

Local crossing minimization  
using simulated annealing

# Introduciton

- Motivation
- Algorithm overview
- Notable technical details
- Challenges encountered
- Further improvement

# Motivation

- 1.As a junior programmer, we planned to learn from the last winning team in the contest who uses simulated annealing approach for the challenge.
- 2.Althought it's rather a black-box to me, the framwork sounds understandable.
- 3.We can't find some way better.

# Algorithm overview

- Workflow briefly:
  - 1. Preprocessing: Generate a good layout then snap to grid
  - 2. Run simulated annealing with the good layout, when the program is “stuck”, restart at most 3 times, in the last time, run until iteration complete.
  - 3. Post-processing: Verify for the validity and resolve if ever needed.
  - 4. Exit.
- The structure is rather standard and simple while I’ve encountered many challenges during implementation.

# Preprocessing

- 1. We first examine the planarity, if the graph is planar, we try to generate a planar embedding.
  - If everything goes well, we obtain a best solution and exit.
  - Possibly, a planar embedding requires greater width or height than what we have, then we resolve this by our legitimize method.
  - Then treat the graph as non-planar and continue.
- 2. Then generate a force directed layout
  - Spring layout by fruchterman-reingold
- 3. Validate and legitimize if need.

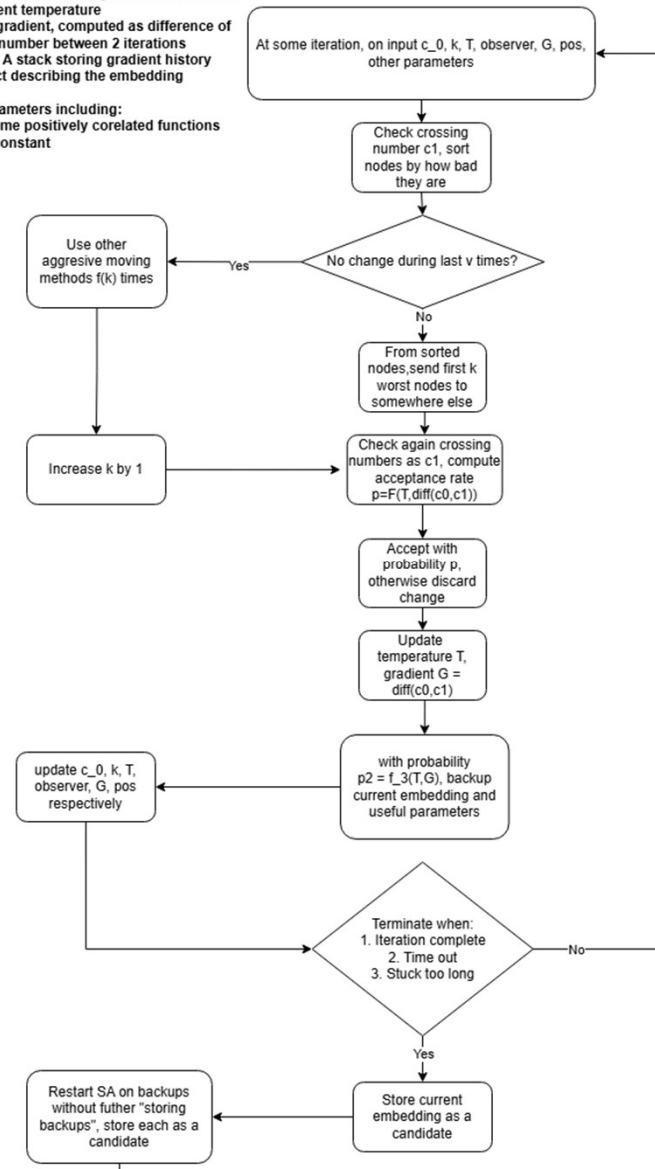
# Simulate annealing

- After preprocessing, we obtained a nice layout. Now we keep trying to improve it.
- The structure of this part is based on the “standard” SA with Temperature, cooling schedule, transition function.
- Briefly explain “standard” simulated annealing approach:
  - With input  $T$ , we check for the difference  $D = \text{Crossing last time} - \text{crossing this time}$ , calculate the acceptance probability  $P = f(T, D)$ , accept to update the drawing with the new one with probability  $p$ . Then reduce  $T$  by a cooling function  $T \rightarrow 0.9 * T$  and go to the next iteration.
- Parameters
  - In practice, we set the initial temperature at 5 and calculate the acceptance probability as  $\langle \text{picture} \rangle$ , cooling function as mentioned  $T = 0.9 * T$ .
- In our algorithm we also keep tracking on the changing rate of the crossing, doing backups here and there, to have better control over the program and collect more informations.

# General structure

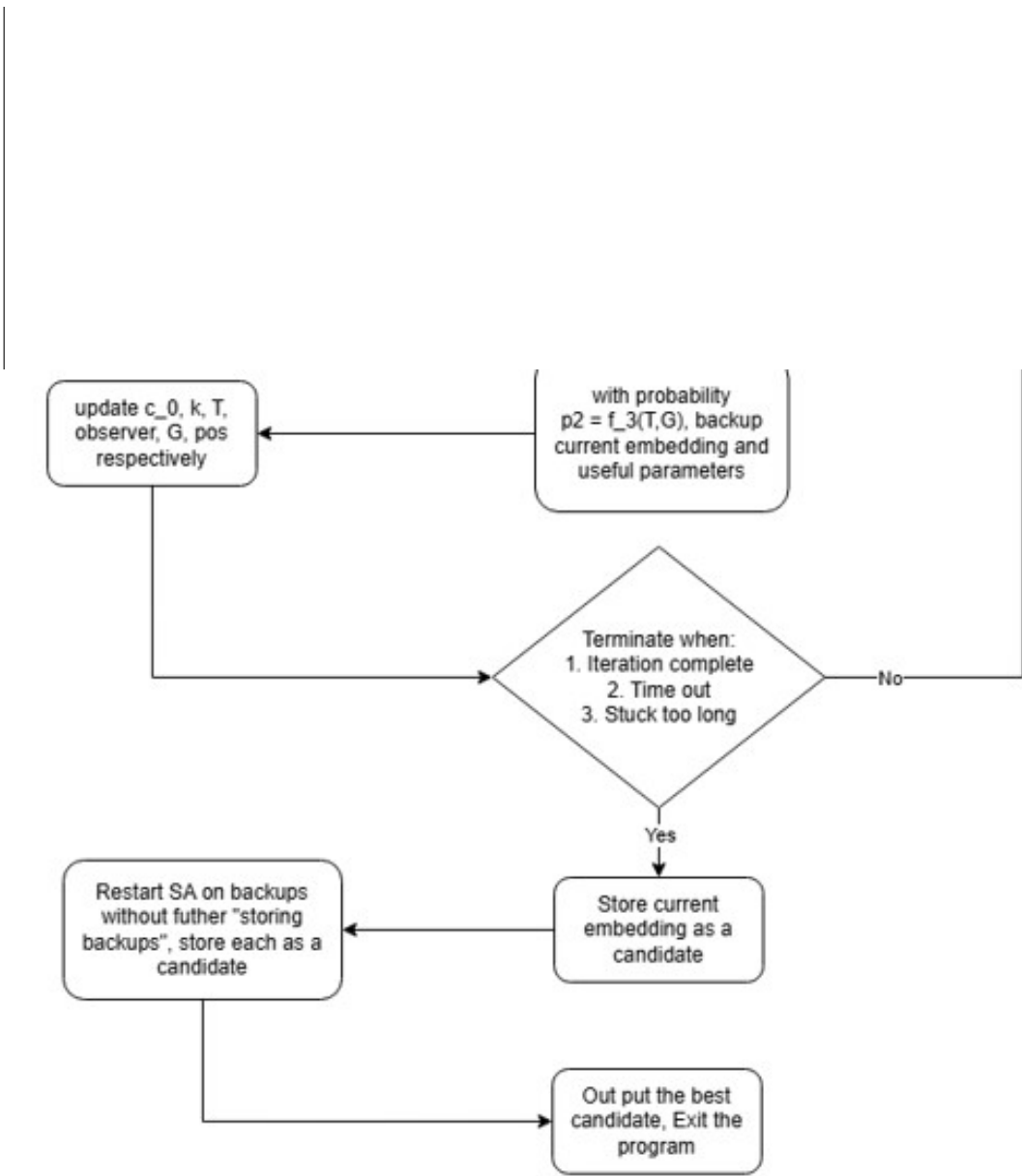
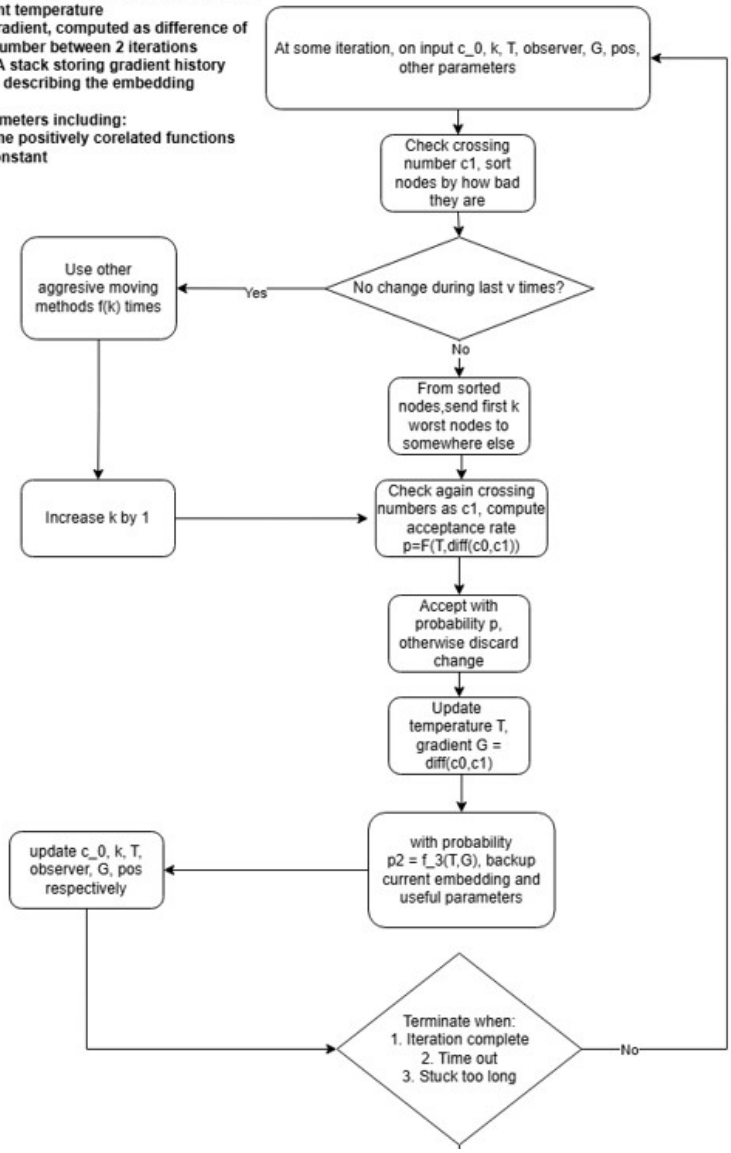
$c_0$ : Crossing number last time  
 $k$ : Number of how many nodes to be moved  
 $T$ : Current temperature  
 $G$ : The gradient, computed as difference of crossing number between 2 iterations  
 observer: A stack storing gradient history  
 pos: A dict describing the embedding

Other parameters including:  
 $F, f_n$ : some positively correlated functions  
 $v$ : some constant



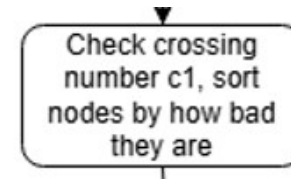
c\_0: Crossing number last time  
k: Number of how many nodes to be moved  
T: Current temperature  
G: The gradient, computed as difference of crossing number between 2 iterations  
observer: A stack storing gradient history  
pos: A dict describing the embedding

Other parameters including:  
F, f\_n : some positively correlated functions  
v: some constant





# Checking crossing numbers

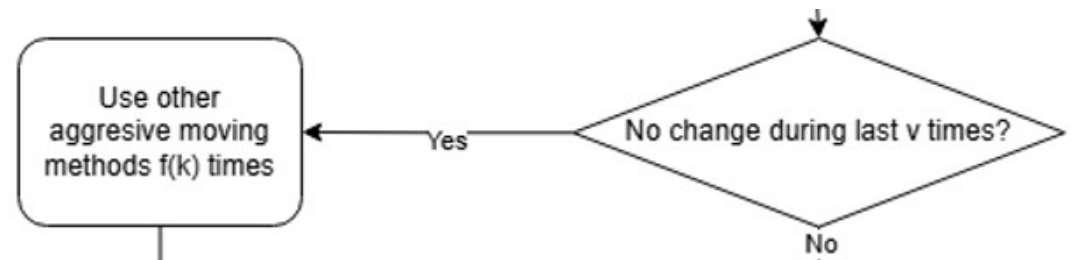


- As the first step and also the most frequently used step(also the most crucial part in terms of time complicity) in our program, it's worthwhile to improve it.
  - 1. First iterate over all combinations  $e_1, e_2$  of the edges, at each step, detect if they crossed each other (and the crossing type), if they do, store them in a dictionary  $CR := \{\text{edge} : \{\text{set of all the edges that crosses it}\}\}$ .
  - 2. When the iteration complete, the crossing number of each edge is naturally computed as the size of  $CR[\text{edge}]$ . And hence, the local crossing number is computed as  $\max(\text{size of } CR[\text{edge}])$ .
  - 3. In the next round of checking crossing numbers, instead of re-calculate the new CR, we make use of the old CR from the last iteration to improve the performance.

## Checking crossing numbers(2)

- Remark:  $CR := \{\text{edge} : \{\text{set of all the edges that crosses it}\}\}$
- Conceptually, we first remove all the “drawing of” edges of the lately moved node  $N$  as such: Let  $E(N)$  be all the edges of node  $N$ , clear  $CR[e]$  for all  $e \in E(N)$ . Then re-compute the new crossed edges for each pair of  $e_1 \in E(N)$  and  $e_2 \notin E(N)$ , store them in new  $CR$  and compute the crossing numbers as usual.
- We then “flatten” the  $CR$  to obtain a sorted list of nodes. We define the “worst level” of a node by this sorted list.
- We can also easily maintain a dictionary to store the punishment of each node during this phase, and re-sort the `worst_level` list by considering the punishment e.g. `worst_level += <input size>` if the node is right on some edge (This is more efficient than validate the movement in every iteration, and fits the contest). But I haven't finish this part yet.

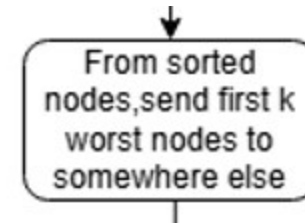
# Adjusting the pace



- . With the computed local crossing numbers, we can now observe the running status of our program. We achieve this by maintaining a stack called `gradient_observer`.
- We don't want to waste our computing resource, so when the program computed identical embedding  $v$  times, where  $v$  is a empirical constant (3 for now), we increase our "step size" to make more aggressive moves or even directly use some weird moves such as swapping or force-directly liked small moves or the "orthogonal rotation".(These might not make too many sense but they are at least computationally low cost.)
- This is learned from Henry's paper. Although I forget most of them now.

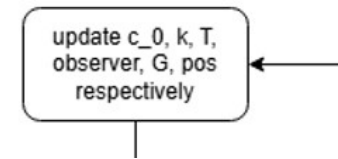
```
gradient_observer = []
```

# Move some nodes



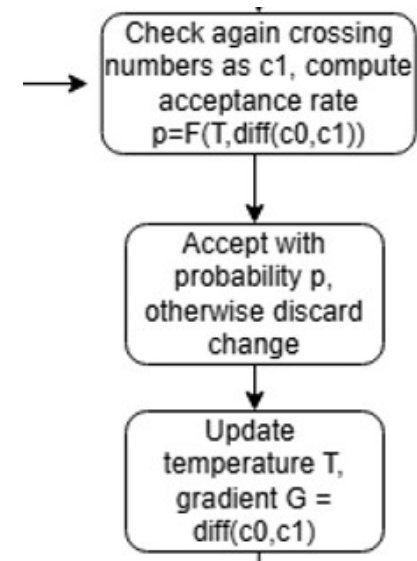
- Unlike the other group who also use SA, we literally just randomly send the nodes to somewhere else.
- To make things valid or good, we avoid the move any node on any other node, this is simply done.
- But for the “on-edge” cases, we either resolve by including punishment to our heuristic function or leave it there and solve them every  $n \in \{\text{big number}\}$  iterations. Since this really takes a while.
- By step size we mean how many nodes we move each iteration, of course, we can always break or backup whenever finding a better result.

Also don't forget to update these..

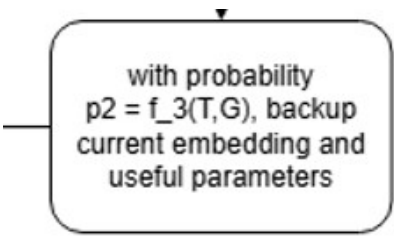


# Accept it or not?

- Typical simulate annealing routine.
- The acceptance probability is computed as  $\text{accp\_p} = e^{\frac{-\text{energy\_diff}}{\text{temperature}}}$  where energy diff:= old\_crossing – new\_crossing.
- The cooling schedule is an exponential one:  
 $T = 0.9 * T$
- When trying to discard, we remember to discard the change of CR as well.



# Backup and restart

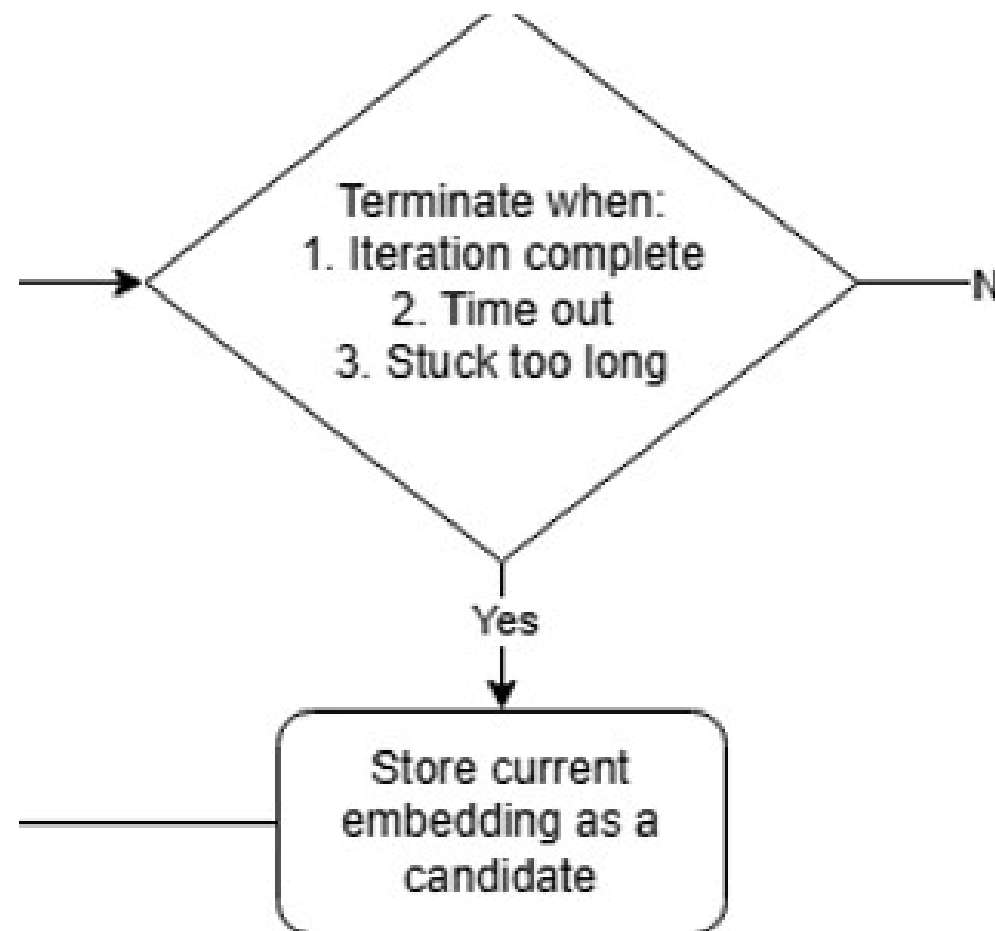


with probability  
 $p_2 = f_3(T, G)$ , backup  
current embedding and  
useful parameters

- As professor Kobourov mentioned, during the SA we might want to restart the whole process if the program sticks, instead of only looking for the good parameters and estimation function.
- We achieve this as such: during some iteration, compute a probability  $p_2$  by a function that positively correlated to temperature  $T$  and the gradient  $G$ . With a limit size as an experimental constant.
- Then we have the chance to restart the program at some point later.

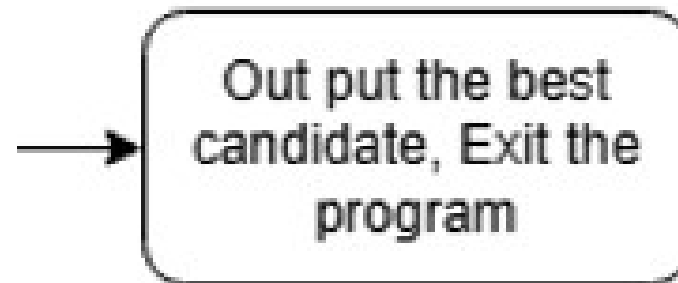
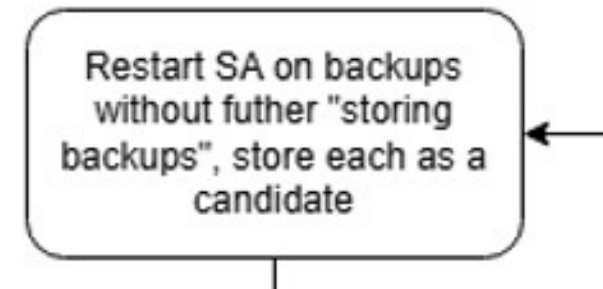
# Termination

- Remark that in the last turn among the re-starts we don't need to terminate for 3. "stuck too long" since we don't have any other thing to try.



## Backup and restart(2)

- As described in the diagram, restart the process (including a force-directed improvement try), then store all those results for the final comparison.
- Final comparison: simply get the one with less local crossing count
- Then exit the program



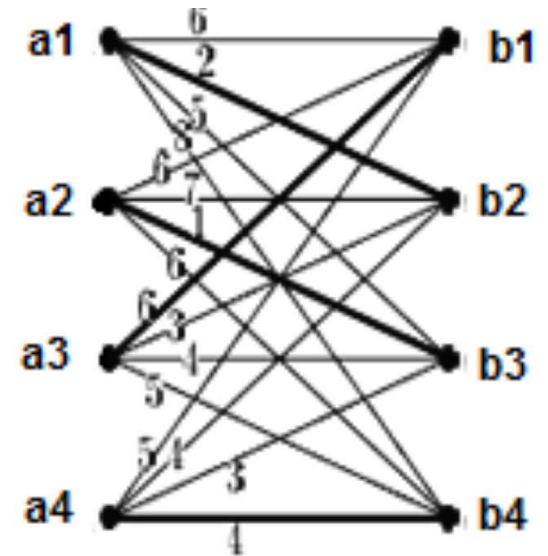


## Other notable details

- One philosophy of heuristic algorithm might be making good use of all “fetchable informations”. During the process of checking for local crossing numbers we can also easily compute the total count.
- Hence, we pass this total count and whenever two embedding has the same local crossing count, we accept the one with better total count.(However, this might make no sense, but it might worth to try things like this)
- They might be some other easily fetchable characteristics to look for.

## Other notable details(2)

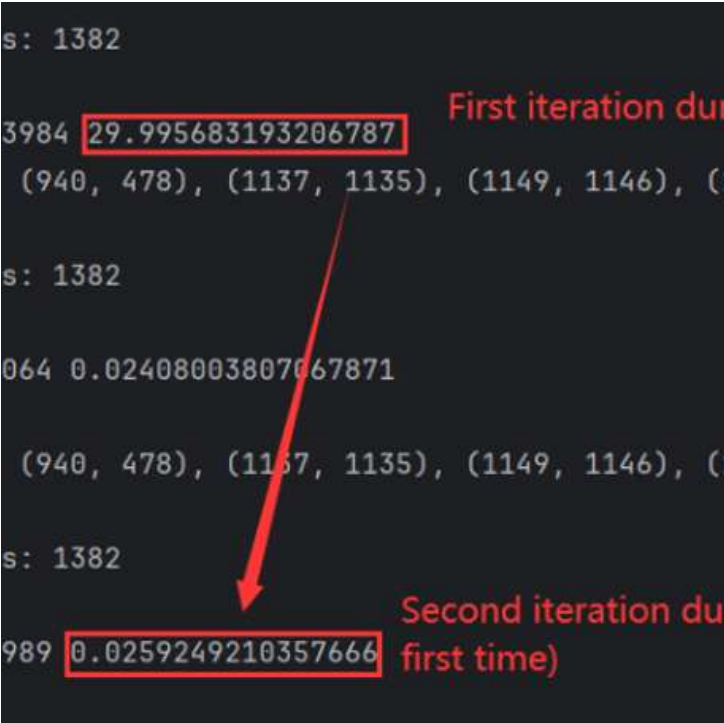
- For snapping/ legitimization we have 2 approaches.(For on-edge crossing and out of scope positions)
- One is the maximum(weighted) matching suggested by my supervisors. But the trivial implementation of it is not efficient for the big inputs. We need to adjust it in the future.
- Another simpler variant is randomly (with random offsets to make things more random) insert the invalid nodes to somewhere close to it. This is a little better but also doesn't work for the big input (such as final 10).
- Remark: This step might results worst local crossing number.



# Challenges encountered

- The biggest challenge I've meet is to find out the way to check crossings faster.
- By taking the advice of my supervisors, the runtime of this part is improved to be 1500 times faster on a very big graph.

Beforehand my approach was just iterate edges 2 times to find out one crossing.



```
s: 1382
3984 29.995683193206787
(940, 478), (1137, 1135), (1149, 1146), (
s: 1382
064 0.024080038070667871
(940, 478), (1137, 1135), (1149, 1146), (
s: 1382
989 0.0259249210357666
```

First iteration du

Second iteration du  
first time)

# Futher improvements

- 1.Improve the performance of critical functions (including legimize/snapping)
- 2.Refactor the structure to collect more informations without increase time complicity.
- 3.Tuning/debug, do more hyperparameter test on more datasets.
- 4.Maybe do some simple pattern distinction to solve special graphs.(assume there are something useful for some certain graphs)

Thanks for your attention.