LKH, Q-LKH, NeuroLKH and further

Jens Matthias Hartsuiker
jenshartsuiker@gmail.com
0001026691

February 2023

1 Introduction

For this project work, we decided to tackle the Traveling Salesman Problem (TSP) due to our long-standing interest in the topic. Our inspiration came from two recent papers VSR-LKH [1] and NeuroLKH [2] that augment the LKH solver (Lin-Kernighan-Helsgaun) [3], one of the current best heuristic solvers for TSP, with either deep- or reinforcement learning techniques.

In the next section, we will introduce the data on which we conducted our tests. Following that, we will provide a simple description of how the LKH solver functions.

Then we briefly explain how the VSR-LKH [1] solver improves the LKH solver by enhancing the local search with standard reinforcement learning and we hypothesized that we could further improve the performance of the VSR-LKH solver by incorporating deep reinforcement learning. In the section thereafter we discuss our results.

Additionally, we introduce the NeuroLKH solver [2] by briefly explaining how it improves the LKH solver by providing better "building blocks" to the local search and we explore the idea of combining it with the ideas presented in [1], which proved to be an intuitive step. The results are presented in the section thereafter.

Finally, we summarize our findings in the last section of this report.

2 Data

For our data, we used the TSP lib data provided in the LKH solver package [3], which can also be directly found at [4] under the heading "Symmetric traveling salesman problem (TSP)". Although the origin of the data is not provided, it is known that TSPLIB is a library of sample instances for the TSP (and related problems) from various sources and of various types [4]. The data set contains a little over a hundred instances.

When training our deep method, more details in section 4, we used all the entries in this library, except those included in the figures below.

3 The LKH solver

In this section, we provide a brief description of the LKH solver [3]. The LKH solver is based on the idea of edge swaps. For a given valid configuration of edges in a TSP instance, there are edges that could potentially shorten the solution. For example, from the perspective of one of the nodes, an edge that is currently being used is not one of the two shortest edges

that the node has. However, it is ambiguous which edges to swap, which edges to swap first, and which ones to replace them with. The LKH solver implements a local search approach that finds local optima. In the standard configuration, the solver does at most a 5-opt swap, which involves removing five edges and inserting five other edges, but this is configurable.

The order of this local search is dependent on the "alpha value" calculated in an earlier stage, which is ultimately dependent on the distance value of the edges. Based on these alpha values, "candidate sets" of edges are constructed, one for each node. The maximum size of these sets is configurable, but the standard value is five. The solver has two phases: first, calculating the alpha values, and second, the local search (swapping) phase. Note that this local search is a deterministic process. The construction heuristic, the algorithm responsible for providing a starting point for the swaps, is non-deterministic and contains random elements. Therefore, the solver returns different solutions between different runs.

Approaching the problem in this way leads to a significantly smaller search space, and it is exactly this reduction that helped [3] and the two papers [2, 1] to achieve their success, even though "brute" deep- and reinforcement learning methods have been attempted before, they were not successful.

4 VSR-LKH and Deep Q-LKH

In [1], the authors improve the local searching process of the LKH by replacing the deterministic edge selection process based on alpha values by a mixture of various (standard) reinforcement learning methods (see [1] for more details). In this work, we focus on their Q-learning implementation (referred to as Q-LKH from here on) for the sake of simplicity. Our goal is to replace the standard Q-learning method with a deep Q-learning method.

Normally, in the problems where the table of Q-values can be contained in memory, like this one, standard Q-learning techniques are preferred over deep Q-learning methods as the standard method converges quicker to the optima. In this respect applying a deep reinforcement method to our problem would not be a good idea.

However with standard Q-learning, the model has to learn all information from scratch for every instance, while a deep Q-learning method can retain/share learned information between different instances. In this respect applying a deep reinforcement method to our problem could be a good idea.

We hypothesise that the ability of a deep method to share information between instances could outperform the slower converging speed w.r.t. to standard reinforcement technique such that the overall performance in terms of the gap w.r.t. the optimal solution and the runtime is better.

The most logical approach to integrating a neural network for this problem would be to directly integrate it within the LKH solver, using a library such as [5]. However, integrating a neural net within the LKH code (which is in C), getting familiar with a new library such as [5], and then training the neural net requires time and resources that we did not have for this project work.

Instead, as a proof of concept, we extracted the Q-values after every 10 runs for each instance of the normal Q-LKH (instances out of our training set only), and then trained a feedforward network on these values in Google Colab. Although this is a "hacky" implementation and may be less accurate, it does offer more analytical possibilities with regard to loss and overfitting. For the testing phase, we predicted the Q-values first in Google Colab

and then loaded them into the normal Q-LKH solver. We refer to this process as the Deep Q-LKH implementation.

To reiterate, the Q-LKH algorithm replaces the deterministic local search process based on alpha-values of the LKH solver by a standard reinforcement learning algorithm based on Q-values. This often outperforms the original search [1]. This could be caused either by the reinforcement method finding local optima sooner (a shorter local search path) or finding different local optima. When implementing our (hacky) deep method our best possible result would be that our model predicts the Q-values in such a way that they are equal to the Q-values of the local optima and thus instantly provide the optimal solution. However it also possible that our deep implementation will give worse results. Our model could predict the Q-values in such a way that the reinforcement process actually converges to a worse local optima than that it would converge to with the starting Q-values as described in [1]. We do not deem this particularly likely, but it is possible.

Training the network only took a few minutes.

5 Deep Q-LKH Results

In this section, we will compare the LKH, Q-LKH, FixQ-LKH (the Q-LKH method with the improved start alpha value, but no learning, see [1]), and our deep implementation in both terms of gap to the optimal solution and in runtime. We will also present the MSE train, validation, and test loss afterward.

For Figures 1 and 2 below, please note that the lines that do not initially appear in the graph have a cumulative sum of zero until they do appear in the graph.

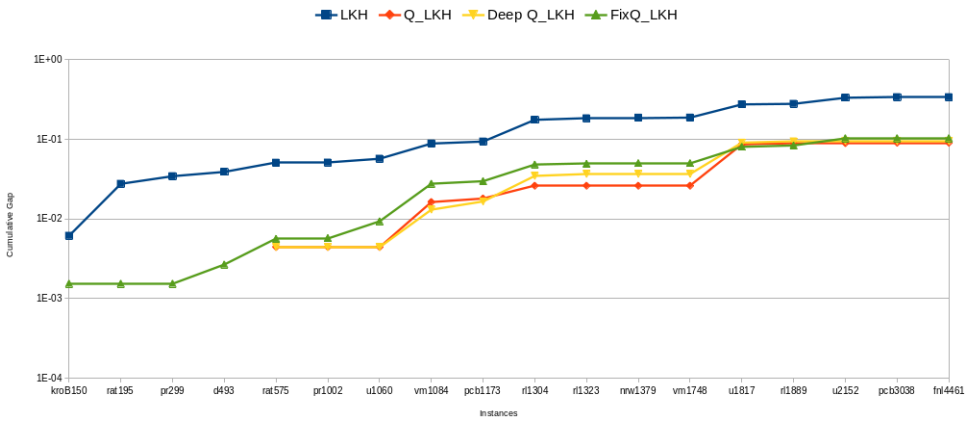

Figure 1: The cumulative gap w.r.t. the optimal solution of our deep Q method compared to various other methods.

Instance	LKH	Q_LKH	Deep Q_LKH	FixQ_LKH
kroB150	0.0062	0.0000	0.0000	0.0015
rat195	0.0277	0.0000	0.0000	0.0015
pr299	0.0345	0.0000	0.0000	0.0015
d493	0.0391	0.0000	0.0000	0.0027
rat575	0.0509	0.0044	0.0044	0.0056
pr1002	0.0512	0.0044	0.0044	0.0058
u1060	0.0572	0.0044	0.0044	0.0094
vm1084	0.0888	0.0163	0.0131	0.0278
pcb1173	0.0940	0.0181	0.0166	0.0299
rl1304	0.1764	0.0262	0.0348	0.0484
rl1323	0.1841	0.0262	0.0368	0.0494
nrw1379	0.1876	0.0262	0.0368	0.0503
vm1748	0.1880	0.0262	0.0368	0.0503
u1817	0.2756	0.0869	0.0910	0.0807
rl1889	0.2799	0.0897	0.0943	0.0840
u2152	0.3339	0.0897	0.0960	0.1035
pcb3038	0.3392	0.0908	0.0960	0.1035
fnl4461	0.3394	0.0908	0.0960	0.1035

Table 1: The raw data corresponding to figure 1

In Figure 1 above, regarding the gap (the difference between the given and optimal solution), we see that FixQ-LKH, Q-LKH, and Deep Q-LKH all outperform LKH by a factor of about 4. Then, we also see that the FixQ-LKH is slightly outperformed by Q-LKH and Deep Q-LKH. Sadly, we see that Deep Q-LKH does not perform better than Q-LKH; instead, it is slightly worse. However, we are hopeful for improvements with this deep technique, as explained later in this section.

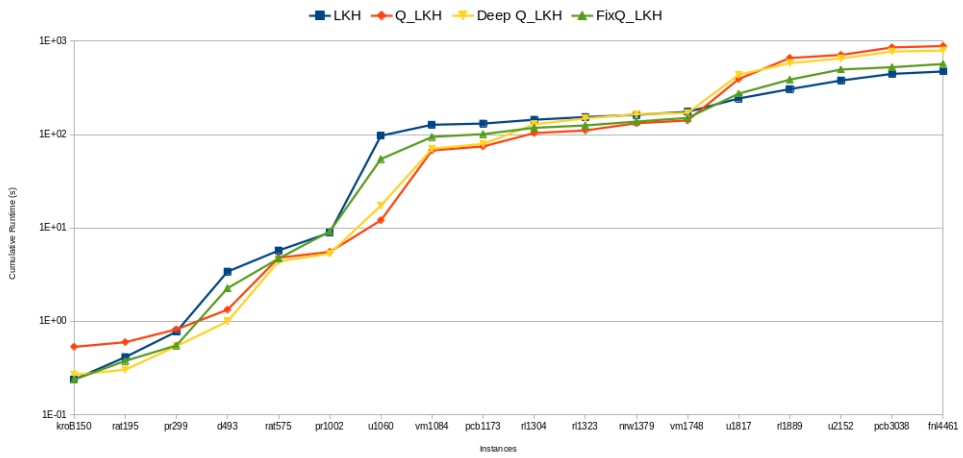

Figure 2: The cumulative runtime of our deep Q method compared to various other methods.

Instance	LKH	Q_LKH	Deep Q_LKH	FixQ_LKH
kroB150	0.2390	0.5340	0.2690	0.2380
rat195	0.4150	0.5990	0.3040	0.3780
pr299	0.7760	0.8220	0.5440	0.5530
d493	3.4120	1.3350	1.0000	2.2680
rat575	5.7410	4.7970	4.3950	4.7160
pr1002	8.9170	5.5480	5.3180	9.2010
u1060	97.2410	12.0900	17.3820	54.7580
vm1084	127.4450	67.6570	70.5220	94.4030
pcb1173	131.2140	74.7330	79.5770	101.1760
rl1304	144.3070	103.7600	128.6640	118.1870
rl1323	154.0430	110.7580	149.0770	125.2680
nrw1379	162.9110	132.2890	165.4140	137.8840
vm1748	175.1500	141.6330	170.2110	150.8850
u1817	243.7000	393.2440	436.0370	274.4370
rl1889	306.7690	660.7550	582.9030	387.9660
u2152	379.0140	711.7310	651.7780	497.4200
pcb3038	446.5680	857.4210	773.6470	526.1750
fnl4461	473.6330	886.4950	793.1430	569.6340

Table 2: The raw data corresponding to figure 2

In Figure 2 above, regarding the runtime, we have a situation analogous to that of the gap above: the more sophisticated the method, the longer the runtime. However interesting to see is that the deep method has a shorter runtime than the standard reinforcement method. In general the improvement in performance seems to be more or less in line with the increase in runtime. It would be up to specific use cases to determine which method to use. Note that the training of the neural network is not included in the runtime.

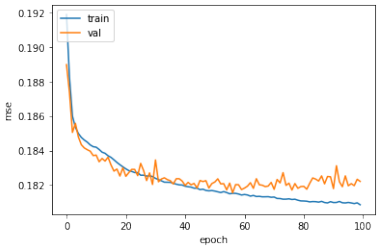

Figure 3: MSE score while training Deep Q-LKH.

As seen in Figure 3 above, there is no overfitting occurring in our model. However, the MSE score is far from optimal, and the MSE score on the test set is 0.183. It is our hope that, in a possible future work, a neural network could be directly implemented within the LKH solver. In the training phase, this will certainly slow down the solver, but hopefully, it will lead to much better performance.
The fact that even with this "hacky" implementation, we already have almost equal performance to the normal Q-LKH we interpret as a good sign. This, as explained in the previous section, because it was also possible that our method would perform worse than the Q-LKH method.

Lastly, keep in mind that this is just the Deep Q-LKH method. We hypothesize that we can achieve better results with a deep VSR-LKH method.

6 NeuroLKH and Merging Q-LKH with NeuroLKH

Paper [2] instead improves the LKH algorithm by providing more accurate/better alpha values, which in turn leads to better candidate sets. This improvement is achieved by utilizing deep neural networks. These candidate sets and alpha values are then fed into the LKH solver, and this entire process is referred to as the NeuroLKH solver. Note that NeuroLKH treats the LKH solver, or more precisely the local search process within the LKH solver, as a blackbox.

Thanks to [2] using deep methods, information is being shared between instances. Although the NeuroLKH and our deep Q-LKH implementation are very similar, the objective of our deep Q-LKH model is fundamentally different from what NeuroLKH aims to achieve. We use deep learning methods in the Q-LKH method because we hope that the model can learn search patterns within the local search process. On the other hand, NeuroLKH aims to provide better building blocks for the local search process to start with, without improving the search process itself. In other words, the NeuroLKH method improves the candidate set, while the (deep) Q-LKH method utilizes the candidate set more effectively.

It was at this point that we realized that these methods complement each other perfectly, and we sought to merge them. We were then also surprised to find out that to our knowledge, this had not been attempted before.

Because NeuroLKH treats LKH as a blackbox, it is sufficient to substitute the LKH code in the NeuroLKH package with the Q-LKH code.

7 Merging Q-LKH and NeuroLKH Results

In this section, we will compare the LKH, Q-LKH, FixQ-LKH, NeuroLKH, and our merged implementation in both terms of gap to the optimal solution and in runtime.

Please note that for figures 4 and 5, the lines that do not initially appear in the graph have a cumulative sum of zero until they do appear.

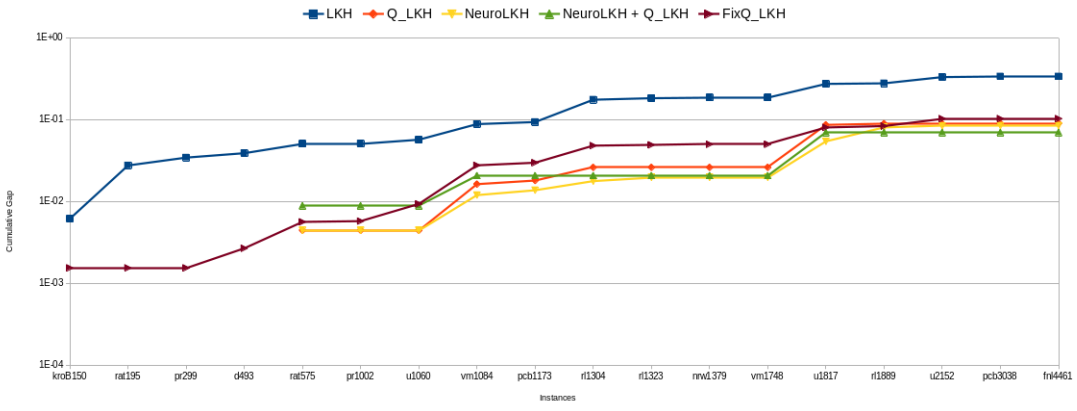

Figure 4: The cumulative gap w.r.t. the optimal solution of our merged method compared to various other methods.

Instance	LKH	Q_LKH	NeuroLKH	NeuroLKH + Q_LKH	FixQ_LKH
kroB150	0.0062	0.0000	0.0000	0.0000	0.0015
rat195	0.0277	0.0000	0.0000	0.0000	0.0015
pr299	0.0345	0.0000	0.0000	0.0000	0.0015
d493	0.0391	0.0000	0.0000	0.0000	0.0027
rat575	0.0509	0.0044	0.0044	0.0089	0.0056
pr1002	0.0512	0.0044	0.0044	0.0089	0.0058
u1060	0.0572	0.0044	0.0044	0.0089	0.0094
vm1084	0.0888	0.0163	0.0120	0.0208	0.0278
pcb1173	0.0940	0.0181	0.0138	0.0208	0.0299
rl1304	0.1764	0.0262	0.0178	0.0208	0.0484
rl1323	0.1841	0.0262	0.0198	0.0208	0.0494
nrw1379	0.1876	0.0262	0.0198	0.0208	0.0503
vm1748	0.1880	0.0262	0.0198	0.0208	0.0503
u1817	0.2756	0.0869	0.0546	0.0701	0.0807
rl1889	0.2799	0.0897	0.0809	0.0701	0.0840
u2152	0.3339	0.0897	0.0844	0.0701	0.1035
pcb3038	0.3392	0.0908	0.0851	0.0701	0.1035
fnl4461	0.3394	0.0908	0.0851	0.0701	0.1035

Table 3: The raw data corresponding to figure 4

In figure 4 we see that LKH is the inferior method, followed mostly by FixQ-LKH. Q-LKH comes next, followed by NeuroLKH, which is outperformed by the merge of Q-LKH and NeuroLKH. Although the merge outperforms the other methods in the end, the margins between these are minimal and performance is comparable.

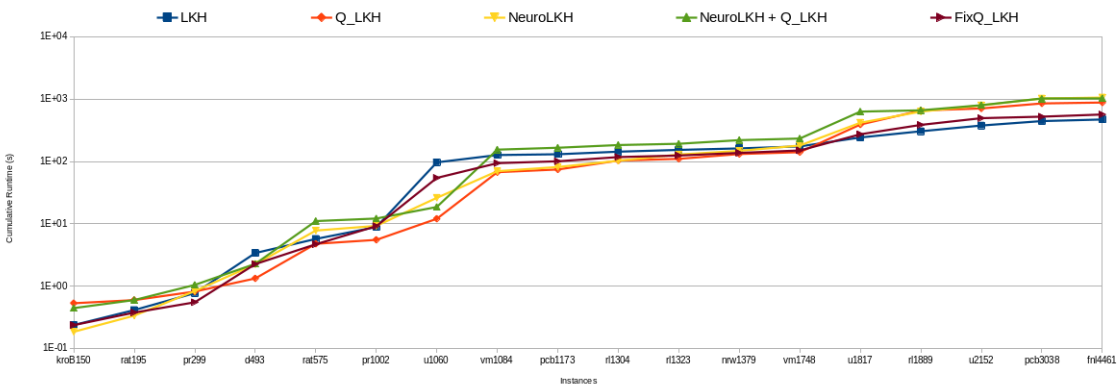

Figure 5: The cumulative runtime of our merged method compared to various other methods.

Instance	LKH	Q_LKH	NeuroLKH	NeuroLKH + Q_LKH	FixQ_LKH
kroB150	0.24	0.53	0.19	0.45	0.24
rat195	0.42	0.60	0.34	0.60	0.378
pr299	0.78	0.82	0.84	1.05	0.553
d493	3.41	1.34	2.24	2.29	2.268
rat575	5.74	4.80	7.83	11.10	4.716
pr1002	8.92	5.55	9.29	12.20	9.201
u1060	97.24	12.09	26.23	18.69	54.758
vm1084	127.45	67.66	70.19	155.51	94.403
pcb1173	131.21	74.73	81.85	166.81	101.176
rl1304	144.31	103.76	103.22	184.37	118.187
rl1323	154.04	110.76	129.93	193.23	125.268
nrw1379	162.91	132.29	148.60	220.97	137.884
vm1748	175.15	141.63	181.12	234.42	150.885
u1817	243.70	393.24	420.24	633.85	274.437
rl1889	306.77	660.76	633.83	665.12	387.966
u2152	379.01	711.73	797.32	803.65	497.42
pcb3038	446.57	857.42	1026.34	1011.82	526.175
fnl4461	473.63	886.50	1066.17	1036.84	569.634

Table 4: The raw data corresponding to figure 5

In figure 5 we arrive at the same conclusion as in the last results section: the more sophisticated the method, the longer the runtime. The improvement in performance seems to be somewhat in line with the increase in runtime. Nice to see is that the merge method performs better than the NeuroLKH method, and its runtime is shorter as well. However w.r.t. to the Q-LKH approach, the increase in runtime is noteworthy while the gain in terms of gap is minimal. It would thus be up to specific use cases to determine which method to use.

Keep in mind that this is just the merge with the Q-LKH method. It is imaginable that we can achieve better results with a merge with the VSR-LKH method.

8 Conclusion

In this section we will reiterate our conclusions and provide our recommendations for future works.

Our conclusions:

- Our deep Q-LKH method slightly under-performs w.r.t. the Q-LKH method in terms of gap to the optimal solutions. However performance is very similar.
- Our deep Q-LKH method slightly outperforms the Q-LKH method in terms of runtime. However performance is very similar.
- Our merged method slightly outperforms the Q-LKH and NeuroLKH methods in terms of gap to the optimal solutions. However performance is similar.
- Our merged method slightly outperforms the NeuroLKH method in terms of runtime. However performance is very similar.

Although our results are not as spectacular as we had hoped for, we do think our results provide justification for further research in the topic: the merge method outperforms the NeuroLKH method in terms of gap and runtime, although marginally, and the deep Q-LKH method performs on par with the Q-LKH method while still having room for improvement. Below we provide some ideas:

- Implement a thorough Deep Q-LKH model, in the case this is a success, investigate a Deep VSR-LKH possibility.
- Research if improvements can be made w.r.t. merging Q-LKH and NeuroLKH, i.e. perhaps bigger candidate sets can be sustained in this merge configuration.
- Merge NeuroLKH with the entire VSR-LKH instead of just Q-LKH.

For anyone continuing this work we recommend strongly to study [3, 1, 2] well. With deep knowledge of these papers it will be intuitive to understand what we have done in our work.

Lastly, source code files related to our work, solely intended for reference, not working code, can be found on our Github [6].

References

[1] J. Zheng, K. He, J. Zhou, Y. Jin, and C.-M. Li, "Combining reinforcement learning with lin-kernighan-helsgaun algorithm for the traveling salesman problem," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 14, pp. 12 445–12 452, May 2021. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/17476

[2] L. Xin, W. Song, Z. Cao, and J. Zhang, "Neurolkh: Combining deep learning model with lin-kernighan-helsgaun heuristic for solving the traveling salesman problem," in *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[3] K. Helsgaun, "Lkh solver source." [Online]. Available: http://webhotel4.ruc.dk/ keld/research/LKH/

[4] G. Reinelt, "Tsp lib source." [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

[5] L. van Winkle, "C neural network library: Genann." [Online]. Available: https://codeplea.com/genann

[6] J. Hartsuiker, "Jensonah/neuro_vsr_lkh." [Online]. Available: https://github.com/Jensonah/Neuro_VSR_LKH