Lin - Kernighan - Algorithm

I decided to draw inspiration from one of the most successful heuristics for the TSP: The Lin-Kernighan Heuristic, proposed in 1973 by Brian W. Kernighan and Shen Lin. It belongs to the class of k-opt heuristics, where k is newly determined on each iteration. In general, a “k-opt move” corresponds to improving a tour by exchanging k of its edges. My implementation of the algorithm consists of 2 functions: improvePath and Lk.

My ImprovePath function is inspired by pseudocode published by D. Karapetyan and G. Gutin. It takes 6 inputs:
- The current path, P, where we think of one edge as missing, to make out a starting node b and an end node e.
- An Integer depth, denoting the current recursion depth.
- A set of “restricted” vertices, the (newly added) outgoing edges of which we will not remove from our graph
- An Integer alpha.
- Integers improveCounter, maxImprovements which let us ensure that the number of recursive calls is polynomial.

If our recursion depth is less than alpha we consider all edges x->y with weight w(x->y) on our path, where x is not "restricted". For each such edge with positive “provisional gain” w(x->y) - w(y->e), we try to remove the edge x->y, instead adding x->e and inverting all edges between y and e, before "closing the cycle" by adding an edge from y to starting node b.

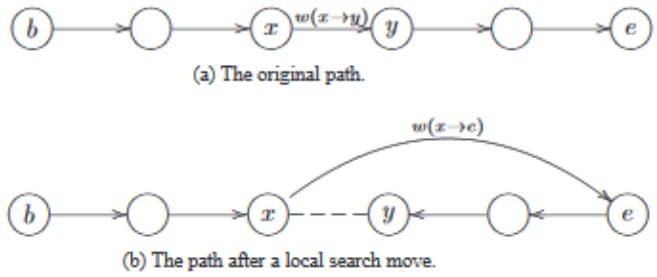


Figure 1: An example of a local search move for a path improvement. The weight of the path is reduced by $w(x \to y) - w(x \to e)$.

If the resulting tour is indeed cheaper than our current one, the function returns the new-found solution immediately. During testing I noticed that this makes the algorithm much faster without generally affecting the “best” solution we find. If this 2-opt move does not yield a faster tour, we try to improve it further (recursively) and, if successful, add it to a list of improvements, the best of which we return if no 2-opt move is successful.

I chose this approach because it balances speed (available 2-opt improvements are accepted immediately) and thoroughness (if no 2-opt improvement is found, more complex improvements are weighed against each other). But considering all potential improvements becomes very inefficient for large graphs. Therefore, we dispense with all paths but the one which gives us the largest “provisional gain” w(x->y) - w(e->x) if our recursion depth is at least alpha, where we choose alpha to be some small integer. I observed alpha=5 to be the most efficient for graphs of reasonable size. Starting at this depth, we only consider one potential improvement, and recursively call improvePath on it, until we find an improvement to our original tour, or until no available path offers a “provisional gain”.

Note that there is no guarantee that the improved tour improvePath returns is the best we can do. Choosing a different permutation as input to our first improvePath call will make the algorithm consider different changes, in a different order. The intuition behind my “LK” function is that we try to improve our current path again and again, cyclically choosing a different node as starting point of our tour, until we have failed to find an improvement after a certain number (m, which I chose to equal the number of nodes for my implementation) of attempts. At this point we accept the solution the algorithm has produced thus far.

We shuffle the current permutation of the cities at the start of an LK call. That is because the solution the algorithm finds depends on the “first path it tries to improve”. It is therefore reasonable to attempt running LK more than once.

The worst-case time complexity of the LK-heuristic is conjectured to be exponential, although in 2009, no trivial upper bound had been found (Helsgaun, 2009). To counter-act this, I added a counter which forces the algorithm to

terminate after n^(2.5) improvePath calls. This specific bound was chosen based on empirical evidence and lets the algorithm finish naturally almost always, since the exponential worst case is very rare, as has been noted in the literature. Ignoring recursion, iterations of improvePath have time complexity O(n^2) for depth<alpha, since a path cost may be computed for each edge. Calls with depth >= alpha are O(n), since computing the provisional gain is O(1) for each of our n current edges, and only one total path cost is computed. Since each improvePath call is counted individually, this leads to an upper bound on the execution time of O(n^(4.5)).

For sufficiently large n, this bound may become unreasonable (as we will see in the Experiments section). More sophisticated implementations (using various metaheuristics etc.) have managed to reduce the average running time of LK to O(2.2) (Helsgaun, 2000).

Let's briefly discuss the pros and cons of my implementation:
Its random nature could be a disadvantage: On one hand, the "best" result it finds is often considerably better than the one found by our previous heuristics. On the other hand, we cannot put an upper bound on the number of times we need to run LK until this best result is returned. Each iteration of LK is reasonably quick for graph sizes of up to a few hundred nodes, but the algorithm may run indefinitely without us ever knowing whether it will eventually spit out an even better solution.
On the upside, this may not be so bad in practice since we can simply let the algorithm run repeatedly until we have obtained a satisfactory solution. Furthermore, adjusting the values of parameters like alpha, maximum recursion depth beta, or the maximum number of "idle iterations" needed for LK to terminate, may help us adjust the algorithm to our needs.
And while my implementation may not be feasible for very large graphs, the results it produces are of a high quality and give it legitimacy to be used for problems it can deal with in an acceptable time frame.

More sophisticated modifications to the LK-heuristic have been made, including "meta-heuristics", heuristic rules which further limit the search to make it faster, or rules that guide the order in which potential improvements are attempted – This is where I could adjust my implementation to enable it to deal with even larger graphs.

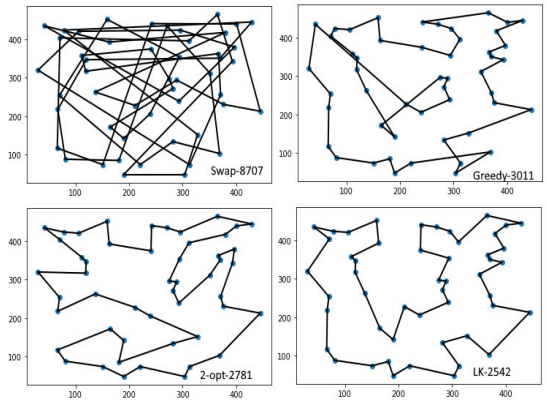
Experiments

In this section, I will summarize the findings I made when conducting computational experiments on our heuristics. To test my LK-implementation, I will note best, worst and median results returned by it on 100 attempts (see function testLK), due to its random nature. I will offer a randomly generated "seed" which may be given to the LK function as an input in order to reproduce the "best" result.
The evaluation platform is based on an Intel Core i7 2.70GHz processor.
I will present my test results as tables, each entry of which will be of the form (cost):(running time in ms).

First, I will evaluate the results achieved by the different heuristics on the TSPs we have been provided with. Example seeds giving the "min" solutions for LK are 0 for sixnodes, 0/74/182 for cities25/50/75.

	swap	2-opt	Swap&2-opt	greed	LK(min)	LK(med)	LK(max)
sixnodes	9 : <1 ms	9 : <1 ms	9 : <1 ms	8 : <1 ms	8 : <1 ms	8 : <1 ms	9 : 1 ms
cities25	5027 : <1	2211 : 2	2233 : 2	2587 : <1	1993 : 8	2000 : 11	2125 : 14
cities50	8707 : 1	2781 : 9	2686 : 12	3011 : 1	2542 : 48	2611 : 64	2710 : 61
cities75	13126 : 2	3354 : 30	3291 : 31	3412 : 1	3029 : 201	3069 : 319	3230 : 156

For the sixnodes case, we find that only the tour [0, 3, 4, 1, 2, 5] (ignoring the choice of starting node) has cost 8. The identity permutation has cost 9, so for Swap or 2-opt to find this solution, a single swap / 2-opt move would have to transform the identity permutation to said solution, which we can check is not the case. The Greedy and (almost always) the LK approach however find this optimal solution very quickly.
For the Euclidean "cities" cases, the fast swap-heuristic gives poor results. Further experiments will let us infer if using it beforehand makes 2-opt generally more efficient. The greedy heuristic is more successful – it does not generally compete for "the optimal solution", but its runtime is extremely short. 2-opt beats the greedy heuristic by 1.7 - 14.5%, still being rather fast. LK beats 2-opt by roughly 5 to 10% if we take its best solution in 100 attempts, but that approach will be unfeasible for larger graphs. Luckily, even the median and worst solutions it finds are competitive.
Plotting the corresponding tours (e.g. for cities50) may give us some further insight into the quality of the heuristics.



Our swap-heuristic on its own fails to find a good tour.

The greedy algorithm is more sensible but has a problem: When making its way around the graph, it "misses" nodes it then needs to get back to when no nearby nodes remain, potentially at a high cost.

2-opt removes all "over-lapping" edges in Euclidean scenarios. To see why, note that it will consider each pair of overlapping edges a-b, x-y and find that the non-overlapping edges a-x, b-y have a smaller combined cost.

Unlike LK, 2-opt however fails to consider more complex moves which may improve the tour further. I conjecture that the 2542-solution the LK function (sometimes) returns is indeed optimal.

To really assess how close our heuristics bring us to the optimal solution, I have generated graphs for which the optimal solution is known. I based my work on an article by Jeffrey L. Arthur and James O. Frendewey (1988).

First I will consider a set of non-metric TSP instances I generated using my function generalTSP which writes a problem declaration to a file "general_graph_X" and returns the graph that corresponds to it (see tests.py for an explanation).
To reproduce the 5 following graphs, run generalTSP(n=N, R=50, rho=0.1, sigma=0.25, seed=100).
They should then have the names "general_graph_X" where X = 904 / 150 / 989 / 345 / 749.
5 corresponding seeds for LK (min) are 505 / 930925 / 4126984 / 19803481 / 40314600.

	2-opt	Swap&2-opt	Greedy	LK(min)	LK(med)	LK(max)	Best	Iden
N = 20	356 : 1 ms	351 : 3 ms	383 : <1 ms	333 : 3 ms	333 : 8 ms	357 : 7 ms	333	602
N = 100	1755 : 43	1782 : 34	1787 : <1	1658 : 744	1685 : 716	1708 : 750	1646	3384
N = 200	3564 : 156	3598 : 204	3622 : 3	3405 : 8747	3437 : 5505	3472 : 5279	3365	6713
N = 300	5442 : 593	5458 : 599	5575 : 6	5235 : 39630	5273 : 34705	5309 : 30966	5172	10172
N = 500	9018 : 1905	9055 : 1790	9121 : 20	8713 : 126307	8748 : 111012	8768 : 100150	8583	17285

Note: Choosing a larger R further increases the running-time of LK, while the other heuristics are largely unaffected by this. This is a shortcoming of my implementation and likely due to the many arithmetic operations on path costs. I assessed the quality of the solutions in terms of where they lie on the interval from the "best cost" to the (presumably very large) cost of the identity permutation. I colored solutions within the "best" 3% green, those within 7% yellow, those within 10% orange and others red.

For graphs of all sizes, LK offers better solutions than the other heuristics. However, running LK 100 times and picking the cheapest solution quickly becomes unbearably inefficient. This is where 2-opt shines: Its solutions are less than 5% more expensive than the best tour found by LK, but for graphs with 500 nodes it is about 500 times as fast as a single iteration of LK. Having conducted these experiments, we can let go of the theory that running the swap heuristic before 2-opt generally has a positive effect on the discovered solution. Like LK, the starting permutation however seems to affect the outcome 2-opt gives in more subtle ways (A different "local minimum" is found).

My function metricTSP produces Travelling Salesman Problems for which the triangle inequality holds.
I will conduct experiments on them to see if our findings differ from the non-metric scenarios.

To reproduce the 5 following graphs, run metricTSP(n=N, R=50, rho=0.1, sigma=0.25, seed=100).
They will correspond to the files "metric_graph_X" where X = 904 / 150 / 989 / 345 / 749.
5 corresponding seeds for LK (min) are 1070/373990/5032921/3714990/79514499.

	2-opt	Greedy	LK(min)	LK(med)	LK(max)	Best	Iden
N = 20	346 : < 1 ms	362 : <1 ms	333 : <1 ms	337 : <1 ms	348 : <1 ms	333	424
N = 100	1737 : 38	1769 : <1	1651 : 1703	1667 : 1347	1684 : 999	1646	2244
N = 200	3488 : 195	3562 : 8	3382 : 21354	3407 : 13477	3425 : 15816	3365	4530
N = 300	5343 : 472	5464 : 6	5205 : 107799	5232 : 113091	5251 : 63629	5172	6930
N = 500	8882 : 1309	9035 : 21	8668 : 584408	8684 : 434476	8697 : 451004	8583	11608

Note: Due to the new upper bound I place on distances in my metric generator, even arbitrary permutations tend to be less far away from the optimal solution than in problems generated by generalTSP, as evidenced by the "Identity" column. Because this makes the interval from the cost of the optimal tour to the cost of the identity matrix much smaller, I am changing the relevant bounds from 3/7/10% to 5/10/17% for green/yellow/orange evaluations.

Other than the observation that, for my metric scenarios, paths tended to be closer to the optimum in general, I could not notice a significant change in the quality of the solutions offered by the different heuristics. Notably however, my LK-implementation was more than twice as slow as in the non-metric case. In fact, I found that the number of improvePath calls my implementation uses for metric scenarios of size n was significantly larger for non-metric graphs. This implies it is harder for LK to find improvements in the metric case, which is interesting and could motivate further research into why exactly this occurs.

Lastly, producing non-trivial Euclidean TSP instances for which we know the optimal solution, that are not simply convex polygons, is rather difficult and an on-going topic of research. Fortunately, being able to plot tours can let us guess how "close to the optimum" the solutions given by our heuristic really are, so I will omit this supplement.
I believe that the preceding experiments have given sufficient insight into the quality and efficiency of our heuristics for graphs of different sizes and types.

References:

Lin, S. and Kernighan, B., ‘An Effective Heuristic Algorithm for the Traveling-Salesman Problem’, 1973

Karapetian, D. and Gutin, G., ‘Lin-Kernighan Heuristic Adaptations for the Generalized Traveling Salesman Problem’, 2011 (Especially the pseudocode under “Algorithm #1”)

Helsgaun, K., ‘An Effective Implementation of the Lin-Kernighan Heuristic’, 2000

Helsgaun, K., ‘General k-opt submoves for the LinKernighan TSP heuristic’, 2009

Arthur, J., Frendewey, J., ‘Generating Travelling-Salesman Problems with Known Optimal Tours’ in ‘The Journal of the Operational Research Society, Vol. 39, No. 2’, 1988