Separating n-points using Greedy approach:

1. Pseudocode:

```
SEPARATE_POINTS_GREEDY( )

//OPERATION: Separate n-points in a graph with axis-parallel lines using Greedy
             approach.
//INPUT: The program will read instances files each containing the co-
          ordinates of the points.
           The points are sorted along x-axis and they won't share each other's axes.
//OUPUT: A Feasible solution containing the set 'S' of axis-parallel lines that
          separate the given instance points such that |S| is minimum. The solutions
          are printed in a file.

1)     //Integer identifiers
2)     file_inst = 1
3)     inst_count = 0
4)     x_lines[] = 0
5)     y_lines[] = 0
6)     max_files = 100
7)
8)     //Structure identifiers
9)     st_points = 0
10)    st_lines = 0
11)
12)    while file_inst < max_files
13)             READ_FILE(file_inst, st_points)
14)
15)             //Sort x and y axes values of the co-ordinates.
16)             QSORT_POINTS(st_points.x[0], st_points.y[0])
17)
18)             MAKE_CONNECTIONS(st_points)
19)
20)             //Finding axis parallel lines for both the axis, X = 0 and Y = 1
21)             FIND_AXIS_PARALLEL_LINES(0, st_points, st_lines)
22)             FIND_AXIS_PARALLEL_LINES(1, st_points, st_lines)
23)
24)             while i < st_lines.x_total and j < st_lines.y_total
25)                      conxn_status = CHECK_LINE_CONNECTIONS(st_lines.x[i], st_points)
26)                      if conxn_status == 1
27)                               COMMIT_LINE(st_lines.x[i], st_points, st_lines)
28)                      endif
29)                      i = i + 1
30)
31)                      conxn_status = CHECK_LINE_CONNECTIONS(st_lines.y[j], st_points)
32)                      if conxn_status == 1
33)                               COMMIT_LINE(st_lines.y[j], st_points, st_lines)
34)                      endif
35)                      j = j + 1
36)             endwhile
37)
38)             SAVE_SOLUTION_IN_FILE(inst_count)
39)             file_inst = file_inst + 1
40)    endwhile
41)
42)    /*****************************************************/
```

```
MAKE_CONNECTIONS(st_points)

//OPERATION: Form connections between all the points/co-ordinates in the graph.
//INPUT: st_points - structure having information on points that are sorted along
         their respective axis
//OUPUT: The structure "st_points" updated with connection information of the co-
         ordinates

1) i = 0
2) j = 0
3)
4) while i < st_points.total
5)        st_points.coord[i].index = i
6)        while j < st_points.total
7)              if i != j
8)                   st_points.coord[i].connection[j] = address_of(st_points.coord[j])
9)                   st_points.coord[i].connection_count =
                                        st_points.coord[i].connection_count + 1
10)                else
11)                     st_points.coord[i].connection[j] = NULL
12)                endif
13)           endwhile
14)endwhile
15)
16)/*****************************************************/

FIND_AXIS_PARALLEL_LINES(axis, st_points, st_lines)

//OPERATION: Recursively divide X and Y axis into half, as a line in the middle of a
             graph disconnects most of the points.
//INPUT: Structure "st_points" having points and connection information
//OUTPUT: Structure "st_lines" having the extracted axis parallel lines of the
graph.

1) //Stop condition for recursion
2) if st_points.begin == st_points.total
3)          return
4) endif
5)
6) //Calculate the half range
7) range = st_points.total
8) if range == 0 OR range == 1
9)          return
10)    else if range%2
11)            middle = (range-1) / 2
12)else
13)            middle = range / 2
14)endif
15)
16)
17)if axis == 0
18)            axis_points = st_points.x[0]
19)    else
20)            axis_points = st_points.y[0]
21)    endif
22)
23)    //Find the intercept and update that in the line information
24)    if middle != 0
25)         point1 = axis_points[ st_points.begin + middle ]
26)         point2 = axis_points[ st_points.begin + (middle + 1)]
27)         length = point2 - point1
28)         half_length = length/2
29)         middle_point = point1 + half_length
30)


31)         if axis == 0
```

```
32)               UPDATE_LINES_INFO(st_lines.x[0])
33)         else
34)               UPDATE_LINES_INFO(st_lines.y[0])
35)
36)          endif
37)
38)          //Recurse on both sides of the middle point
39)          if range != 2
40)                //From left to middle
41)                st_points.total = st_points.begin + middle
42)                FIND_AXIS_PARALLEL_LINES(0, st_points, st_lines)
43)
44)                //From middle to end
45)                st_points.begin = st_points.begin + middle
46)                FIND_AXIS_PARALLEL_LINES(0, st_points, st_lines)
47)          endif
48)     endif
49)
50)/*****************************************************/

CHECK_LINE_CONNECTIONS(st_axis_line, st_points)

//OPERATION: Check for connection(edge) between two points
//INPUT: Structure "st_axis_line" having the given axis's parallel lines
//OUTPUT: Connection status: 1 = TRUE or 0 = FALSE


1)     axis = st_axis_line.axis
2)     intercept = st_axis_line.intercept_point
3)     right_point = GET_NEAREST_RIGHT_POINT(axis, intercept, st_points)
4)
5)     if axis == 0
6)               ptr_points = st_points.x[0]
7)     else
8)               ptr_points = st_points.y[0]
9)     endif
10)
11)    i = 0
12)    j = 0
13)
14)    while i < (right_point+1)
15)           j = right_point+1
16)           while j < st_points.total
17)                if (st_points.coord[ptr_points[i]->index].connection[ptr_points[j]
                    ->index]) != NULL
18)                     return 1
19)                endif
20)                j = j + 1
21)             endwhile
22)
23)             i = i + 1
24)    endwhile
25)    return 0
26)
27)    /*****************************************************/

COMMIT_LINE(line, st_points, st_lines)

//OPERATION: Add the line into the solution and disconnect all the
                    connections which the line crosses
//INPUT: Structure "line" to be added into the solution and "st_points"
         having all the points
//OUTPUT: Structure "st_lines" with the added lines

1) axis = line.axis
2) intercept = line.intercept
3) right_point = GET_NEAREST_RIGHT_POINT(axis, intercept)
```

```
4)
5) //Store the lines that are added to the solution
6) n_lines = 0
7) st_lines.added_lines[n_lines] = line
8)
9) if axis == 0
10)            ptr_points = st_points.x[0]
11)else
12)            ptr_points = st_points.y[0]
13)endif
14)
15)i = 0
16)j = right_point
17)
18)while i <= right_point
19)            j = right_point + 1
20)            while j < st_points.total
21)                  DISCONNECT_POINTS(ptr_points[i]->index, ptr_points[j]->index)
22)                  j = j + 1
23)            endwhile
24)            i = i + 1
25)endwhile
26)
27)n_lines = n_lines + 1
28)st_lines.added_line_count = n_lines
29)
30)    /*****************************************************/
```

2. Run-time Analysis:

The implementation for SEPARATE_POINTS_GREEDY( ) is a standalone algorithm that reads co-ordinate values from multiple files, finds their axis parallel lines, separates the given co-ordinates with minimal number of axis parallel lines and stores the result into multiple files.

As given in the problem statement, the input instances would have atmost 100 points and we can have upto 100 instances, having various combinations of the points (sorted along x-axis). We have in **lines 12 to 40** a while-loop that cycles over the given number of instance files. So, at the maximum we have '**n**' time complexity for this loop.

Line 16 is a sorting operation using Quick-Sort, that, in worst-case, would reach **n^2**.

Line 18 is a function that does connecting the points of the graph. Its implementation, as given by the lines 61-74, contains two while-loops each for n-points. So, it would be **n^2** complexity for this operation.

Lines 21 and 22 finds the axis parallel lines of both the axes. The function, which is implement in the lines 85 to 133, is a recursive function, that runs in 'nlogn' time. For 2 axes it would be **2nlogn**.

Lines 24 to 36 has an inner while-loop that runs up to maximum number of the respective co-ordinate axes, which is '**n**' for the total n-points. Within it, it calls the function Commit(), that at maximum runs in **n^2** time. Hence the whole loop of this code section may take upto **n^3** time to run.

Overall, we have:

n(n^2 + n^2 + 2nlogn + n^3) = n(2n^2 + 2nlogn + n^3)
= 2n^3 + 2(n^2)logn + n^4

Omitting lower order terms, we get:

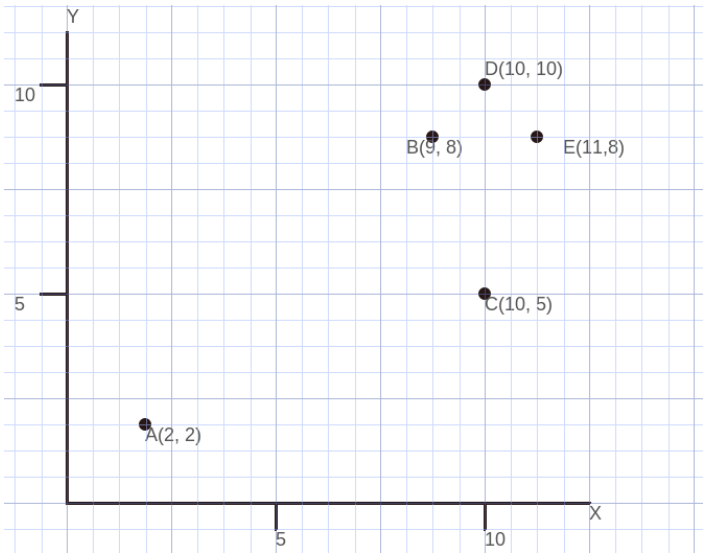
O(n^4), which is the complexity of the algorithm that separates n-points in a graph using greedy approach.

3. Failure Case of the greedy approach:

We have assumed in our algorithm that no 2 points share the same ‘x’ or ‘y co-ordinate axis. But, if we alter this assumption the algorithm won’t find an optimal solution, as the greedy approach fails for arbitrary inputs that have significant overlaps.

For example, consider an instance with 5 points in the graph:

A(2, 2), B(9, 8), C(10, 5), D(10, 10), E(11, 8)



We can see that there exists overlaps for the points B & E, and C & D for co-ordinate values y = 8 and x = 10 respectively.

The Greedy method gives the following solution (with 4 lines) :

v 10.0
h 8.0
v 9.5
h 6.5

While an optimum solution for the same would be:

v 10.5
v 9.5
h 6.5

Again, if we alter the same input and ensure that the points have no-overlap co-ordinates, we get an optimal solution. Let the altered points be:

A(2, 2), B(8, 8), C(9, 5), D(10, 10), E(11, 7)

For this input instance, we get:

v 9.5
h 7.5
v 8.5

- which is an optimum solution.

Thus, we can find that the Greedy approach would fail to return an optimum solution if we have inputs of the graph with overlaps.

References:

1) Greedy algorightm code: https://github.com/nickziv/sep_pts, Author: nickviz
2) Book: Field-Programmable Logic and Applications, by Reiner W. Hartenstein, Andres Keevallik. Page: 131, Section: 2.2, Successive bipartitioning.
2) http://alg12.wikischolars.columbia.edu/file/view/GREEDY.pdf/305349730/GREEDY.pdf