Grid Decomposition — Finding the Closest or Close Pairs

Yufei Tao

Chinese University of Hong Kong

We will apply divide and conquer to solve another fundamental problem in computational geometry: closest pair. Recall that all of our divide-and-conquer algorithms so far work by reducing a d-dimensional problem to a $(d-1)$-dimensional one. This, however, will not be our approach today. Instead, we will introduce a grid decomposition technique, which allows us to deal with the closest pair problem directly in d-dimensional space.

Grid decomposition is a very useful technique in general. To further illustrate its power, we will deploy it to solve the close pairs problem (which is very similar to, but yet different from, the closest pair problem).

Closest Pair and Close Pairs

Let P be a set of points $\mathbb{R}^d$. The objective of the closest pair problem is to output a pair of distinct points $p, q \in P$ that have the smallest distance to each other, or formally:

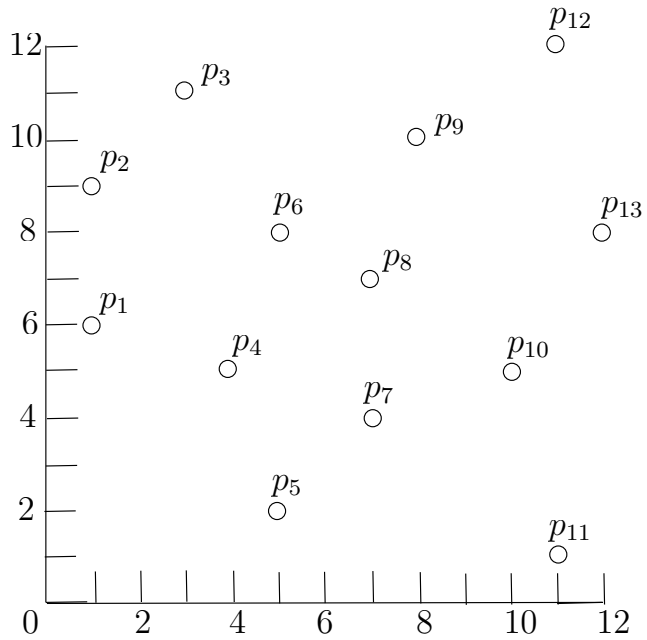
$$dist(p, q) \quad = \quad \min_{p_1, p_2 \in P,\ p_1 \neq p_2} dist(p, q).$$

where $dist(.,.)$ represents the Euclidean distance of two points.

Let P be a set of points $\mathbb{R}^d$, and r a real value. The objective of the close pairs problem is to output all pairs of distinct points $p, q \in P$ satisfying:

$$dist(p, q) \quad \leq \quad r.$$

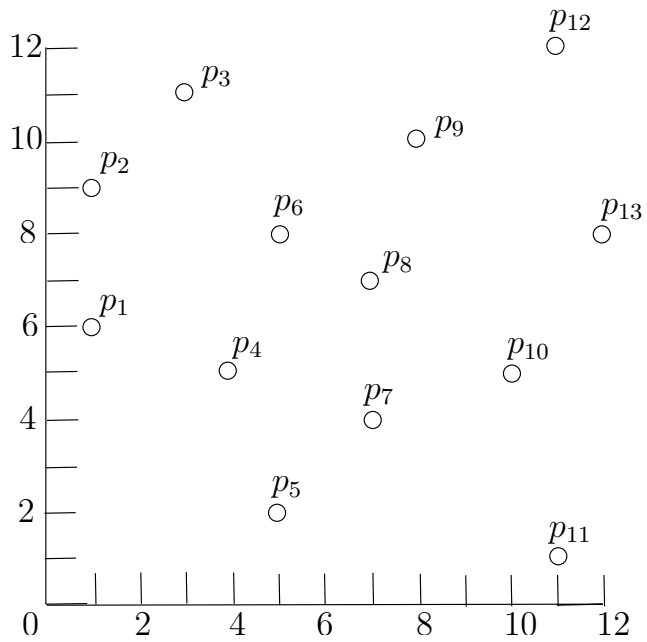
Example: Closest Pair



The answer is (p_6, p_8).

Example: Close Pairs

Assume $r = 4\sqrt{2}$.



The answer is $\{(p_1, p_4), (p_1, p_2), (p_2, p_3), (p_2, p_6), (p_2, p_4), ...\}$.

Applications

- "Find the closest pair of airplanes at 12pm on 1 Jan 2017."
- "Find the closest pair of earth quake locations in the last 5 years."
- "Find the two customers whose profiles are most similar to each other."
- ...

All the above applications have their counterparts with respect to the close-pairs problem, e.g., "find all pairs of airplanes that were within 10km at 12pm on 1 Jan 2017".

Next, we will first deal with the closest pair problem, and then attend to the close pairs problem. Note that both problems can be easily solved in $O(n^2)$ time, , where $n = |P|$. We, on the other hand, aim to solve the first problem in $O(n \log n)$ expected time, and the second in $O(n + k)$ expected time, where k is the number of pairs reported.

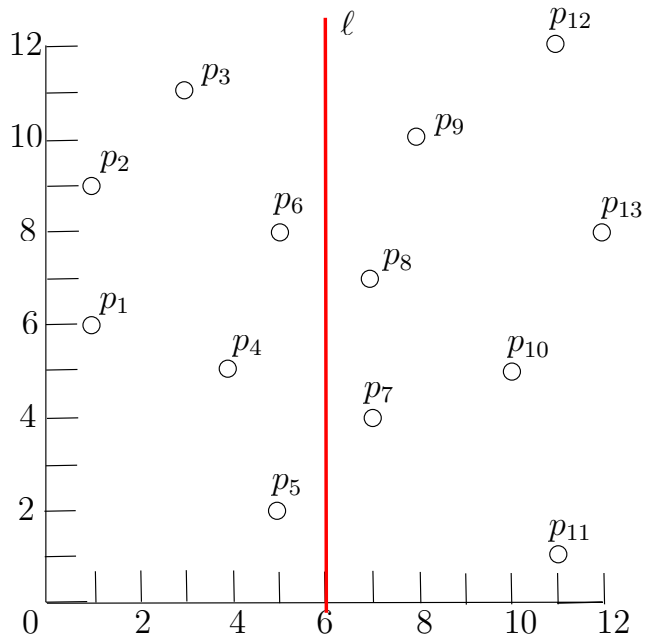
Warm Up: Closest Pair in 1D

Think: How to solve the the 1D closest pair problem in $O(n \log n)$ time?

Hint: Sorting suffices.

Closest Pair in 2D

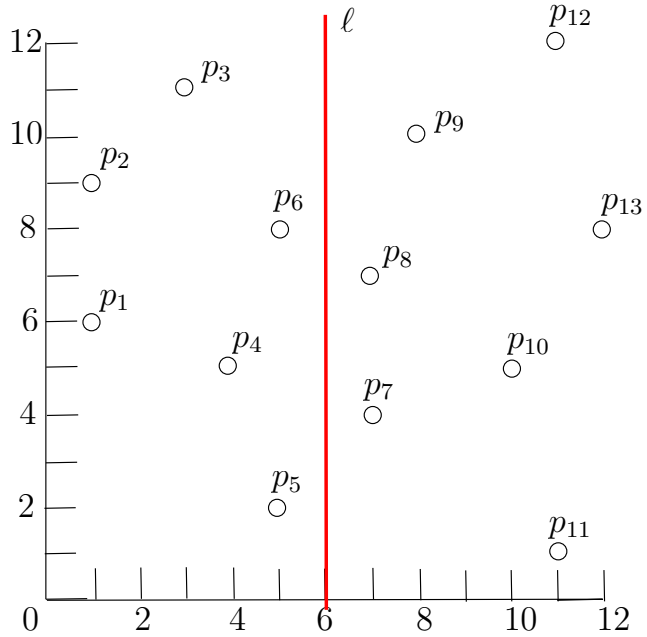
Let us now turn our attention to 2D space. Naturally, we divide P evenly using a vertical line ℓ, such that there are $n/2$ points on each side. Let P_1 (or P_2) be the set of points on the left (or right) of ℓ. We recursively find the closest pair in P_1, and then in P_2, respectively.



In the above example, the closest pair of P_1 is (p_2, p_3), and that of P_2 is (p_7, p_8).

Closest Pair in 2D

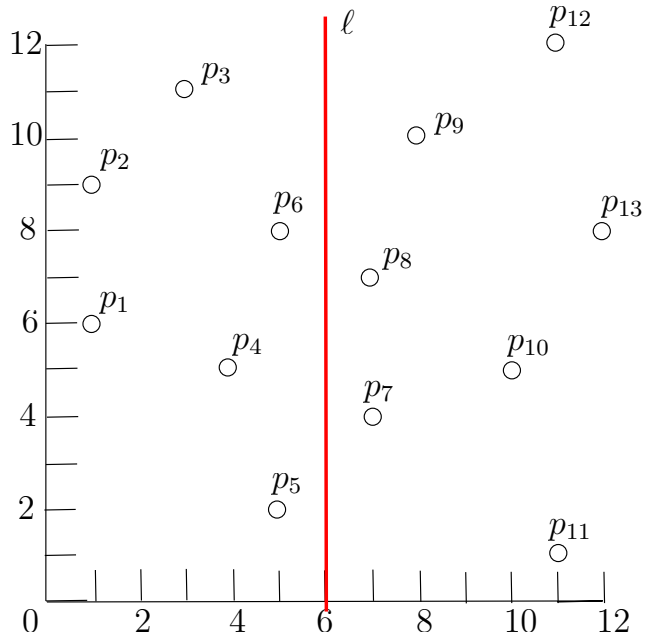
We need to "merge" the two halves to find the global closest pair. It suffices to find the closest pair (p_1, p_2) satisfying $p_1 \in P_1$ and $p_2 \in P_2$—namely, p_1, p_2 come from different sides. Call it the crossing closest pair.



In the above example, the crossing closest pair is (p_6, p_8). The global closest pair must be among the two "local" pairs (p_2, p_3), (p_7, p_8), and the crossing pair (p_6, p_8).

Closest Pair in 2D

We now explain how to find the crossing closest pair. Let r_1 be the distance of the closest pair in P_1, and r_2 be the distance of the closest pair in P_2. Define $r = \min\{r_1, r_2\}$.

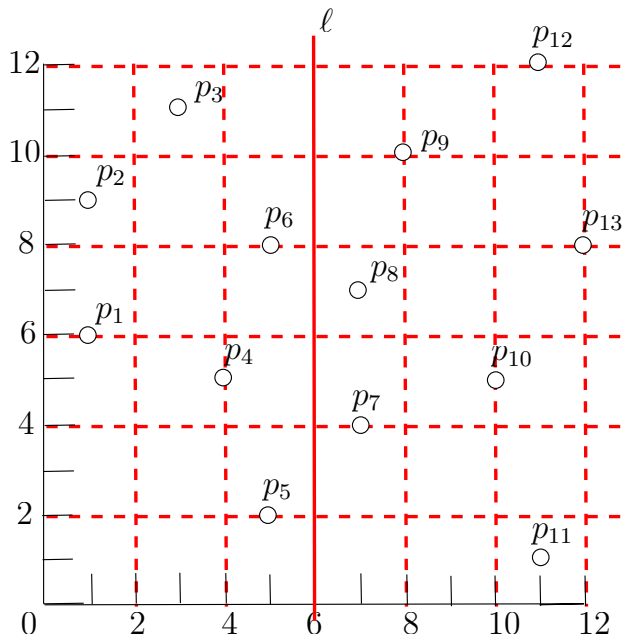


In the above example, $r_1 = \sqrt{8}$, $r_2 = 3$, and $r = \min\{r_1, r_2\} = \sqrt{8}$.

Observation: We care about the crossing closest pair only if its distance is smaller than or equal to r.

Closest Pair in 2D

Impose an arbitrary grid G onto the data space, where (i) each cell is an axis-parallel square with side length $r/\sqrt{2}$, and (ii) ℓ is a line in the grid.



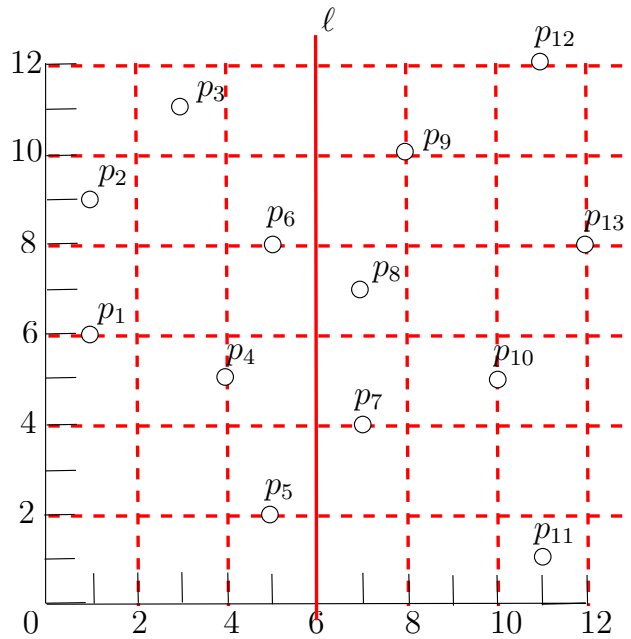
Each point p can be covered by at most 4 cells (e.g., p_9 is covered by 4 cells, but p_8 by only 1 cell).

Closest Pair in 2D

For each cell c, we denote by $c(P)$ the set of points in P that are covered by c.

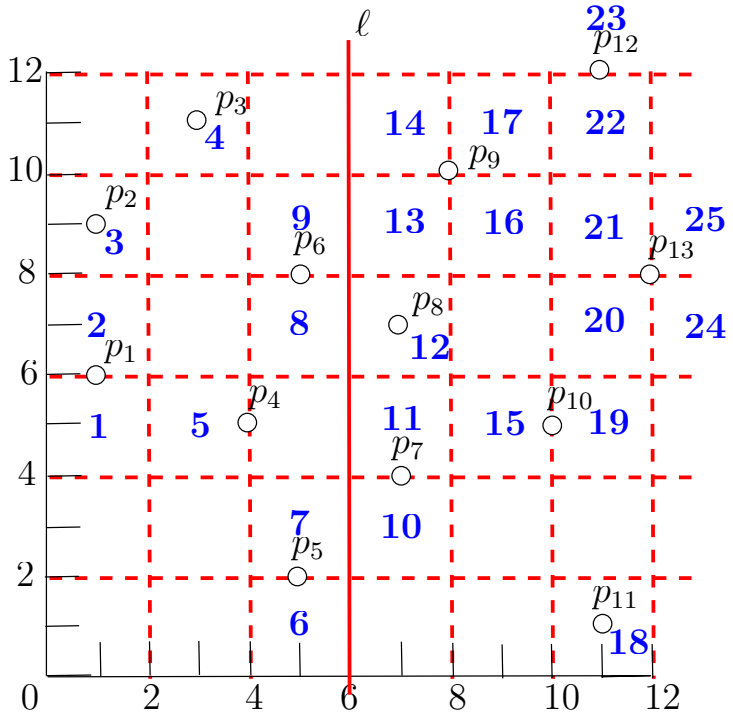
Observation: For every c, $|c(P)| \leq 2 = O(1)$!

Proof: Note that the diagonal of c has length r. If c covers more than 2 points, at least 2 points have distance less than r—contradicting the definition of r! □



Closest Pair in 2D

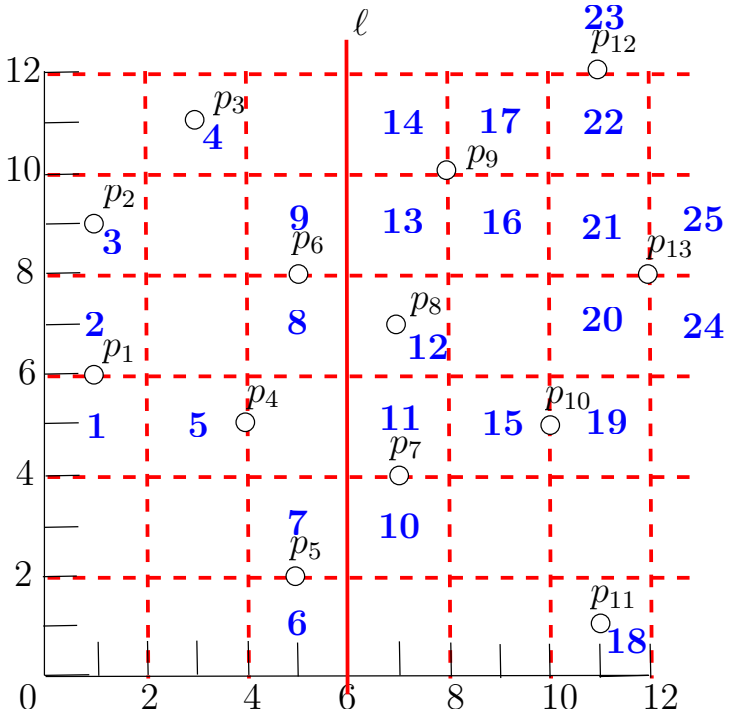
Group the points by the cells they belong. A cell is non-empty if it covers at least one point. There can be at most $4n$ non-empty cells.



In the above example, there are 25 non-empty cells.

Closest Pair in 2D

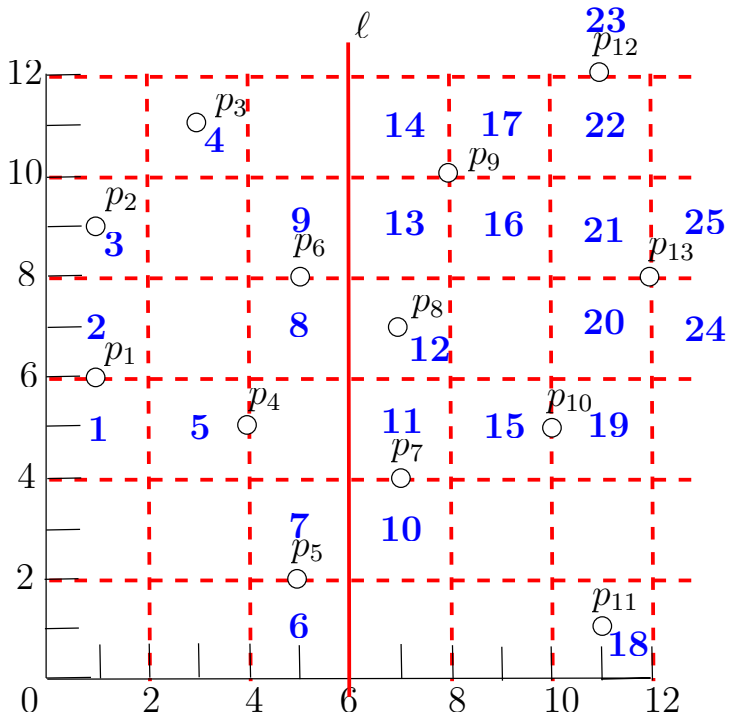
Each cell can be uniquely identified by the coordinates of its centroid—which we refer to as the id of the cell. Using hashing, by spending $O(n)$ expected time in total, we can create for each cell c, a linked list containing all the points in $c(P)$ (i.e., the set of points covered by c).



Closest Pair in 2D

Let c_1, c_2 be two non-empty cells. We say that c_1 is an *r*-neighbor of c_2 (and vice versa) if their mindist is at most r.

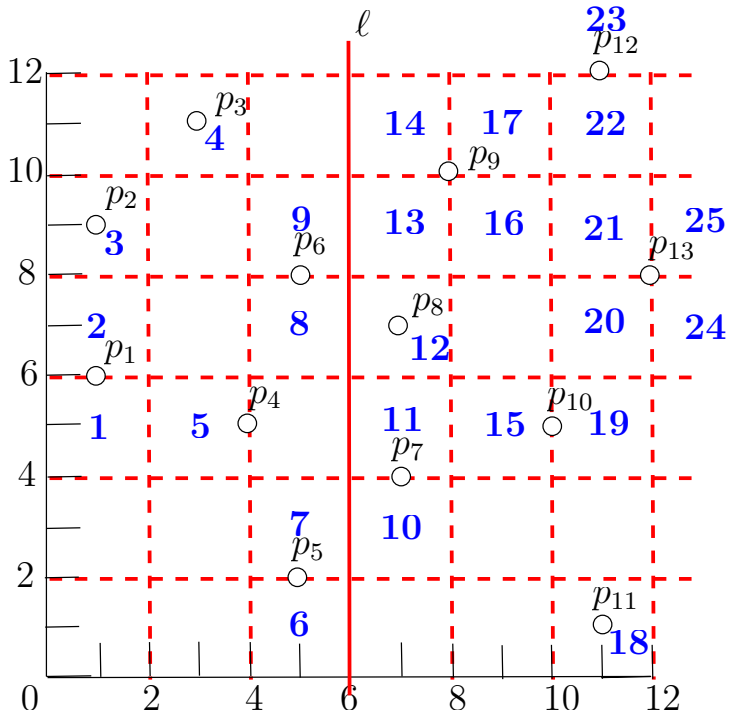
To find a crossing closest pair within distance r, it suffices to consider (the points in) non-empty cells c_1, c_2 satisfying (i) c_1 is on the left of ℓ, and c_2 is on the right, and (ii) c_1 and c_2 are r-neighbors.



For example, Cell 11 is an r-neighbor of Cell 5, while Cell 15 is not. In other words, we need to consider the cell pair (5, 11), but not (5, 15).

Cells 1, 2, 3, 18, 19, 20, 21, 22, 23, 24, and 25 can be immediately discarded (why?).

Observation: Each non-empty cell c on the left of ℓ has at most $10 = O(1)$ r-neighbor cells on the right of ℓ.



For example, for Cell 8, we need to consider 8 pairs: (8, 10), (8, 11), (8, 12), (8, 13), (8, 14), (8, 15), (8, 16), (8, 17).

Closest Pair in 2D

The above discussion motivates the following algorithm for finding a crossing closest pair within distance r:

1. **for** every non-empty cell c_1 on the left of ℓ
2. **for** every r-neighbor cell c_2 of r_1 on the right of ℓ
3. calculate the distance of each pair of points $(p_1, p_2) \in c_1(P) \times c_2(P)$
4. **return** the closest one among all the pairs inspected at Line 3, if the pair has distance at most r.

As mentioned, for each c_1, there are $O(1)$ cells c_2 that need to be considered. Since $c_1(P)$ and $c_2(P)$ each contain at most 2 points, each execution of Line 3 takes only $O(1)$ time. The overall algorithm takes $O(n)$ expected time in total.

Think: How to find the cells c_2 for each c_1 in $O(1)$ expected time? Hint: by hashing on the cell ids.

Closest Pair in 2D: Analysis

Let $f(n)$ be the expected running time of our algorithm, it follows that

$$f(n) \quad \leq \quad 2 \cdot f(n/2) + O(n)$$

while $f(n) = O(1)$ for $n \leq 2$.

The recurrence solves to $f(n) = O(n \log n)$.

Closest Pair in d-dimensional Space

Our algorithm can be extended directly to d-dimensional space.

The only difference is that, we should impose a d-dimensional grid, where each cell is a d-dimensional square with side length $r/\sqrt{d}$ (this ensures that a diagonal of every cell has length r).

Recall that, to ensure $O(n \log n)$ running time, we relied on the following fact in 2D:

> Each non-empty cell c_1 on the left of ℓ has $O(1)$ r-neighbor cells c_2 on the right of ℓ.

We will prove that this is true regardless of d (as long as d is a constant, i.e., it has nothing related to n).

Plugging this fact into the earlier analysis immediately shows that the algorithm runs in $O(n \log n)$ expected time for any constant d.

Closest Pair in d-dimensional Space

Lemma: In d-dimensional space, each non-empty cell c_1 on the left of ℓ has $O(1)$ non-empty cells c_2 on the right of ℓ satisfying $mindist(c_1, c_2) \leq r$.

Proof: Extend each boundary face of c_1 outwards by a length of r. This gives a d-dimensional square of side length $2r + r/\sqrt{d}$. The square intersects with at most

$$\left\lfloor 2 + \frac{2r + r/\sqrt{d}}{r/\sqrt{d}} \right\rfloor^d \leq \left\lfloor 2\sqrt{d} + 3 \right\rfloor^d = O(1).$$

cells. □

Remark: The calculation in the proof is very loose. For example, many of the cells in the "enlarged" square are on the left of ℓ, but are counted anyway. However, the number of cells is still $O(1)$ even after the over-counting.

The success behind our grid decomposition technique in the closest-pair problem comes from the property that, each cell in the grid has $O(1)$ r-neighbor cells.

We now proceed to tackle the close-pairs problem by essentially using the same property. Recall that our objective is to achieve $O(n+k)$ expected time, where k is the number of pairs reported.

Recall the definition of the close-pairs problem.

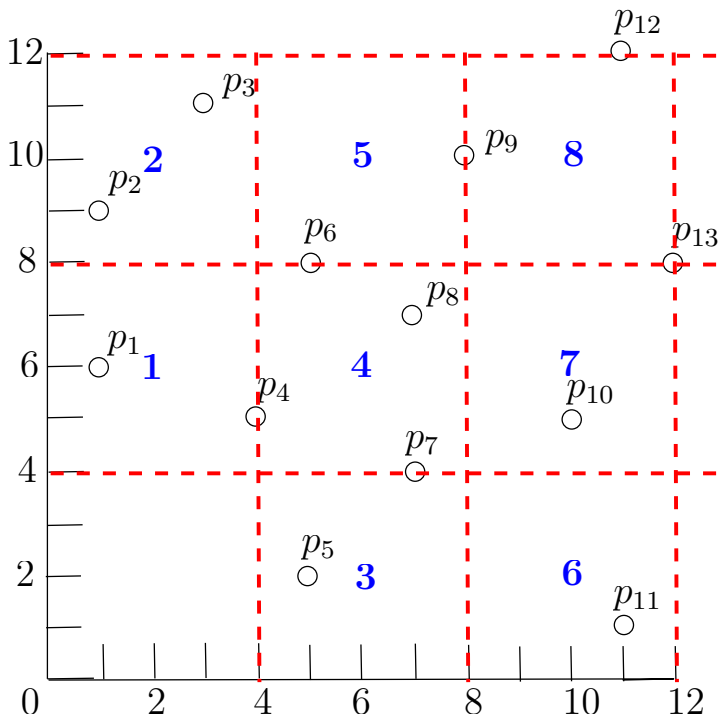
Let P be a set of distinct points $\mathbb{R}^d$, and r a real value. The objective of the close pairs problem is to output all pairs of distinct points $p, q \in P$ satisfying:

$$dist(p, q) \quad \leq \quad r.$$

We will focus on 2D space, because the algorithm can be directly extended to arbitrary dimensionalities.

Close Pairs in 2D

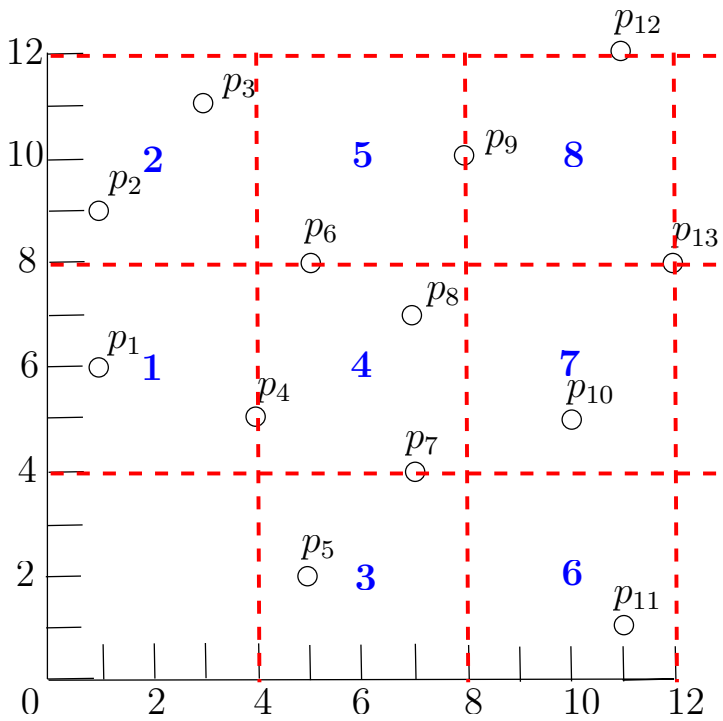
We will explain the algorithm using the same dataset and $r = 4\sqrt{2}$.



Step 1: Impose an arbitrary grid where each square cell has side length $r/\sqrt{2} = 4$. Identify all the non-empty cells (there are 8 such cells in our example).

Close Pairs in 2D

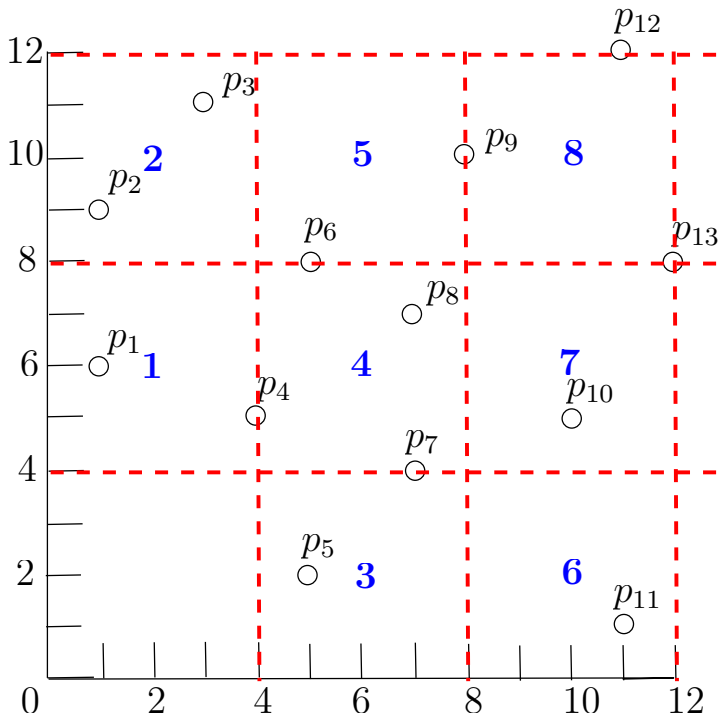
Step 2: For each cell c, let $c(P)$ be the set of points covered by c. Simply report all pairs of distinct points in $c(P)$—notice that any two points in the same cell must have distance at most r.



For example, 1 pair is reported for Cell 1, and 3 pairs for Cell 8.

Close Pairs in 2D

Step 3: For each cell c_1, identify all of its r-neighbor cells c_2. For every c_2, inspect all pairs of distinct points $(p_1, p_2) \in c_1(P) \times c_2(P)$, and report the ones within distance at most r.



For example, from Cells 2, 4, inspect all the 8 pairs in $\{p_2, p_3\} \times \{p_4, p_6, p_7, p_8\}$, and report $(p_2, p_4), (p_2, p_6), (p_3, p_6)$.

Close Pairs in 2D: Analysis

Next, we will prove that our algorithm runs in $O(n + k)$ expected time. At first glance, this may look a bit surprising. Recall that in Step 3, for each pair of r-neighbor cells (c_1, c_2), we spend a quadratic amount of time $O(|c_1(P)||c_2(P)|)$, but risk finding no answer pairs at all. Indeed, the core of the analysis is to show that the total time of doing so is bounded by $O(n + k)$.

We will focus on Steps 2 and 3 because Step 1 obviously takes $O(n)$ expected time (by hashing).

Close Pairs in 2D: Analysis (Step 2)

Let $c_1, c_2, ..., c_m$ be the non-empty cells, for some $m \geq 1$. Define $n_i = |c_i(P)|$, namely, the number of points covered by c_i, for each $i \in [1, m]$. Clearly $\sum_{i=1}^{m} n_i \geq n$.

The cost of Step 2 is obviously

$$\sum_{i=1}^{m} O(n_i^2)$$

Notice that

$$k \geq \sum_{i=1}^{m} n_i(n_i - 1)/2 = \frac{1}{2}\sum_{i=1}^{m} n_i^2 - \frac{1}{2}\sum_{i=1}^{m} n_i \geq \frac{1}{2}\left(\sum_{i=1}^{m} n_i^2\right) - \frac{n}{2}.$$

We thus have

$$\sum_{i=1}^{m} O(n_i^2) = O(n + k).$$

Close Pairs in 2D: Analysis (Step 3)

We will prove that the cost of Step 3 is $\sum_{i=1}^{m} O(n_i^2)$, and therefore, bounded by $O(n+k)$.

Let c_i and c_j be a pair of r-neighbor cells. Step 3 spends $O(n_i \cdot n_j)$ time to process $c_i(P) \times c_j(P)$. Clearly:

$$n_i \cdot n_j \leq (n_i^2 + n_j^2)/2.$$

Close Pairs in 2D: Analysis (Step 3)

The total cost of Step 3 can be written as

$$O\left(\sum_{i=1}^{m} \sum_{j:\ c_j \text{ is an } r\text{-neighbor of } c_i} n_i \cdot n_j\right)$$

$$= O\left(\sum_{i=1}^{m} \sum_{j:\ c_j \text{ is an } r\text{-neighbor of } c_i} (n_i^2 + n_j^2)\right)$$

(by the inequality of the previous slide)

As a cell has $O(1)$ r-neighbor cells, we know that each n_i^2 can appear only $O(1)$ times in the above summation. Therefore, the summation is bounded by $O(\sum_{i=1}^{m} n_i^2)$.

We now conclude that the running time of our close-pairs algorithm is $O(n + k)$ expected.