Evolutionary Computation Lab V

Piotr Kaszubski 148283

Sunday, November 26, 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

Incorrect inter-swapping in reports 3 and 4

I discovered that I was incorrectly passing parameters to a function responsible for inter-swapping nodes. As a result, all of the inter-swaps were semi-random. It was an easy fix, and it didn't change the results much (mostly there were improvements by up to a thousand, in some cases the result got worse by up to a thousand). The results in this report have been updated.
Fixed in 1da9927.

3 Pseudocode

The logic behind the delta cache is rather complicated, so I think it's better, to our own benefit, if I explain the *strategy*, rather than transcribing tons of functions to a not much more legible form here.

- The cache is comprised of 3 `N`×`N` matrices (`N` being the number of nodes in the solution), one for:
  - **inter-route node swaps**
  - **intra-route node swaps**

  - **intra-route edge swaps**

  These matrices are indexed by node IDs, and for each pair of nodes store a cached delta, or an empty value (internally represented as `LONG_MIN`).

- Whenever we need to compute a delta for any type of swap, we check if a cached value exists in the corresponding matrix. If so, we return it without doing any of the regular computation. Otherwise, after the computation, we store the new delta in the matrix.
  - After an inter-route node swap, *update*:
    - the node that just got added to the graph,
    - its two neighboring nodes.
  - After an intra-route node swap, *update*:
    - the two nodes that got swapped,
    - their neighboring nodes (four nodes in total).
  - After an intra-route edge swap, *update*:
    - all nodes within the segment that got ”reversed”,
    - the neighboring nodes of the segment (two in total).[1]
- An *update* operation for a node is defined as follows:
  - For the inter-route node swap cache matrix:
    - Recompute non-empty deltas between the target node and all nodes currently **outside** of the graph.
    - Delete deltas between the target node and all nodes currently **inside** the graph.[2]
  - For the intra-route node and edge swap cache matrices:
    - Recompute non-empty deltas between the target node and all nodes currently **inside** the graph.

[1] I tried making it work with only 4 node updates for the segment endpoints, but it didn't work out.

[2] It wasn't immediately obvious to me why this is needed, but not doing it caused inconsistencies to emerge in the matrix over time. It makes sense – an inter-route swap between two nodes inside the graph is an illegal operation.

4 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)

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)

5 Visualizations

5.1 TSPA.csv

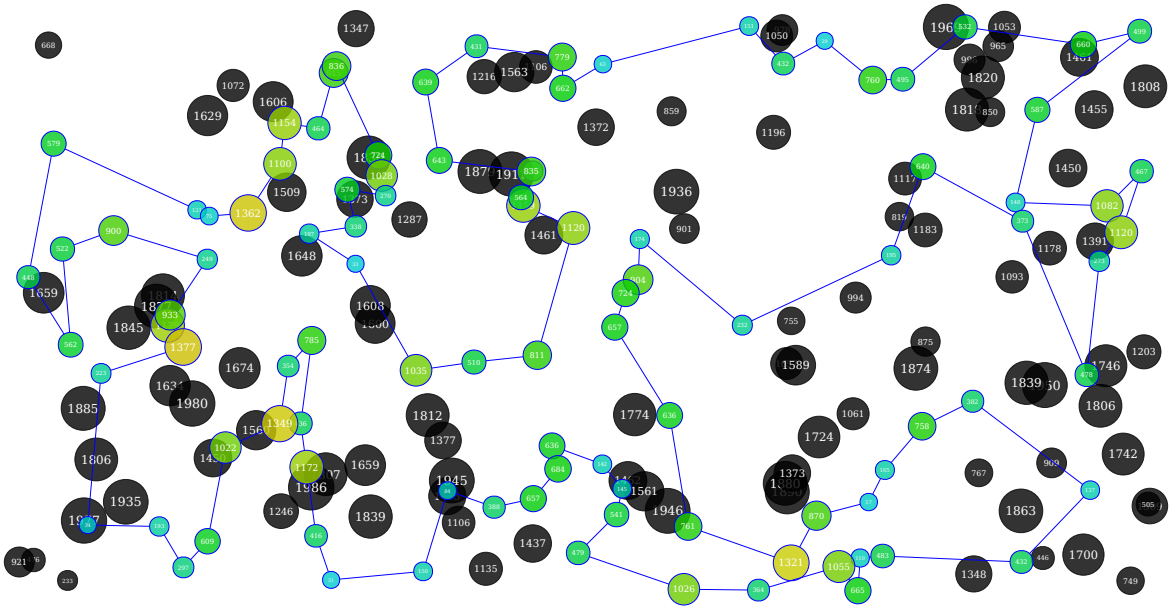

Figure 1: Best lsd-steepest-random solution to TSPA (75,315)

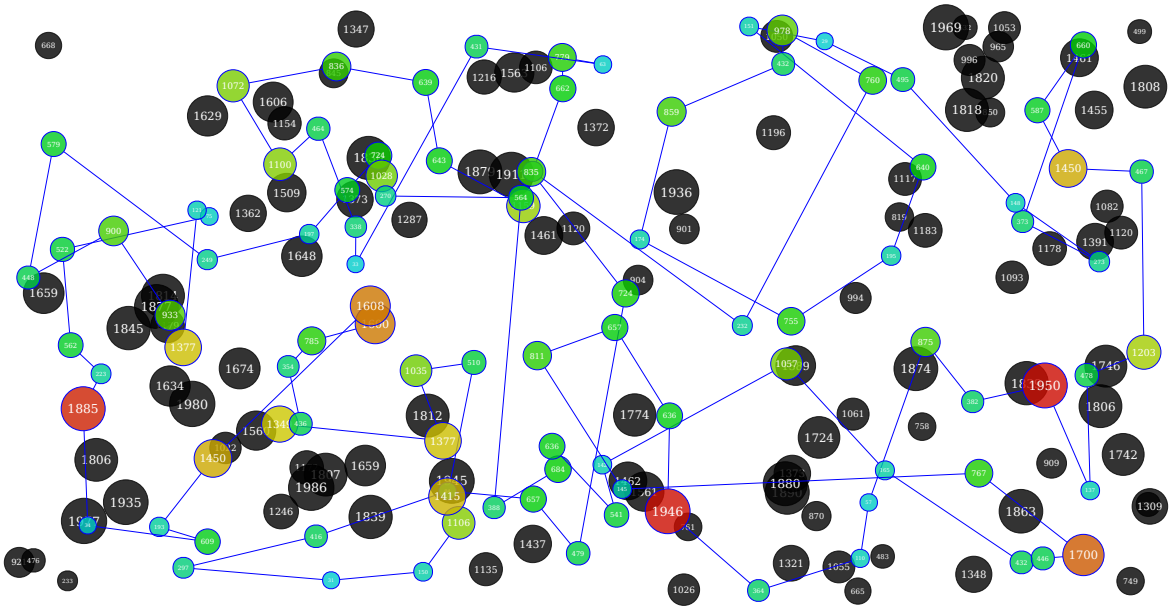

Figure 2: Best lscd-steepest-random solution to TSPA (82,350)

5.2 TSPB.csv

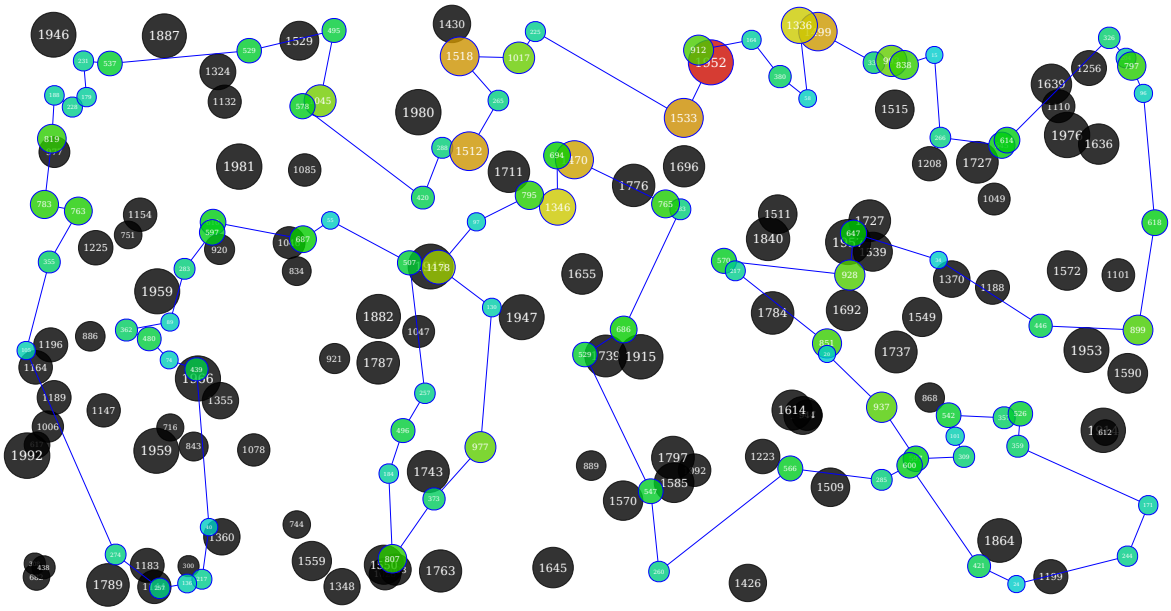

Figure 3: Best lsd-steepest-random solution to TSPB (68,623)

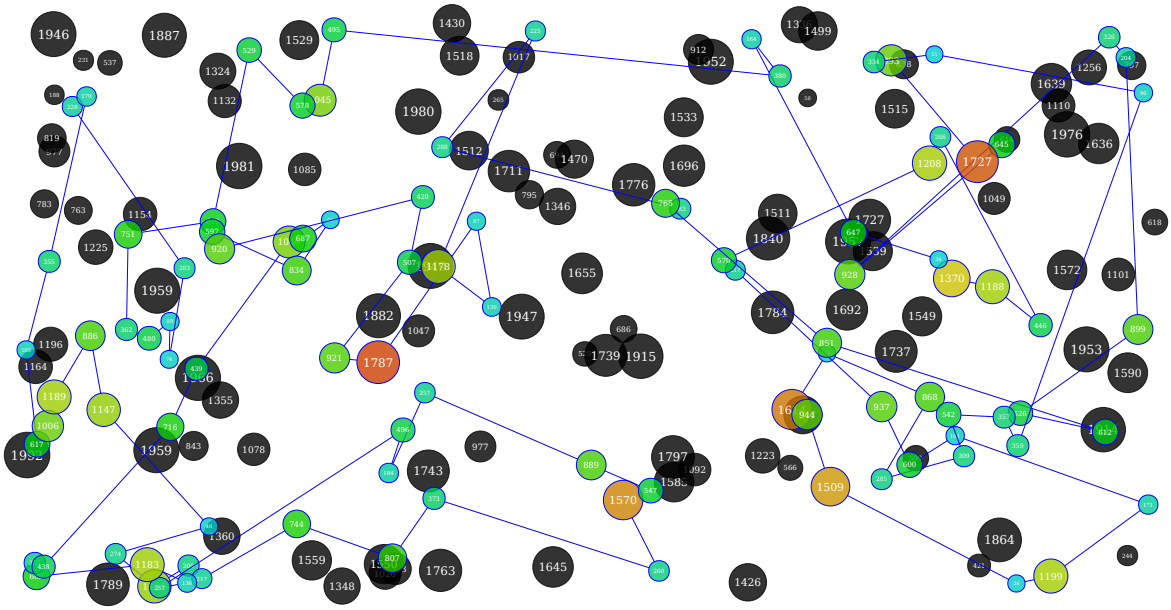

Figure 4: Best lscd-steepest-random solution to TSPB (74,393)

5.3 TSPC.csv

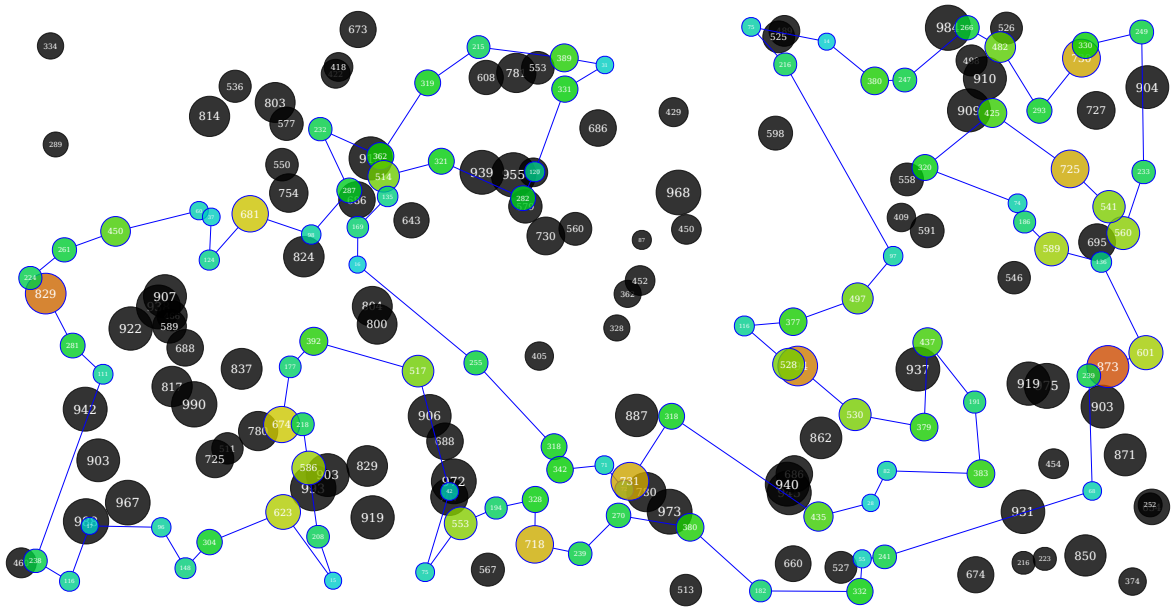

Figure 5: Best lsd-steepest-random solution to TSPC (49,257)

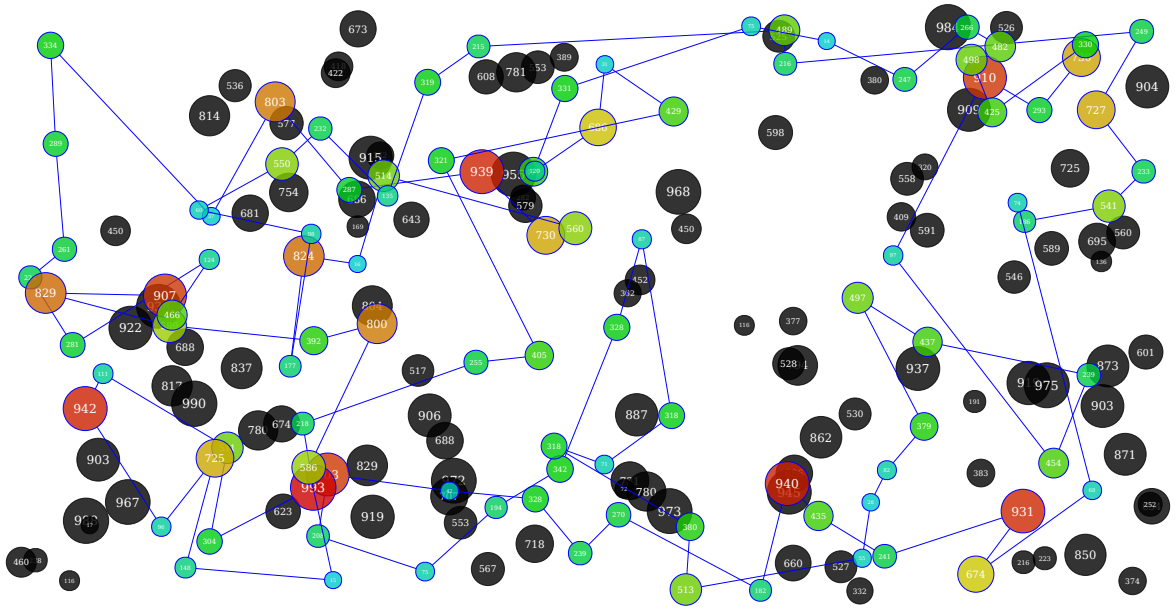

Figure 6: Best lscd-steepest-random solution to TSPC (55,074)

5.4 TSPD.csv

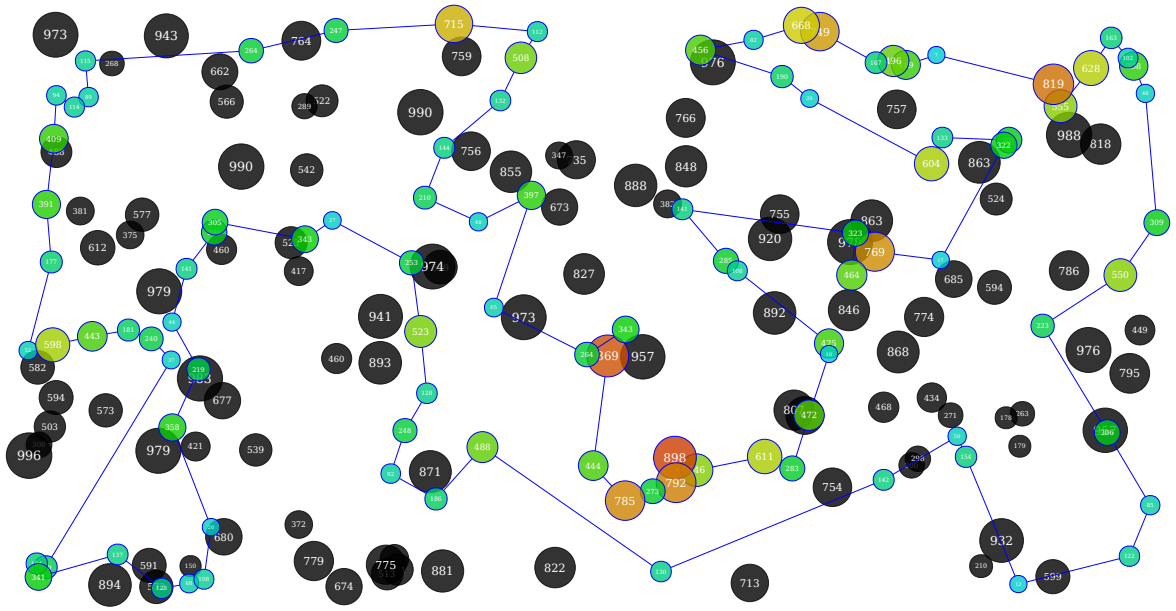

Figure 7: Best lsd-steepest-random solution to TSPD (45,351)

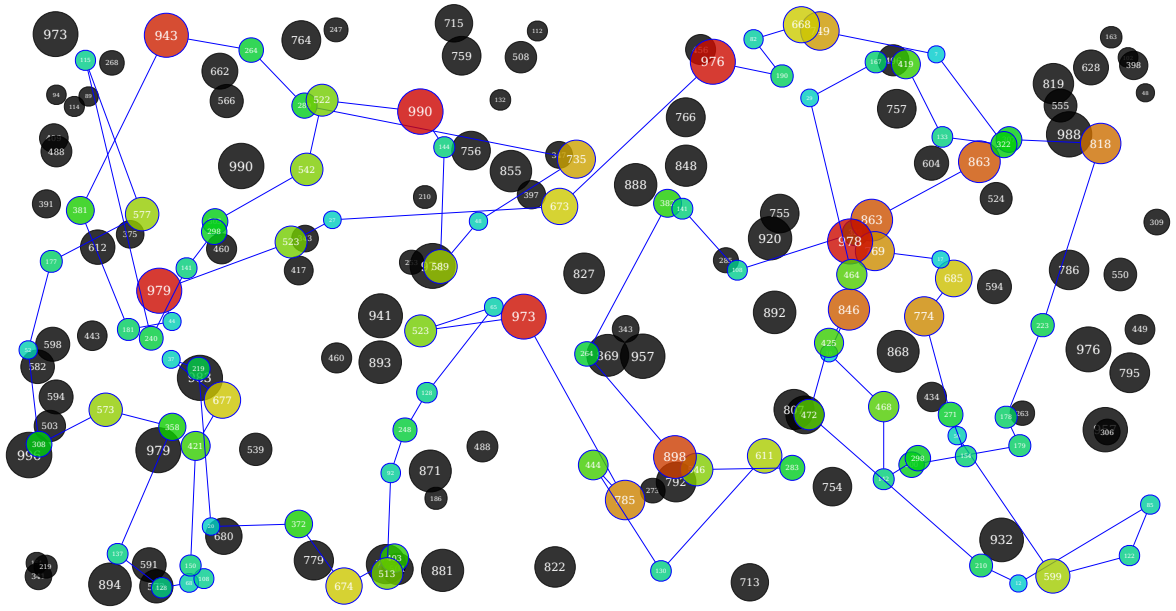

Figure 8: Best lscd-steepest-random solution to TSPD (51,897)

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

To address the elephant in the room, my implementation of delta cache speeds the algorithm *without* candidate moves, but slows down the one *with* candidate moves. This is a surprising result, so I profiled my programs with `gprof` and I'm including the first few lines from their flat profiles below:

`ls-steepest-random`

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
34.43	20.36	20.36	1079710000	0.00	0.00	tsp_graph_evaluate_inter_swap
33.39	40.11	19.75	545253550	0.00	0.00	tsp_nodes_evaluate_swap_nodes
16.38	49.79	9.68	545253550	0.00	0.00	tsp_nodes_evaluate_swap_edges
8.20	54.64	4.85				sp_stack_get
3.59	56.77	2.12	800	0.00	0.07	lsearch_steepest
3.48	58.83	2.06				sp_stack_peek
0.58	59.17	0.34	33694	0.00	0.00	tsp_nodes_swap_edges
0.00	59.17	0.00	512823	0.00	0.00	tsp_nodes_swap_nodes

`lsd-steepest-random`

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
16.03	5.19	5.19	1079710000	0.00	0.00	tsp_graph_evaluate_inter_swap_with_delta_cache
15.90	10.34	5.15	140098289	0.00	0.00	tsp_nodes_evaluate_swap_nodes
10.28	13.68	3.33	148181901	0.00	0.00	tsp_graph_evaluate_inter_swap
10.04	16.93	3.25	545253550	0.00	0.00	tsp_graph_evaluate_swap_nodes_with_delta_cache
9.88	20.13	3.20				sp_stack_get
8.83	22.99	2.86	800	0.00	0.03	lsearch_delta_steepest
8.60	25.78	2.79	545253550	0.00	0.00	tsp_graph_evaluate_swap_edges_with_delta_cache
7.32	28.15	2.37	140098289	0.00	0.00	tsp_nodes_evaluate_swap_edges
5.96	30.08	1.93	1361338	0.00	0.00	tsp_graph_update_delta_cache_for_node
5.81	31.96	1.88				sp_stack_peek

For local search without candidate moves, the time spent inside evaluation functions dropped from 50 seconds to only about 11 seconds!

lsc-steepest-random

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
33.42	4.51	4.51	105882052	0.00	0.00	tsp_nodes_evaluate_swap_nodes
21.27	7.38	2.87	112972806	0.00	0.00	tsp_graph_evaluate_inter_swap
19.79	10.05	2.67	800	0.00	0.02	lsearch_candidates_steepest
5.78	10.83	0.78	56845336	0.00	0.00	tsp_nodes_evaluate_swap_edges
5.26	11.55	0.71				sp_stack_get
4.08	12.10	0.55	31840000	0.00	0.00	tsp_heap_push
1.89	12.35	0.26	8000000	0.00	0.00	tsp_nodes_swap_nodes_adds_candidate
1.70	12.58	0.23	1	0.23	12.56	run_lsearch_algorithm

lscd-steepest-random

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
22.64	3.66	3.66	98267679	0.00	0.00	tsp_nodes_evaluate_swap_nodes
14.23	5.96	2.30	1311342	0.00	0.00	tsp_graph_update_delta_cache_for_node
10.95	7.73	1.77	800	0.00	0.02	lsearch_candidates_delta_steepest
9.59	9.28	1.55	63986193	0.00	0.00	tsp_graph_evaluate_inter_swap
7.92	10.56	1.28	74051064	0.00	0.00	tsp_nodes_evaluate_swap_edges
6.77	11.66	1.10	112972806	0.00	0.00	tsp_graph_evaluate_inter_swap_with_delta_cache
6.68	12.74	1.08				sp_stack_get
5.07	13.56	0.82	105882052	0.00	0.00	tsp_graph_evaluate_swap_nodes_with_delta_cache

With candidate moves enabled, far less time is spent evaluating (about 8 seconds). Caching brings that down to 6.5 seconds, but the gain is simply too small to outweigh the cost of caching.

This report proved exceptionally difficult to finish, so it's good we got 2 weeks for it. I'm not completely happy with my implementation, because I had to cut corners in a few places due to lack of time. It's possible that my code does more updating than strictly needed, thus slowing it down.

Still, I am glad about the results, especially after doing the profiler breakdown.