Scott Wiedemann

September 13, 2010

CSCI 562

**Closest Pairs Project**

1. Describe your algorithm in English or pseudocode and argue that it works correctly.

Description:

My algorithm is quite simple and can be thought of as a sliding window on a high level. The steps below outline the general procedural of the algorithm, but do not describe how the code was optimized to obtain low run times in practice. Refer to the source code for these details.

1. For a given set of points, read all points into an array, *points*, of length *n*, that contains the coordinates of each point, *(x, y)*.
2. Sort this array with respect to *x*.
3. Calculate the Euclidian distance between of the first two points in the array (the two left most points) and assign it to *delta*, which will be used to store the minimum distance of the entire set of points.
4. Outer loop: Index, *right*, will start at 1 and go to *n*.
5. Inner loop: Index, *left*, will start at *right*-1 and decrease while *points[left].x* > *points[right].x* – *delta*. Meaning that *x* of the current point of interest, *points[left]*, lies within delta to the left of the index right.
6. Within the inner loop, the *y* of the current point of interest, *points[left].y*, must be less than *points[right].y* + *delta*,and greater than *points[right].y* – *delta*. This step is only done to avoid commuting a square root more often. (As done when finding Euclidian distance.)
7. Finally, if the Euclidian distance between *points[left]* and *points[right]* is less than *delta*, assign this value to *delta*. This is the smallest distance found so far.

Proof of correctness (by contradiction):

Given a set of points *P*, consider two points, *p1* and *p2,* in *P* which have the smallest Euclidian distance *delta*true, and where *p1.x* < *p2.x*. Furthermore suppose these points lie to the left of *points[right].x* – *delta* (the algorithm missed the points). *delta*true is also less than the current *delta* (*delta*true< *delta*). When *points[right].x* was equal to *p2.x*, *p1.x* was either less than or equal to *p2.x* – *delta* (*p1.x* ≤ *p2.x* – *delta),* or greater than *p2.x* – *delta* (*p1.x* > *p2.x* – *delta*). If *p1.x* ≤ *p2.x* – *delta* this contradicts that *delta*true< *delta*. If *p1.x* > *p2.x* – *delta* then the algorithm will update *delta* to *delta*true (*delta* = *delta*true).

1. What is the time complexity of your algorithm?

The time complexity of my program is *O(nlog(n))* which is caused by the sorting step (quick sort). Disregarding the sorting step (assume input data is already sorted), the time complexity of the solve routine is dependent on the distribution of the data. (Specifically on how *delta* is updated.) In practice, the time complexity seems to be *O(nlog(n))* at worst, and for typical distributions, generally quite better(closer to *O(n) than O(nlog(n))*).

1. Submit a print-out of your code. If you are unable to get your code to compile/run, please state this explicitly.

Refer to attached code.

1. Submit your code/executable to Blackboard to be run on a set of inputs devised by the grader. Your code will be run on an Alamode lab machine.

Complete.

1. Submit your code for robot judging and submit evidence that it ran successfully.

