Evolutionary Computation Lab I

Piotr Kaszubski 148283

Sunday, October 15, 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 Random solution

Input A set `N` of nodes.

Output An ordered subset `N'` of `N`, with half of `N`'s cardinality.

Steps

1. Initialize `N'` to an empty list.
2. While `[ len(N') < len(N)/2 ]`, do:
   (a) Choose a random node `n` from `N`.
   (b) Remove `n` from `N`.
   (c) Append `n` to `N'`.

2.2 Nearest neighbor

Input A set `N` of nodes.

Output An ordered subset `N'` of `N`, with half of `N`'s cardinality.

Steps

1. Initialize `N'` to an empty list.
2. Choose a random node `prev_n` from `N`.
3. Remove `prev_n` from `N`.
4. Append `prev_n` to `N'`.
5. While `[ len(N') < len(N)/2 ]`, do:
   (a) Find the node `n` from `N` with the smallest distance to `prev_n`.
   (b) Set `prev_n` to `n`.
   (c) Remove `n` from `N`.
   (d) Append `n` to `N'`.

2.3 Greedy cycle

Input A set `N` of nodes.

Output An ordered subset `N'` of `N`, with half of `N`'s cardinality.

Steps

1. Initialize `N'` to an empty list.
2. Choose a random node `n1` from `N`.
3. Remove `n1` from `N`.
4. Append `n1` to `N'`.
5. Find the node `n2` from `N` with the smallest distance to `n1`.
6. Remove `n2` from `N`.
7. Append `n2` to `N'`.
8. While `[ len(N') < len(N)/2 ]`, do:
   (a) Let `found_n` be an empty value.
   (b) Let `lowest_delta` be infinity.
   (c) For each pair `(n,o)` of adjacent nodes from `N'`, do:
      i. Find the node `p` from `N`, such that the sum `s` of its distances from `n` and `o` is minimal.
      ii. Let `delta` be `s` minus the distance between `n` and `o`.
      iii. If `delta < lowest_delta`, then:
         A. Set `found_n` to `p`.
         B. Set `lowest_delta` to `delta`.
   (d) Remove `found_n` from `N`.
   (e) Append `found_n` to `N'`.

3 Results

ALG.	FILE	min	avg	max
random	TSPA.csv	241,510	266,062	308,034
	TSPB.csv	241,731	266,549	293,093
	TSPC.csv	189,473	215,587	239,581
nn	TSPA.csv	110,035	116,145	125,805
	TSPB.csv	106,815	116,181	124,675
	TSPC.csv	62,629	66,196	71,616
cycle	TSPA.csv	113,298	123,691	129,175
	TSPB.csv	111,981	120,922	131,174
	TSPC.csv	67,077	72,771	75,763

4 Visualizations

4.1 TSPA.csv

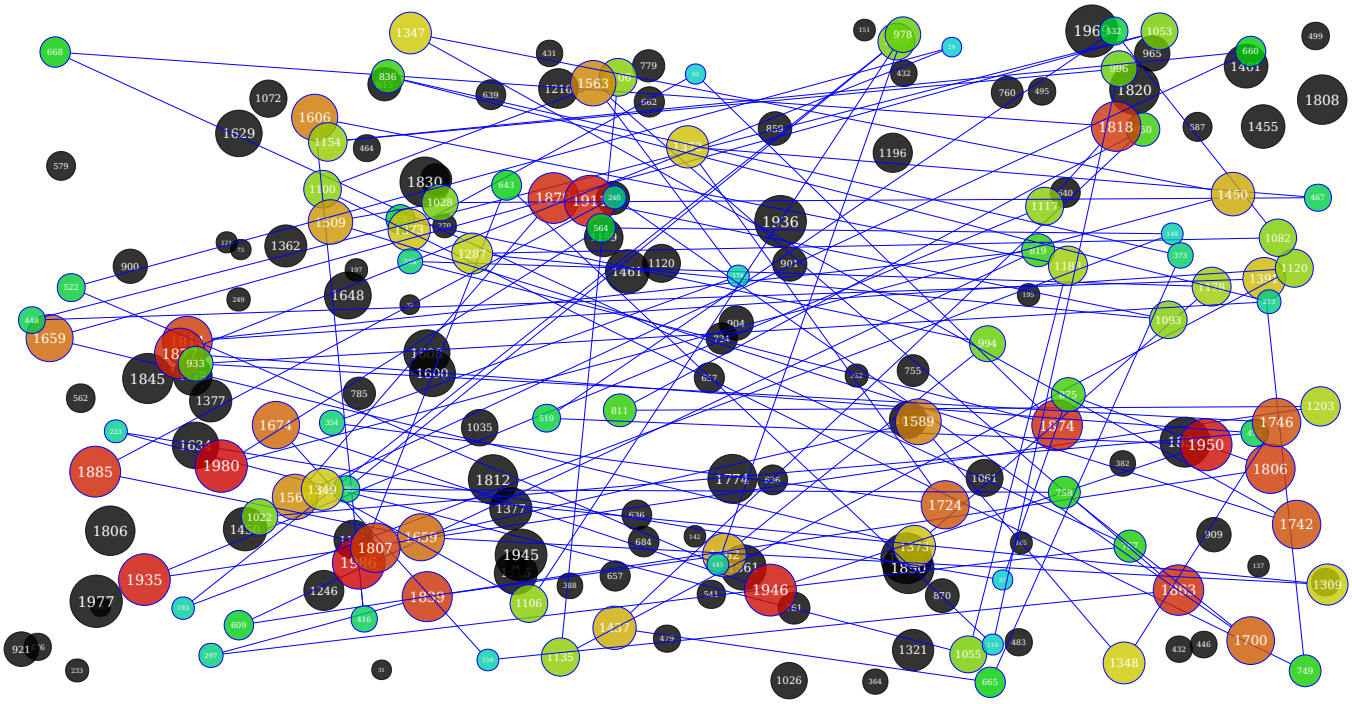

Figure 1: Best random solution to TSPA (241,510)

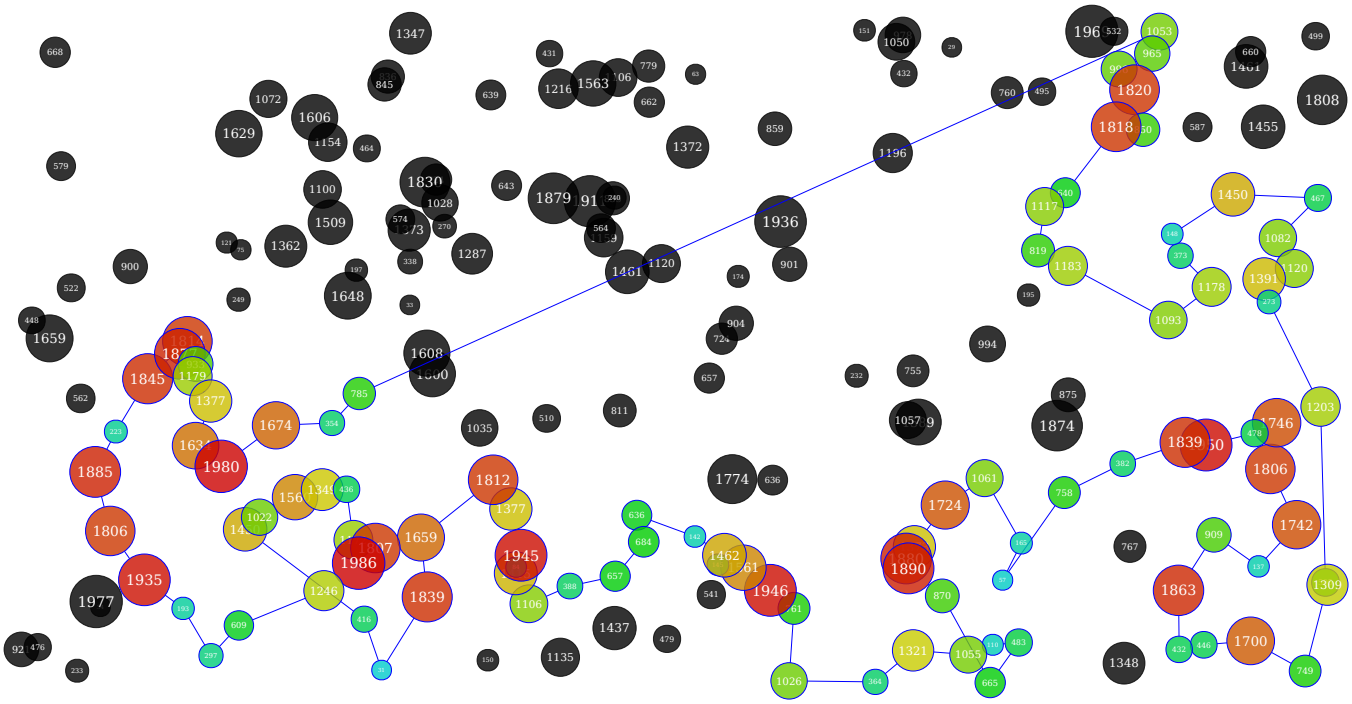

Figure 2: Best nearest neighbor solution to TSPA (110,035)

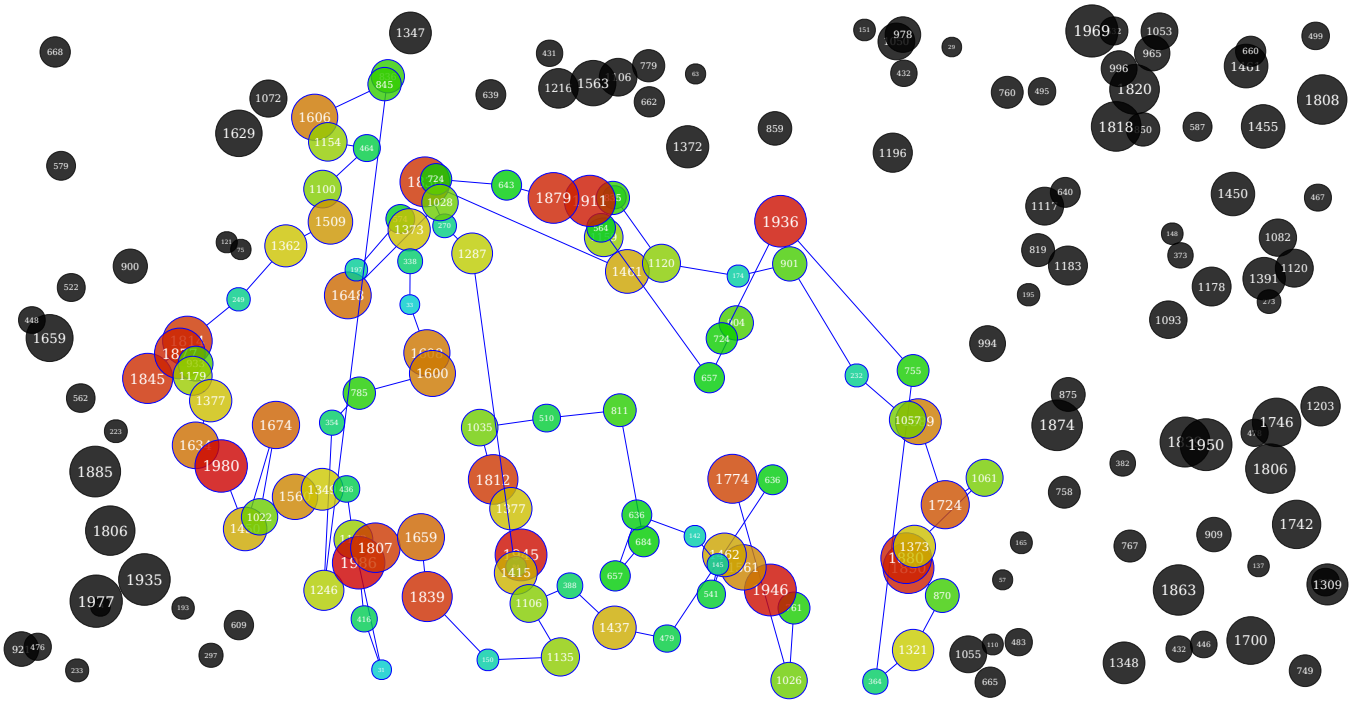

Figure 3: Best greedy cycle solution to TSPA (113,298)

4.2 TSPB.csv

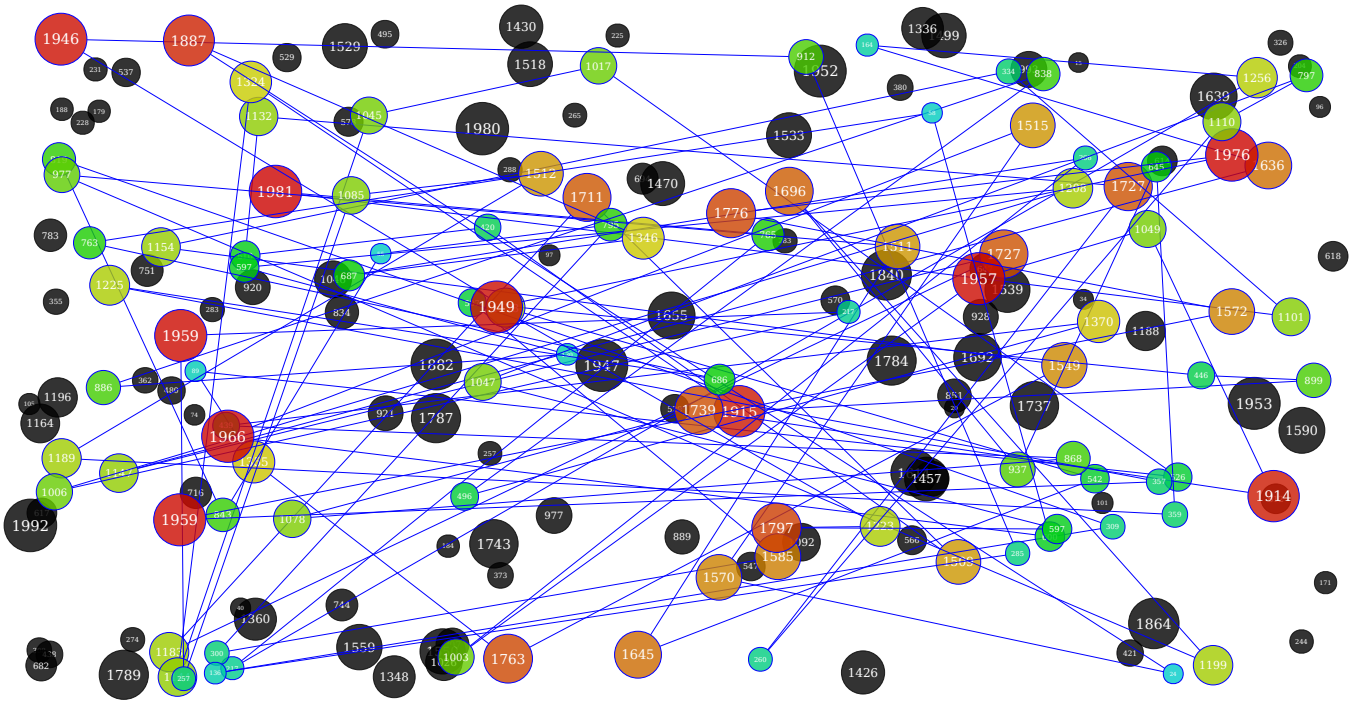

Figure 4: Best random solution to TSPB (241,731)

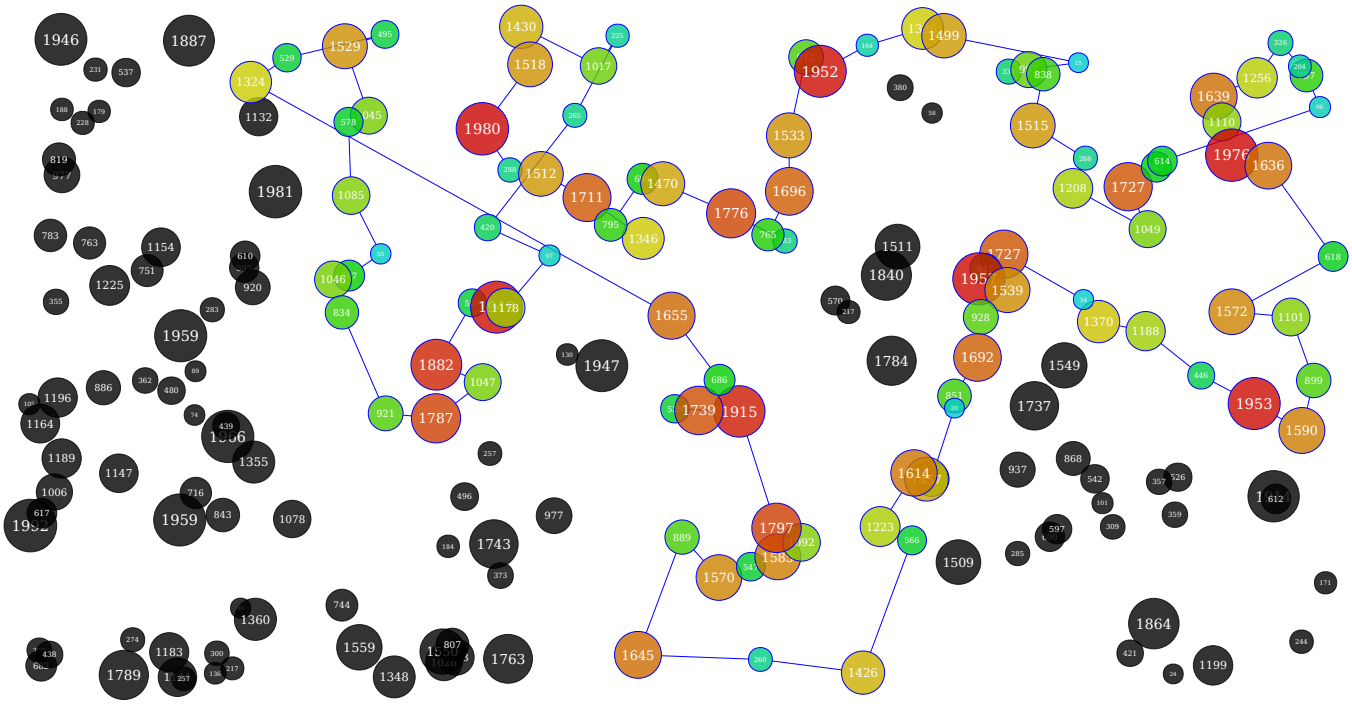

Figure 5: Best nearest neighbor solution to TSPB (106,815)

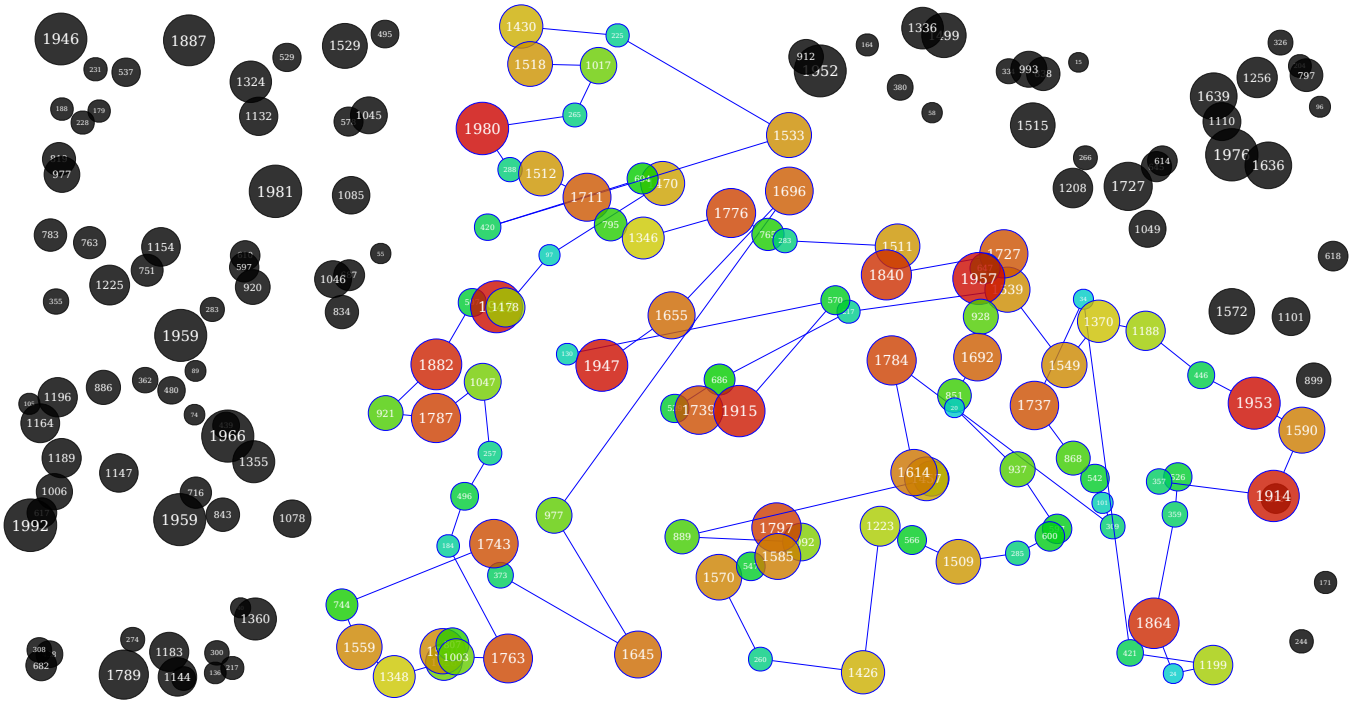

Figure 6: Best greedy cycle solution to TSPB (111,981)

4.3 TSPC.csv

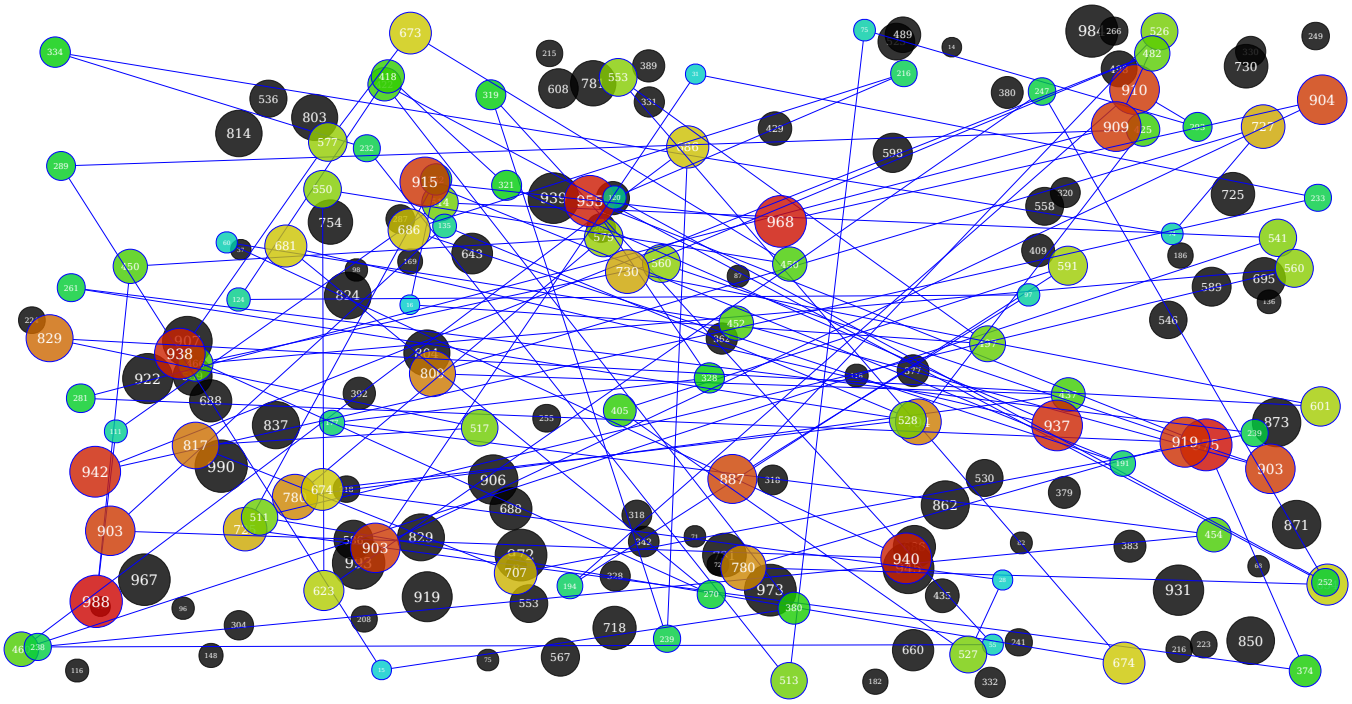

Figure 7: Best random solution to TSPC (189,473)

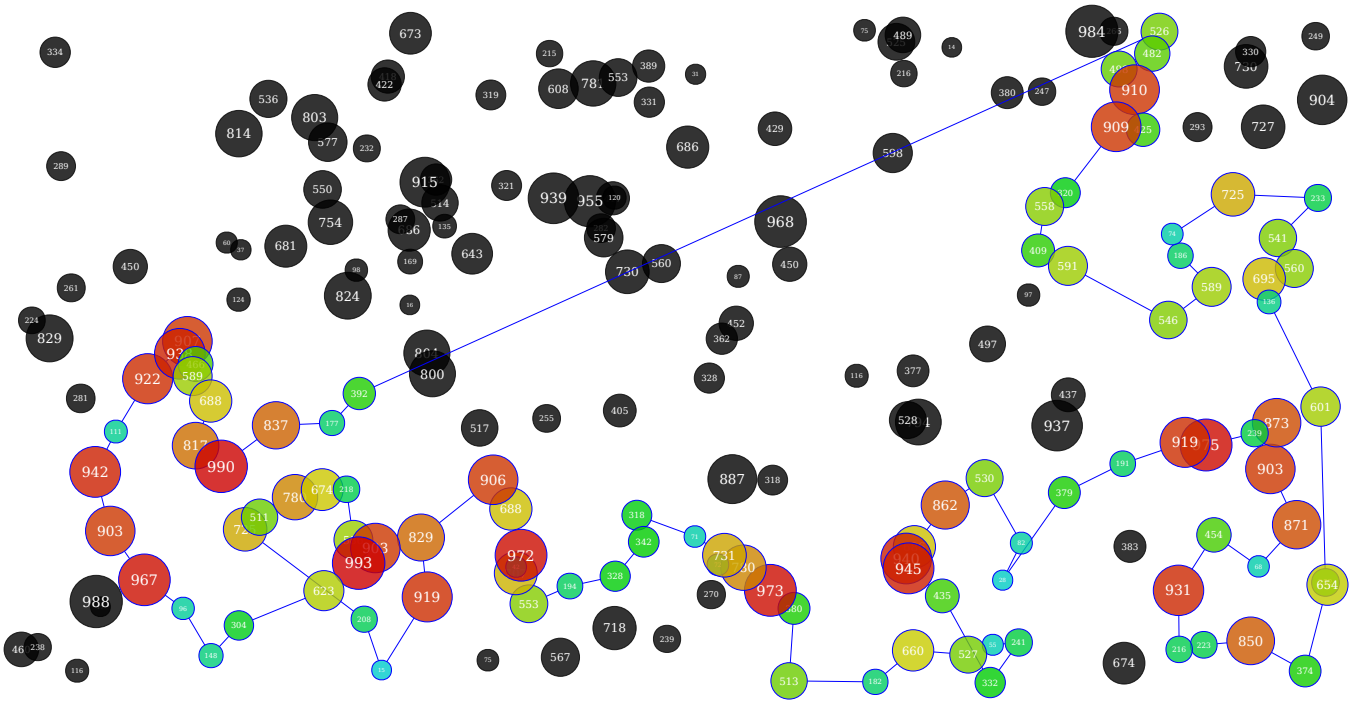

Figure 8: Best nearest neighbor solution to TSPC (62,629)

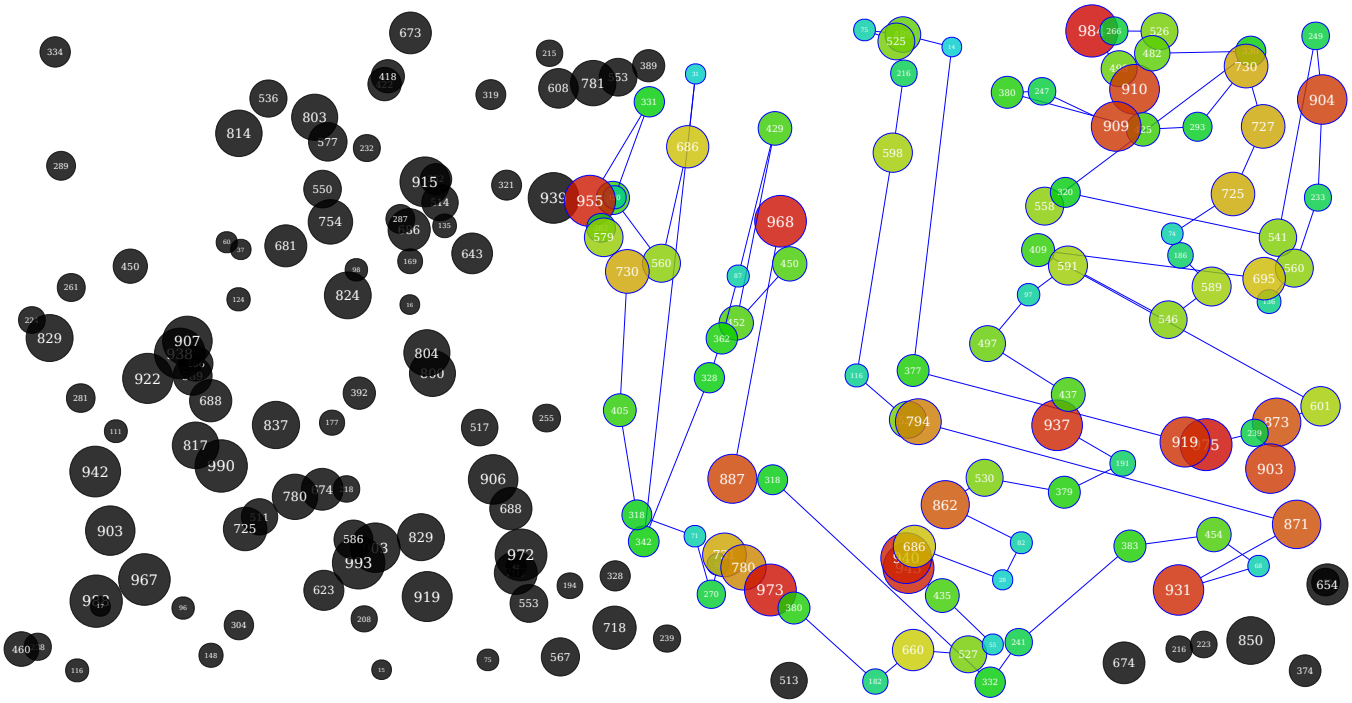

Figure 9: Best greedy cycle solution to TSPC (67,077)

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

The nearest neighbor method actually tends to outperform greedy cycle, despite the latter being slightly more complex and thus making it seem like it should be better.

Before I *consciously* read the part of the task description that specifies to use a cached distance matrix, I was just computing euclidean distance from scratch each time it was needed. This led to pretty bad times[1]. I tried a hashmap cache approach, however it actually made things far worse, because computing euclidean distance is *not* slower than hashing 2 points, walking through a bucket list and comparing keys. It's not very common that functions that are already small and quite fast are the bottleneck, so I was very surprised at this result. Serves as a reminder to stop and think more often whether a tool is right for the job before we sink time into it.

[1] "bad times" = >8 seconds. For comparison, the current and final version of the program using the distance matrix takes about 5 seconds to run on my machine.