Evolutionary Computation Lab VI

Piotr Kaszubski 148283

Monday, December 4, 2023

Contents

1 Problem description

We are given three columns of integers with a row for each node. The first two columns contain `x` and `y` coordinates of the node positions in a plane. The third column contains node costs.

1. Select exactly 50% of the nodes (if the number of nodes is odd we round the number of nodes to be selected up).
2. Form a Hamiltonian cycle (closed path) through this set of nodes such that the sum of the total length of the path plus the total cost of the selected nodes is minimized. The distances between nodes are calculated as Euclidean distances rounded mathematically to integer values.

The distance matrix should be calculated just after reading an instance and then only the distance matrix (no nodes coordinates) should be accessed by optimization methods to allow instances defined only by distance matrices.

2 Pseudocode

2.1 Perturbation

I define a perturbation as applying **10** uniformly random (with replacement) swaps (can be both intra- and inter-swaps).

This is the most important part of my algorithm that commenses after the initial linear search (expressed in Python for ease of reading):

```
perturb_delta = -1;
anneal_temp = 600.0 * 1.2;
while perturb_delta < anneal_temp and clock() < deadline:
        score_before = evaluate(graph)
        perturb(graph);
        lsearch_steepest(graph);
        score_after = evaluate(graph)
        perturb_delta = score_after - score_before;
        anneal_temp /= 1.2;
```

Basically, I enter a loop where a perturbation is applied, linear search performed, and then if the delta is lower than some threshold, we continue. This threshold starts at **600.0** and is divided by **1.2** after every successful perturbation.

The parameters **10**, **600.0** and **1.2** were chosen experimentally.

2.2 Perturbation (simpler)

After presenting the idea of the perturbation with annealing on the labs, I found out that I slightly misunderstood the requirements, and, more importantly, I learned that a much simpler version of this algorithm works better:

```
while clock() < deadline:
        perturb(graph);
        lsearch_steepest(graph);
```

The report will generally focus on the simpler approach, as it was the one intended for this report, and the one that gives better results. However, I will still include the results of the more complicated variant. To distinguish between the two, the more complicated one will be referred to as "i$_a$ls-steepest-random" or "I$_a$LS" (subscript "a" for "annealing").

3 Results

ALG.	TSPA TSPB	TSPC TSPD
ls-steepest-random	77,866 (75,315–81,017) 71,322 (68,623–76,002)	51,453 (49,257–53,785) 48,234 (45,351–51,534)
lsc-steepest-random	89,127 (82,350–101,152) 84,469 (74,393–96,069)	62,892 (55,074–74,956) 60,203 (51,897–67,824)
lsd-steepest-random	77,866 (75,315–81,017) 71,322 (68,623–76,002)	51,453 (49,257–53,785) 48,234 (45,351–51,534)
lscd-steepest-random	89,127 (82,350–101,152) 84,469 (74,393–96,069)	62,892 (55,074–74,956) 60,203 (51,897–67,824)
msls-steepest-random	75,048 (74,178–75,644) 68,211 (67,828–68,622)	49,128 (48,311–49,620) 45,649 (45,075–46,032)
ils-steepest-random	74,255 (73,754–74,535) 67,389 (67,008–67,744)	48,312 (47,936–48,646) 44,616 (44,105–45,127)
i$_a$ls-steepest-random	74,893 (74,137–75,593) 68,070 (67,676–68,496)	48,969 (48,662–49,308) 45,505 (45,016–46,114)

Table 1: Average, minimum and maximum scores of found solutions

ALG.	TSPA TSPB	TSPC TSPD
ls-steepest-random	85.737 (71.796–105.250) 88.181 (73.757–107.447)	85.470 (72.226–98.981) 87.450 (67.483–102.099)
lsc-steepest-random	21.973 (17.070–28.766) 22.002 (17.069–29.329)	20.936 (15.504–30.904) 21.179 (15.651–28.530)
lsd-steepest-random	43.684 (33.639–66.647) 43.794 (35.071–54.709)	42.885 (33.038–57.912) 43.747 (30.598–74.864)
lscd-steepest-random	28.267 (18.670–39.157) 28.695 (19.072–38.583)	27.632 (18.752–39.882) 27.669 (19.578–40.003)

Table 2: Average, minimum, maximum running times per instance (ms)

ALG.	TSPA TSPB	TSPC TSPD
msls-steepest-random	16.339 (15.744–16.838) 16.606 (16.074–17.191)	16.064 (15.530–16.725) 16.282 (15.941–16.693)
ils-steepest-random	16.327 (16.323–16.337) 16.328 (16.323–16.337)	16.328 (16.324–16.334) 16.328 (16.323–16.338)
$i_a$ls-steepest-random	16.335 (16.323–16.387) 16.333 (16.324–16.380)	16.342 (16.323–16.395) 16.342 (16.323–16.403)

Table 3: Average, minimum, maximum running times for MSLS, ILS and I_aLS (s)

ALG.	TSPA TSPB	TSPC TSPD
ils-steepest-random	1,868.5 (1,805–1,920) 1,774.8 (1,599–1,857)	1,663.0 (1,601–1,835) 1,642.5 (1,535–1,829)
$i_a$ls-steepest-random	1,149.3 (1,091–1,209) 1,117.7 (1,056–1,178)	1,207.8 (1,139–1,249) 1,150.0 (1,067–1,194)

Table 4: Average, minimum, maximum number of times linear search was run

3.1 Best solutions

All best solutions are available in the repository, as always. I export them as `*.graph` files, which follow a very simple text encoding:

```
<no_vacant_nodes>;<no_active_nodes>
<x>;<y>;<cost>                          \
<x>;<y>;<cost>                           | vacant nodes
...                                     /
<x>;<y>;<cost>                          \
<x>;<y>;<cost>                           | active nodes
...                                     /
```

”Vacant” nodes are the nodes not used in the solution. ”active” nodes are the ones in the solution. Since all graphs consist of 200 nodes and all solutions are 50% active, to find the list of nodes in the solution, simply look at the last 100 lines of each `*.graph` file.

4 Visualizations

4.1 TSPA.csv

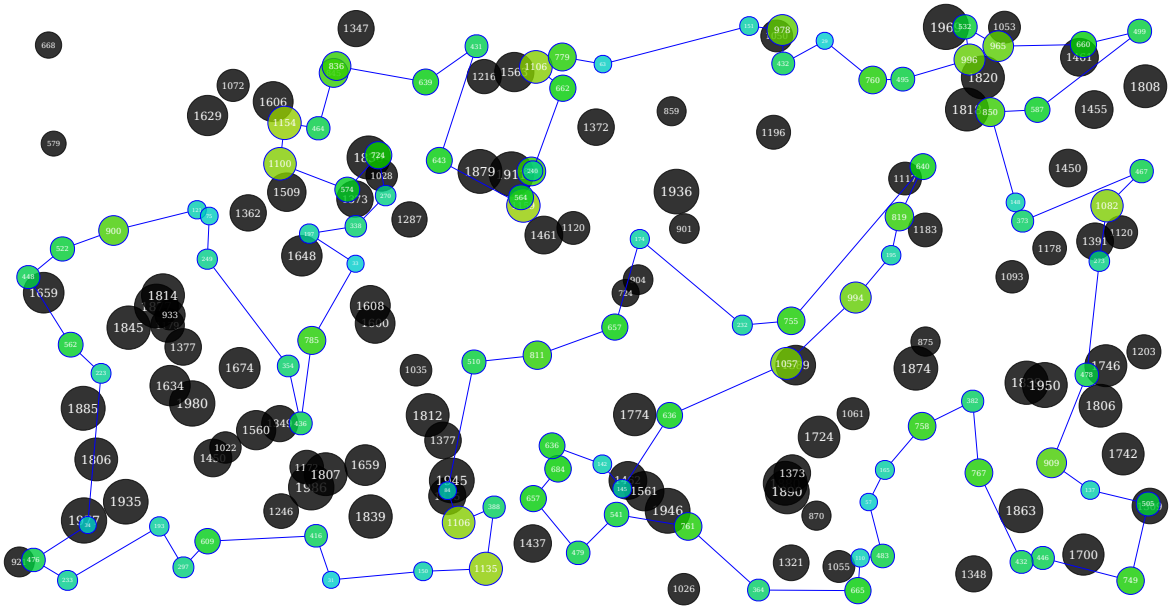

Figure 1: Best msls-steepest-random solution to TSPA (74,178)

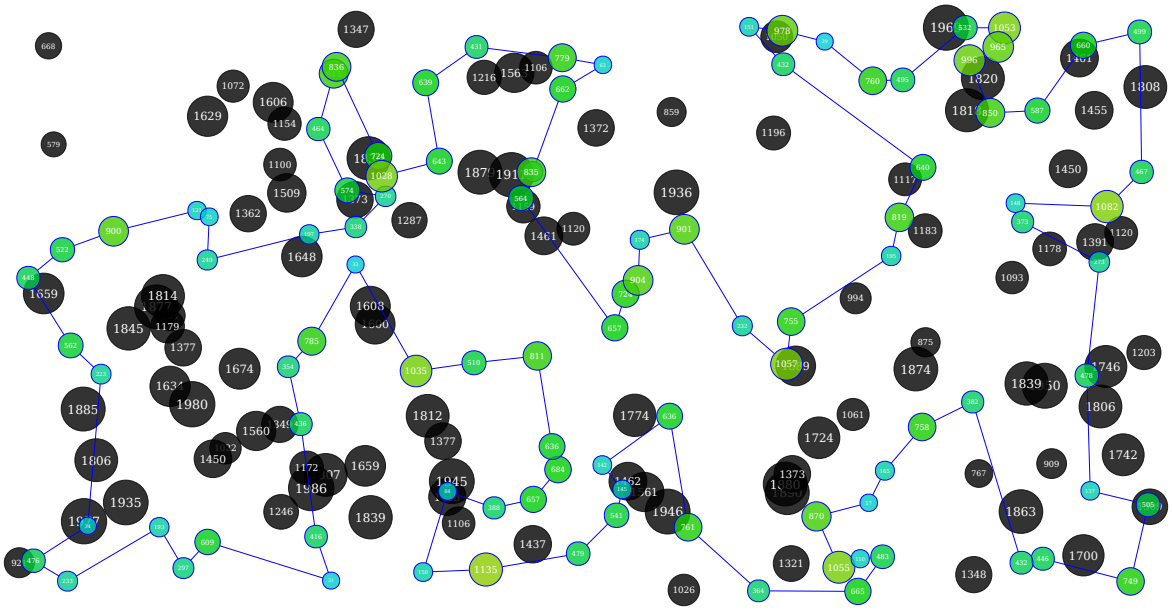

Figure 2: Best ils-steepest-random solution to TSPA (73,754)

4.2 TSPB.csv

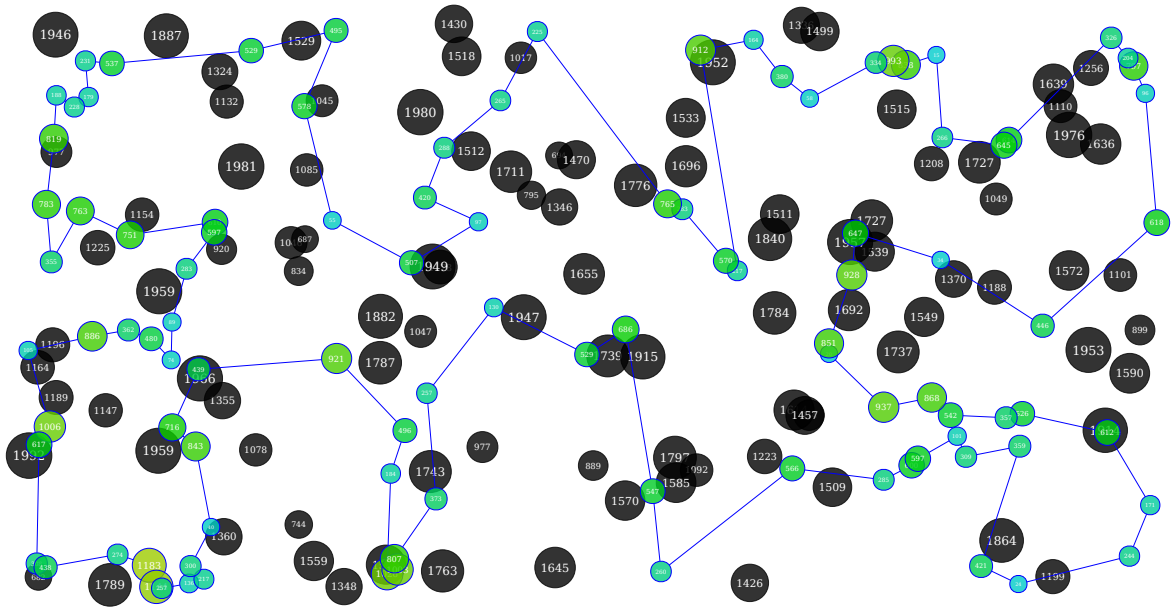

Figure 3: Best msls-steepest-random solution to TSPB (67.828)

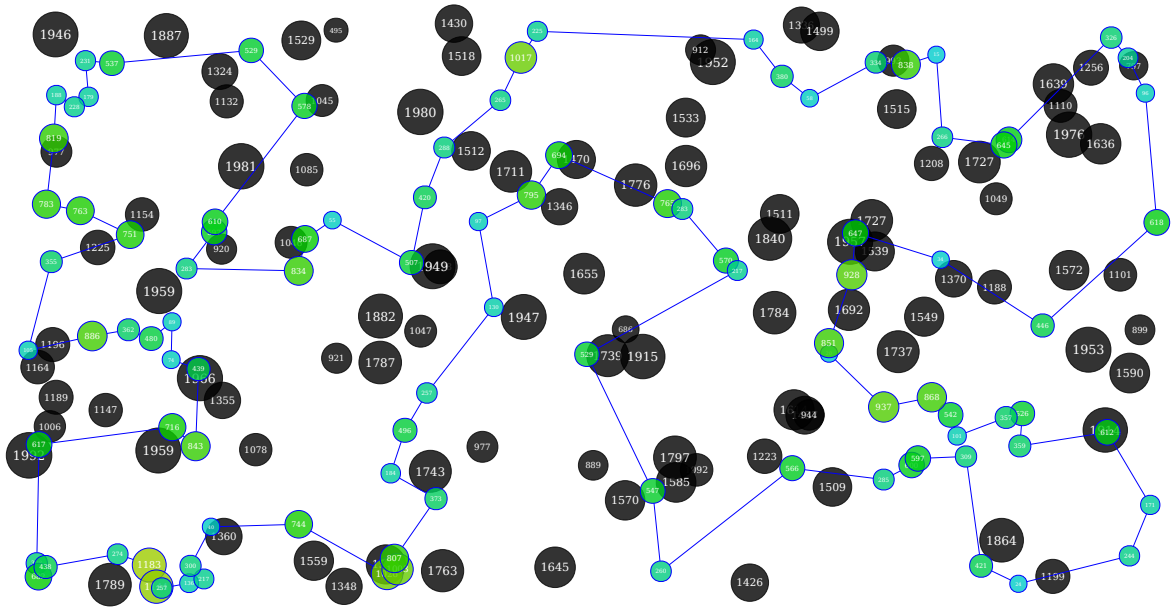

Figure 4: Best ils-steepest-random solution to TSPB (67,008)

4.3 TSPC.csv

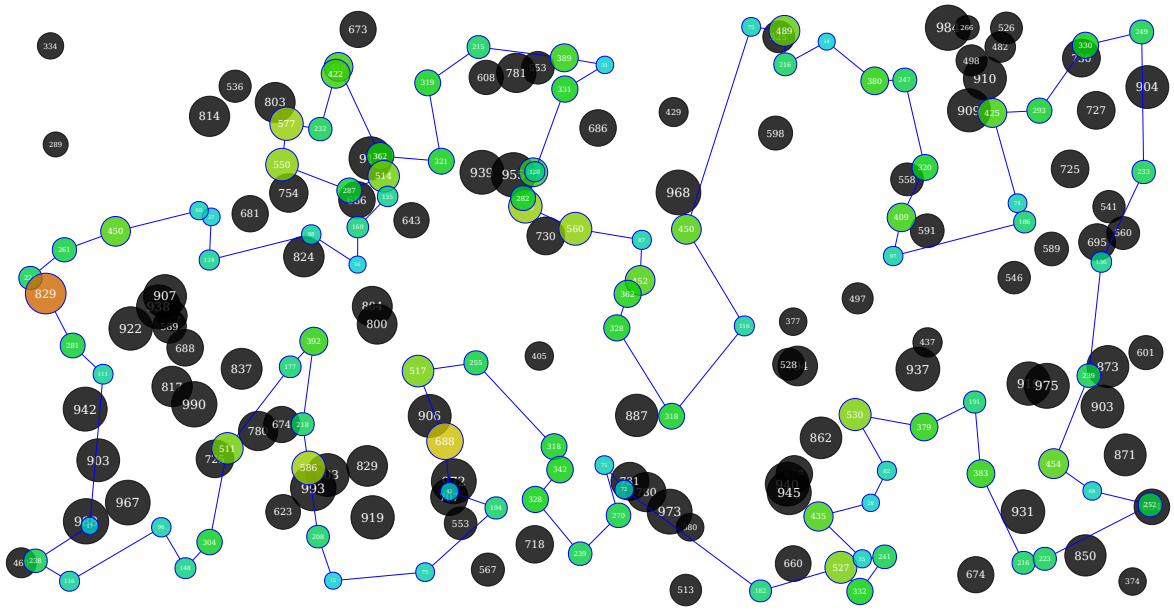

Figure 5: Best msls-steepest-random solution to TSPC (48,311)

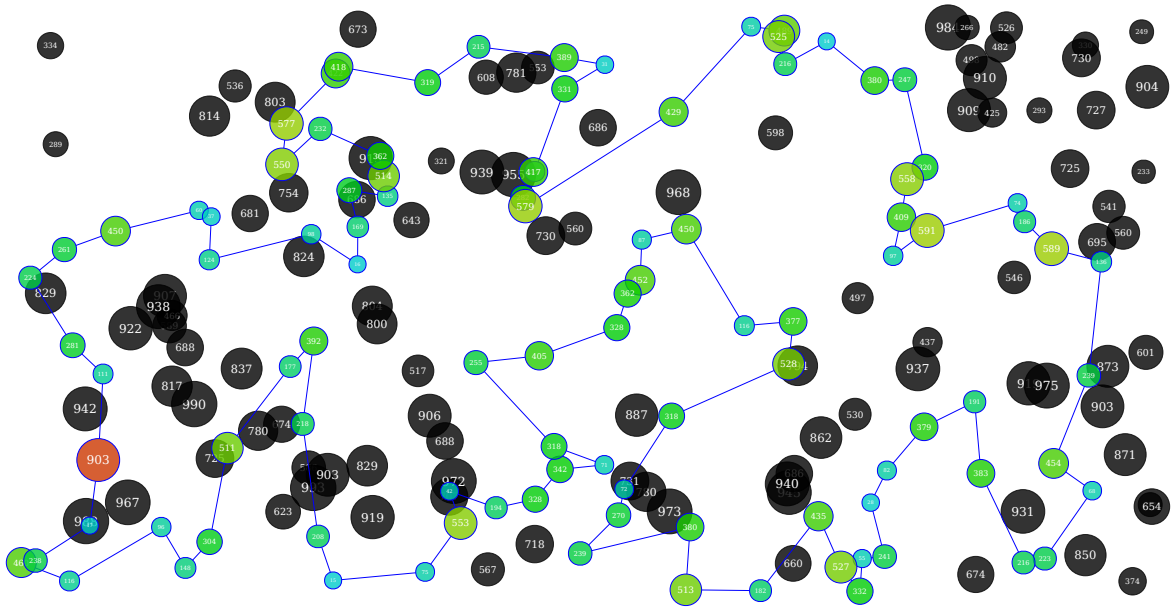

Figure 6: Best ils-steepest-random solution to TSPC (47,936)

4.4 TSPD.csv

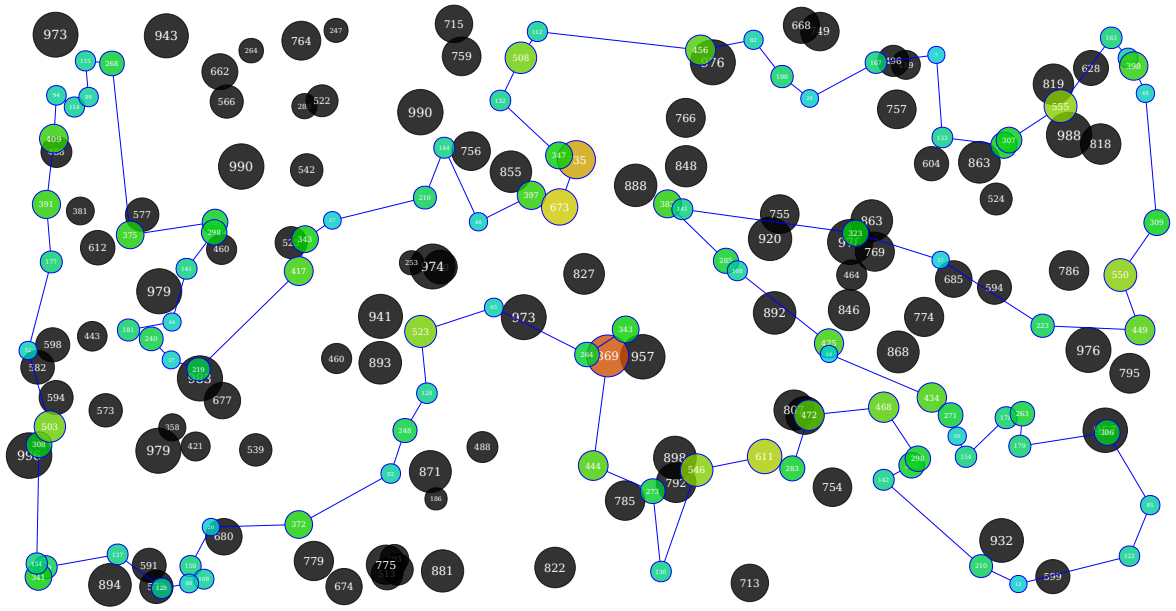

Figure 7: Best msls-steepest-random solution to TSPD (45,075)

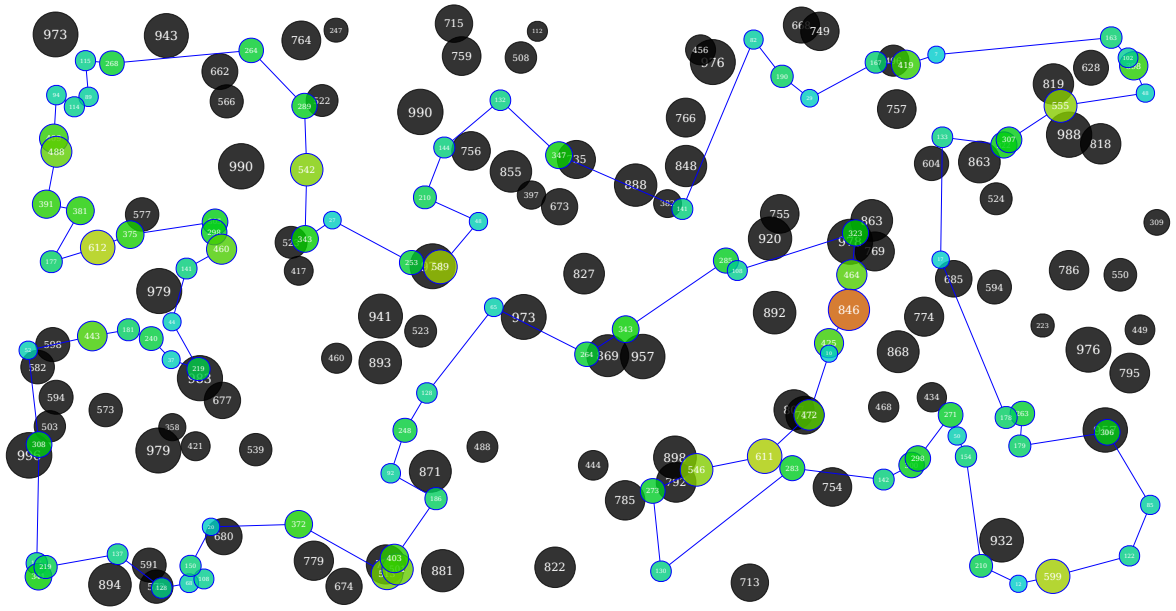

Figure 8: Best ils-steepest-random solution to TSPD (44,105)

5 Source code

The source code for all the experiments and this report is hosted on GitHub:
https://github.com/RoyalDonkey/put-ec-tasks

6 Conclusions

6.1 Original conclusions (before simpler perturbation)

As expected, iterated linear search with perturbations outperformed the more naïve multiple-start local search. The margin isn't great, but considering the flexibility of I_aLS (tweakable perturbation function and timeout), it is a significantly more versatile approach, while still being very simple in its concept.

Despite their promising results, both MSLS and I_aLS can't really be compared with the previously tested techniques, due to the many times longer execution time that was given to them. In the case of I_aLS, this could of course be tweaked, but capping it at 1 linear search's time would make it identical to just linear search. Both MSLS and I_aLS are really only viable as extensions to linear search, with the assumption that we have enough time for many runs.

6.2 Additional conclusions (after adding simpler perturbation)

Well, simpler means better a lot of the time, and this is one of those times. There is a very noticeable improvement in results, which can be attributed to the increased number of instances that are explored. No time is wasted on extra evaluations, comparisons, etc. This is very clearly visible on Table 4, where ILS manages to run linear search many more times.