Evolutionary Computation Lab III

Piotr Kaszubski 148283

Tuesday, November 7, 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 Corrections from previous reports

While working on this report, I discovered a trivial but serious bug that unfortunately affected the previous 2 reports: when exporting the found solutions (for the visualization plots), I was *maximizing* the score instead of minimizing. That means all the plots you have seen thus far have been the *worst* solutions, not best. The scores in tables were unaffected by this error.
Fixed in c2604bc.

I rerun all experiments from report 1 and 2 and pushed the new plots. If you wish, you can examine them here:

- updated report 1 plots
- updated report 2 plots

3 Pseudocode

3.1 InitMoves

Input set of graph nodes `G`, set of candidate nodes `N`

Output A set `M` of all possible moves. Each move is comprised of the *move type* (intra-route node swap, intra-route edge swap, or inter-route node swap) and *indices* of the nodes/edges involved.

Steps

```
1. Initialize M to an empty list.
2. For i = 1...sizeof(G), do:
   (a) For j = i...sizeof(G), do:
       i. Add (i, j, intra-route node swap) move to M.
       ii. Add (i, j, intra-route edge swap) move to M.
3. For i = 1...sizeof(G), do:
   (a) For j = 1...sizeof(N), do:
       i. Add (i, j, inter-route node swap) move to M.
```

Observation

- Since moves are index-based, and all graphs we will be dealing with have an equal number of nodes, `InitMoves()` always has the same return value. To simplify things, I will call it without parameters.

3.2 Greedy local search

Input set of graph nodes `G`, set of candidate nodes `N`

Output A set of graph nodes `G'`$\in$ (`G`$\cup$`N`), *at least* as good as `G` w.r.t. the objective function, where `sizeof(G')` = `sizeof(G)`.

Steps

```
1. Let M = InitMoves().
2. Let did_improve = true.
3. While did_improve, do:
   (a) Set did_improve = false.
   (b) Let M' be a copy of M.
```

```
(c) While sizeof(M') > 0, do:
    i. Pop a random move from M' into m.
    ii. Compute score delta d of m.
    iii. If d < 0, then:
        A. Perform m.
        B. Set did_improve = true.
        C. Break to outer loop.
```

3.3 Steepest local search

Input set of graph nodes `G`, set of candidate nodes `N`

Output A set of graph nodes `G'`$\in$ (`G`$\cup$`N`), *at least* as good as `G` w.r.t. the objective function, where `sizeof(G')` = `sizeof(G)`.

Steps

```
1. Let did_improve = true.
2. While did_improve, do:
    (a) Set did_improve = false.
    (b) Find a move m from InitMoves() with the lowest delta d.
    (c) If d < 0, then:
        i. Perform m.
        ii. Set did_improve = true.
```

Remarks

- I used `InitMoves()` here for simplification, but the actual implementation does not waste resources on constructing a complete neighborhood and simply iterates through the possible moves one-at-a-time.

4 Results

ALG.	TSPA TSPB	TSPC TSPD
greedy-random	264,383 (240,873−292,180) 267,346 (240,224−297,124)	214,934 (185,725−239,535) 218,750 (194,342−243,930)
greedy-nn	87,649 (84,471−95,013) 79,304 (77,448−82,669)	58,877 (56,304−63,304) 54,405 (50,335−59,846)
greedy-cycle	77,069 (75,666−80,321) 70,685 (68,764−76,324)	55,845 (53,226−58,876) 55,055 (50,409−60,077)
2-regret	103,385 (86,224−119,577) 97,807 (83,179−113,449)	66,161 (58,778−71,821) 63,387 (53,396−69,441)
wsc	87,553 (80,427−101,498) 82,983 (73,310−95,733)	60,334 (54,306−66,962) 58,478 (51,420−66,263)
ls-greedy-random	77,819 (75,305−81,728) 71,093 (68,171−76,696)	51,615 (48,989−54,563) 48,712 (46,240−53,155)
ls-greedy-preset	74,819 (74,690−74,888) 67,955 (67,748−68,316)	53,080 (53,080−53,080) 45,721 (44,698−46,494)
ls-steepest-random	77,866 (75,315−81,017) 71,322 (68,623−76,002)	51,371 (49,257−53,785) 48,234 (45,351−51,534)
ls-steepest-preset	74,837 (74,837−74,837) 67,852 (67,852−67,852)	53,080 (53,080−53,080) 45,556 (45,556−45,556)

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

ALG.	TSPA TSPB	TSPC TSPD
ls-greedy-random	35.826 (31.185−43.658) 36.761 (31.362−43.052)	36.884 (31.790−52.520) 38.939 (30.861−50.768)
ls-greedy-preset	5.050 (3.089−11.703) 4.435 (3.041−8.031)	2.471 (1.739−3.529) 8.352 (5.892−11.703)
ls-steepest-random	82.528 (70.451−117.119) 84.666 (73.417−98.120)	82.132 (72.866−96.202) 83.055 (68.014−94.368)
ls-steepest-preset	7.415 (7.290−12.772) 6.738 (6.681−7.263)	2.442 (2.429−2.567) 24.005 (23.762−30.697)

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

5 Visualizations

5.1 TSPA.csv

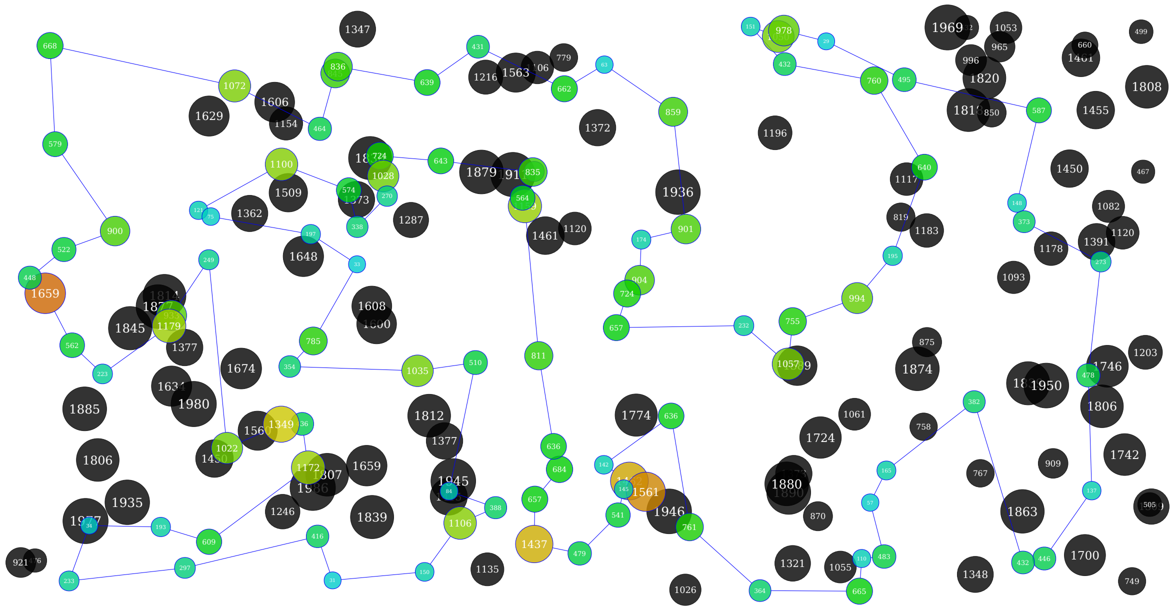

Figure 1: Best ls-greedy-random solution to TSPA (75,305)

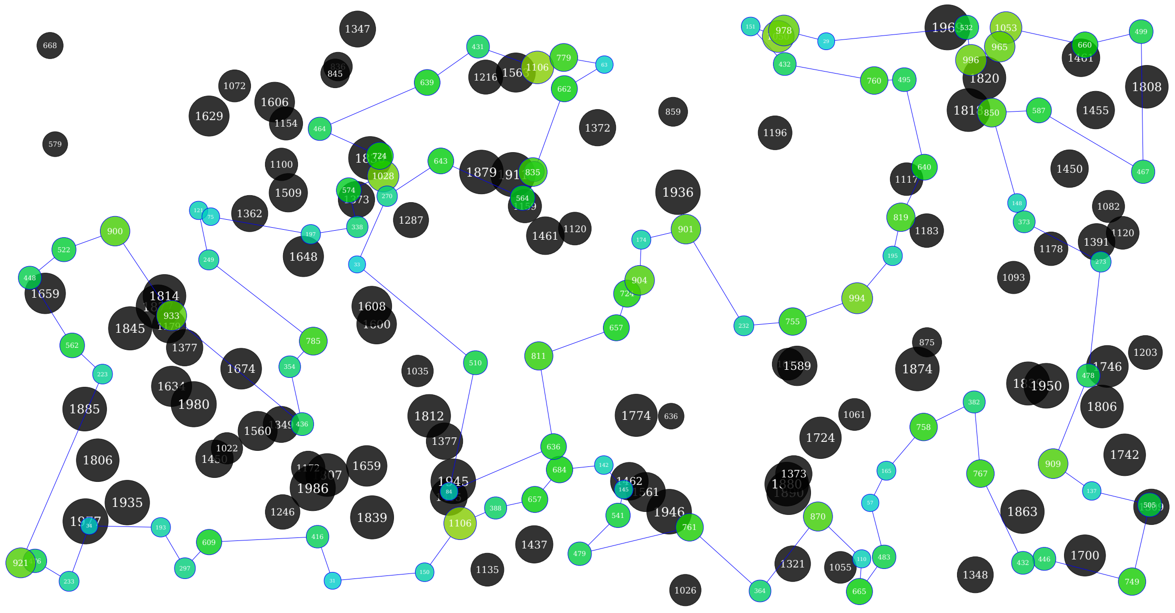

Figure 2: Best ls-greedy-preset solution to TSPA (74,690)

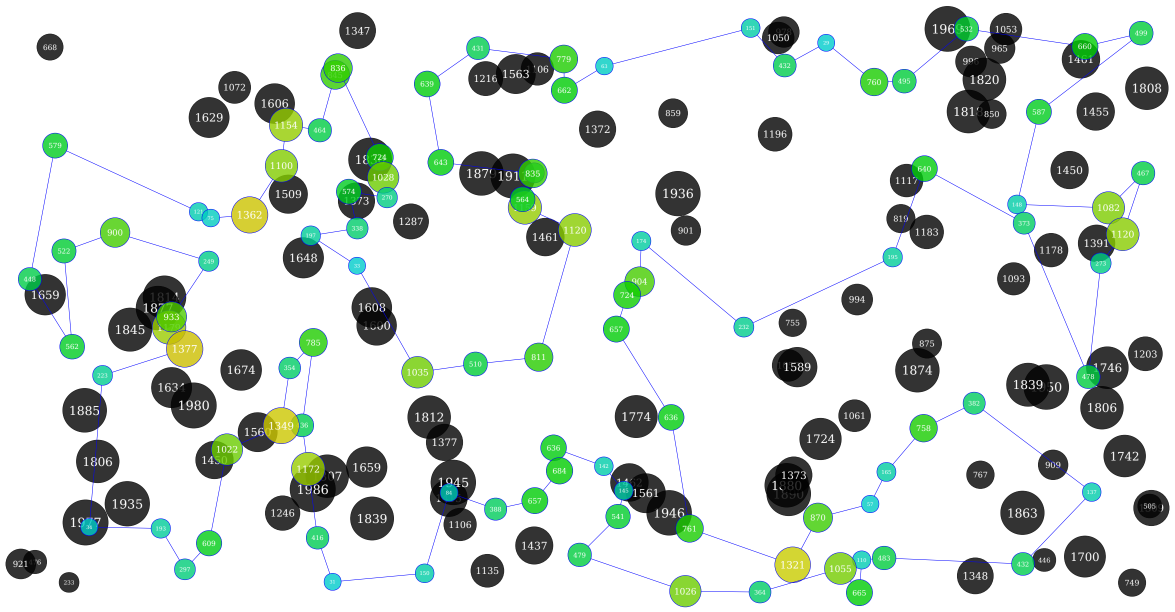
Figure 3: Best ls-steepest-random solution to TSPA (75,315)

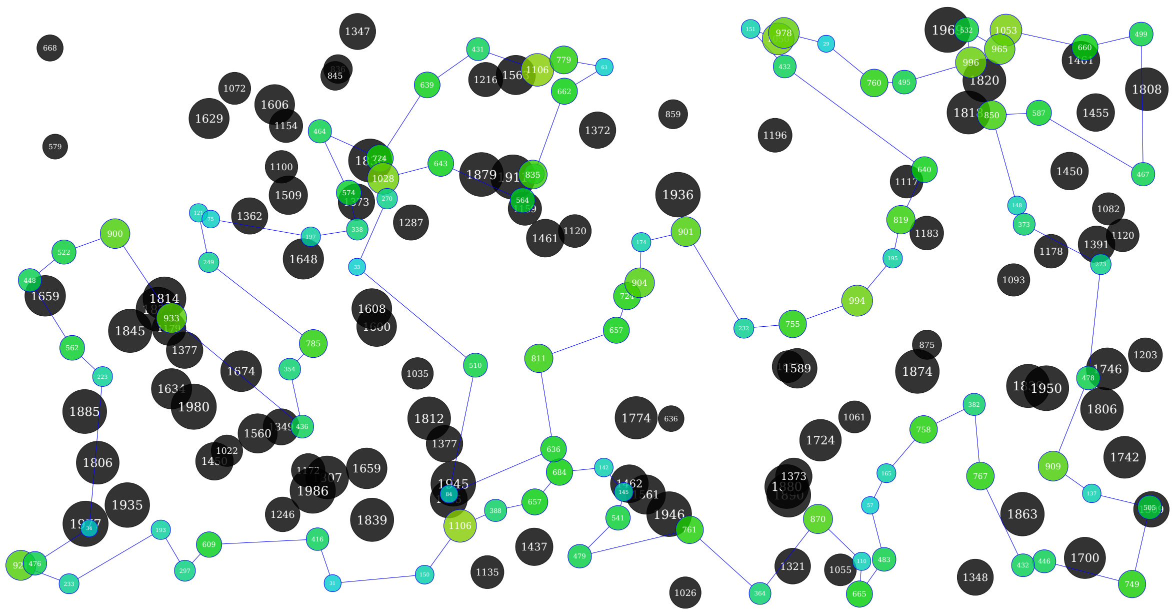
Figure 4: Best ls-steepest-preset solution to TSPA (74,837)

5.2 TSPB.csv

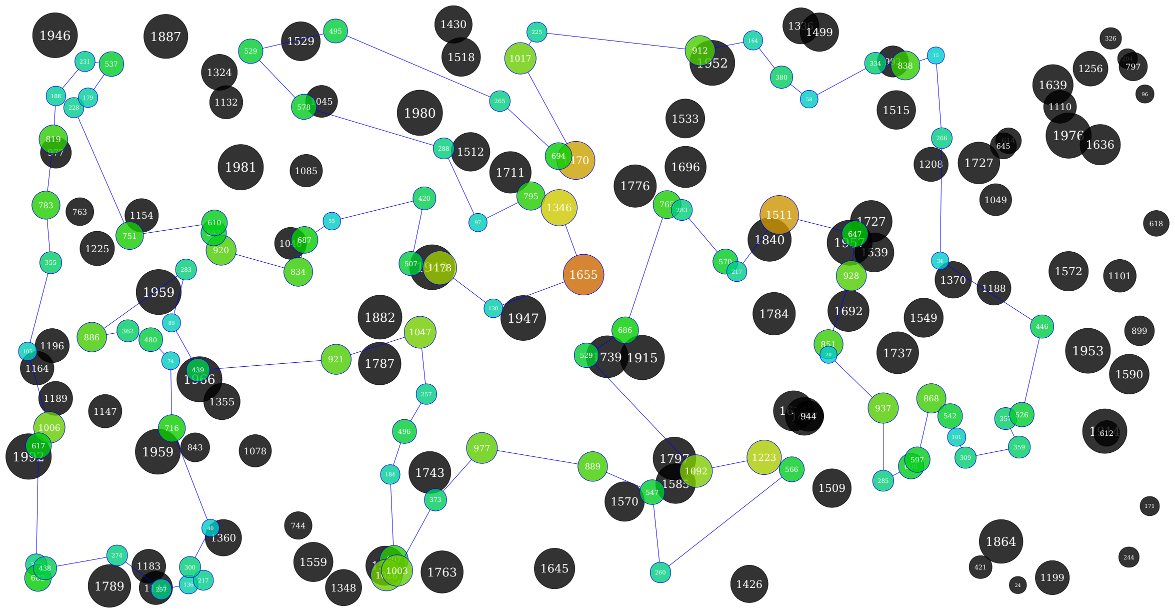
Figure 5: Best ls-greedy-random solution to TSPB (68,171)

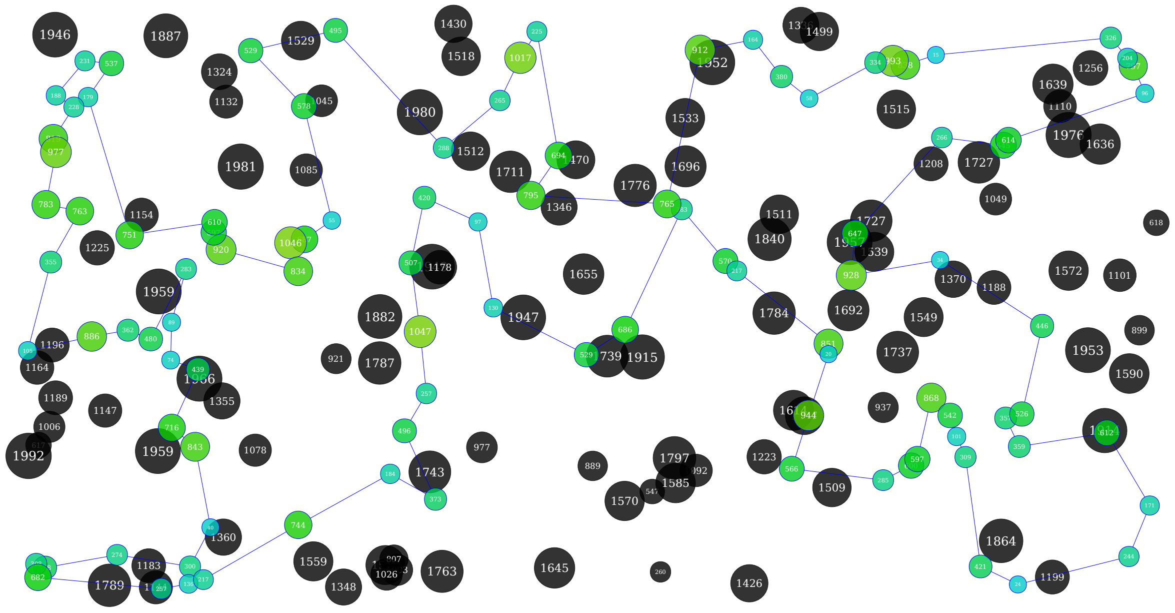
Figure 6: Best ls-greedy-preset solution to TSPB (67,748)

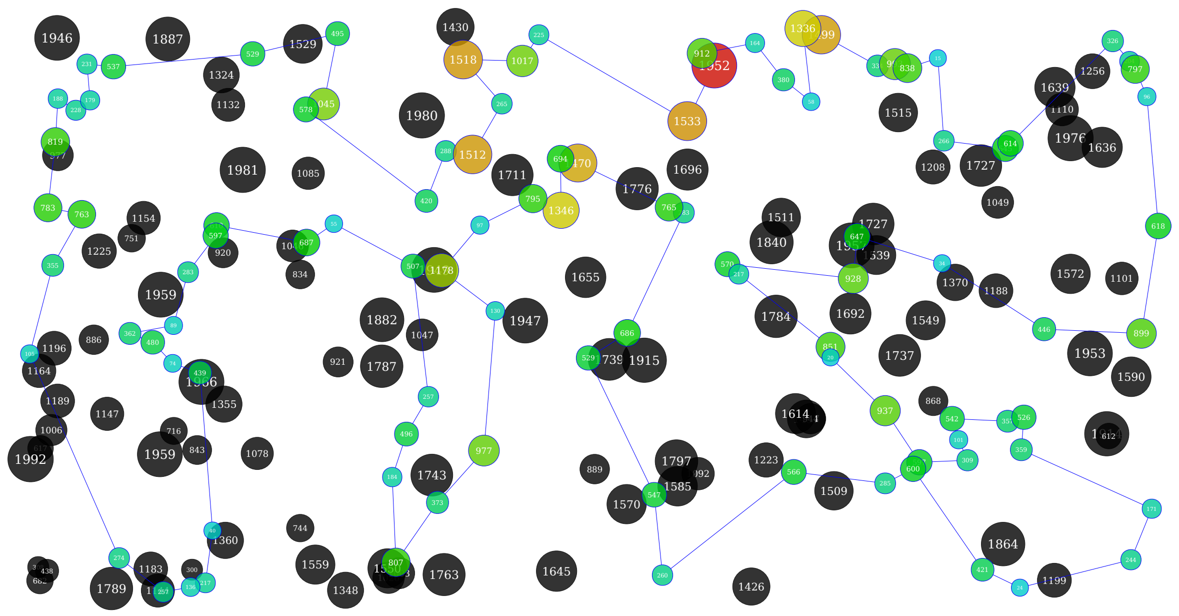
Figure 7: Best ls-steepest-random solution to TSPB (68,623)

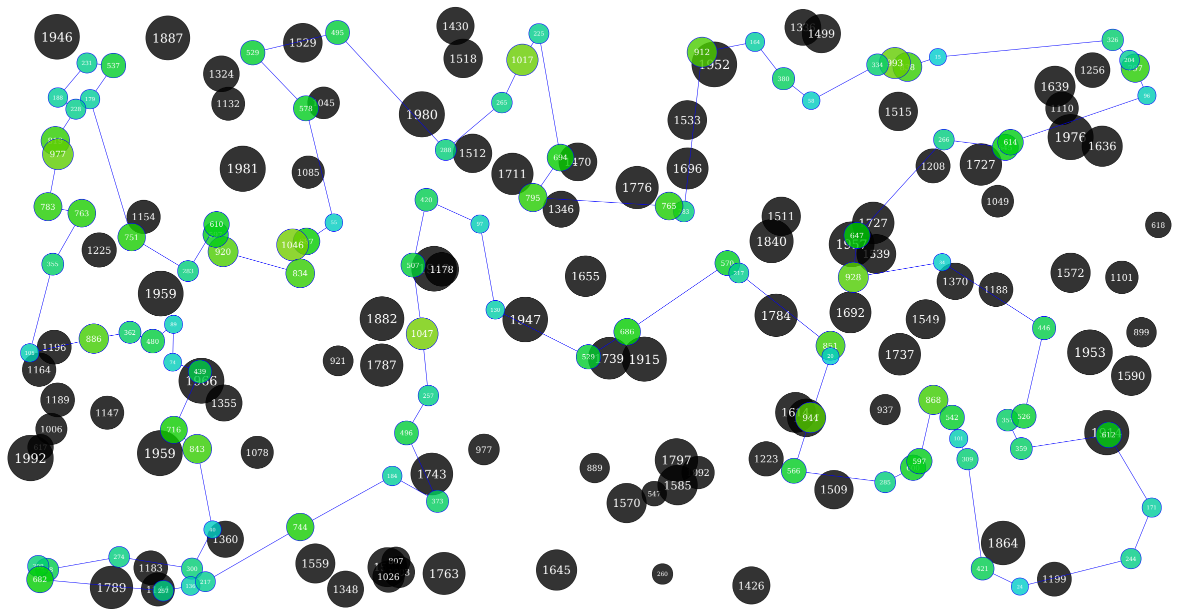
Figure 8: Best ls-steepest-preset solution to TSPB (67,852)

5.3 TSPC.csv

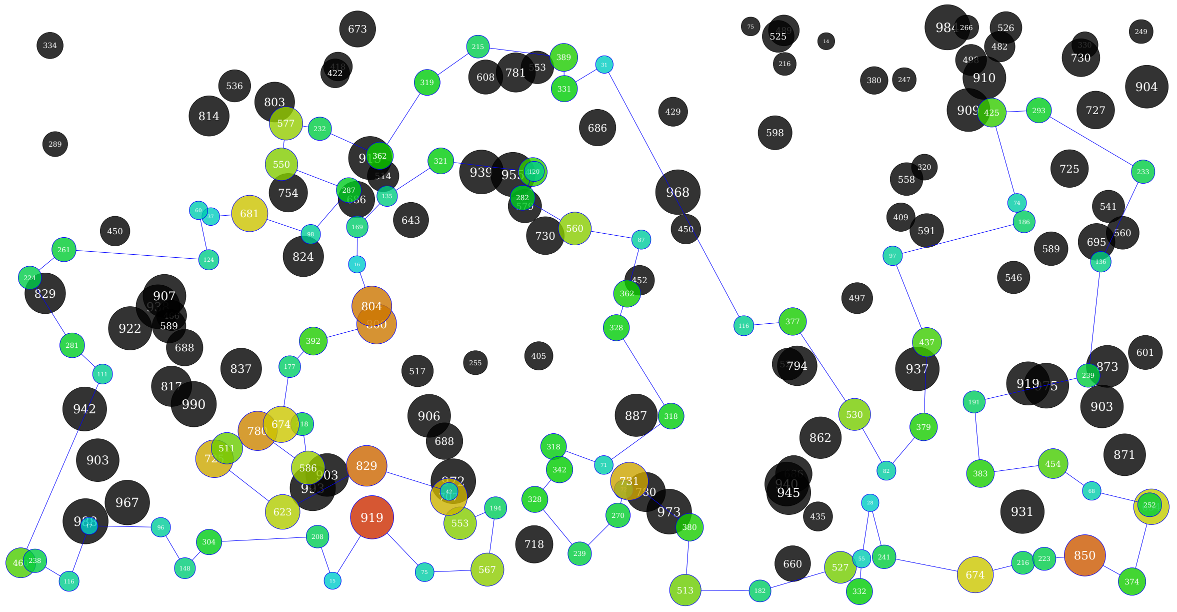

Figure 9: Best ls-greedy-random solution to TSPC (48,989)

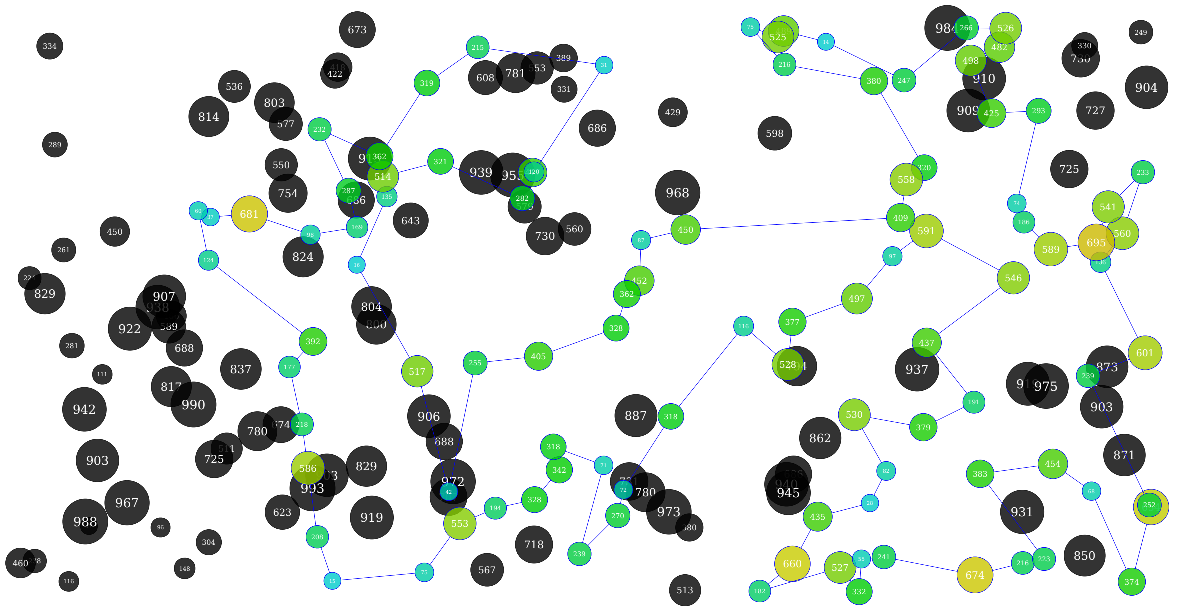

Figure 10: Best ls-greedy-preset solution to TSPC (53,080)

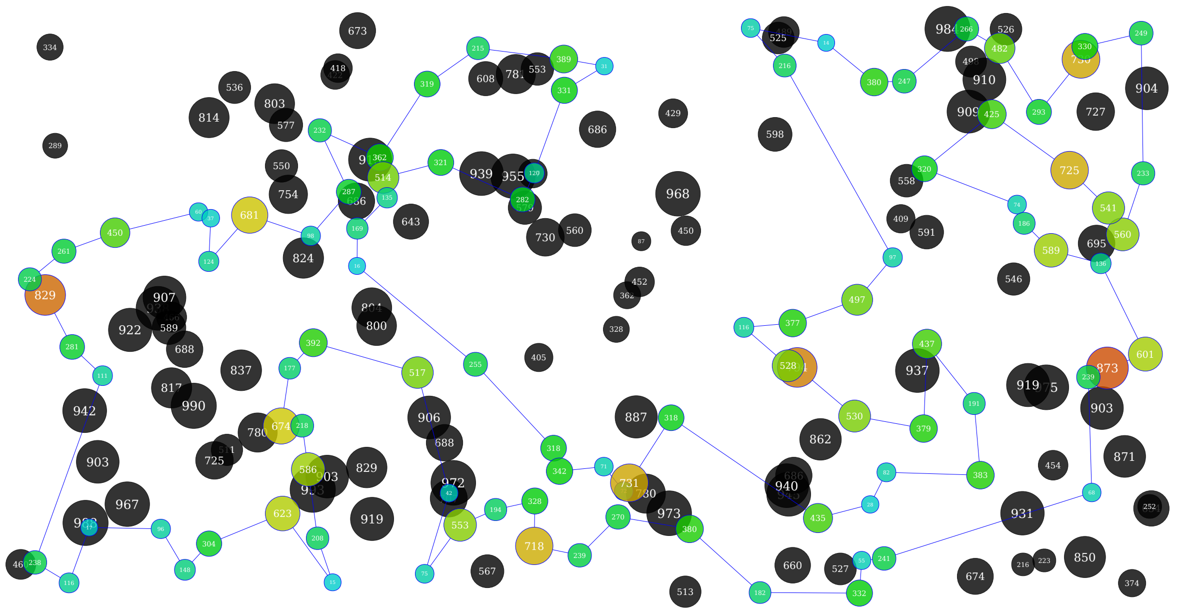
Figure 11: Best ls-steepest-random solution to TSPC (49,257)

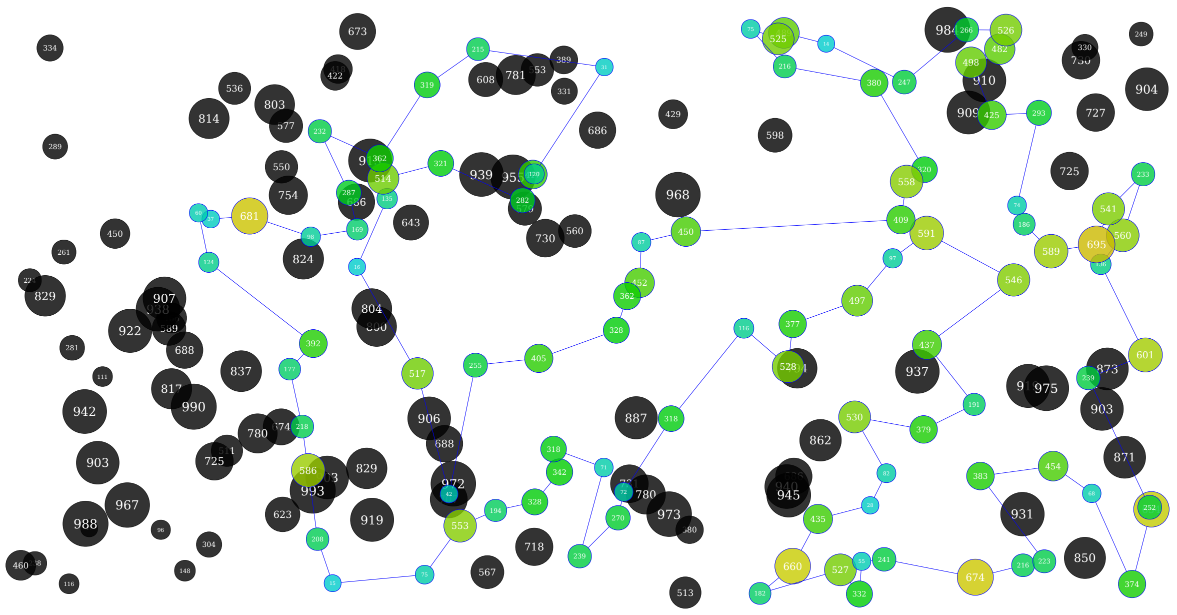
Figure 12: Best ls-steepest-preset solution to TSPC (53,080)

5.4 TSPD.csv

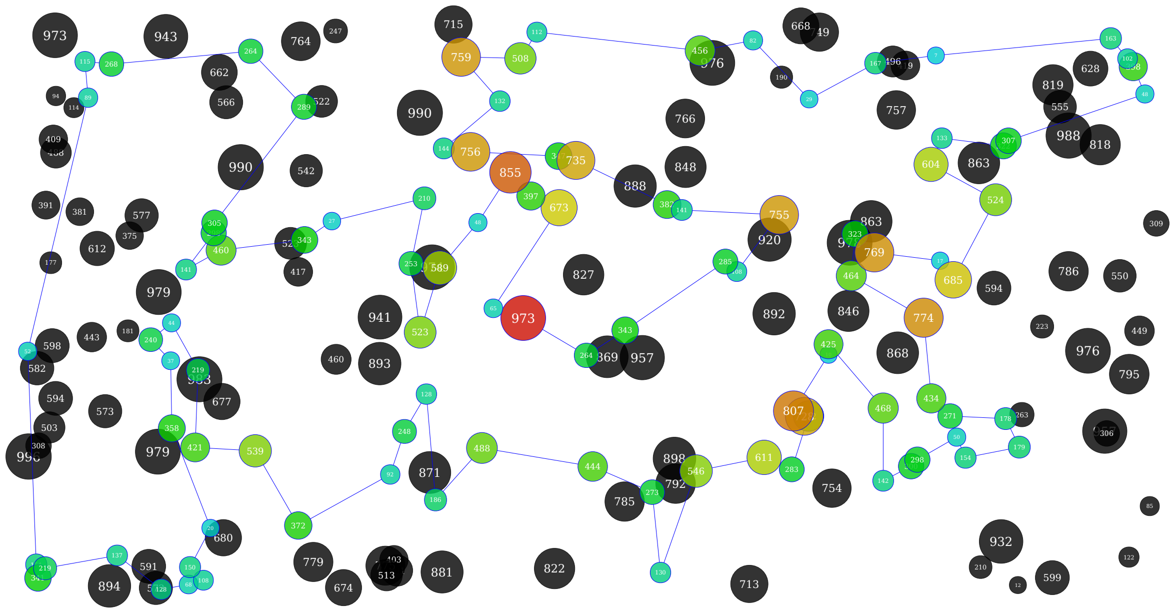

Figure 13: Best ls-greedy-random solution to TSPD (46,240)

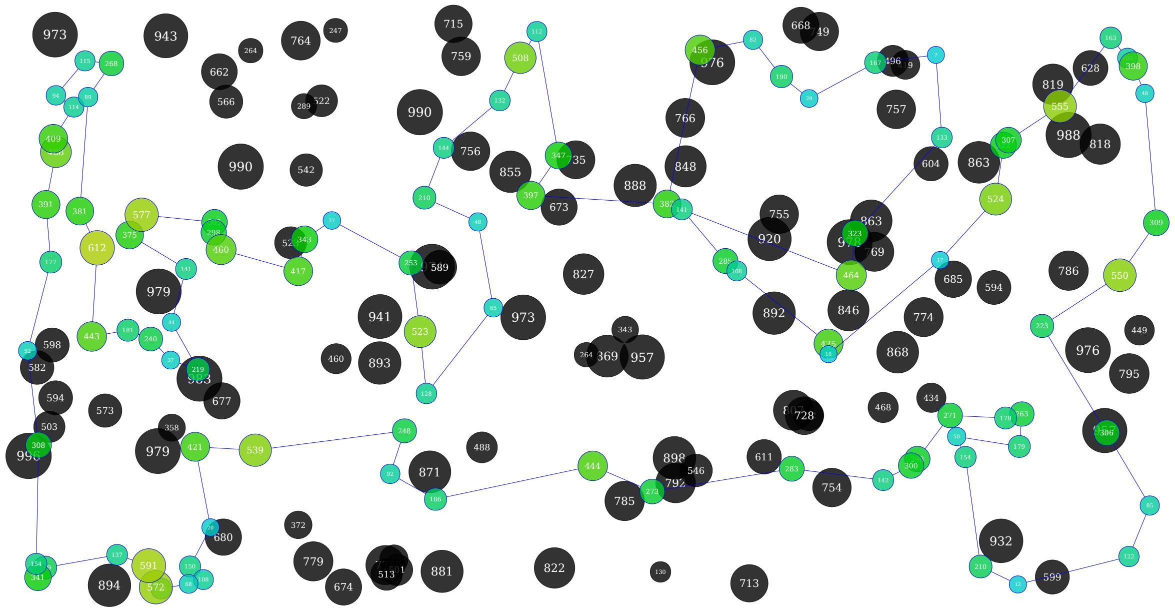

Figure 14: Best ls-greedy-preset solution to TSPD (44,698)

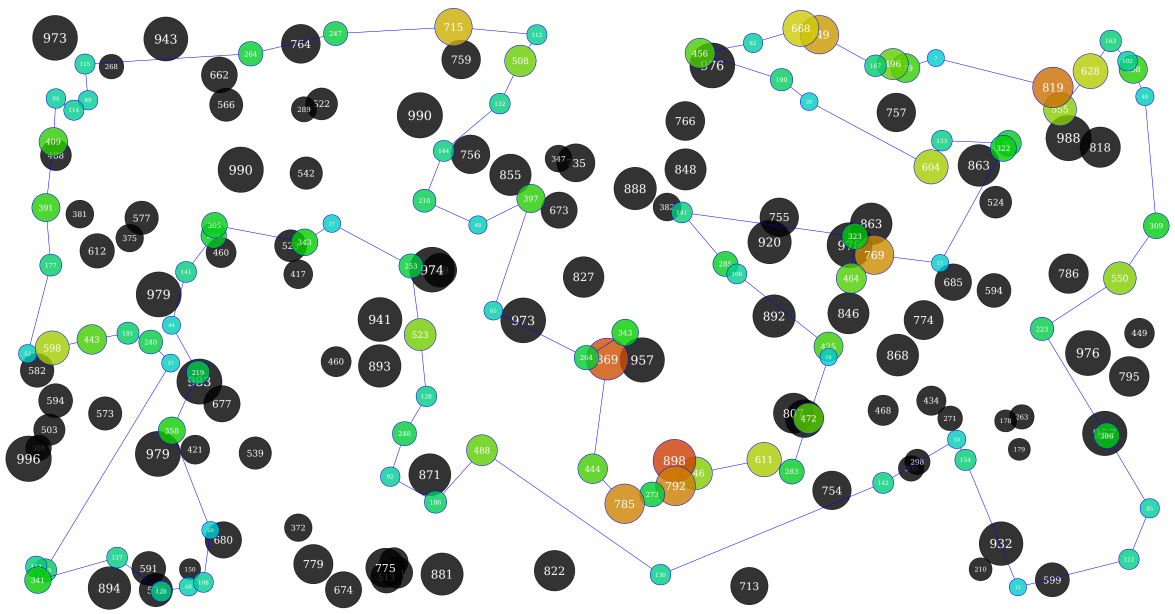
Figure 15: Best ls-steepest-random solution to TSPD (45,351)

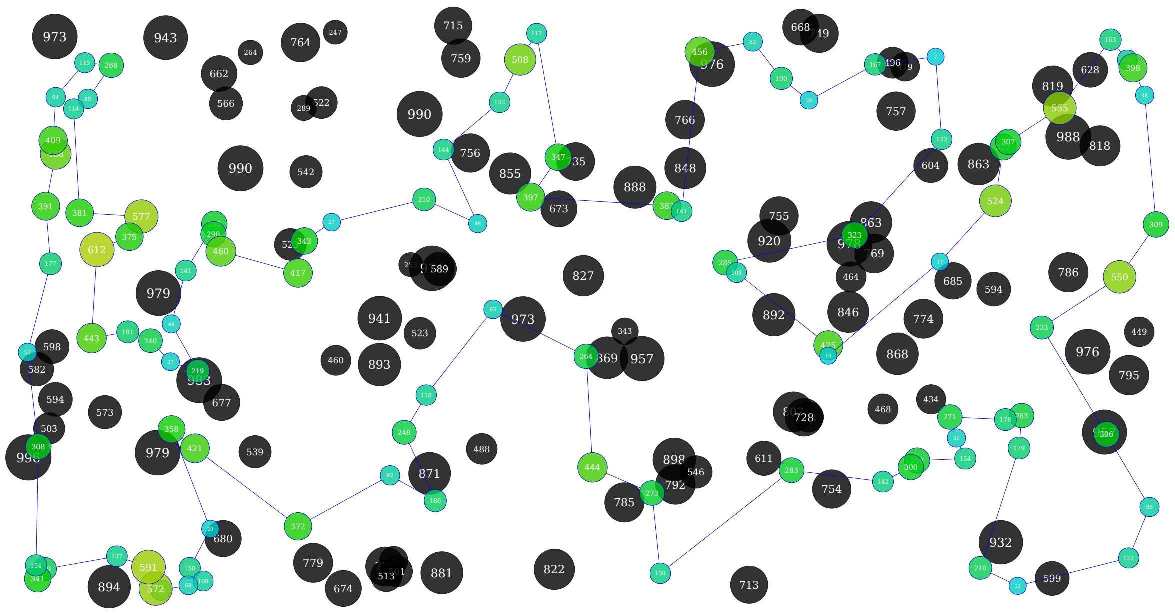
Figure 16: Best ls-steepest-preset solution to TSPD (45,556)

6 Source code

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

7 Conclusions

I'm generally pleased with the results – local search methods managed to improve solutions in almost all cases and in some by a considerable margin.

An interesting anomaly with **TSPC** happened; it appears that the preset solution I took from nearest neighbor run of the first report is, by some insane(?) happenstance, a local optimum. This has caused both greedy and steepest local searches to not be able to find a single improvement, understandably so. As a result, the random-initialized variants beat the greedy-initialized one, which in turn have equal average-min-max values.[1]

Similarly to report 2, I ended up implementing a data structure for this assignment, which I ended up not using at all. I really need to stop doing that.

I thought a min-heap would be handy for steepest neighborhood scan, because I figured that it's a huge waste to rescan everything every iteration, when we're only changing tiny parts of the graph at a time. This idea is good, but then I forgot about it, ended up doing it the naïve way, finished the report[2], and once I was past the finish line, I did not feel like coming back to optimize it anymore.

[1]It was later brought to my attention that I probably misunderstood the assignment a bit, and we were supposed to start local search 200 times from graphs that were obtained from a greedy method being started from different nodes. I instead read it as starting the *local search* each time from a different node, which doesn't make a lot of sense and I chose to ignore this requirement entirely. I always start all preset local searches from the same solutions. My apologies, if I wasn't already past the schedule I would correct this. On the flip side, this gave me the opportunity to notice and discuss the local optimum situation, which is pretty interesting.

[2]FYI, the running time for all the experiments in this report was about 1:53 minutes.