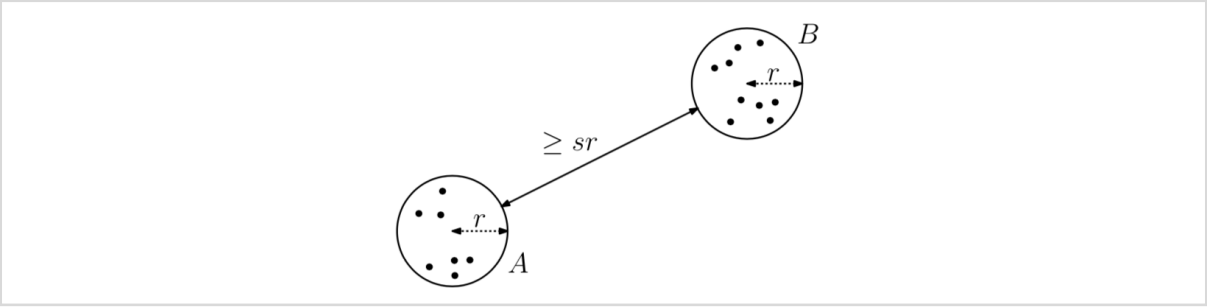
CS 6301.008.18S Lecture—March 27, 2017

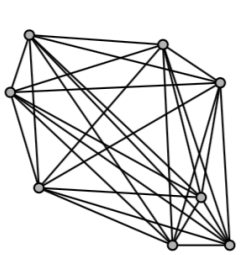
Main topics are #well_separated_pair_decompositions.

Well Separated Pair Decomposition (WSPD)

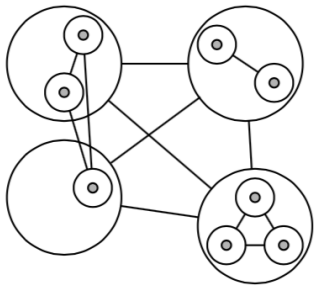
- On Thursday, we discussed approximation algorithms for the k-center clustering problem.
- One motivation for those algorithms is that we cannot solve k-center clustering exactly in polynomial time if k is part of the input.
- However, we also saw an approximation scheme for when k is not part of the input but we want a near-linear time algorithm for the problem.
- Today, we're going to discuss another tool where near-linear time approximation algorithms are just one application.
- To motivate it, consider the following setting.
- In the n-body problem, you're given a large collection of bodies, say in space, and you want to simulate gravitational and other interactions between them.
- Well, just knowing how each body's mass is affecting the other bodies appears to require Omega(n^2) distance calculations. If you're trying to do a simulation, performing the calculations can be very expensive.
- But consider the following situation: you have two sets of bodies, each in a different galaxy. In each galaxy, the bodies are relatively close together, but the two galaxies are very far apart as galaxies tend to be. If we want to know how bodies in one galaxy are affecting the other, we may as well treat each galaxy as one big body.
- Let's make this more precise for today's lecture. We're given a set of n points P in R^d.
- Let s > 0 be a value we call the *separation factor*.
- We say two disjoint subsets A and B of P are *s-well separated* if A and B can be enclosed in Euclidean balls of radius r such that the closest distance between the balls is ≥ sr.



- Note by this definition, for any a ≠ b in P, {a} and {b} are s-well separated for *any* s > 0.
- So here's the surprising thing. There are (n choose 2) different pairs of points in P.
- However, it is possible to find a small number of pairs of subsets (think little pairs of galaxies) from P that such that pairs of subsets are all s-well separated and every pair is represented in some pair of subsets.



28 pairs

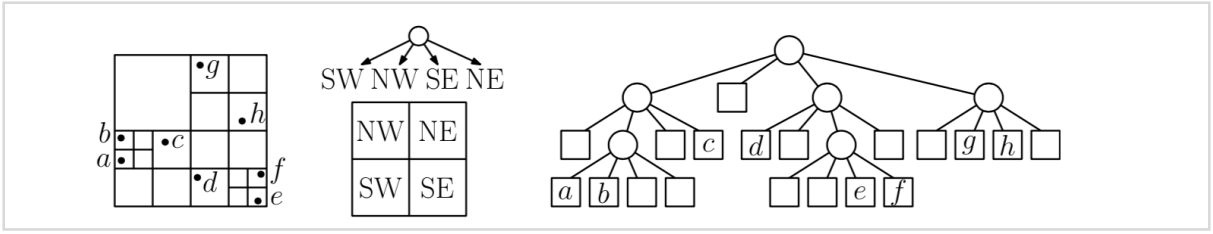


12 well-separated pairs

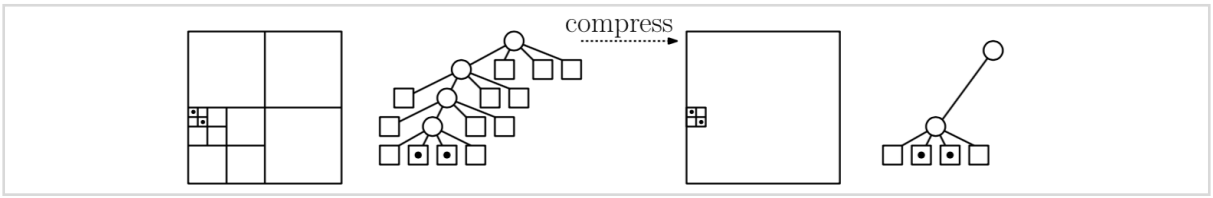
- Let's break down that claim piece-by-piece.
- Given disjoint sets A and B, let A otimes B be the set of distinct unordered pairs from A and B, so A otimes B = {{a, b} | a in A, b in B, a ≠ b}.
- An *s-well separated pair decomposition* (s-WSPD) is a collection of pairs of subsets of P denoted ({A_1, B_1}, {A_2, B_2}, …, {A_m, B_m}} such that
  1. A_i, B_i subset P for all 1 ≤_i ≤_m
  2. A_i intersect B_i = emptyset for all 1 ≤_i ≤_m
  3. union_{i=1}^m A_i otimes B_i = P otimes P
  4. A_i and B_i are s-well separated for all 1 ≤_i ≤_m
- So, we have pairs of disjoint subsets, and every pair of points from P appears in at least one subset. Also, each pair of subsets is s-well separated.
- In other words, the distance between any pair of points can be approximated by the distance between the subsets containing that pair.
- We can build an s-WSPD by building O(n^2) pairs of singletons, one per pair of points from P.
- But when s and the dimension d are constants, you can actually get away with only O(n) pairs. The constant in that big-oh is proportional to s^d.

Quadtrees

- In order to find those O(n) pairs, we'll have to work with a new kind of geometric data structure called a quadtree (called an octree in 3D).
- The quadtree is a hierarchical decomposition of space. Each node is associated with a hypercube region of space called a cell.
- For simplicity, let's scale everything so P lies in the unit hypercube [0, 1]^d. This hypercube corresponds to the root of the quadtree.
- Now we iteratively add nodes to the tree as follows.
- Let's say we haven't processed node u or its cell.
- If the cell contains one or fewer points, we're done processing u.
- Otherwise, we subdivide the cell into 2^d hypercubes of side length half that of the original cell and make one node per new hypercube as a child of u.



- Notice how within each level of the tree, the cells are all the same size and pairwise disjoint.
- Also, each cell has a side length of exactly 1 / 2^i for non-negative integer i.
- This is not the same as a kd-tree! First, each cell in the quadtree has 2^d children instead of 2. Also, *space* is partitioned evenly, not the number of points.
- In practice, what I just described tends to yield relatively small trees, but in the worst case it may actually have more than O(n) nodes!
- The problem is when there is a cluster of points very close together relative to their surroundings, the quadtree may contain an arbitrarily long *trivial path* leading to the cluster in which only one child of each node on the path has a cell containing any points.

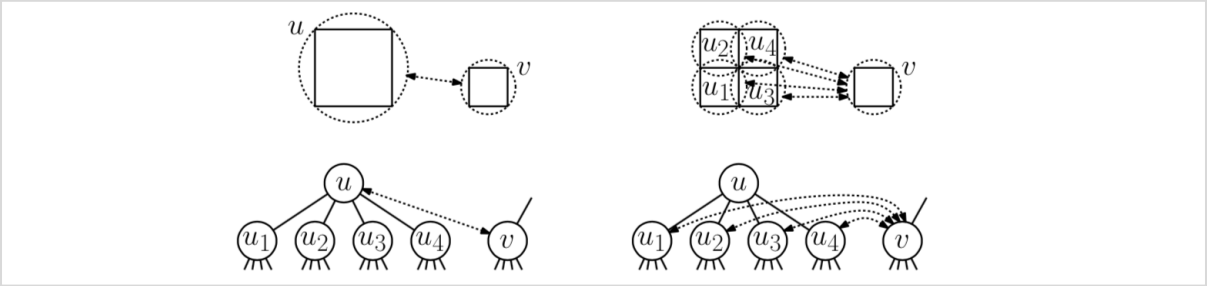


- To deal with this issue, we have to use *path compression* where we replace the whole trivial path with a single node associated with the smallest hypercube holding the cluster.
- Compressing every trivial path results in the *compressed quadtree*.
- In a compressed quadtree, every internal node separates a pair of points. Therefore, there are O(n) nodes in the tree.
- I won't give the algorithm here, but we can construct the compressed quadtree for any set P in O(n log n) time.
- Some final bits of notation before we build a WSPD.
  - For a node u, let P_u be the set of points in its cell.
  - Let level(u) = -log_2 x where x is the side length of the cell. In the uncompressed quadtree, the level is the distance from the root.
  - However, if u contains one point, we'll say level(u) = infinity.

Constructing the WSPD

- So, we're given n points in P in R^d and s > 0. We'll build an s-WSPD of size O(s^d n) in O(n log n + s^d n). Here, I'm assuming d is a constant.
- First, build a compressed quadtree in O(n log n) time.
- Each pair {A_i, B_i} will equal {P_u, P_v} for a pair of nodes u and v in the quadtree, so we'll concentrate on outputting pairs of nodes.

- The algorithm recursively takes pairs of nodes u and v and tries to find well separated pairs containing every pair from P_u oplus P_v.
- If either cell is empty, we ignore the pair. If they are both leaves, we output {u, v}.
- Otherwise, assume level(u) ≤ level(v).
- Take the two smallest Euclidean balls of equal radius surrounding u and v's cells. If s-well separated then output {u, v}. Otherwise, recursively apply this procedure for each of the 2^d pairs {u_i, v} where u_i is a child of u.

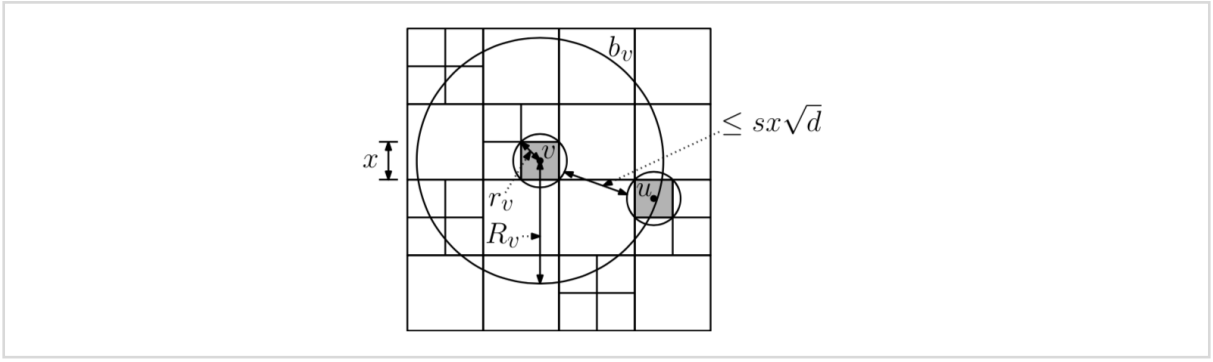


- Here's some pseudocode. ws-pair(u, v, s) makes well separated pairs from P_u and P_v with separation factor s. The initial call is ws-pair(u_0, u_0, s) where u_0 is the root of the quadtree.
- ws-pairs(u, v, s):
  - if u and v are leaves and u = v, return emptyset
  - if P_u or P_v is empty, return emptyset
  - else if u and v's cells are s-well separated, return {{u, v}}
  - else
    - if level(u) > level(v), swap u and v
    - let u_1, …, u_{2^d} denote children of u
    - return union_{i=1}^{2^d} ws-pairs(u_i, v, s)
- You can verify this algorithm is correct by induction.
  - In the first three cases, it verifies that P_u oplus P_v consists of zero or one s-well separated pairs.
  - Otherwise, it inductively finds well separated pairs for every P_{u_i} oplus P_v, and together these cover all pairs of P_u oplus P_v.
- Notice how each *ordered* pair from P oplus P appears exactly once. You can modify the algorithm slightly to detect and remove redundant pairs (P_u, P_v) and (P_v, P_u).

Analysis

- To make the analysis easier, let's assume today that the quadtree is not compressed but still has size O(n). Then children nodes have cells with exactly half the side length of their parent's cell.
- This assumption also implies that when calling ws-pairs(u, v, s), the sizes for u and v differ by a factor of at most 2. Remember, we always split the larger cell.

- We'll also assume s ≥ 1, since that's the interesting case anyway.
- We will count the number of calls to ws-pairs since it bounds both the running time and the number of pairs in the output.
- Say a call is *terminal* if it doesn't reach the final else clause.
- Since the quadtree and call tree have arity 2^d, we'll just count the *non-terminal* calls and multiply by 2^d (a constant) for our final count.
- We'll use a charging argument. Every non-terminal call reaches that final else clause where it splits u's cell and does not split v's cell. We will charge each non-terminal call to node v and show each node is charged at most O(s^d) times. There are O(n) nodes in the quadtree so that's O(s^d n) calls overall.
- A node v is charged to only when a call is non-terminal, meaning some u is *not* s-well separated from v.
- Let x be the side length of v's cell. The side length of u's cell is at most 2x. The smallest radius needed to enclose either cell in a ball is x sqrt(d). And since the balls are not s-well separated, their distance from each other is at most s x sqrt(d).
- In particular, the centers of the balls are at most R_v = x sqrt(d) + x sqrt(d) + s x sqrt(d) ≤ 3sx sqrt(d) apart.
- OK, so all u have cells touching a ball of radius R_v centered at the center of v's cell. The way the procedure works, no single ws-pair uses the same u and v twice. Each u in a non-terminal call has side length x or 2x. The cells are pairwise disjoint since each {p, q} appears exactly once.
- But you can prove there are at most O((R_v / x)^d) = O((3sx sqrt(d) / x)^d) = O(s^d) disjoint cells of side length x or 2x touching that ball.



- So, we charge to v O(s^d) times as promised.
- Meaning we output O(s^d n) pairs.
- And the total running time is O(n log n + s^d n).
- Thursday, we'll discuss applications of WSPDs for approximation algorithms and related concepts.