6.7 Incremental construction

Given a set $S = \{p_1, p_2 \ldots p_n\}$ of n points on a plane, we want to find a pair of points $q, r \in S$ such that $d(p, q) = \min_{p_i, p_j \in S} d(p_i, p_j)$ where $d()$ computes the Euclidean distance between two points. The pair (q, r) is known as the closest pair and it may not be unique. Moreover, the closest pair has distance zero if the points are not distinct [4]

In one dimension, it is easy to compute the closest pair by first sorting the points and choosing an adjacent pair which has the minimum separation. A trivial algorithm is to compute all the $\binom{n}{2}$ pairs and choose the minimum separated pair, so we would like to design a significantly faster algorithm.

A general approach to many similar problems is given in Figure 6.8. The idea is to maintain the closest pair in an incremental fashion, so that in the end we have the required result.

```
Algorithm 1: Closest pair($S$)
   Input $P = \{p_1, p_2 \ldots p_n\}$ ;
1  $S = \{p_1, p_2\}$ ; $\mathcal{C} = d(p_1, p_2)$; $j = 2$ ;
2  while $j \leq n$ do
3      if $d(p_j, S) < \mathcal{C}$ then
4          $\mathcal{C} = d(p_j, q)$ where $q = \arg\min_{p \in S} d(p_j, p)$
5      $P \leftarrow P \cup \{p_j\}$  $j \leftarrow j + 1$
6  Output $\mathcal{C}$ as closest pair distance.
```

Figure 6.8: Incremental Algorithm for closest pair computation

While the correctness of the algorithm is obvious, the analysis depends on the test in line 3 and the update time in line 5. For simplicity, let us analyze the running time for points in one dimension. Suppose the distances $d(p_{j+1}, S_j)$ are decreasing where $S_j = \{p_1, p_2 \ldots p_j\}$. Then the closest pair distance $\mathcal{C}$ is updated in every step. To find the closest point from p_{j+1} to S_j, we can maintain S_j as a sorted set and we can find the closest point from p_{j+1} using a binary search in $O(\log j) = O(\log n)$ time. Overall, the algorithm takes $O(n \log n)$ time which is the same as presorting the points.

For points on a plane, we have to design a data structure to efficiently perform the test in line 3 and update in line 5. Trivially it can be done in $O(n)$ steps leading to an $O(n^2)$ time algorithm. Instead we analyze the algorithm for a random ordering of

[4] So the lower bound for element distinctness would hold for the closest pair problem.

points in S. This will potentially reduce the number of updates required significantly from the worst case bound of $n-2$ updates. Let q_i denote the probability that point p_i causes an update when the points are inserted in a *random order.* A random ordering corresponds to a random permutation of points in P. To avoid extra notations let us assume that $p_1, p_2 \ldots p_n$ is according to a randomly chosen permutation.

We can restate our problem as
When $p_1, p_2 \ldots p_i$ is a random ordering of the set of points $P = \{p_1, p_2 \ldots p_i\}$ what is the probability that p_i defines the closest pair ?

Suppose the closest pair is unique, i.e., $\mathcal{C} = d(r, s)$ for some $r, s \in \{p_1, p_2 \ldots p_i\}$. Then, this probability is the same as the event that $p_i = \{r, s\}$. The total number of permutations of i objects is $i!$ and the total number of permutations with r or s as the last element is $2(i-1)!$. So the probability that p_i defines $\mathcal{C}$ equals $\frac{2(i-1)!}{i!} = \frac{2}{i}$. In a random permutation of n elements, the previous argument holds for a fixed set of i points. The *law of total probability* states that

$$\Pr[A] = \Pr[A|B_1] \cdot \Pr[B_1] + \Pr[A|B_2] \cdot \Pr[B_2] + \ldots \Pr[A|B_k] \cdot \Pr[B_k] \quad \text{for disjoint events } B_1, B_2 \ldots \tag{6.7.6}$$

In the above situation B_i represent each of the $\binom{n}{i}$ possible choice of i elements as the first i elements and by symmetry the probabilities are equal as well as $\sum_i \Pr[B_i] = 1$. Since $\Pr[A|B_i] = \frac{2}{i}$, the unconditional probability of update in the i-th step is $\frac{2}{i}$.

This is very encouraging since the expected update cost of the i-th step is $\frac{2}{i} \cdot U(i)$ where $U(i)$ is the cost of updating the data structure in the i-th step. Therefore even for $U(i) = O(i \log i)$, the expected update time is $O(\log i) = O(\log n)$.

The situation for the test in line 3 is somewhat different since we will execute this step regardless of whether update is necessary. Given S and a new point p_i, we have find the closest point from p_i and S (and update if necessary). Suppose the closest pair distance in S is D, then consider a $D \times D$ grid of the plane and each point of S is hashed to the appropriate cell. Given the new point $p_i = (x_i, y_i)$, we can compute the cell as $\lceil \frac{x_i}{D} \rceil, \lceil \frac{y_i}{D} \rceil$. It can be seen (Figure 6.9) that the closest point to p_i is within distance D then it must lie in one of the neighboring grid cells, including the one containing p_i. We can exhaustively search each of the nine cells.

Claim 6.1 *None of the cells can contain more than 4 points.*

This implies that we need to do at most $O(1)$ computations. These neighboring cells can be stored in some appropriate search data structure (Exercise **??**) so that it can be accessed in $O(\log i)$ steps. In line 4, this data structure can be rebuilt in $O(i \log i)$ time which results in an expected update time of $O(\log i)$. So, the overall expected running time for the randomized incremental construction is $O(n \log n)$.