Outline page

- Section1
- Section2
- Section3
- ..

Simulated annealing with magical tuning

- Some figure

Motivations

- text

Overall structure

- Some figure

Important components

Notable details

- Punishment avoiding:
  - 1.Iterate over the nodes again, collect whatever placed on an edge or another node by the adapted_is_intersected() function.
  - 2.Walk in expanding circles, examine in each step if it's still placed on somewhere not wanted, if not, fix it in the place.
  - 3.Do 2 iteratively for each node collected.

Also SA can resolve this itself if we change some code.

Supportive functions

One interesting approach

- As Mr. Luttenburger proposed at the beginning of the class(although I didn't really get his approach), doing some kind of separations first might help us understand the local structure better. At least it sounds nice for the graphs that are too big to understand by a glance.
- Here I introduce my approach: Separation-> locally improve->reconnection. I know this is not very imformative without futher motivations, lets dive into some details.

Overall structure:

- Some figure

Separation: Bcc-decomposition

- Reason:
  - 1.Famous, Available in the library I use (networkx).
  - 3.Fast, in either max[O(E), O(V)]. (by Tarjan et al.), and you don't need to frequently run it.
  - 4.Make better use of planar embedding algorithm: Connecting a single vertex or adding a single edge can easily break a nice planar graph. (e.g. making it 1-planar) If we don't have mechanism to easily examine and produce 1-planar/<small number>-planar embedding, we kind of wasted the power of planar embedding algorithm.
  - Remark: Decomposition ignores some edges i.e. not bcc part, we need to store them then add them back later.

Separation: Hide the rest

- After obtaining bccs, there might be still line segments overlaps some bcc while not belonging to any of its edge. We simply ignore them and do nothing.
- Alternatively, we can move them aside, but we don’t focus on this now.

Local improvement: planar embedding try

- Reason:
  - 1. Straightforward: As discussed, planar embedding is suitable for this “planarity minimization” task since is has the term “planar” in it’s name. It only take care of crossing numbers which might make it more suitable than some other approaches e.g. force directed algorithm since they care about other things.
  - 2.Famous, In the lib.

Local improvement(2): Force directed

- Reason:
  - 1.Clearly not every bcc is planar, we need a back up solution.
  - 2.Force directed algorithm also works OK for the smaller parts.
  - 3.Famous, available in the lib.
  - 4.Definitly better than nothing.

- Notice:
  - Can append some random search afterwards.

Reconnection: Enumerate automorphism

- Remark: (Loosely) Graph Automorphism >= line symmetry of the Graph or rotation symmetry or point Symmetry of the graph.
- Reason:
  - Reconnecting edges might produce more crossings than before(Luckly we can always have some back up, but we hope to have something new).
  - Modifying drawing (here: the planar component) in a symmetry fashion won't change the planarity.
  - It can be used to swap positions, we don't map/swap any node, instead we only map/swap the position.(e.g. rename the nodes n->n', copy position as pos', then pos[n]=pos[auto(n')])
- Problem:
  - No algorithm found in the lib
  - ∈NP(?)

Reconnection(2): Orthogonal Rotation

- Reason:
  - 1.Very easy to implement.
  - 2.Fast.
  - 3.Enumerates some symmetry drawing.

As a sub-function:

- So far this approach sounds like a competitor to per-processing algorithms, I don't think it can beat the reinmann-gold there.
- Hence, we try to plug it in the simulated annealing approach as a variant of the local improvement methods/transition functions.(i.e. replace random search or something force directed).

As a sub-function:

- A simple conversion is such: Compute the bcc first, during the SA we keep tracking the small-scaled(in terms of euclidean distance) bccs, whenever the program is stuck, we look for those small-scaled bccs and cut every node inside the surrounding rectangle(or some neighborhoods?) as a sub-graph for each bcc, then we run the algorithm on these bccs. This tracking thing might be a bit difficult maintain, but won't last long.

Futher improvements

- 1.Improve the performance of critical functions
- 2.Refactor the structure to collect more informations without increase time complicity.
- 3.Tuning/debug
- 4.Do some simple pattern distinction to solve special graphs.

Thanks for the attention