Evolutionary Computation Lab IX

Piotr Kaszubski 148283

Monday, January 8, 2024

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 Recombination operators

Both recombination operators start by initializing an empty child and adding **Common Edges** from both parents to it. Then, the two options are:

op1 – fill the remaining nodes randomly,

op2 – fill the remaining nodes using the *greedy cycle* strategy from report 1 (also used in report 7).

Common Edges algorithm The algorithm is based on Edge Recombination Crossover (ERX) from the lectures, with a small modification:

1. Select a random node from either parent (that is not already present in the child). If there are no such nodes, terminate.
2. Select subsequent elements randomly from a set of all nodes which **share an edge** with the last selected element in either of the parents. If there are no such nodes, go back to 1.

Difference with ERX The lecture slides mention only the ”subsequent” element, which suggests we consider only one of two adjacent nodes per parent. I chose to relax this mechanism, allowing up to 4 continuations per iteration, instead of 2. It seemed like a good idea, because a cycle of nodes is bidirectional. But I don’t know if it actually makes things better or worse. At the very least it should reduce the likelihood of premature convergence, I suppose.

2.2 Runtime constraint

To keep comparisons consistent with previous algorithms, I set the same CPU time limit to all the evolutionary methods presented in this report (16.323 CPU seconds).

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)
lsns-nolsearch-from-greedy	73,777 (73,061−74,941) 68,001 (66,750−69,953)	48,571 (47,475−49,299) 45,345 (44,229−46,356)
lsns-nolsearch-from-lsearch	73,774 (73,221−74,547) 68,217 (66,814−70,149)	48,391 (47,514−49,624) 45,249 (43,954−46,160)
lsns-lsearch-from-greedy	73,609 (**72,859**−74,404) 66,987 (66,755−67,337)	48,454 (47,562−49,040) 44,708 (43,869−46,217)
lsns-lsearch-from-lsearch	73,607 (72,967−74,319) 67,112 (**66,585**−67,673)	48,180 (**47,406**−49,137) 44,574 (**43,884**−45,715)

ALG.	TSPA TSPB	TSPC TSPD
evo-op1	75,752 (74,909–76,582) 68,656 (67,989–69,463)	49,601 (48,387–50,805) 46,330 (45,247–47,714)
evo-op2	75,766 (74,701–76,827) 68,713 (67,872–69,598)	49,712 (48,367–50,408) 46,284 (45,364–47,530)

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

4 Visualizations

4.1 TSPA.csv

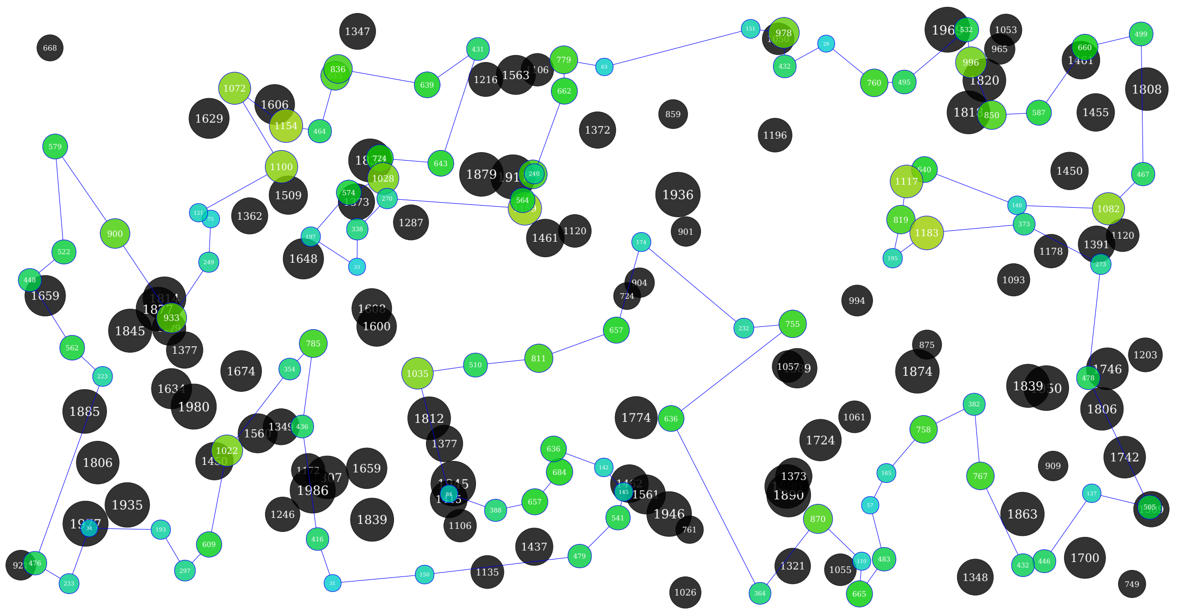

Figure 1: Best evo-op1 solution to TSPA (74,909)

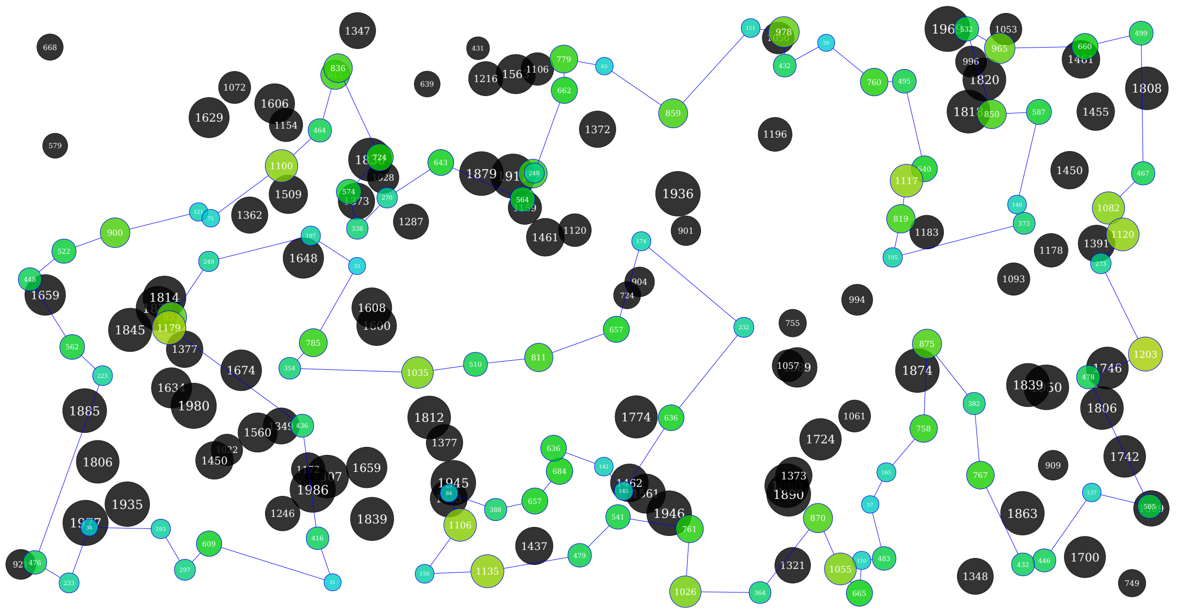
Figure 2: Best evo-op2 solution to TSPA (74,701)

4.2 TSPB.csv

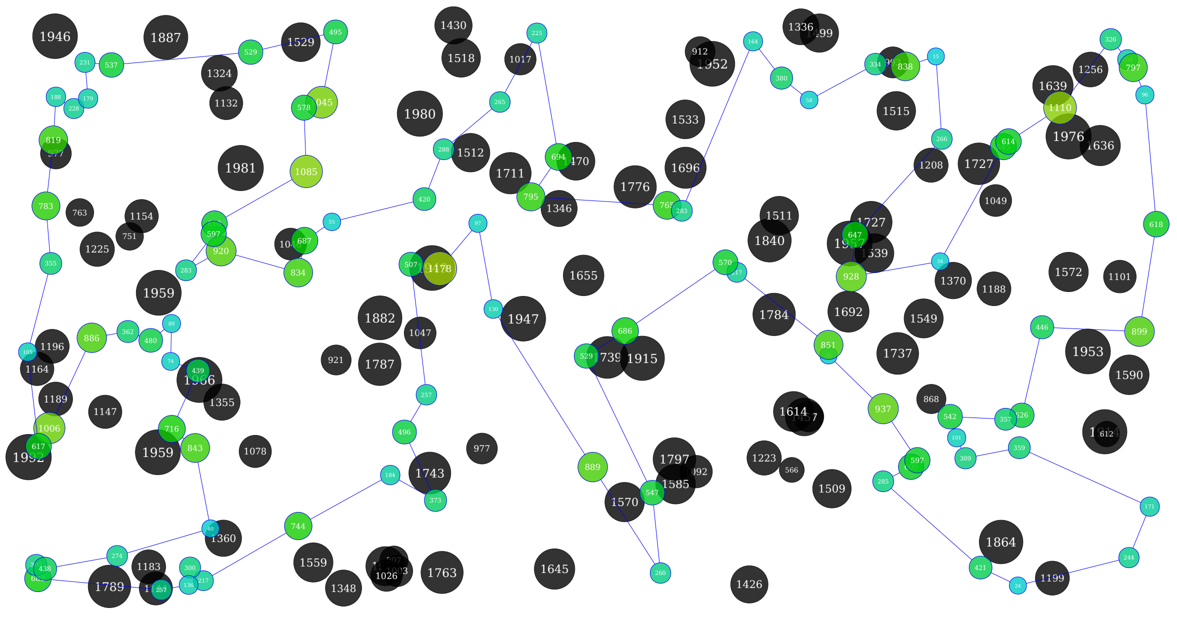
Figure 3: Best evo-op1 solution to TSPB (67,989)

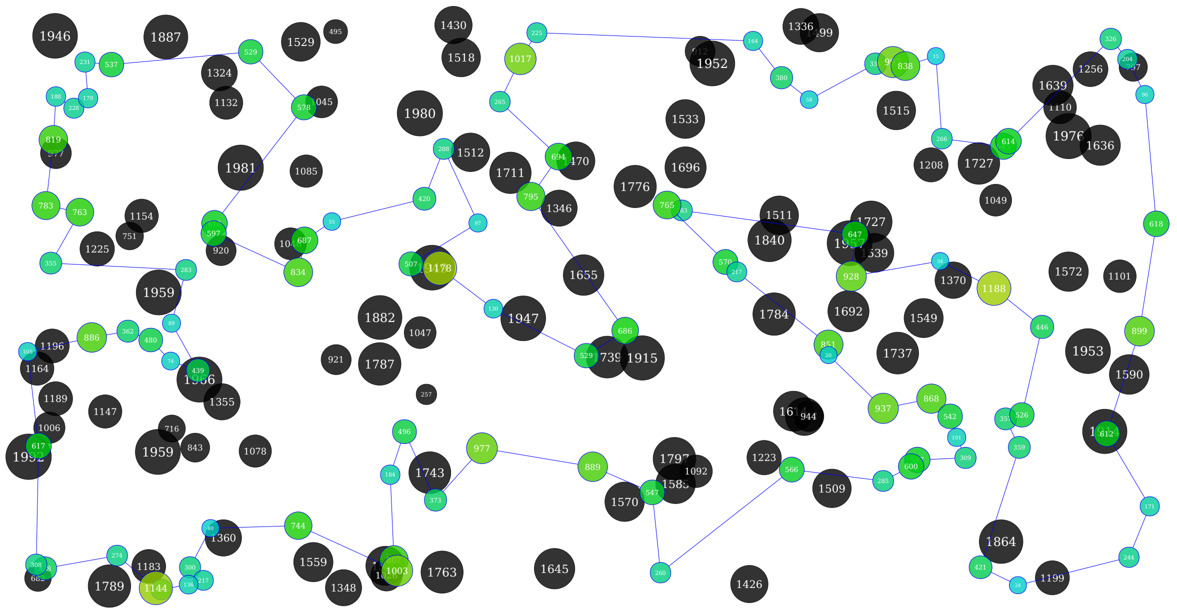
Figure 4: Best evo-op2 solution to TSPB (67,872)

4.3 TSPC.csv

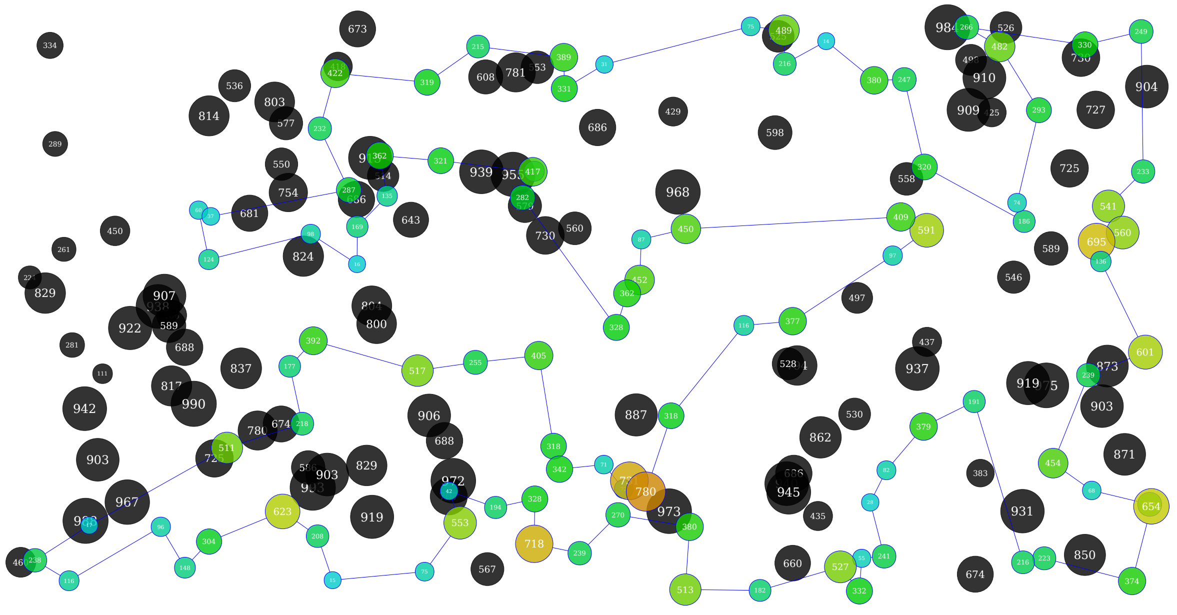
Figure 5: Best evo-op1 solution to TSPC (48,387)

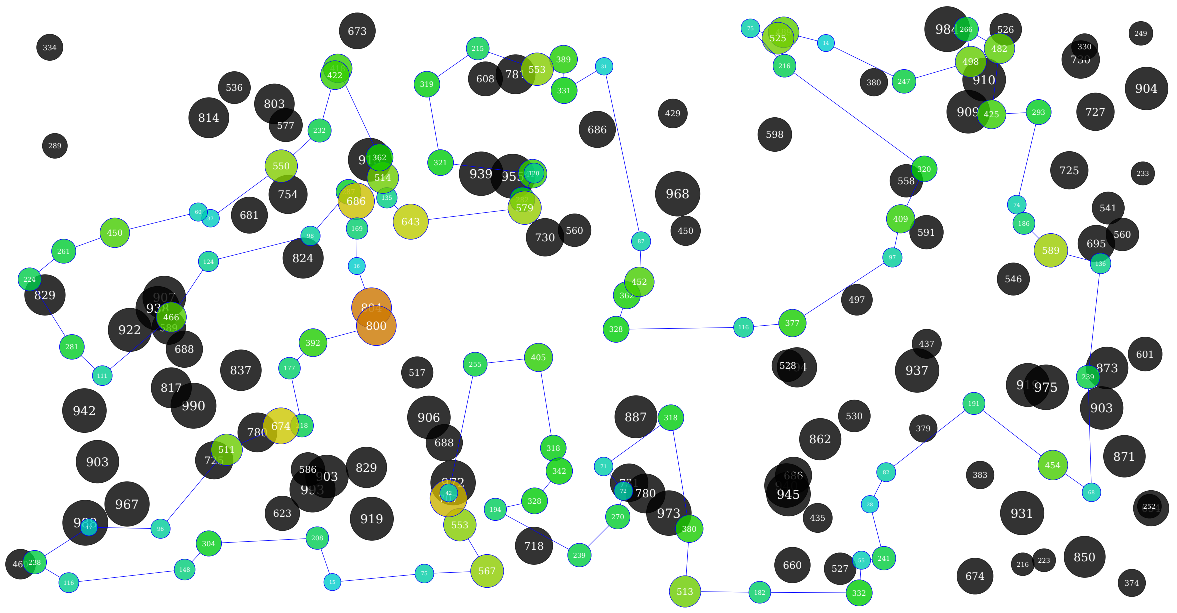
Figure 6: Best evo-op2 solution to TSPC (48,367)

4.4 TSPD.csv

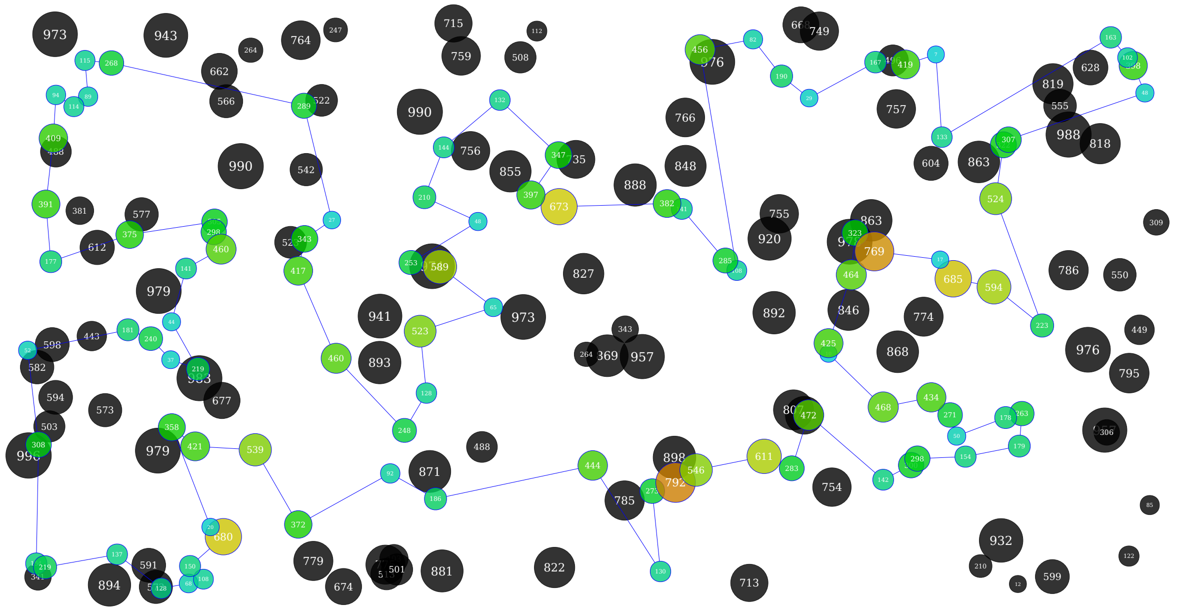
Figure 7: Best evo-op1 solution to TSPD (45,247)

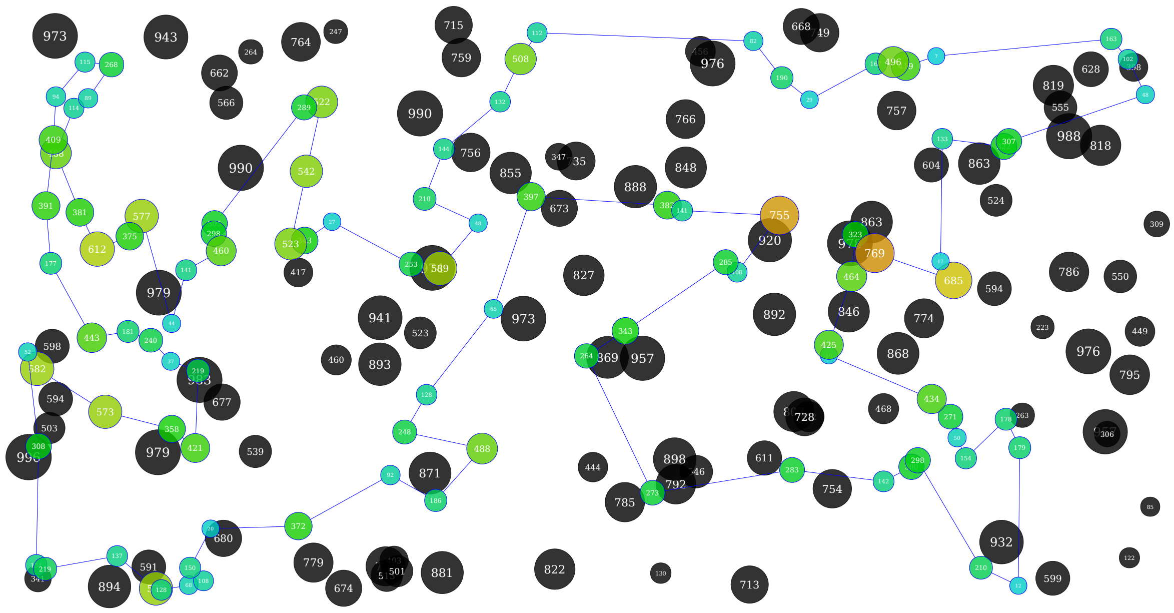
Figure 8: Best evo-op2 solution to TSPD (45,364)

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

Evolutionary methods take considerably more effort to implement, and in their most basic form perform quite average. However, their major strength is that, once the framework has been established, it's relatively easy to swap and compare different parameters and components.

I'm surprised that **op1** and **op2** performed pretty much the same. I was expecting **op2** to be at least noticeably better. But it's possible that this similarity is caused by **Common Edges** filling up the vast majority of children, leaving little difference to be made by random or repair operators. If I wasn't running late, I would test this hypothesis, but I want to get this report out as soon as possible. Maybe I will find more insight for report 10, because I will probably try to develop some kind of hybrid evolutionary algorithm for it, basing it off of this report.