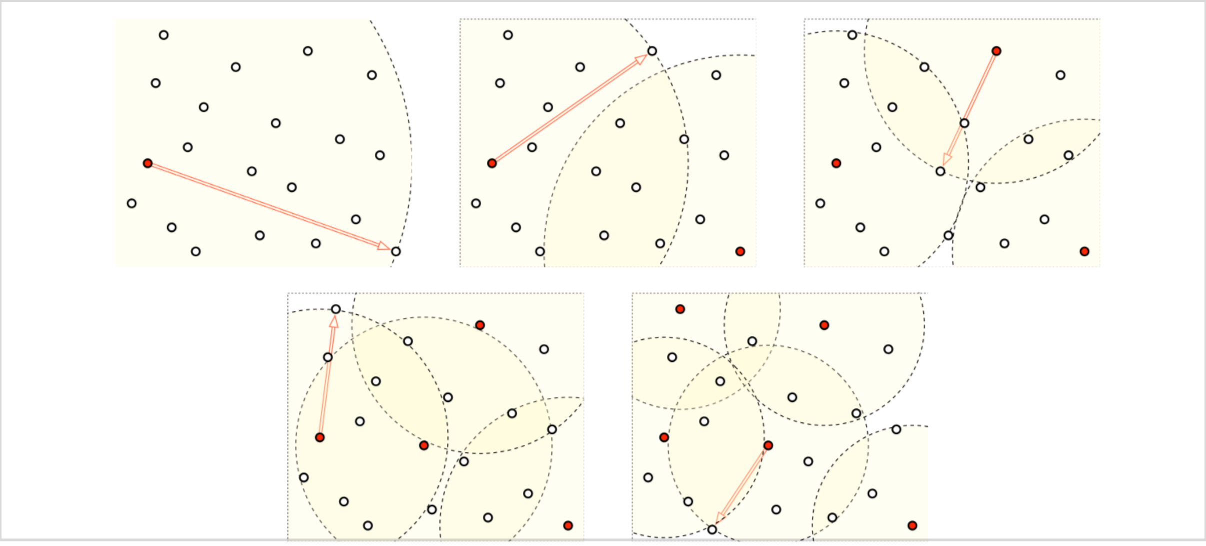
- So it seems very unlikely that we can solve k-center clustering in polynomial time.
- But what if we don't require the best solution, just one that is pretty good. Can we design an algorithm and prove that it is "good enough"? What does that mean?

Approximation Algorithms

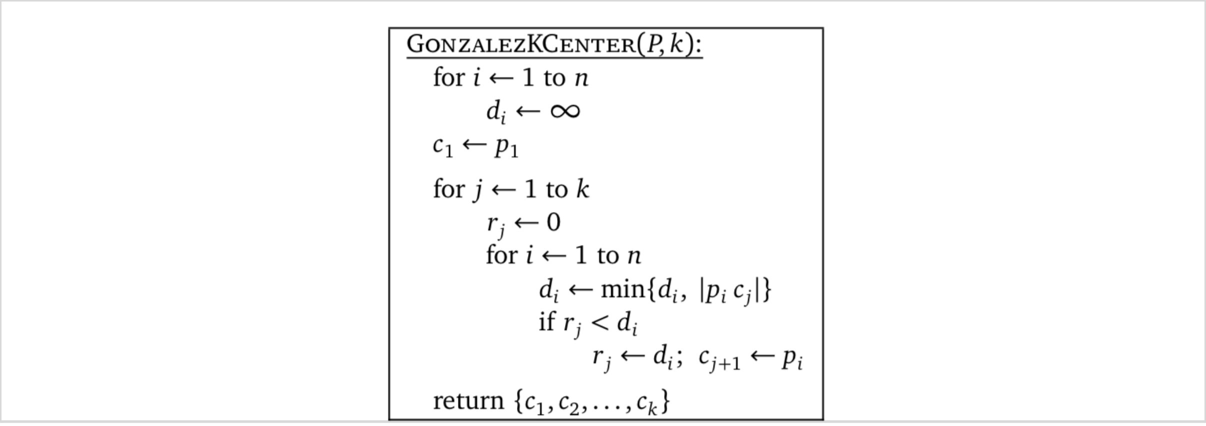
- Let's step back from k-center and consider an *arbitrary* optimization problem (of which k-center is just one example).
- For an input X, let OPT(X) denote the value of the optimal solution given X. So for k-center clustering, OPT(X) would be the smallest possible maximum radius.
- Now, suppose we have some algorithm A for the problem. A may not find the optimal solution, but we should measure how good a job A does. Let A(X) be the value of the solution computed by A.
- A is an alpha(n)-approximation algorithm if and only if
    - OPT(X) / A(X) ≤ alpha(n) and
    - A(X) / OPT(X) ≤ alpha(n)
    - for all inputs X of size n.
- alpha(n) is called the *approximation factor* of the algorithm. Sometimes it will be a constant like 2 or 5. Sometimes it will be an increasing function of n like ln n.
- Depending on if the problem is a minimization problem or maximization problem, exactly one of those inequalities is interesting.
- For minimization problems like k-center clustering, the second inequality tells us the algorithm A always finds a solution of cost *at most* alpha(n) times the optimal.
- For maximization, the first inequality tells us our algorithm finds a solution of size *at least* 1 / alpha(n) times optimal.
- In either case, a 1-approximation algorithm always finds an optimal solution.
- Our goal is to design approximation algorithms that have as small an approximation ratio alpha(n) as possible.

Greedy 2-approximation

- So back to k-center clustering.
- There's a natural heuristic for this problem first analyzed by Gonzales in 1985.
- Pick an arbitrary point of P as the first center point.
- Then, iteratively pick a center point from P that is as far away from the other points you picked as possible. After all, it's the one point responsible for the current value of your solution!



- Here’s a relatively efficient way to run the algorithm. d_i is the distance from p_i to the closest center chosen so far. r_j is the max radius after adding j centers.



```
GonzalezKCenter($P$, $k$):
  for $i \leftarrow 1$ to $n$
      $d_i \leftarrow \infty$
  $c_1 \leftarrow p_1$
  for $j \leftarrow 1$ to $k$
      $r_j \leftarrow 0$
      for $i \leftarrow 1$ to $n$
          $d_i \leftarrow \min\{d_i, |p_i c_j|\}$
          if $r_j < d_i$
              $r_j \leftarrow d_i$; $c_{j+1} \leftarrow p_i$
  return $\{c_1, c_2, \ldots, c_k\}$
```

- The algorithm runs in O(nk) time.
- So the heuristic seems reasonable, but can we guarantee it actually finds a good solution?
- Theorem: GonzalezKCenter computes a 2-approximation to the optimal k-center clustering.
- Proof: Let OPT denote the cost of the optimal k-center clustering.
    - Each center point c_j is distance at least r_{j-1} from any center c_i with i < j. And since we’re picking farthest centers each time, r_i ≥ r_j for any i < j.
    - Therefore, ||c_i c_j|| ≥ r_{j-1} ≥ r_k for all i < j ≤ k+1
    - By the pigeonhole principle, at least one cluster in the optimal clustering contains two of c_1, c_2, …, c_{k+1}. Let c_i and c_j be these centers.
    - By triangle inequality, ||c_i c_j|| ≤ 2OPT. In particular, 2OPT ≥ r_k.

Approximation Schemes

- So can we do better? It turns out it is NP-hard to get a better than 1.8 approximation as long as k is part of the input.
- But even if we assume k is a small constant, say 16, the optimal n^O(sqrt(k))-time algorithm still has a pretty bad dependency on n.

- We can reduce that running time by settling for an *approximation scheme*.
- An approximation scheme takes the input for the problem and a value eps > 0, and then computes a (1 + eps)-approximation for the problem.
- Often the running time increases quickly as eps gets very small, but for any constant eps, the running time may be reasonable.
- Here's an approximation scheme for k-center clustering:
    1. Compute a 2-approximation using GonzalezKCenter(P, k). Let r be the cost of the clustering.
    2. Consider a grid of squares in the plane where each square has side length delta = eps r / (2 sqrt(2)). Let Q be a subset of P with one point per non-empty grid cell.
    3. Compute an *optimal* set of k centers for Q. Return these k centers as the centers for P.
- The first phase takes O(nk) time. Note that OPT ≤ r ≤ 2OPT.
- For the second phase, we associate with each point p = (x, y) in P a grid cell denoted with the pair (floor(x / delta), floor(y / delta)).
- Then, we sort the points by their grid cell in O(n log n) time so that pairs in the same cell end up adjacent in order. Afterward, we can remove points sharing a cell in linear time by sweeping through and keeping only the first point for each cell.
- Or we could use a hash table to group up points and discard those sharing a cell in O(n) time total.
- So why is it useful just to consider Q? Well, P can be covered by k balls of radius OPT. You can only fit O(OPT^2 / delta^2) = O(1 / eps^2) cells in a single ball of that size. So, |Q| = O(k / eps^2).
- Note that |Q| gets bigger as eps gets smaller and our approximation improves.
- But then we can use that optimal algorithm to find the k centers for Q in (k / eps^2)^O(sqrt(k)) time.
- So the whole algorithm takes O(kn + n log n + (k / eps^2)^O(sqrt(k))) time. If k and eps are both constants, then this algorithm runs in O(n log n) time. It's only O(n) if you use a hash table!
- But is it an approximation scheme?
- The optimal clustering of Q consists of k balls. Each has radius OPT(Q) ≤ OPT(P), since we're only covering a subset of P.
- Each point of P \ Q shares a grid cell with a member of Q, meaning it is distance at most delta sqrt(2) away from a point in Q.
- So, if we increase the radius of each ball by delta sqrt(2), we'll cover every point in P.
- The radius of the larger balls is at most OPT(Q) + delta sqrt(2) ≤ OPT(P) + eps r / 2 ≤ OPT(P) + eps OPT(P) = (1 + eps) OPT(P).

Final Points

- There was nothing that special about using the plane.
- The approximation scheme works for points in R^d for any constant d. You'll just get a running time of O(kn + n log n + (k / eps^**d**)^O(sqrt(k))) instead.
- The 2-approximation algorithm actually works with any *metric space*. A metric space is any collection of objects that comes with non-negative distances between the objects that obey the triangle inequality. So shortest paths in a weighted, undirected graph for example.