Report

LPA - Programming problem #1

Team - Passevite

Algorithm description

Optimize the positioning of vertices in $\mathbb{R}^2$ points in order to minimize the number of intersections between edges

```
1: procedure OPTIMIZE_INTERSECTIONS(vertex, lastVertexPosition)
2:     if vertex has no edges then                                  ▷ Recursive step if ¬ last vertex
3:         if vertex ¬ last one then return OPTIMIZE_INTERSECTIONS(vertex, lastVertexPosition)
4:         else
5:             bestSolution ← temporarySolution                     ▷ Base case, being the last vertex
6:             return 1 if a solution with no intersections was found, 0 otherwise
7:     for ∀ ℝ² points ∈ pointSet do
8:         if ℝ² point is free then
9:             ℝ² point ← Occupied
10:            for ∀ edges ∈ vertex do
11:                for ∀ edges ∈ (temporarySolution \ edges ∈ vertex) do.
12:                    if edges intersect then
13:                        Update counter for edge crossings
14:                        if edge crossing count≥ bestSolution then    ▷ Rejection Condition
15:                            Free the point being used and get the next one (goto line 7)
16:                Add vertex edge to temporarySolution
17:            Add vertex to temporarySolution
18:            if vertex ¬ last one then                            ▷ Recursive step if ¬ last vertex
19:                if OPTIMIZE_INTERSECTIONS(vertex, lastVertexPosition) = 1 then return 1
20:            else
21:                bestSolution ← temporarySolution                 ▷ Base case, being the last vertex
22:                if number of intersections in temporarySolution = 0 then return 1
```

In order to improve the performance of our approach, and to avoid the massive repetition of edges intersection calculations, before starting to check for the optimal placement of the vertices

on the $\mathbb{R}^2$ points given in the input, we've created an `array` to store information about all possible intersections between 2 segments made by 4 vertices which are set in any of the $\mathbb{R}^2$ points available . Another trick to optimize our solution was to store the information on the vertex with the higher ID, while reading the input information regarding the edges (colliders). This way, when we're assigning a vertex to a given $\mathbb{R}^2$ point, we guarantee that all the other vertices needed to complete their edges were already previously set.

Intersection procedure

Our approach to detecting the intersection of colliders is based on the concept of the orientation of an ordered triplet of points in the plane. Consider that A and B are the two colliders we are detecting an intersection for, and that A.1 is the source device for A, and A.2 is the target device for A (the same goes for B). The principle of this approach is that two colliders intersect if and only if one of the following conditions is verified:

- The orientations of the triplet (A.1, A.2, B.1) and the triplet (A.1, A.2, B.2) are different;
- The orientations of the triplet (B.1, B.2, A.1) and the triplet (B.1, B.2, A.2) are different;
- The two segments are collinear, and either their x-projections or y-projections intersect;

The orientation of a triplet of points can be clockwise, counter clockwise or collinear. To determine the orientation of a triplet of ordered points (A, B, C), we consider the segments AB and BC, then, we compute the slopes of these two segments, and compare them:

- If $\text{slope}_{AB} < \text{slope}_{BC}$, then the triplet (A, B, C) is counter clockwise orientated
- If $\text{slope}_{AB} > \text{slope}_{BC}$, then the triplet (A, B, C) is clockwise orientated
- If $\text{slope}_{AB} = \text{slope}_{BC}$, then the triplet (A, B, C) is collinear

Data structures

For storing information about a point in the plane, we have the `struct` $point_2d$, that has fields for the coordinates x and y `int`, and also a `boolean` $occupied$, which describes if any device is occupying that specific point in the plane at each moment. The `struct` $edge$ represents one collider, and has fields a and b, which are integers referring to the ID of the source and target devices for that collider. The `struct` $vertex$ refers to one device in the problem, and saves an `array` of the IDs of all its neighbours (this is, the other devices that are connected to this device by a collider) - field $edge$ (as by the assignment we knew that at most there could be 28 edges, this `array` is allocated with a size

of 28 `ints`); the number of neighbours of this vertex ($edge\_count$); and the boolean $seen$, where being "seen" means that this vertex is either the start or the end of an edge. The `struct` $solution$ is where we save the vertices location, the edges and the number of collisions found. This structure is employed several times throughout the program. Each time we finish placing all the edges on the plane, and meet a solution where the number of intersections is lower than the best one found until that moment, we copy the current solution to a variable $best_solution$. The variable $temp_solution$ is used for "building" a solution, that is, for registering which device is placed on which point of the room, and the number of intersections for that arrangement. The global variables of the problem are an array of 81 `structs` $point_2d$, $points$, an array of 81 `structs` $vertex$, $vertices$, an `array` for storing the pre-processed intersections, $intersect$, the two `structs` $solution$, $best_solution$ and $temp_solution$, and some auxiliary variables, such as the number of points in the plane, the number of devices to be placed and the number of colliders.

Correctness

Our algorithm returns the lowest number of intersections between edges of a given graph $G = \{V, E\}$, after positioning all its vertices on $\mathbb{R}^2$ points of a given set $P = \{(x1, y1), (x2, y2), ..., (xi, yi)\}$.
Base case:

- Having all vertices with edges assigned a point $p \in P$
- Having an edges intersection count bigger that the best one we have so far $EI' > EI$

Inductive hypotesis: Assume we are putting vertex i, and currently number of intersections is EI (before we assign our vertex), the number of intersections after positioning the vertex is EI' and and the best solution so far has bEI. By **contradiction**, we assume that after positioning vertex i there is some other solution $eBEI$ that is better than EI'. In this case we wold have that for $eBEI < EI'$, then, $EI' < EI$. Then we have a **contradiction** of our assumption, wich stated that befor putting the vertex we had EI intersections, and the condition $EI \leq EI'$ is always true.

Algorithm Analysis

Regarding memory complexity, we globally allocate all the data structures needed to support our algorithm to the higher bounds of the assignment and in total we have $\approx$ 41,2`MB`. Since all the support information is already stored, the recursive function only allocates a total of 4 variables and use 2 input parameters to the function, hence we can say that our approach has a memory complexity of $\mathcal{O}(n)$.

In terms of time complexity, our algorithm follows a design pattern of subtract and conquer, which means that in each recursive call will have a subproblem of size `n - 1`. Having such design, we can

calculate its time complexity using the Muster Theorem[3]. Let $T(n)$ be a function defined on positive values of n, and having the property:

$$T(n) \leq \begin{cases} c & \text{if } n \leq 1, \\ aT(n-b) + f(n) & \text{if } n > 1, \end{cases}$$

for some constants c, $a > 0$, $b > 0$, $d \geq 0$, and function $f(n)$. If $f(n)$ is in $O(n^d)$, then:

$$T(n) \; is \; in \begin{cases} \mathcal{O}(n^d) & \text{if } a < 1, \\ \mathcal{O}(n^{d+1}) & \text{if } a = 1, \\ \mathcal{O}(n^d a^{n/b}) & \text{if } a > 1, \end{cases}$$

In our case, the number of subproblems is $|P|$, so $a = |P|$, since in each recursion we make one new call to the recursive function per available point to test, and $b = 1$, as our problem decreases by a constant factor of 1 in each subsequent call, thus, we can denote our approach as being defined by

$$T(n) \leq \begin{cases} c & \text{if } n \leq 1, \\ PT(n-1) + f(n) & \text{if } n > 1, \end{cases}$$

and $f(n) = P\lfloor \frac{E^2}{4} \rfloor \implies \mathcal{O}(1)$. As shown above, we can then see that the time complexity of our algorithm is $\mathcal{O}(P^n)$.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest, Clifford Stein (2001) *Introduction to Algorithms*, The MIT press, 2nd Edition

[2] Jeff Erickson (2018) *Algorithms*, 0th Edition (pre-publication draft)

[3] David Saunders - saunders@udel.edu - *CISC320 Algorithms - Recurrence Relations Master Theorem and Muster Theorem* - `https://www.eecis.udel.edu/~saunders/notes/recurrence-relations.pdf`

Team members

Name	Student ID	Signature
João Soares	2009113061	
José Ferreira	2014192844	
Madalena Santos	2016226726	