# COMP 5703 - Finding the closest pair of points

Katie Duong - 100801959

October 30, 2015

**Abstract**

We consider the problem of computing the closest pair in a set of $n \geq 2$ points. "Closest" refers to the usual Euclidean distance: the distance between two points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ is

$$d(p_1, p_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

I will present a randomized incremental algorithm. This algorithm adds the points one by one, and maintains the closest pair. It stores the points in a grid, with cellsize the current closest pair distance. This grid is used to update the closest pair when the next point is added. If the grid is stored using a binary search tree, the expected running time is $O(n \log n)$. Using hashing, the expected time improves to $O(n)$.

## 1 Introduction

Given a set $S$ of $n$ points in the plane, we want to find the pair of points closest to each other: Return the pair of points $P$, $Q$ in $S$ such that:

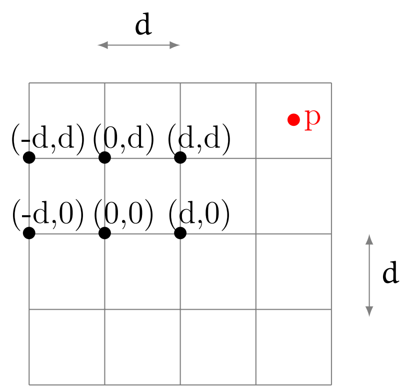
$$d(P, Q) = \min\{d(p, q) : p, q \in S, p \neq q\},$$

where $d(p, q)$ is the Euclidean distance between $p$ and $q$.

In this paper I will present an incremental algorithm for this problem. The worst-case running time is $O(n^2 \log n)$. I will use randomization to get an expected running time of $O(n \log n)$. Using hashing, the running time improves by $\log n$.

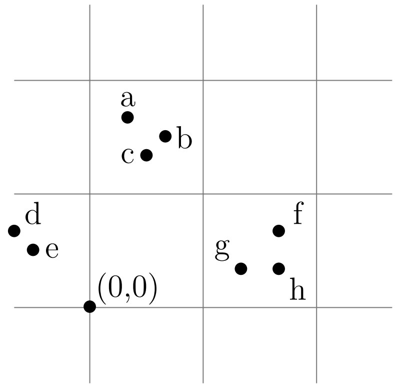
## 2 Grids

A $d$-grid is a grid with cells of sidelength $d$. Each cell has lower-left corner with coordinates $(id, jd)$ and upper-right corner with coordinates $((i + 1)d, (j + 1)d)$, for integers $i$ and $j$.




The $id$ of a cell with lower-left corner $(id, jd)$ is $(i, j)$. The point $p = (p_x, p_y)$ is in the cell with $id = (\lfloor p_x/d \rfloor, \lfloor p_y/d \rfloor)$.

## 2.1 Example




The cell with $\mathtt{id} = (1, 0)$ contains the points: f, g and h
The cell with $\mathtt{id} = (-1, 0)$ contains the points: d and e
The cell with $\mathtt{id} = (1, 1)$ contains the points: a, b and c

## 2.2 Storing points in a grid

To store a set $S$ of points in a $d$-grid:

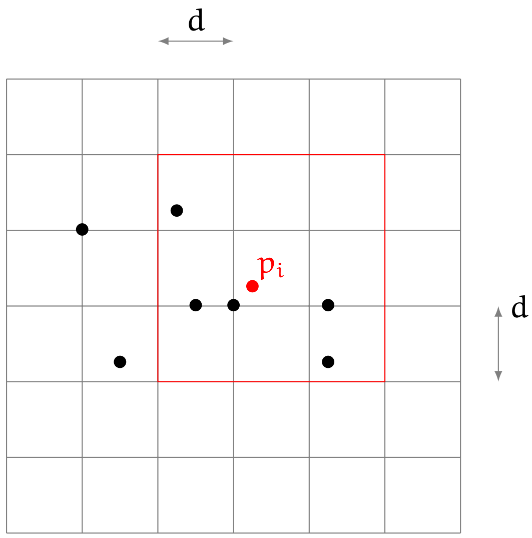
1. For each $p \in S$ we compute the id of the cell containing $p$.
2. Store the id's of the non-empty cells in a binary search tree or a hash table.
3. With each non-empty cell, store a list of all points in this cell.

# 3 The Algorithm

Let $S = \{p_1, p_2, \dots, p_n\}$ be a set of $n$ points in the plane. The algorithm computes a closest pair in the set $S_i = \{p_1, p_2, \dots, p_i\}$ for $i = 2, 3, \dots, n$.

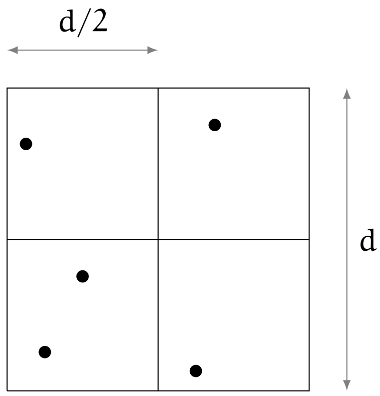
1. $i := 2$; CP-distance $= d(p_1, p_2)$;
2. for $i = 3, 4, \dots, n$, given the CP-distance $d$ of $S_{i-1}$, compute the CP-distance of $S_i$.

In the for-loop, we store the points of $S_{i-1}$ in a $d$-grid. To compute the CP-distance of $S_i$, we compute the distance beween $p_i$ and all points in the 9 cells around $p_i$. If the CP-distance does not change, we add $p_i$ to the $d$-grid. If the CP-distance changes, we store all points of $S_i$ in a new grid.



## 3.1 How many points in one cell?

One cell has at most 4 points. To prove this by contradiction, we assume the number of point in one cell is at least 5. Split this cell into 4 cells of size $d/2$. Then one of the small cells has 2 points, their distance is at most $\sqrt{2}d/2 < d$. This is a contradiction, because $d$ is the CP-distance.




## 3.2 Time for one iteration

To update the CP-distance $d$, we need to find 9 cells, each having at most 4 points. This will take $O(\log i)$ if we store the grid in a binary search tree and $O(1)$ if we store it in a hash table. Computing at most 36 distances takes $O(1)$.

If $d$ did not change: insert the id of $p_i$'s cell into the binary search tree, takes $O(\log i)$, or into the hash table, takes $O(1)$ time.

if $d$ changed: Store the points of $S_i$ in the grid for the new $d$. With a binary search tree takes $O(i \log i)$ and hash table $O(i)$.

## 3.3 Total running time

The total running time:

1. $O(n^2 \log n)$ using a binary search tree.
2. $O(n^2)$ using a hash table.

Consider the following example:


In each iteration, $d$ changes and we take a new grid:
Iteration 1: store $p_1$ and $p_2$, we get CP-distance $= d(p_1, p_2)$.
Iteration 2: add $p_3$, we get CP-distance $= d(p_2, p_3)$.
Iteration 3: add $p_4$, we get CP-distance $= d(p_3, p_4)$.
Iteration 4: add $p_5$, we get CP-distance $= d(p_4, p_5)$.
In this case, the total running time is $\Theta(n^2 \log n)$ using a binary search tree and $O(n^2)$ using hashing.

Consider the same example, but number the points from right to left:


The grid-size never changes:
Iteration 1: store $p_1$ and $p_2$, we get CP-distance $= d(p_1, p_2)$.

Iteration 2: add $p_3$, we get CP-distance $= d(p_1, p_2)$.

Iteration 3: add $p_4$, we get CP-distance $= d(p_1, p_2)$.

Iteration 4: add $p_5$, we get CP-distance $= d(p_1, p_2)$.

In this case, the total running time is $\Theta(n \log n)$ using a binary search tree and $O(n)$ using hashing.

# 4 The randomized algorithm

The running time depends on the numbering of the points. We take a random numbering of the points, take a random permutation $p_1, p_2, \ldots, p_n$.

The random variable $X$ denotes the total running time. First, I will determine the expected value $E(X)$ when hashing is used.

Define

$$X_i = \begin{cases} 1 & \text{if the grid changes in iteration } i \\ 0 & \text{if the grid does not change in iteration } i \end{cases}$$

Then

$$X = O(n) + \sum_{i=3}^{n} X_i \cdot O(i)$$

and

$$E(X) = O(n) + \sum_{i=3}^{n} E(X_i) \cdot O(i).$$

We know that

$$E(X_i) = \Pr(X_i = 1).$$

In iteration $i$, we insert $p_i$ into the grid for $S_{i-1}$. Afterwards, we have the grid for $S_i$. The grid changes if and only if the CP-distance of $S_i$ is smaller than the CP-distance of $S_{i-1}$. Imagine that we run the algorithm backwards: in this iteration, we pick a random point in $S_i$ and delete it.

Let $(a, b)$ be the CP-pair in $S_i$. The CP-distance changes if and only if $a$ or $b$ is deleted. Each point of $S_i$ has a probability of $1/i$ of being deleted. Therefore,

$$E(X_i) = \Pr(X_i = 1) = 2/i.$$

We get

$$E(X) = O(n) + \sum_{i=3}^{n} \frac{2}{i} \cdot O(i) = O(n) + \sum_{i=3}^{n} O(1) = O(n).$$

If we use a binary search tree, then

$$X = O(n \log n) + \sum_{i=3}^{n} X_i \cdot O(i \log i)$$

and

$$E(X) = O(n \log n).$$

# 5 Lower bounds

Consider the following problem. Given numbers $a_1, \ldots, a_n$, decide if they are all different. Algorithms for this problem that use $+$, $-$, $\times$, $/$, $\sqrt{}$, $<$, $\leq$, and nothing else, take $\Omega(n \log n)$ time. This was proved in [7] for deterministic algorithms and in [6] for randomized algorithms.

The numbers $a_1, \ldots, a_n$ are all different is the same as their CP-distance is $> 0$. The same lower bounds hold for the closest pair problem.

The randomized algorithm in this paper takes $O(n)$ expected time if hashing is used. This is not a contradiction, because hashing uses more operations, such as the modulo-operation. That means, the hashing based algorithm is not in the same model of computation as in [6, 7].

# References


[1] Sariel Har-Peled. *Geometric Approximation Algorithms.* American Mathematical Society.

[2] Jon Kleinberg, Eva Tardos, *Algorithm Design*, Chapter 13. Addison Wesley.

[3] Michiel Smid. *The closest pair problem: a randomized incremental algorithm.* Carleton University

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms.* The MIT Press Cambridge, Massachusetts London, England.

[5] Mordecai Golin, Rajeev Raman, Christian Schwarz, Michiel Smid. *Simple randomized algorithms for closest pair problems.* Nordic Journal of Computing, 2, 1995, pages 3-27.

[6] Dima Grigoriev, Marek Karpinski, Friedhelm Meyer auf der Heide, Roman Smolensky. *A lower bound for randomized algebraic decision trees.* Computational Complexity, 6, pages 357-375, 1996.

[7] Michael Ben-Or. *Lower bounds for algebraic computation trees.* Proc. 15th ACM Symposium on Theory of Computing, pages 8086, 1983.