2nd Lab Class – Part B – Divide and Conquer

Instructions

- Download the zipped file **DA_TP02b_unsolved.zip** from the course's Moodle area and unzip it. It contains the folder **TP2b**, with the *.h* and*.cpp* files, as well as files with datasets of points needed for this lab class.
- Copy the whole folder to the root of the CLion project provided in the first lab class.
- Edit the *CMakeLists.txt* in the root of the project, so as to set up an executable run/debug configuration for **TP2b**, accordingly:
  - Add the source files for the **TP2a** class:
    ```
    file(GLOB TP2b_FILES CONFIGURE_DEPENDS "TP2b/*.cpp")
    ```
  - Add executable target with source files listed in *TP2b_FILES* variable:
    ```
    add_executable(TP2b main.cpp ${TP2b_FILES})
    ```
  - Link the Google Test library to the *TP2b* target:
    ```
    target_link_libraries(TP2b gtest_main gmock_main)
    ```
- Do "*Load CMake Project*" over the file *CMakeLists.txt*. The **TP2b** configuration should now appear and be available from the "*Select Run/Debug Configuration*" drop-down list, on the upper-right corner of the CLion IDE.
- Compile and run the project after selecting the **TP2b** configuration.
- IMPORTANT: to read text files in I/O mode, you may need to tell CLion where such files are located. For exercise 1, you will need to read text files with datasets of points, located in the same folder as the sources files (i.e. **TP2b**). In the case you want to have the files elsewhere, you can either:
  - Reset the path (either relative or absolute) of the files containing the datasets of points, updating the value of `REL_PATH` in file *ex1.cpp*:
    `#define REL_PATH "../TP2b/"` (it should work as is, though!)
  - Redefine the IDE environment variable "Working Directory" for the **TP2b** configuration, through menu Run > Edit Configurations… > Working Directory. Make sure you change the code of file *ex1.cpp* accordingly, where it depends on the macro `REL_PATH`

Exercises

1. Closest pair problem

Suppose *P* is a list of points on a plane. If *p1=(x1,y1)* and *p2=(x2,y2)*, the Euclidean distance between *p1* and *p2* is given by:

$$[(x_1 - x_2)^2 + (y_1 - y_2)^2]^{\frac{1}{2}}$$

Along with the **.h** and **.cpp** files, you have been given data files with a (power of 2) number of random points. In the case where there are two points with the same coordinates, those are the closest points, with distance 0 between them.

a. Implement the ***nearestPoints_BF*** function following a brute force algorithm, and write down the time it takes to run.

b. Implement the ***nearestPoints_DC*** function using the divide and conquer algorithm described further down (except for the last part that requires a second list). Write down the time it takes to run and compare it to the values written down for point *a)*.

c. Implement the ***nearestPoints_DC_MT*** function, a multi-threaded version of the divide and conquer algorithm. Compare the performances obtained for different sizes of input data and different numbers of threads.

d. Indicate a loop invariant and loop variant for the main loop of the function in *a)*, and show they satisfy the properties needed to prove the algorithm correctness.

e. Prove that the time complexity of the divide and conquer algorithm in b) is *O(N log^2 N).*

f. Implement the function ***Result nearestPoints_BF_SortByX(vector<Ponto> &vp)*** that refines the brute force algorithm with an initial sorting by X. Execute the tests and check that the execution time is very good for random points, but not for points that differ only in Y.

Divide and Conquer Algorithm to compute the closest pair of points

(adapted from M.A. Weiss, "Data Structures and Algorithms Analysis in C++", 3rd edition –chapter 10, pages 430-435)

Suppose *P* is a list of points on a plane. If *p1=(x1,y1)* and *p2=(x2,y2)*, the Euclidean distance between *p1* and *p2* is given by:

$$[(x_1 - x_2)^2 + (y_1 - y_2)^2]^{\frac{1}{2}}$$

The objective is to find the two closest points. If there are two points with the same coordinates, those are the two closest, with distance 0.

If there are N points, then there are *N(N-1)/2* pairs of distances. One could look through all of them with a very simple exhaustive search (brute force) algorithm. However, that algorithm would have complexity *O(N^2).* With a divide and conquer algorithm like the one described below, one can guarantee *O(NlogN)* complexity.

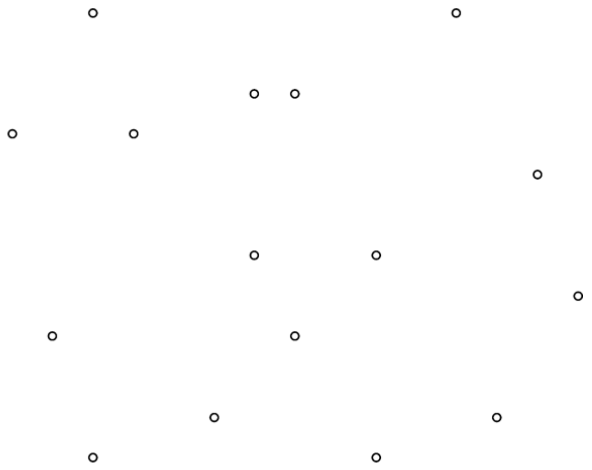


Figure 1 – A small set *P* of points

Figure 1 shows a small set of *P* points. If the points were sorted by their value in the abscissa (*x* axis), one could draw an imaginary vertical line which divides the set in two halves P_L and P_R. Given this division, either both points of the closest pair are in P_L, both are in P_R or one is in P_L and one is in P_R. We can call the distances between them d_L, d_R and d_C, as shown in figure 2.

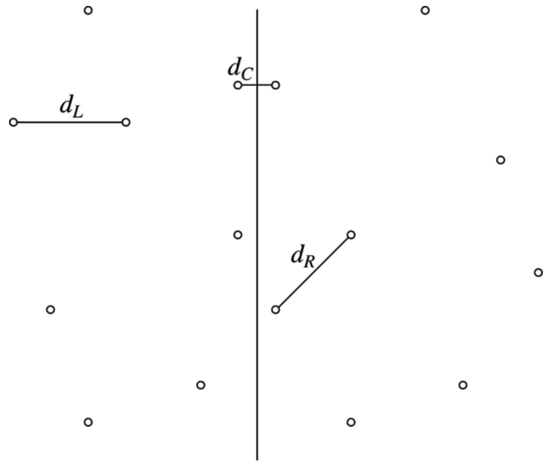


Figure 2 – Set *P* divided into P_L and P_R , with the minimum distances shown.

Computing d_L and d_R can be done recursively. O the problem is then to compute d_C. In order to guarantee a *O(NlogN)* complexity algorithm (needed to sort the values), it must be possible to compute d_C in *O(N)*.

Consider $\delta = (d_L, d_R)$. One only has to compute d_C if it is smaller than δ. With that in mind, it can be said that the two points which define d_C should be less than δ distance from the dividing line. We will name this area the **strip**.

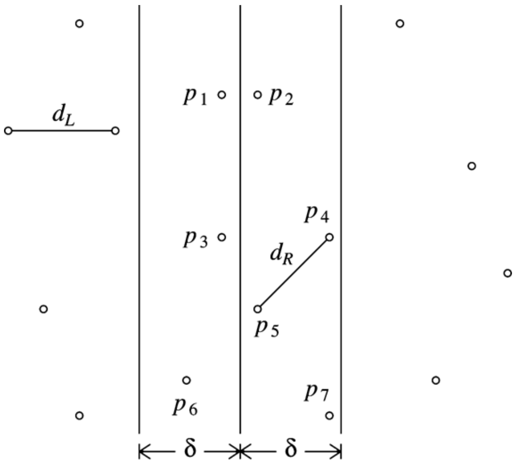
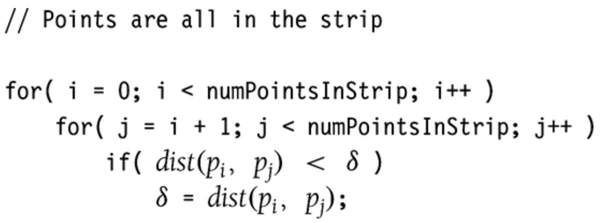


Figure 3 – Two bands containing all of the points considered for the strip d_C

There are two strategies to compute d_C. For a very large set of uniformly distributed points, the number of points expected to be contained in the strip is very small – on average there will be $O(\sqrt{N})$ points. In that case, the brute force algorithm can be used in this strip in time $O(N)$. The pseudo-code for this strategy is shown in figure 4.

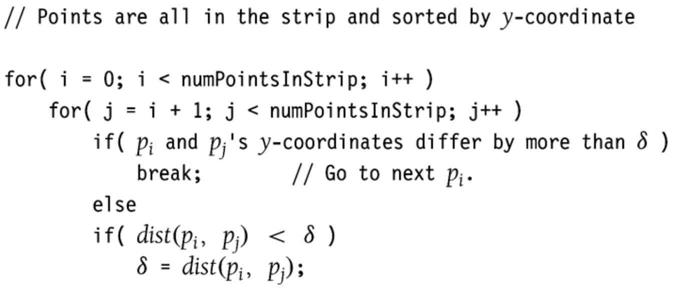


```
// Points are all in the strip

for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( $dist(p_i, p_j) < \delta$ )
            $\delta$ = $dist(p_i, p_j)$;
```

Figure 4 – Brute force algorithm to compute *min(δ, dC)*

In the worst case scenario, every point can be in the strip. In this case, the brute force strategy does not run in linear time. It is necessary to look closely at the problem in order to improve the algorithm: the *y* coordinates of the two points which define d_C should differ, at most, δ; otherwise, $d_C > \delta$. Suppose the points in the strip are sorted by their *y* coordinate. If the y coordinates of points p_i and p_j differ more than δ, the algorithm skips to point p_{i+1}. This simple modification is implemented in the algorithm shown in figure 5.



```
// Points are all in the strip and sorted by $y$-coordinate

for( i = 0; i < numPointsInStrip; i++ )
    for( j = i + 1; j < numPointsInStrip; j++ )
        if( $p_i$ and $p_j$'s $y$-coordinates differ by more than $\delta$ )
            break;        // Go to next $p_i$.
        else
        if( $dist(p_i, p_j) < \delta$ )
            $\delta$ = $dist(p_i, p_j)$;
```

Figure 5 – Improved computation of *min(δ, d_C)*

This simple additional test has a very significative effect on the algorithm's behavior, because for each point p_i, very few points p_j are examined (if their coordinates differ by more than δ, the internal *for* loop is terminated). Figure 6 shows, for example, that for point *p3*, only points *p4* and *p5* are less than δ distance away on the vertical axis.

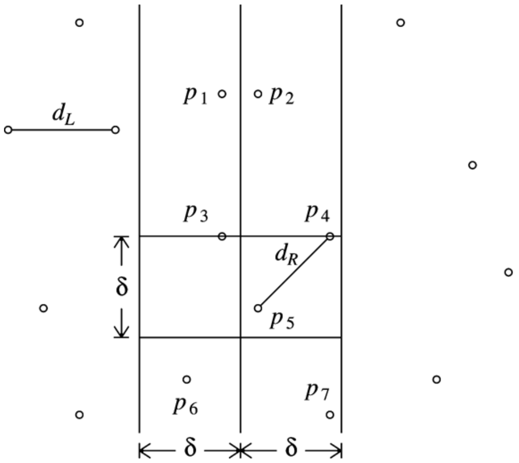


Figura 6 – Only points p_4 e p_5 are considered in the second *for* loop

In the worst case scenario, for any point p_i at most seven points p_j will be considered. The reason for this is simple to understand: these points must be contained in a δxδ square on the left half of the strip or a δxδ square on the right half of the strip. On the other hand, all points in each δxδ square are at least δ distance from each other. Worst case scenario, each square contains four points, one in each corner. One of those points is p_i, leaving, therefore, at most seven points to be considered. This case is illustrated in figure 7. Even in points p_{L2} and p_{R1} have the same coordinates they can be different points. In this analysis, the important thing to notice is that the number of points in the λ by 2λ rectangle is *O(1)*.

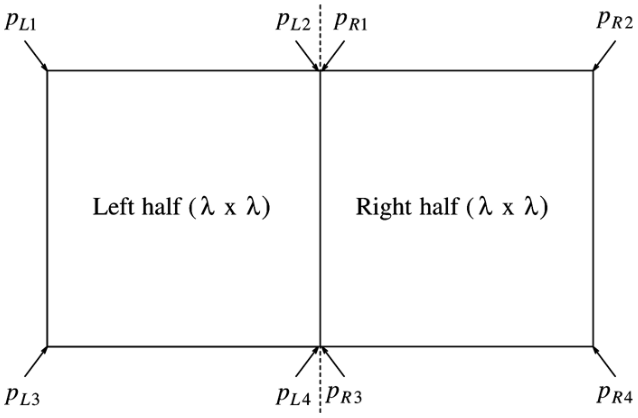

p_{L1}
p_{L2}
p_{R1}
p_{R2}
Left half (λ x λ)
Right half (λ x λ)
p_{L3}
p_{L4}
p_{R3}
p_{R4}

Figure 7 - There at most eight points in the rectangle, each sharing two coordinates with other points.

Given that there are at most seven points to be considered for each p_i, the time needed to compute a d_C better than δ is *O(N)*.It seems, then, that a *O(NlogN)* solution for the closest pair problem has been found, based on the recursive call of the left and right halves plus the linear time to combine the results.

However, this solution is not effectively *O(NlogN)*.It has been assumed that the list of points is sorted. If this operation is done in each recursive call, then there is work of complexity *O(NlogN)* to consider, which results in an overall complexity of $O(N\log^2 N)$. This is not too bad when compared to brute force’s $O(N^2)$. It is, however, relatively easy to reduce the complexity of each recursive call to *O(N)*, therefore guaranteeing overall complexity *O(NlogN)*.

The idea is to maintain two lists: one contains the points sorted by the *x* axis, while the other contains the points sorted by the *y* axis. This implies a first step where the points are sorted, with complexity *O(NlogN).* Let these lists be names *P* and *Q*, respectively. P_L and Q_L are the lists passed into the left half recursive call, while P_R e Q_R are the lists passed into the right half recursive call. List *P* is easily split in half. Once the dividing line is known, *Q* is traversed sequentially, placing each element in Q_L or Q_R as appropriate. It is easy to verify that both Q_L and Q_R are automatically sorted by *y* as they are filled in. Once the recursive call returns, all points in *Q* whose *x* coordinate does not belong within the strip are removed. That way, *Q* contains all of the points which are within the strip, already sorted by *y*.

This strategy guarantees that the overall algorithm has complexity *O(NlogN),* because the only extra processing which is done is done with complexity *O(N).*

2. The maximum subarray problem

Considering the same description for the maximum subarray problem of exercise 2 of the first practical class, implement the function *maxSubsequence* below using a divide and conquer algorithm instead.

```
int maxSubsequenceDC(int A[], unsigned int n , int &i, int &j)
```

The function returns the sum of the maximum subarray, for which *i* and *j* are the indices of the first and last elements of this subsequence (respectively), starting at 0.

For example: **A** = [−2, 1, −3, 4, −1, 2, 1, −5, 4]
Solution: [0, 0, 0, 1, 1, 1, 1, 0, 0], as subsequence [4, −1, 2, 1] (i = 3, j = 6) produces the largest sum, 6.