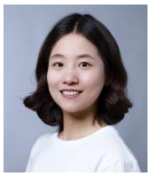
Our Solution for the Minimum Feedback Challenge

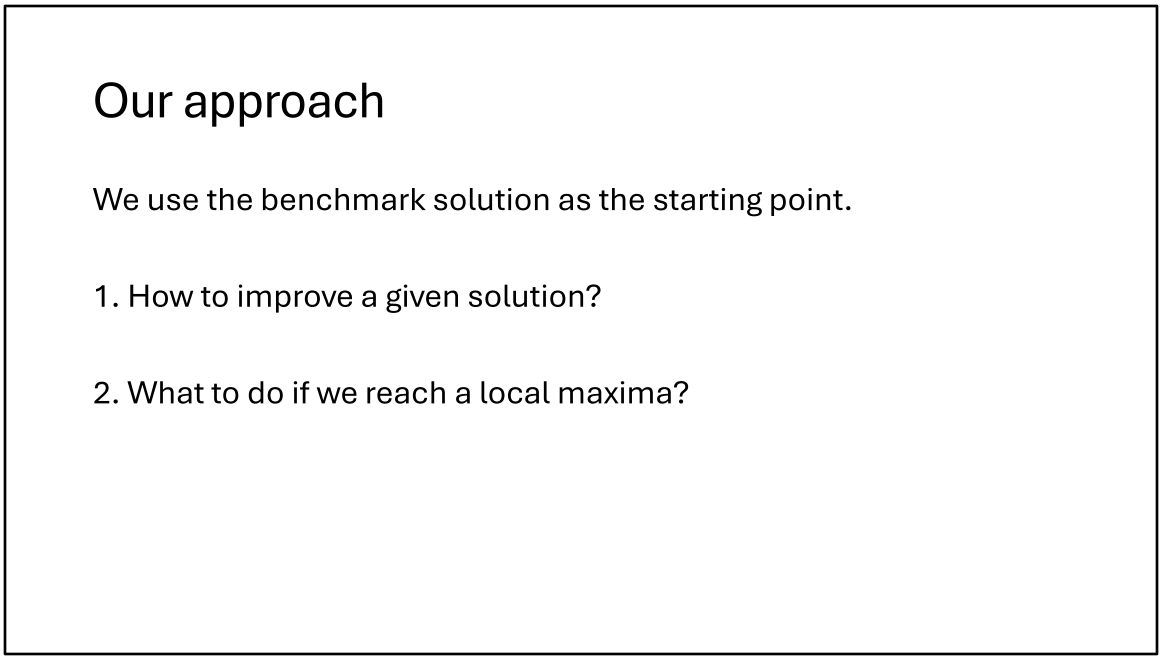
Tatsuo Okubo

with Runxuan Tang and Xin Zheng



https://github.com/CIBR-Okubo-Lab/min_feedback_final

Hello everyone! I am Tatsuo Okubo from Chinese Institute for Brain Research, Beijing, and I will present our solution for the challenge. This work is done together with two members in my group, Runxuan Tang and Xin Zheng. All the codes are available at this GitHub repo, if you would like to know more in detail.



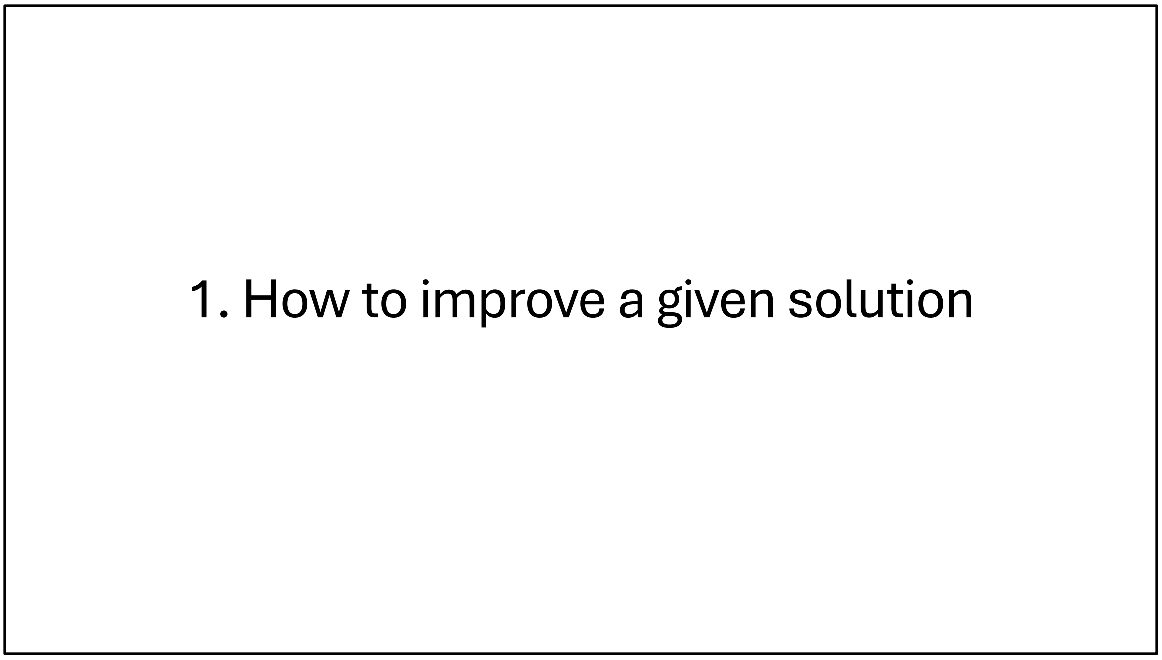
Our approach

We use the benchmark solution as the starting point.

1. How to improve a given solution?

2. What to do if we reach a local maxima?

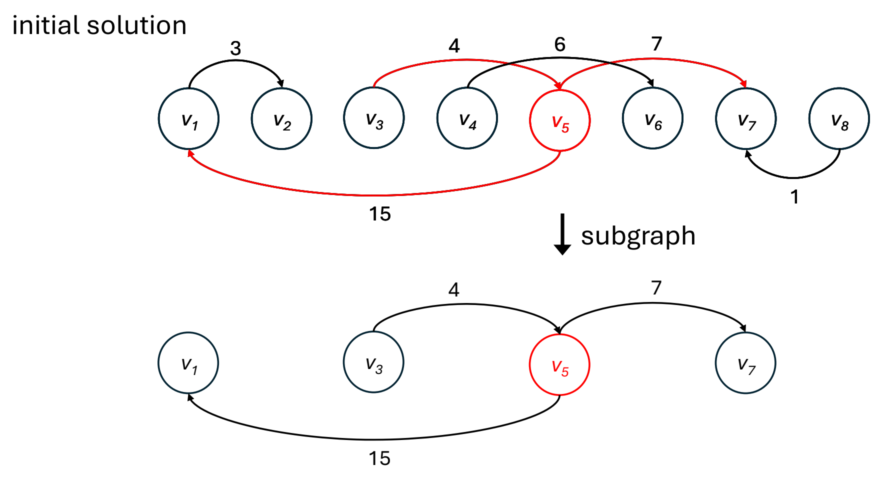
Our approach was to take a benchmark solution and keep on making small changes to improve it.
The two key questions are (1) how to improve given a solution and (2) what to do if we reach a local maxima, and I will explain these two in turn.



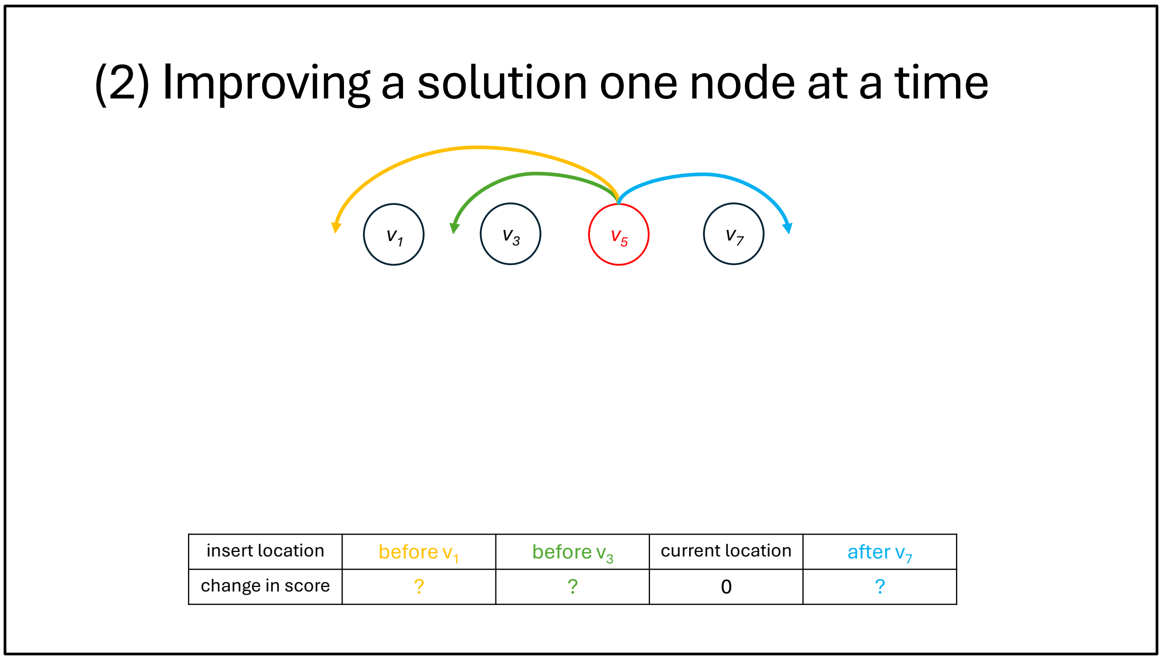
1. How to improve a given solution

So the first section is on how to impove a given solultion.

Improving a solution one node at a time



To improve a given initial solution, we randomly chose a single node, in this example node v_5, and asked "where can we put this node v_5 to improve the score?"
To answer that question, we actually don't need to examine every possible location in the entire solution, but instead only need to consider the insertion locations with respect to nodes that are connected to v_5. In other words, we created a subgraph of node v_5 by extracting all the nodes that are directly connected to v_5. The connectome is sparse, so this makes the search space much smaller.

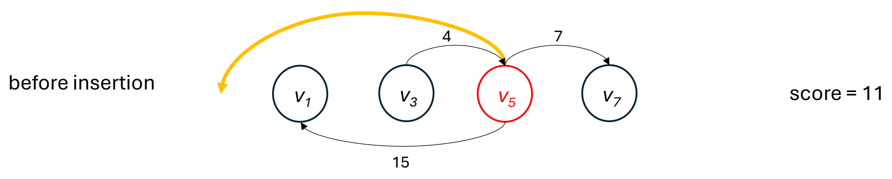


(2) Improving a solution one node at a time

insert location	before v_1	before v_3	current location	after v_7
change in score	?	?	0	?

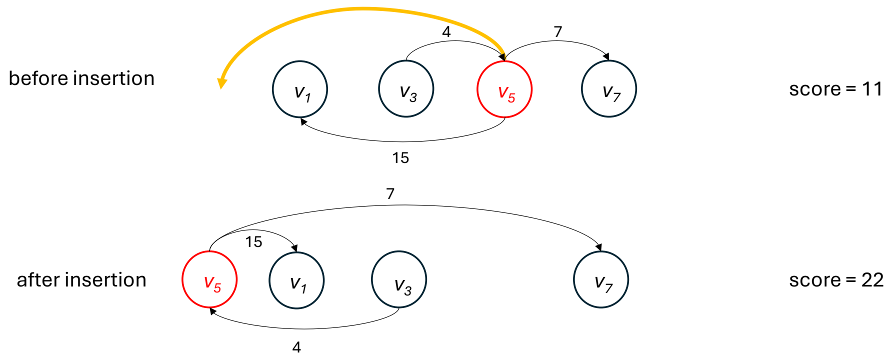
Then, we want to test all the potential insertion location of v_5 within this subgraph and calculate the change in the score, and basically fill this table. Note that when we test each insertion location, we don't need to calculate the full score using all the nodes, we only need to calculate the change in the score, which is much faster to calculate.

(2) Improving a solution one node at a time



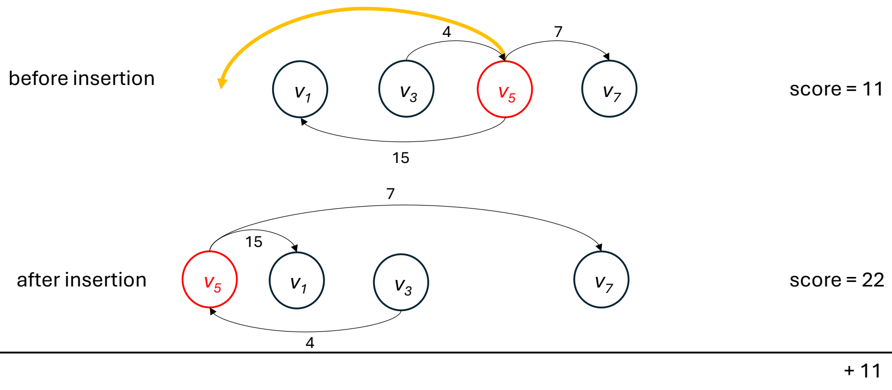
Just to give you one example, if this is the current solution, the score of this subgraph is 4+7=11.

(2) Improving a solution one node at a time

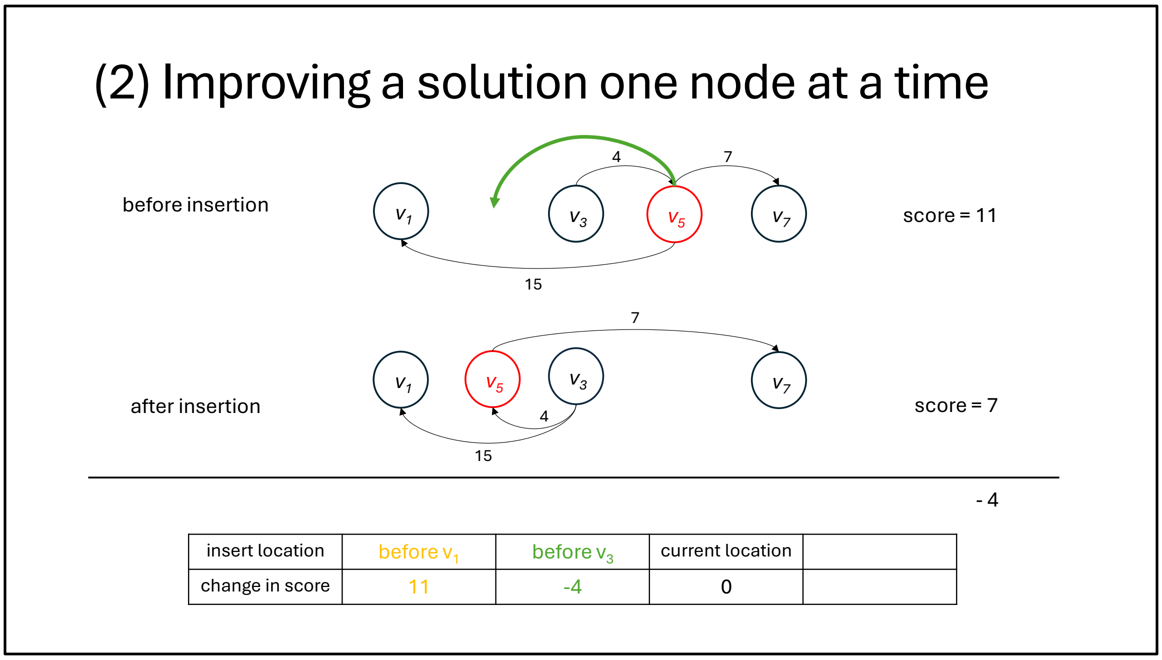


If we insert v_5 at the very beginning, the score will be 15+7=22, so increase in the score is +11.

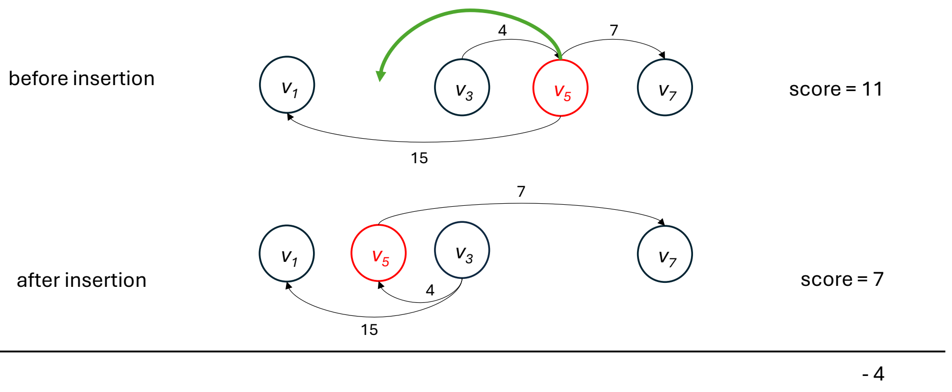
(2) Improving a solution one node at a time



insert location	before v_1		current location	
change in score	11		0	

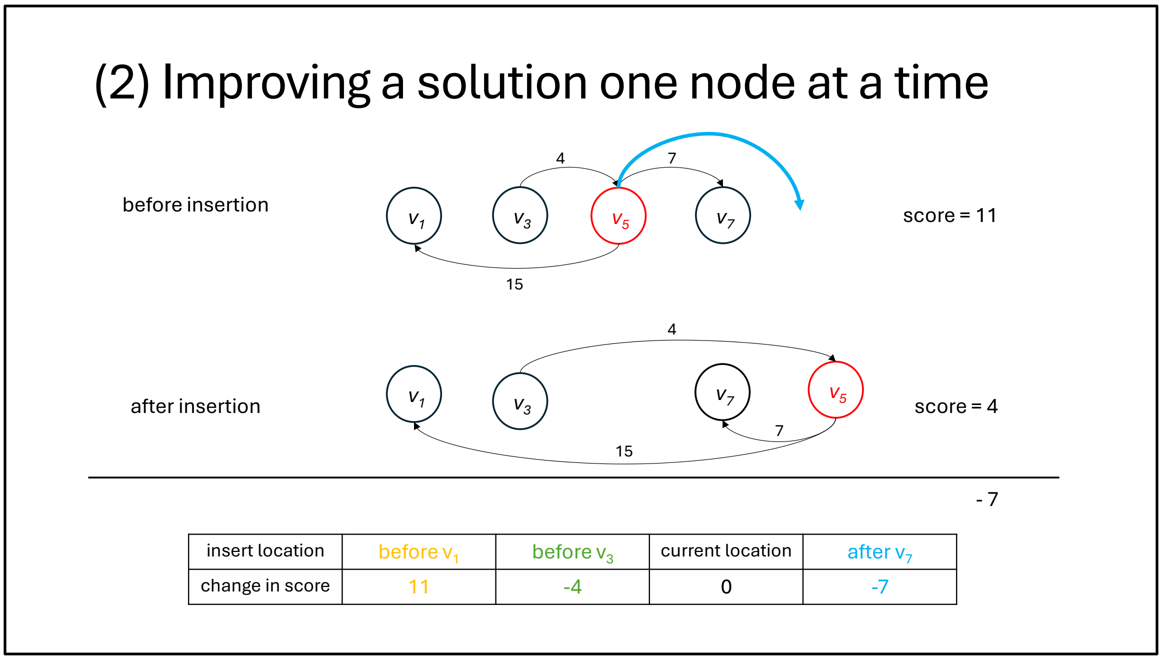


(2) Improving a solution one node at a time

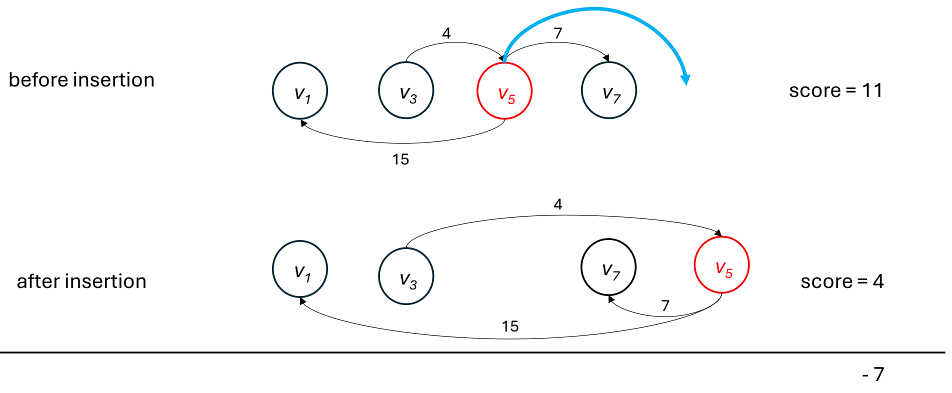


insert location	before v_1	before v_3	current location	
change in score	11	-4	0	

We do this calculation for different possible insertion locations.

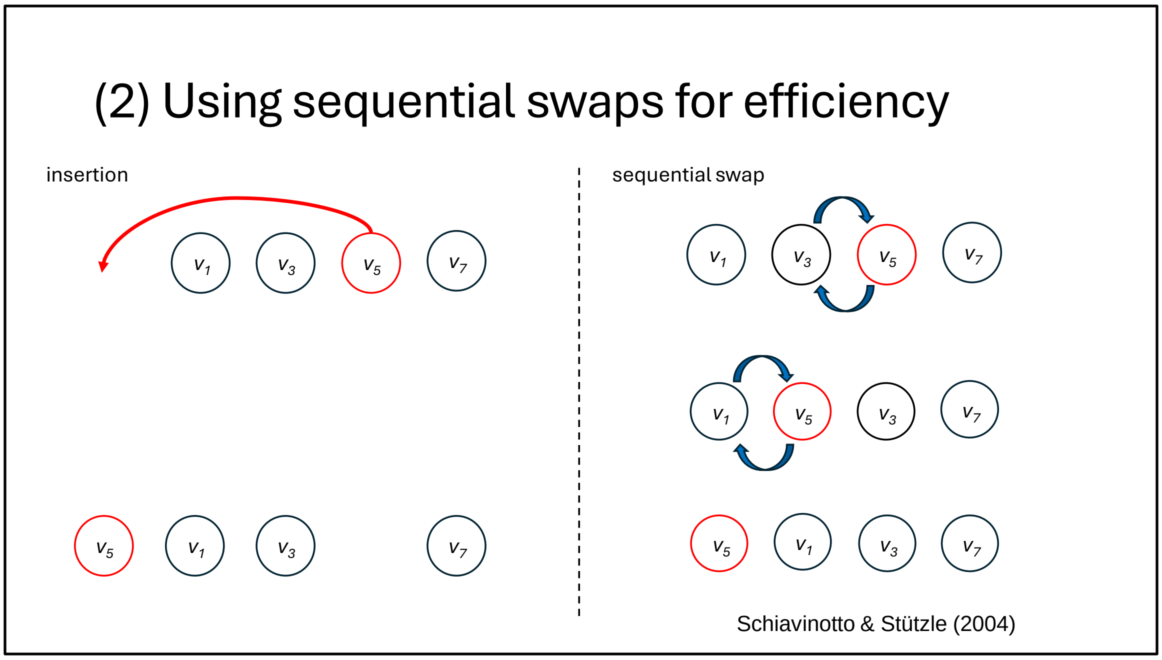


(2) Improving a solution one node at a time

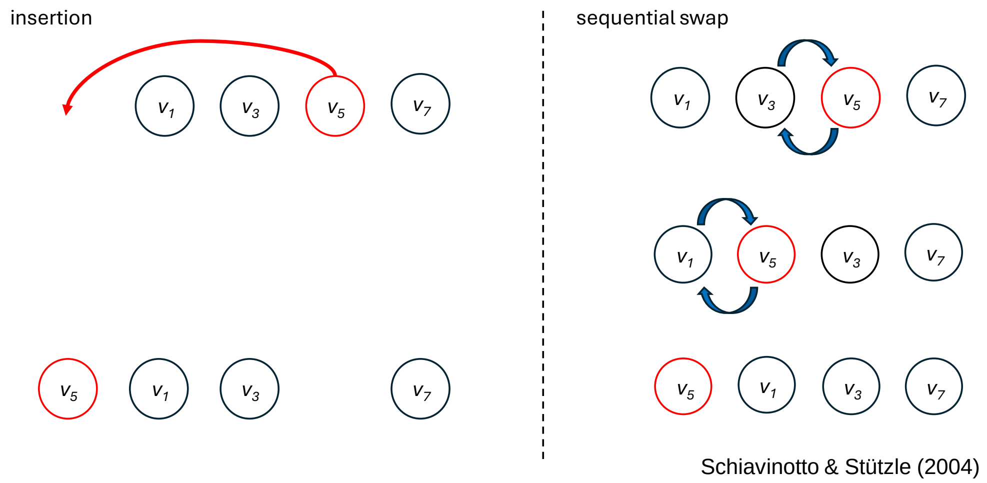


insert location	before v_1	before v_3	current location	after v_7
change in score	11	-4	0	-7

Once we fill the table, we know that inserting v_5 before v_1 is best, so we actually implment that insertion and then put these four nodes back in the original solution.



(2) Using sequential swaps for efficiency

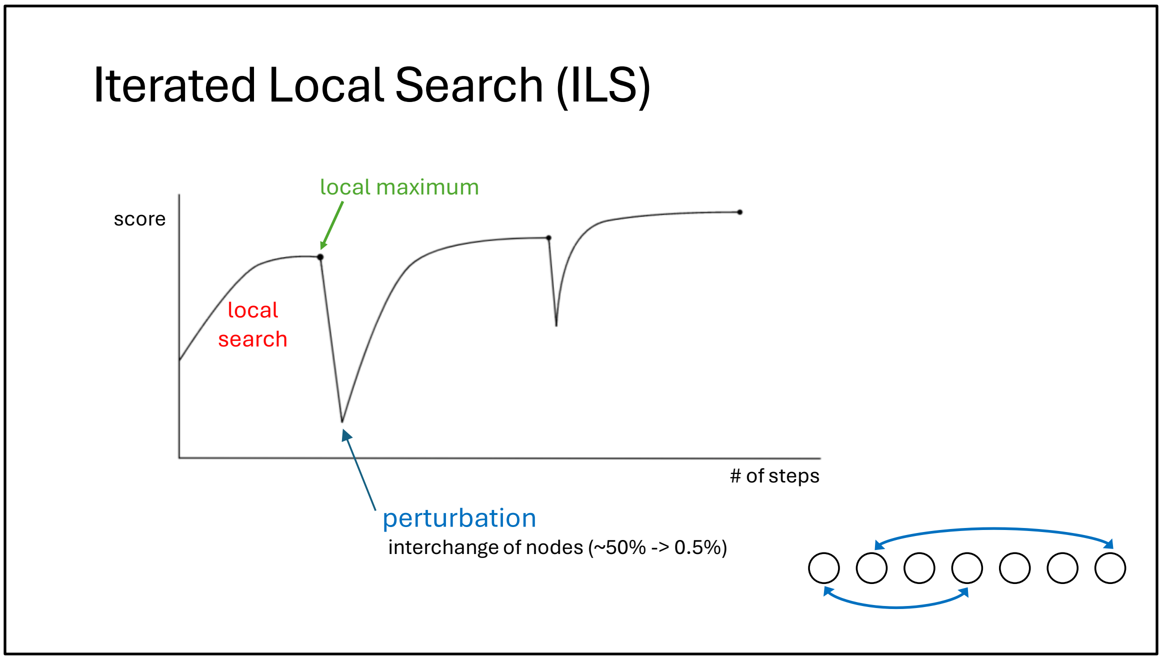


Schiavinotto & Stützle (2004)

We used one trick from this paper that improves the efficiency when filling out the table I showed you. Instead of trying all possible insertion location from the very beginning to the end, we performed sequential swaps. This is because any insertion can be expressed as a sequence of swaps, and we need these intermediate steps anyway, this allowed us to fill the table more efficiently.

This is how we improve the score for one node. We then went through all the nodes in several rounds until the score stopped increasing. However, this is local search, so it could easily get stuck in a local maximum.

2. How to get out of local maximum

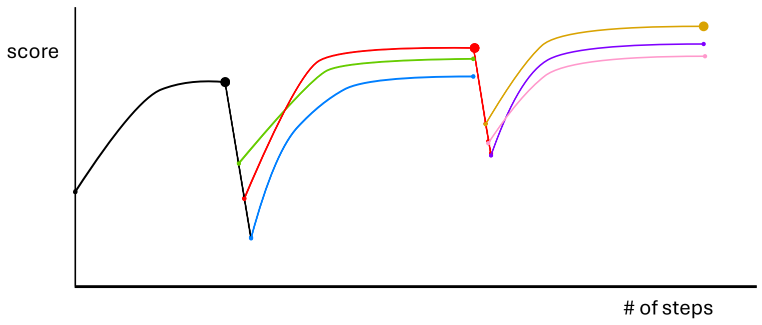


Iterated Local Search (ILS)

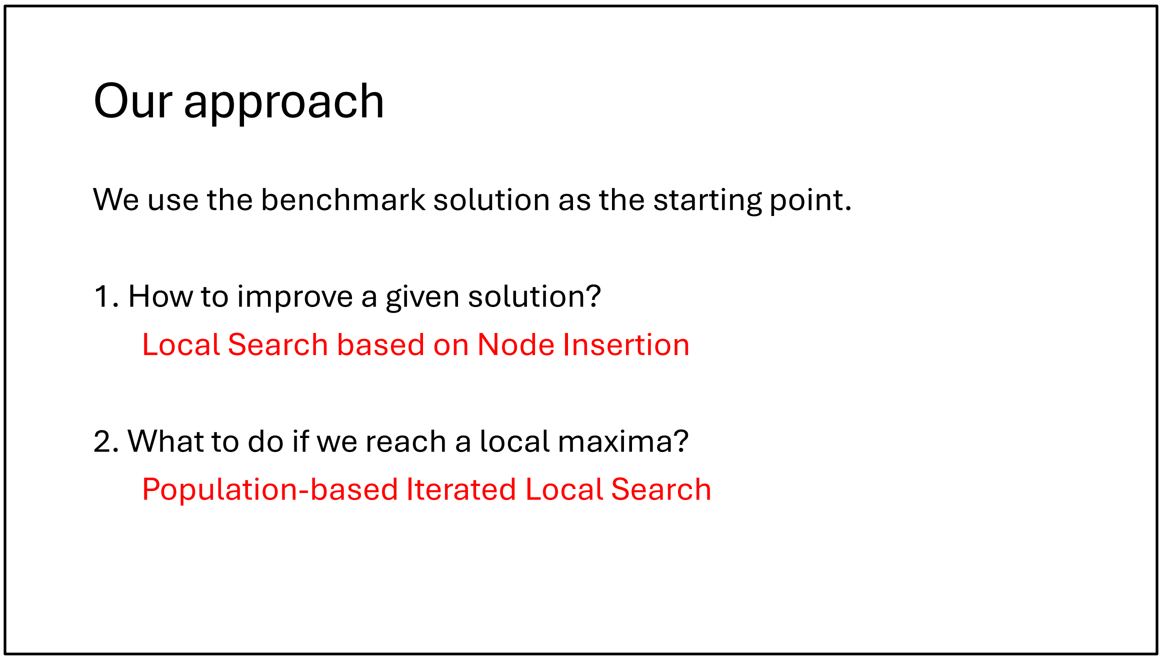
Next, I would like to explain how we tried to avoid getting stuck at local maxima.

Once we reached a local maxima after local search based on node insertion, we choose randompair of nodes and interchanged their order. The fraction of nodes that were interchanged was an important parameter we changed as the optimization progressed, but in general, we started from a large fraction of about a half in the beginning to a small number, like 0.5%, towards the end.

Population-based Iterated Local Search



In reality, we modified this itereated local search to introduce population-based approach. That is, once we reach a local maximum, we pertub this solution by different amounts of perturbtions and each of them performs local search. At the end, we pick a single solution that had the maximum score, and pertubed that solution in different ways and started a local searchs. We typically ran 15-20 parallel local searchs.



Our approach

We use the benchmark solution as the starting point.

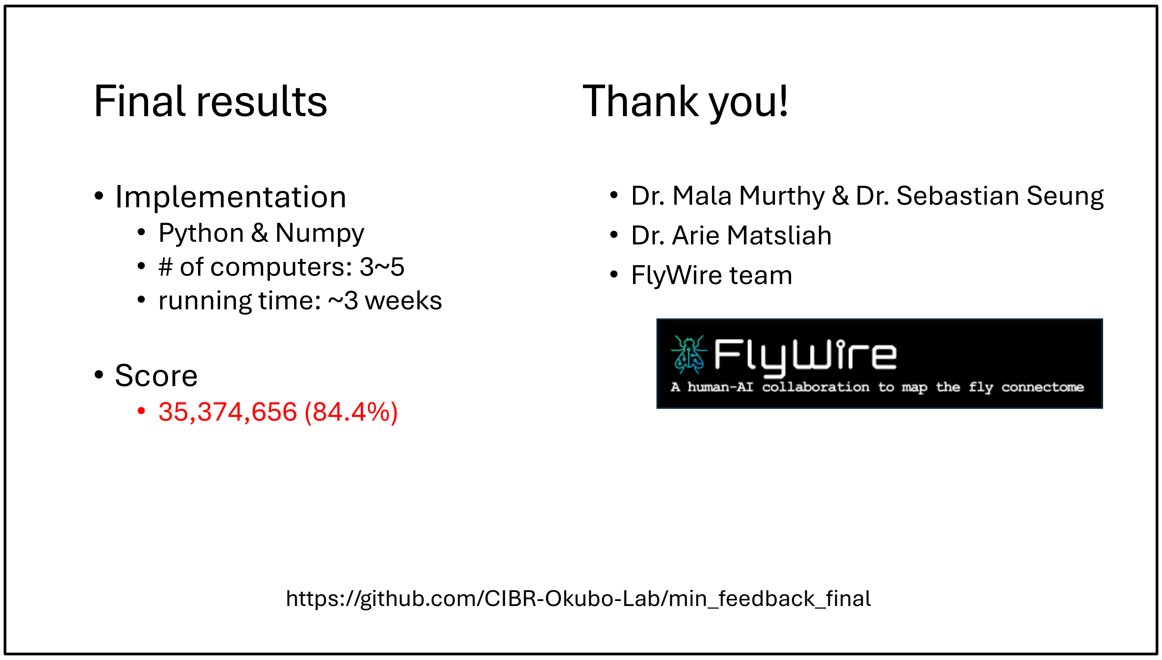
1. How to improve a given solution?

   Local Search based on Node Insertion

2. What to do if we reach a local maxima?

   Population-based Iterated Local Search

To summarize our approach, we used local search based on node insertion to improve on the given benchmark solution. Once we reached a local maxima, we used population-based iterated local search.



Final results

- Implementation
  - Python & Numpy
  - # of computers: 3~5
  - running time: ~3 weeks
- Score
  - 35,374,656 (84.4%)

Thank you!

- Dr. Mala Murthy & Dr. Sebastian Seung
- Dr. Arie Matsliah
- FlyWire team

FlyWire
A human-AI collaboration to map the fly connectome

https://github.com/CIBR-Okubo-Lab/min_feedback_final

We implemented this algorithm in python and numpy and ran it on 3-5 computers for about 3 weeks, and our final score was 84.4%, and once again, more detailed explanation can be found in this GitHub repo. In the end, I would like to thank Arie and the entire FlyWire for this opportunity!