Randomized algorithm for Closest Pair Problem

The Closest Pair Problem is a Las Vegas type of algorithm. This means that the algorithm itself will always produce either the correct result or inform us about the failure. The algorithm has a very simple problem to solve. Given a set of points, \( P = \{ p_1, p_2, \ldots, p_n \} \), in a plane, we want to find a particular pair of points \( \{ p_1, p_2 \} \), in \( P \) which are closest together. In this case, we consider \( P_1 \) to have coordinates \( (x_1, y_1) \) and \( d(p_1, p_2) \) to represent the standard Euclidean distance between the points. Therefore, we must find points that minimize \( d(p_1, p_2) \).

Solution: Brute force and Divide and Conquer

The first type of algorithm we know to solve this issue is brute force. This type of algorithm will compare each point to each of the other points and figure out the minimum distance. This will produce an inefficient run time of \( O(n^2) \). We then examine the Divide and Conquer approach. We want to sort the points according to their \( x \)-coordinates, splitting the points into two equal-sized subsets. We then solve the problem recursively on the left side and on the right side along with the points lying on left and right. We then compare the minimum points on each side and generate the most minimal solution. This algorithm is more efficient than the brute force, producing a run time of \( O(n \log n) \).

Solutions: randomized algorithm

The next algorithm, the one we are interested in, is the Closest Pair via Randomization. This algorithm is the most efficient out of the three, with the run time of \( O(n) \). We will first examine how the algorithm runs and then examine how it came to be linear in runtime. We want to consider the points in a random order and assume that all points are within the unit square. We initially set \( \delta = d(p_1, p_2) \) to be the minimal distance. The algorithm then adds another point in constant time. We want to subdivide the unit square into \( \delta/2 \times \delta/2 \) squares. There are two properties one must take into account when subdividing squares to determine the closest pair. The first one being that if \( p_1 \) and \( p_2 \) are in the same subsquare, then they are the closest pair and, thusly, less than \( \delta \). We also want to keep in mind that when
checking the proximity of points, it is sufficient to check at most 25 points, which you get by forming a 5x5 grid around the points. We then store the number of the subsquare it belongs to. Every time δ is updated, we update the subdivision and the number of subsquares associated with points.

This poses two challenges. The first being that we need to find which points are contained in the subsquare close to the subsquare containing the new point, very quickly so we can maintain the O(n) time. We can solve this problem by introducing hash tables. The universe U is the set of N^2 subsquares. The set S of elements in the dictionary will be the set of subsquares occupied by points p_1, p_2, ..., p_i. Therefore, every subsquare in the dictionary also stores the index of the point it, solving the first challenge. The second problem we can face is having a constant time for updating δ. We have observed that checking if a subsquare close to the subsquare with the new point contains a point and checking the distance to this point can be done in expected constant time.

**Analysis and Conclusion**

Furthermore, we want to analyze how the algorithm could produce such an efficient run time of O(n). We can observe that distance computations, lookup operations and MakeDictionary only take O(n) time. The only part we might have an issue with is the insert. Examining this, we can figure out the number of insert operations. We can use X_i to represent a random variable which equals 1 if, when adding a point, the distance changes and 0 otherwise. We consider the total number of operations to be n + Σ_i i X_i as well as the probability of X_i to be one is 2/i. Let us think of the minimum distance between p_1 and p_2. There are two ways the point p_i causes the minimum distance to decrease; either p_i is equal to p_1 or p_i is equal to p_2, thusly, the probability is 2/i. We can, therefore, say that E(X) = n + Σ_i i X_i ≤ n + Σ_i i 2/i = 3n, which is in constant, linear time. We already know that there are O(n) distance computation, O(n) lookup operations, O(n) MakeDictionary, and we now know that insert also has O(n) operations due to the 3 is negligible. Therefore, we can observe that the Randomized Algorithm for Closest Pair problem is the most efficient as it runs in O(n) time.
References:

• http://www.cs.cmu.edu/~ckingsf/class/02713/lectures/lec31-random.pdf
• http://www.cs.tufts.edu/comp/250P/classpages/classes.html
• http://www.cs.arizona.edu/classes/cs545/fall09/closestpair.prn.pdf
• Chapter 13.7 in Kleinberg - Tardos book
• http://www.cs.sfu.ca/CourseCentral/405/abulatov/lectures/20.pdf
• http://www.yovisto.com/video/9660
• http://bigocheatsheet.com/
• http://stanford.edu/~ejdai/notes/cs265notes.pdf
• http://www.iitg.ac.in/rinkulu/algorithms/slides/lasvegas-closestpp.pdf