CpSc 2120: Algorithms and Data Structures

Instructor: Dr. Brian Dean — Fall 2019
Webpage: `http://www.cs.clemson.edu/~bcdean/` — MW 2:30-3:45
Handout 12: Lab #7 — Daniel 313

1 STL Practice: Finding the Closest Pair of Points

In lab #3, we used a "spatial hashing" approach to find the closest pair of points among 1 million points in the 2D plane (recall that the simple $\Theta(n^2)$ approach of simply checking all pairs of points would be far too slow for a data set of this size). In this lab, we will see yet another approach for solving this problem, which will help us gain familiarity with using the C++ Standard Template Library (STL). This approach runs in $O(n \log n)$ time, so it should be competitive with the spatial hashing approach; moreover, it does not depend on points being uniformly distributed, as was the case with spatial hashing.

The input file for this lab is the same as with lab #3:

`/group/course/cpsc212/f19/lab03/points.txt`

It contains one million points in the 2D plane, one per line.

2 The Algorithm and its Implementation

We will be storing points in STL pairs; for example, we could declare a point p using:

```
pair<double,double> p;
```

or if we wanted to give the `pair<double, double>` type a friendlier name:

```
typedef pair<double,double> Point;
Point p;
```

Afterwards, `p.first` refers to the x coordinate of point p, and `p.second` refers to its y coordinate. We will store our points in an STL vector:

```
vector<Point> v;
```

A new point (x, y) can be added to the end of this vector very easily, by simply writing

std::make_pair

```
v.push_back(make_pair(x,y));
```

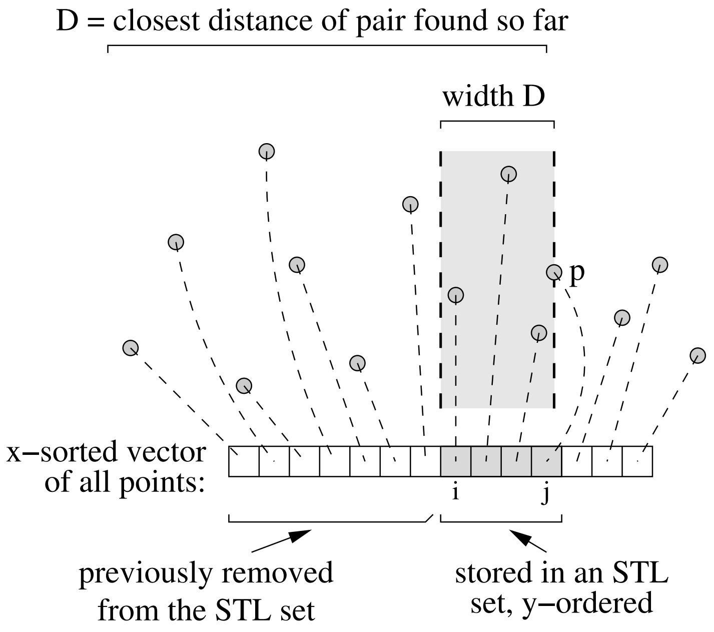
D = closest distance of pair found so far
width D
p
x−sorted vector of all points:
i
j
previously removed from the STL set
stored in an STL set, y−ordered

Figure 1: A sweep line approach for computing the closest pair of points in $O(n \log n)$ total time.

Using this approach, you should be able to read the input without pre-assuming that there are 1 million points (since you don't need to worry about pre-allocating space for the vector, as it grows as needed); rather, you should continue to read points from the input file until you reach the end of the file.

Our algorithm can be described as a type of "sort and scan" or "sweep line" approach. We first sort all the points on x (breaking ties arbitrarily). This is easy using the STL sort command:

```
sort(v.begin(), v.end());
```

This is one of the benefits of using a "pair" to hold each point instead of an arbitrary struct. The STL sort command knows how to compare two pairs (by comparing their first elements, and breaking ties on their second elements); hence the STL sort command can easily sort a vector of pairs. If we were instead inclined to store points in a structure of our own design, we would need to to define an auxiliary comparison function and pass this along to the STL sort function.

We now sweep from left to right across the 2D plane, visiting all points in increasing order of x. As we scan, we will maintain the closest distance D among all points we have scanned so far. At the end of the algorithm, D tells us our final answer.

As we encounter each point `v[j]`, we need to potentially update D by checking `v[j]` against the points we have already scanned (those on our left), to see if any of them are closer than D units of distance from `v[j]`. To do this quickly, we make a few observations. First, points farther than D units from `v[j]` in just the x dimension are clearly not worth checking. We therefore only need to check `v[j]` against points in a width-D strip to the left of `v[j]`, as shown in Figure 1. To maintain the contents of this strip, we keep track of an index i (as shown in the figure) that tracks the left endpoint of the strip. As you advance j, the index i should also advance as needed.

You should store the contents of the width-D strip in an STL set, ordered by y. This means you will need to swap the x and y coordinates of a point pair before inserting them into the set; for example, to insert the point `v[j]` into an STL set this way:

store all points in strip D (height is much larger than 2D), then for each j in the strip, calculate distance

```
set<Point> s;
s.insert(make_pair(v[j].second, v[j].first));
```

As j increases, you should insert points into the set, and as i increases, you should remove points from the set, so the set always contains exactly the elements in the strip. Moreover, since the set is keyed on y, this now facilitates fast searching based on y coordinate for points within the strip.

To check for points closer to `v[j]` than distance D, we start at `v[j]` in the STL set and check its successor elements in turn (e.g., the elements within the strip having the next-largest y coordinates), stopping when we reach an element differing from `v[j]` by more than D units in the y dimension. We then do the same thing to check predecessor elements. A key observation is that we will only need to check at most a constant number of successors and predecessors, since these elements are all at most D units apart from each-other, and hence there cannot be too many of them within a $D \times 2D$ box. In terms of how we implement this using the STL, we first call `find` to return an iterator (think of this as a pointer) to the record in the set corresponding to `v[j]`. We can then increment and decrement this iterator to move to successors and predecessors, as illustrated in the following code example:

```
set<Point>::iterator it;
it = s.find(Point(v[j].second, v[j].first));
it++; // move to successor
if (it == s.end()) ... // have we reached the one element past the end of the set?
it = s.find(Point(v[j].second, v[j].first));
it--; // move to predecessor
if (it == s.begin()) ... // have we reached the beginning of the set?
// it->first is the y coordinate of the point currently referenced by it
// it->second is the x coordinate of the point currently referenced by it
```

The total running time of this algorithm is $O(n \log n)$, since it makes n insertions and finds, at most n deletions, and $O(n)$ successor/predecessor calls in an STL set; each of these operations takes $O(\log n)$ time.

3 Grading

When submitting your code, please name your file `main.cpp`, and please output only the distance between the closest pair of points, with no other debugging output (to facilitate easier grading).

For this lab, you will receive 8 points for correctness and 2 points for having well-organized, readable code. Zero points will be awarded for code that does not compile, so make sure your code compiles on the lab machines before submitting!

Final submissions are due by 11:59pm on the evening of Friday, October 25. No late submissions will be accepted.