An efficient graph neural network to solve the Traveling Salesman Problem : report

Sacha Braun
sacha.braun@polytechnique.edu
École Polytechnique
Palaiseau, France

Abstract

The Traveling Salesman Problem (TSP) is a classic optimization problem in the field of computer science, operations research, and mathematics. It is a combinatorial problem that can be formulated as follows: Given a set of cities and the distances between each pair of cities, the objective is to find the shortest possible tour that visits each city exactly once and returns to the starting city. This problem is studied in graph theory as it can be reformulated as finding the shortest n-cycle in a complete weighted graph of size n.
Many problems in various fields can be reformulated as a TSP, such as DNA sequencing. However, the TSP is known to be NP-hard. Therefore, finding efficient algorithms able to return sub-optimal solutions for the TSP is a of a particular interest and useful and many research areas.
In this report, we study a a framework to find sub-optimal solutions for TSP. The method divides in two parts. The first one is to train a graph neural network on TSP with fixed sizes to return for each edges of the graph the probability that this edge will be used in the optimal solution of the TSP. The second one is to search optimal paths using those probabilities. Described as follow, the model is non-autoregressive which implies to study how to efficiently scale this model to larger graphs.
Our contribution starts with a re-implementation from scratch of the original methods, which are the convolutional graph neural network and the beam search. We also propose a novel decoding method inspired by the upper confidence bound for tree search. We show that this decoding strategy performs significantly better on TSP, with a gap to optimal tour to 11.08% compared to the 47.76% we get from beam search for a 50 nodes problem. Those results are highly correlated to our CPU capacities which does not allow to train the neural network properly, and several experiments should be conducted in order to evaluate the gap between the two methods. We also propose a novel approach to scale the model we have for small TSP to larger ones. With our CPU capacity, we achieved 10.09% gap to optimal when we scaled a TSP20 model to a 50 nodes one, whereas the original paper could only achieve a 34.46% gap.
Our conclusions are that the overall strategy of the original paper is efficient for small TSP, but is non-robust to large scale problems, i.e. when the number of nodes is larger than 100. The non-auto-regressive model allows to fasten the computation cost, but makes the model very specific to a certain number of nodes. We propose a novel approach to scale those models, but further research including training the models with GPU capacity are left to be performed.

Sacha Braun. 2023. An efficient graph neural network to solve the Traveling Salesman Problem : report

1 Introduction

The Traveling Sales Man problem can be described as follow. Given a complete undirected weighted graph of size n, $\mathcal{G} = \{\mathcal{U}, \mathcal{E}, w(i,j), i, j \in [\![1, n]\!]\}$, the goal is to find a permutation π of the nodes in the range $[\![1, n]\!]$ that minimizes the sum of edge weights:

$$\sum_{i=0}^{n-1} w(\pi(i+1), \pi(i))$$

where $\pi(0) = \pi(n)$, and the weights represent the distance between the two nodes.

The Traveling Salesman Problem is known to be NP-hard (1972). As a consequence, when the number of nodes is large, no computational algorithm can except to find an optimal solution in a reasonable time. Some solver such as Concorde (D. Applegate and Cook 1998) uses linear programming with carefully handcrafted heuristics to find solutions up to tens of thousands of nodes, but with prohibitive execution times. The largest TSP solved by Concorde to date has 109,399 nodes with running time of 7.5 months (Chaitanya K. Joshi 2020). Therefore, research on this area mainly focus on finding sub-optimal solutions for the TSP in a reasonable computation time complexity for large graphs.

An algorithm which return close to optimal solutions of a TSP instance in a reasonable time would be a significant step forward in several fields. Indeed, this problem has various applications in many different domains. In the molecular biology, one of the most challenging issues is to read an unknown DNA sequence. It has been shown that this problem can be reformulated as a selective TSP (Jacek Blazewicz 1999). In medicine, the MRI scan time can be long to compute (up to 60min depending on the number of pictures). Using a TSP allows to shorten the reconstruction of the 3D image

MVA 2023, December 2023, Paris, FR

and decrease the reconstruction error due to noise (Nicolas Chauffert 2016). In the industry, many applications can be seen. A famous and intuitive one is the order-picking in a warehouse, where one has to collect n objects at different positions in the warehouse in the shortest time (H. Donald Ratliff 1983). Another beautiful application of the TSP is for continuous line drawing of an image (Bosch and Herman 2003). Some results of this problem can be seen in Appendix (6).

Previous approaches to solve TSP using neural networks has been studied. One famous one uses the algorithm REINFORCE from Reinforcement Learning to build a solution in an autoregressive framework (Wouter Kool 2019). At each time step, the model output the edge that it finds the most promising to add to the partial tour, given the partial tour that has already been created, and the nodes which are left to be visited. The reward of the model is the generated tour's length. This autogressive framework allow to scale the algorithm to problems of various sizes, but require a long computation time.

The approach studied in (Chaitanya K. Joshi 2019), which is the one we are studying, is based on a more general previous work (Alex Nowak 2017) that tries to build a non-auto-regressive graph neural network that aim at solving the problem

$$\text{minimize} \quad \|AX - XB\|^2 \tag{1}$$

$$\text{subject to} \quad X \in \Pi \tag{2}$$

where Π is the set of permutation matrix of size n x n.
The model is trained for a fixed size n on a various collection of matrix A and B. As we will see later (2), the architecture of the model is the same for all classes of matrices A and B which leads to poor results when the model is trained matrices representing the graph a TSP. The neural network is trained to return the probability for each edge that it will be used in the optimal tour, and then uses beam search (Mark F. Medress 1977) to output the shortest path found by the model. The paper we study (Chaitanya K. Joshi 2019) tries to overpass the architecture difficulties, as they use this previous work to start a new one. The preprocessing is different, including a primal community detection filter. The model's architecture is inspired by a more recent one (Xavier Bresson 2017) more suited for connected graphs. In this report, we study the architecture of this new graph convolution layer compared to the one of the previous paper. We also try to compare Beam Search with a reinforcement learning technique based on the principle of optimism in face of uncertainly, and empirically show that this new method surpass Beam Search for all experiments we conducted, in our CPU setting (meaning we use a non-optimal model). Finally, we empathize that this work is promising, but that the most difficult part of this algorithm is the generalization of the model to large graph where supervised learning is not possible. Indeed, the studied algorithm is only effective for problems that we already know how to solve efficiently, and therefore does not provide any significant improvement directly to the TSP. As the paper we study tried to use models trained from a small TSP problem to a larger one but does not provide any information about the methodology they used, we tried our personal method, that can achieve significant better results compared to the one of the original paper. We combined our search algorithm that is robust to errors of the model, with a sampling procedure of small TSP in a large TSP instance.

2 Methodology

2.1 Dataset and input layer

In the next section, we will refer to n as the number of nodes of the TSP. We only focus on the 2D-euclidean representation of the TSP, which means that each node can be placed on the 2D-euclidean square $[0, 1]^2$.

The data used consist of n nodes randomly generated in the square space. A graph of size n representing different the associated TSP is then generated, and the edges are labelled thanks to the solution given by Concorde Solver (an edge used in the optimal tour found by Concorde is classified as 1, and 0 otherwise).

The data is pre-processed for nodes and edges through an input encoder layer. The initial nodes features is the position of the node on a 2D space, that is being linearly encoded in a hidden space with h features (for each node). The edges represents the euclidean distance between each node, and each distance is being encoded in a $h/2$ dimensional space. We concatenate those features with a $h/2$ dimensional vector, that is computed as a linear transformation of the k-nearest neigh-boors matrix δ, for which the values depends on the connection level between the edges (0 if their are not the k-nearest neightboors, or 1 otherwise. It may also be 2 for self-connections). This matrix δ is the first main difference between this method and the previous ones. This matrix allows to encode in the model the connections between the edges, and act as an feature to help the decision process of the model. Intuitively, one can expect the ground truth to have many connections between two edges in the same cluster, and a few number of connections between two edges from different clusters.

2.2 Graph convolutional layer

The graph convolutional layer is composed of l layers, where we map edges and nodes features. Each layer is of the form :

$$x_i^{l+1} = x_i^l + \text{ReLU}\left(\text{BN}\left(W_1^l \cdot x_i^l + \sum_{j=i} \eta_{ij}^l \odot W_2^l \cdot x_j\right)\right)$$

$$e_{ij}^{l+1} = e_{ij}^l + \text{ReLU}\left(\text{BN}\left(W_3^l \cdot e_{ij} + W_4^l \cdot x_i^l + W_5^l \cdot x_j^l\right)\right)$$

Compared to the model (F. Scarselli 2009) that is used in (Alex Nowak 2017) to approximate solutions of the TSP, this model leverage from an LSTM structure (K. Tai 2015), to use a dense map that works as an attention map. Intuitively, this mapping allows each node to leverage the information from the edges which are the most relevant for this node. In (Alex Nowak 2017), the model used has a one shot nature as it is supposed to work on many quadratic problems. Therefore, the model was not optimized for complete graph as it used the features from each nodes uniformly which can explain why it performs poorly. Indeed, a graph convolutional network learn from the underlying graph structure to reduce the node encoding. Since the graph associated to a TSP is complete, there is hardly any graph structure to exploit. In contrast, the dense map allows to leverage for each nodes the features that seems to be the most important for this specific node. The model learns some underlying connections between the graph, even if it is a complete graph, giving more importance to some edges, and less to others.

For each layer, we perform a batch normalization as the spectral radius of the learned linear operator $W_1^l \cdot x_i^l + \sum_{j=i} \eta_{ij}^l \odot W_2^l \cdot x_j$ can grow as l increases. This would lead to instability. Furthermore, adding a batch normalization allows to use much higher learning rates and converge quicker to a solution (S. Ioffe 2015).

We also build the graph convolutional layer with a residual structure. The idea behind the residual structure is that the desired underlying mapping $\mathcal{H}(x)$ can also be approximated as the mapping $\mathcal{F}(x) := \mathcal{H}(x) - x$. (K. He and Sun. 2016) empirically showed that it is easier to optimize the residual mapping than to optimize the original one as the learned residual functions have small responses in general.

In the end, the model output with a Multi Layer Perceptron (MLP) for each edges a 2D array representing (up to a softmax) a feature that is interpreted as probabilities that this edges will, and will not be used in the optimal tour.

2.3 Search algorithms

2.3.1 Beam Search. The search algorithm implemented in the original paper is the Beam Search one (BS) (Mark F. Medress 1977). This algorithm uses the output probabilities of the model to search the most probable tour. It iterates the same procedure $\max_{\text{iter}}$ times, that is : choose a starting node uniformly at random among the nodes. While the tour is not finished, scale transform the probabilities of its non-visited neighbors in order to sample its next neighbors among those probabilities. An algorithm is provided in Appendix (6). In the end, we can either choose to evaluate beam search with the length of the shortest tour found, or the length of the most probable tour found. Obviously, and as shown in the original paper (Chaitanya K. Joshi 2019), the first option is more relevant which is why we choose to evaluate this method. However, we empathize that this search algorithm only relies on the output of the model, and is therefore non-robust to errors. If an edge is used in the optimal tour, but that its probability according to the model is close to zero, then we cannot expect the beam search to find the optimal tour. This problem motivated the study of a new decoding strategy that would use the length of the tour found to update its search method.

2.3.2 MCTS based on UCB search. In the reports, we choose to go beyond beam search and tried to implement our own decoding strategy. We replaced beam search by a more metric-driven search technique to compare those two searches. This methods leverage bandits theory (T Lattimore 2020) to integrate upper confidence bound in the tree search. We adapted the Monte Carlo Tree Search (MCTS) in (Rémi Leblond 2021) that tries to find the most probable path in a classic auto-regressive machine translation model. In our case, we want to find the tour minimizing the length of TSP. Therefore, we compute each tour selecting our action sequentially in an auto-regressive manner such that at each time step, the edge we add to our partially generated tour is selected according to :

$$a_{t+1} = \underset{a \in \mathcal{A}_{a_t}}{\operatorname{argmin}} Q(a_t, a) - \pi_\tau(a|a_t) * \frac{\sqrt{\sum_b N(a_t, b)}}{1 + N(a_t, a)}$$

We update the Q value online according to $\tilde{Q}(a_t, a)$ that represents the average tour length we got in the end of previous sampling procedures when we choose to go to the node a. We then re-scale this value to $[0, 1]$ to get $Q(a_t, a) = \frac{\tilde{Q}(a_t, a) - \min \tilde{Q}}{\max \tilde{Q} - \min \tilde{Q}}$. A simplified version of our algorithm can be seen in the Appendix (6). The second value represents an upper confidence bound we have on the value $\tilde{Q}$. We compute a first upper confidence bound derived from Hoeffding Inequality (Hoeffding 1963), and we biased this confidence bound with a probability mapping of the next possible states. This probability mapping is directly derived from the output of the model, such that $\pi_\tau(a|a_t) = \frac{p(a|a_t)^{\frac{1}{\tau}}}{\sum_b p(b|a_t)^{\frac{1}{\tau}}}$ where p is the probability output of our model, and where the sum is calculated over all the next possible edges.

Intuitively, we try to find the node from which previous explorations returned smaller length paths. We use the principle of optimism in face of uncertainly to add a confidence margin to the results we got in previous experiments. It is also important to note that compared to (Rémi Leblond 2021) for which the Q value comes from the probabilities input, we do not have any prior information about a tour length. To overpass this issue, we could have fixed the Q value to $-\infty$ for unexplored node, but this would not allow us to leverage the upper confidence bound that is derived from the model, as the Q value would blow up compared to the confidence

margin. To this end, we fixed the initial Q value of unexplored node to be $C * \pi_\tau(a|a_t) * \frac{\sqrt{\sum_b N(a_t, b)}}{1 + N(a_t, a)}$ where C is a large negative number. Therefore, when the policy needs to choose between two unexplored nodes, we will always prefer the one with the highest probability of being in the true TSP tour according to the model.

We make the hypothesis that this method is more robust to the errors of the output of the machine learning model. Indeed, our MCTS methods use both the output of the model, and the reward given by the average tour lengths during the search. If the model output a very small probability for an edge that should be used in the optimal tour, the beam search should fail to use this edge. By construction, at some point our method will try a tour with this edge, and update the Q value of the output tour length. If this tour is indeed really short, the Q value will overlap the bias induce by the probability distribution understand that this edge could be more use-full.

However, this price to pay for a more effective search is an increased computation time. Where Beam Search is faster and can be optimized for batches and GPU, the MCTS based method which works in an auto-regressive manner, cannot be used for batches, and need to update the values of the nodes after each iteration. This also induce an extension of the memory capacity needed. Note that the auto-regressive usage is still really different from (**?**), as it is just used for the decoding strategy and is not computing the output of a graph neural network after each iteration (compared to RNN for instance), so the computation time of our method is smaller.

2.4 Generalisation to larger problems

In the original paper (Chaitanya K. Joshi 2019), some results when they tried to generalize one model to variable problem sizes is provided. However, they do not provide any information about the methodology that has been used. We could also not find any information in the original code of the paper. To this end, we developed our own algorithm. When one tries to use a model trained for a TSPn, where n represent the size of the TSP, to output a tour on a TPSm where $m > n$, we just sample randomly n nodes from the TSPm to a TSPn problem. We run our model on the TSPn instance and repeat the process several times. Then, we aggregate the results of the outputs of the neural network by taking for each edges the average probability that the model returned. We then use our MCTS based method to find the shortest tour. This method will produce probabilities that are more noisy compared to a model that could have been trained directly on this instance. Therefore, we can expect the beam search to perform poorly on this new instance, where the MCTS could overpass this difficulty.

3 Results

3.1 Implementation

In our study, we implemented from scratch the graph model described in (Chaitanya K. Joshi 2019). We re-used the code to generate a dataset, as well as most of the trainable structure from the original repository[1]. We also re-used the function to plot the results, but we implemented from scratch our Beam Search decoding, as-well as the MCTS we described earlier. We also implemented our method to generalize one model to larger problems.

Our experiments were limited by the CPU capacities of our computer. That is why all the experiments are conducted with a batch size of 15, on 16 epochs, with a hidden dimension of 50 on 3 layers. The precise setting for each size of a TSP (such as the number of clusters) can be found in the "default-size-.json" in our github repository[2].

We conducted experiments for TSP of size 10, 20, 30 and 50. For each experiment, we trained our model, and tested the two decoding strategies (Beam Search and MCTS) for randomly generated TSP that where not in the training dataset. As metric to evaluate those results we used the averaged gap to optimality which is calculated by $G = \frac{1}{m} \sum_{1 \leq i \ m} (\frac{l_i}{l_i^*} - 1)$ above the m TSP solved. On all the generated tests, with the same number of searches allowed for MCTS and beam search, our method perform better than the beam search.

Table 1. Gap to optimality for our different models with respect to the decoding method used after 1,000 iterations for both decoding strategies

Model	Beam Search	MCTS
TSP10	1.62%	**0.0%**
TSP20	17.07%	**2.25%**
TSP30	27.93%	**5.39%**
TSP50	47.76%	**11.08%**

We would however like to draw attention to the fact that these experiments were carried out on a CPU machine with a very small neural network. Therefore, the beam search which highly rely on the accuracy of the model is more likely to fall into local optimum, where as the MCTS only uses the outputs of the model to bias the search of the minimal length. Several experiments should be conducted to assert whether or not MCTS has a significant advantaged on Beam Search. A non-exhaustive list a experiments are : 1) comparing the two methods on more accurate models. 2) Compare the complexity and the time taken by the two methods to solve a TSP instance. 3) Compare the two methods on larger graphs. One should expect that the convergence rate of MCTS grows faster than the beam search as the number of nodes grows.

[1] https://github.com/chaitjo/graph-convnet-tsp
[2] https://github.com/ElSacho/GNN-For-Salesman-Prob

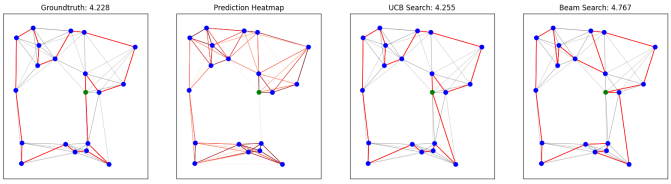

Figure 1. The output best tour derived from each method on a TSP20. From left to right, the ground truth from Concorde, the edges probabilities from the model, MCTS, Beam Search

Some additional results and figures can be seen in Appendix 6).

We also empirically showed that the outputs of the model have an importance with our MCTS based method, trying a MCTS decoding strategy for one TSP with either the output probabilities of the model, or random probabilities. We respectively jump from a gap top optimality to 1.79% to 77.72%. Indeed, one could argue that the MCTS is just a search that does not significantly leverage the outputs of the model. We showed this is not true.

We also tested our implementation of the new generalized model, adapting a model trained on a TSP20 to a TSP50. We show that the beam search does not seem to work with this model : indeed, the noise in the output probabilities is larger since the model does not solve one instance of the TSP50, but k instances of a TSP20, generated with a random extraction of 20 nodes from the original TSP50. However, the MCTS which is more robust to wrong probabilities performs significantly better, with a gap top optimality to 10.5%. This is actually better than the results that the original paper achieved (Chaitanya K. Joshi 2019) (34.46%). Those results are really promising as our models are only trained with a CPU capacity, and the architecture is only composed of 3 layers. Further experiments could implement our algorithms, but for larger models.

Table 2. Gap to optimality when generelizing a TSP20 to a TSP50

Paper studied	MCTS	Beam Search
34.46%	**10.21%**	300.41%

We also studied how the number of TSP20 instances k that be derive from the original TSP50 impact the final gap we get Fig. 2. Surprisingly, we do not need a large number of instances, and just 30 seems to be enough.

All those results can be found on the github repository[3] with a specific notebook for each experiment. We also provides more results on the Appendix (6).

[3] https://github.com/ElSacho/GNN-For-Salesman-Problem

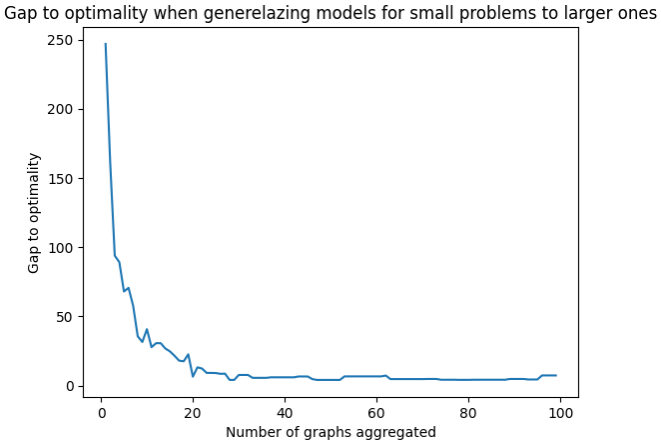

Figure 2. Gap to optimality with respect to the number of instances

4 Limits

As the model needs a dataset of pre-labeled ground truth tour, this method is not mature to solve large TSP instances, and can only solve TSP instances that other algorithms could already solve. In fact, most of the time, the model performs poorly than most of the existing heuristic searches, even if the gap is small. The only main advantage is the computation time, that is not so small for large TSP (around 18 min according to (Chaitanya K. Joshi 2019) for a TSP of size 50). The main challenge of this method is to generalize it to larger TSP.

Indeed, the main objective of TSP is to provide solutions for large TSP instances, with more than 100 nodes, as solvers can provide solutions close to optimality in a very short time. The auto-regressive architecture of the model allows to fasten the computation time, but it makes generalisation more difficult to handle. Further research from the same author tries to build another model that will be robust to scaling (Chaitanya K. Joshi 2020). Do do so, they combine the auto-regressive work from (Wouter Kool 2019) and combined it with their model (Chaitanya K. Joshi 2019). Unfortunately, this work also fails when it needs to generalize its results for larger TSP instances.

One of the major limitation of our model, is that it does not learn to optimize a tour, but rather learn to mimic the results of a heuristic search. Therefore, the model is dependent of the initial training, and may produce poor results for data it has never seen. For instance, the training phase was made with points randomly distributed on the $[0, 1]^2$ square, but for real world applications those points could share a particular structure. If we just think about warehouse handling for instance, the structure of the TSP can be really different as the points should be distributed on parallel lines

representing the aisles of the warehouse.

We also argue that this method uses a 2D simplification of the problem, as the positions of the nodes are encoded. Indeed, many TSP problems are not that straightforward as the length to go from a point A to a point B is not always the length of a straight line. Therefore, generalisation to more complex TSP could be on the agenda to evaluate the performance of this method.

5 Conclusion

The paper studied provides a framework that outperform all previous non auto-regressive models to provided sub-optimal solutions for the Traveling Salesman Problem (TSP). Its model architecture merged previous work on the TSP, and attention models that suit better complete graphs. The algorithm can provides suboptimal solution in a quick time compared to other methods.

One of the main limitation of the algorithm is that we cannot generalize the framework for any TSP. We need to train a model for each sizes of a TSP. Therefore, the model is only trainable for small TSP and we still need to find how to generalize to larger one. This task appears to be hard, as the authors tried to work on it in a new paper, but did not come up with a satisfactory solution.

Further research could try other algorithms to generalize one model of small size to larger ones. The decoding strategy can also be studied. For instance, the output probabilities of the model could be used to bias the search of an existing efficient heuristic. This would fasten the computation time. One of the main questions behind those strategies could be : can we find a hierarchical structure in TSP that would allow to divide one TSP in several smaller ones.

References

RE Miller JW Thatcher , Richard M. Karp. 1972. Reducibility among combinatorial problems. *MR* (1972).

Afonso S. Bandeira Joan Bruna Alex Nowak, Soledad Villar. 2017. A Note on Learning Algorithms for Quadratic Assignment with Graph Neural Networks. *arXiv:1706.07450* (2017).

Robert Bosch and Adrianne Herman. 2003. Continuous Line Drawings via the Traveling Salesman Problem. *DBLP* (2003).

Louis-Martin Rousseau Thomas Laurent Chaitanya K. Joshi, Quentin Cappart. 2020. Learning the Travelling Salesperson Problem Requires Rethinking Generalization. *International Conference on Principles and Practice of Constraint Programming , arXiv:2006.07054* (2020). CQRL2020.

Xavier Bresson Chaitanya K. Joshi, Thomas Laurent. 2019. An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem. *INFORMS* (2019).

V. Chvatal D. Applegate, R. Bixby and W. Cook. 1998. On the Solution of Traveling Salesman Problems. *EMS Press* (1998).

A. C. Tsoi M. Hagenbuchner G. Monfardini F. Scarselli, M. Gori. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* (2009).

Arnon S. Rosenthal H. Donald Ratliff. 1983. Order-Picking in a Rectangular Warehouse: A Solvable Case of the Traveling Salesman. *INFORMS* (1983).

Wassily Hoeffding. 1963. Probability inequalities for sums of bounded random variables. *J. Amer. Statist. Assoc.* (1963).

Marta Kasprzak Wojciech T Markiewicz Jacek Blazewicz, Piotr Formanowicz. 1999. DNA Sequencing With Positive and Negative Errors. *Journal of Computational Biology: a Journal of Computational Molecular Cell Biology* (1999).

S. Ren K. He, X. Zhang and J. Sun. 2016. Deep Residual Learning for Image Recognition. *Computer Vision and Pattern Recognition* (2016).

C. Manning. K. Tai, R. Socher. 2015. Improved Semantic Representations from Tree-Structured Long Short-Term Memory Networks. *Association for Computational Linguistics (ACL)* (2015).

James W. Forgie C. C. Green Dennis H. Klatt Michael H. O'Malley Edward P. Neuburg Allen Newell Raj Reddy H. Barry Ritea J. E. Shoup-Hummel Donald E. Walker William A. Woods Mark F. Medress, Franklin S. Cooper. 1977. Speech Understanding Systems. *Artificial Intelligence* (1977).

Jonas Kahn Philippe Ciuciu Nicolas Chauffert, Pierre Weiss. 2016. A projection algorithm for gradient waveforms design in Magnetic Resonance Imaging. *IEEE Transactions on Medical Imaging* (2016).

Laurent Sifre Miruna Pislar Jean-Baptiste Lespiau Ioannis Antonoglou Karen Simonyan Oriol Vinyals Rémi Leblond, Jean-Baptiste Alayrac. 2021. Machine Translation Decoding beyond Beam Search. *arXiv:2104.05336* (2021).

C. Szegedy. S. Ioffe. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *International Conference on Machine Learning (ICML)* (2015).

C Szepesvári T Lattimore. 2020. Bandit algorithms. *Cambridge University Press* (2020).

Max Welling Wouter Kool, Herke van Hoof. 2019. Attention, Learn to Solve Routing Problems! *ICLR* (2019).

Thomas Laurent Xavier Bresson. 2017. Residual Gated Graph ConvNets. *arXiv:1711.07553* (2017).

6 Appendix

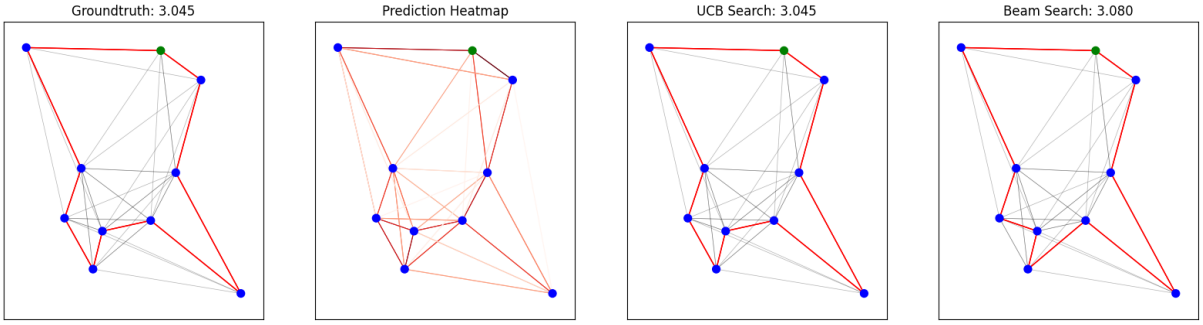

Figure 3. The output best tour derived from each method on a TSP10. From left to right, the ground truth from Concorde, the edges probabilities from the model, MCTS, Beam Search

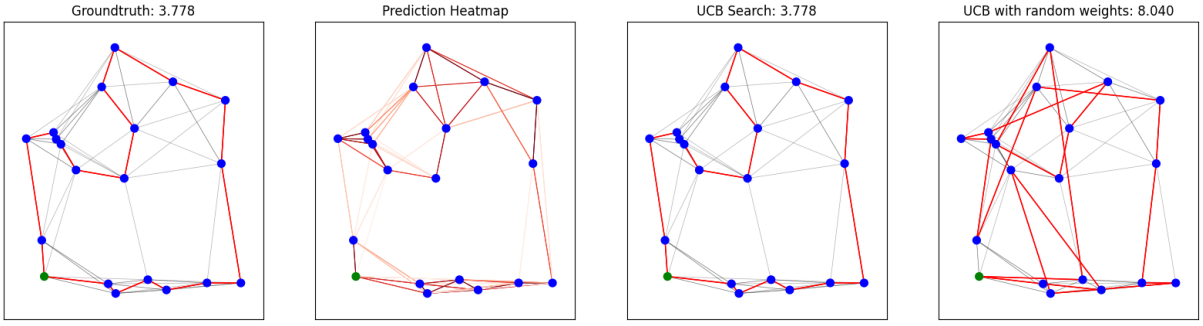

Figure 4. The output best tour derived from each method on a TSP30. From left to right, the ground truth from Concorde, the edges probabilities from the model, MCTS, MCTS with random weights

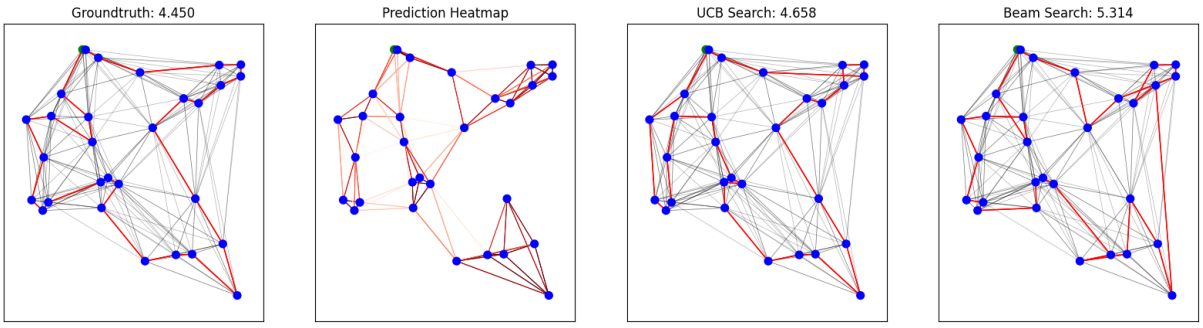

Figure 5. The output best tour derived from each method on a TSP50. From left to right, the ground truth from Concorde, the edges probabilities from the model, MCTS, Beam Search

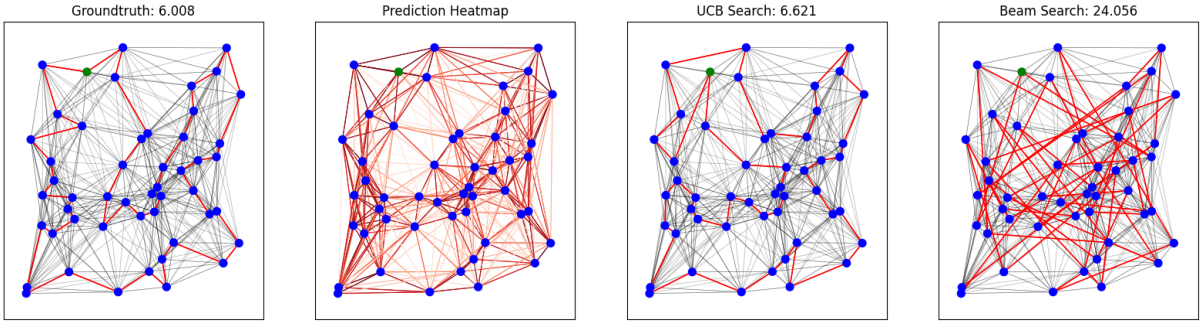

Figure 6. The output best tour derived from each method on a TSP50 **when generelizing a TSP20 model**. From left to right, the ground truth from Concorde, the edges probabilities from the model, MCTS, Beam Search

Algorithm 1: Beam Search

Input: Length between the edges matrix L, probability output p, max_{iter}
Output: Best tour found
for *each step in* max_{iter} **do**
1 Choose one node i at random in the graph;
while *One node has not been visited* **do**
2 Normalize the probabilities (p_{ij}) for all the nodes j that has not been visited yet ;
3 Update i choosing the next node at random with respect to (p_{ij});
4 Add the length to the original node;
5 Compute the length of the tour, if it the smallest one, save it;
return *The tour that generated the smallest tour length*;

Algorithm 2: MCTS

Input: Length between the edges matrix L, probability output p, max_{iter}, large number C, root node R
Output: Best tour found
1 Normalize the probabilities $p_{R,node}$;
for *each children of* R **do**
2 Initialize its Search value $V(node) = p_{R,node} * C$
for *each step in* max_{iter} **do**
3 Start from the root node R;
while *One node has not been visited* **do**
4 Select the next node with the lowest V value. If the partial tour has not been explored yet, initialize it with $V(next\_node) = p_{actual\_node,next\_node} * C$;
5 Add the length to the original node;
6 Compute the length of the tour l, if it the smallest one, save it;
node = last_node_visited;
while *node is not* R **do**
7 Calculate Q = average length of the generated tour that pass by this nodes and all its parent nodes;
8 Calculate $UCB = p_{parent,node} * \frac{\sqrt{\sum_{children\_of\_the\_parent} N(parent, children\_of\_the\_parent)}}{1 + N(parent, node)}$;
9 Update $V(node) = Q - UCB$;
10 node = parent;
return *The tour that generated the smallest tour length*;

One beautiful application of the TSP is for continuous line drawing, that is te draw an image with a single line. One method (Bosch and Herman 2003) divide the target picture into squares based on the grayscale value we have, and run a TSP using those squares as nodes. Here is a picture they could get (Fig 7). To see how good this result can be, please do not look at the picture to closely (the smaller the picture is, the better it is).

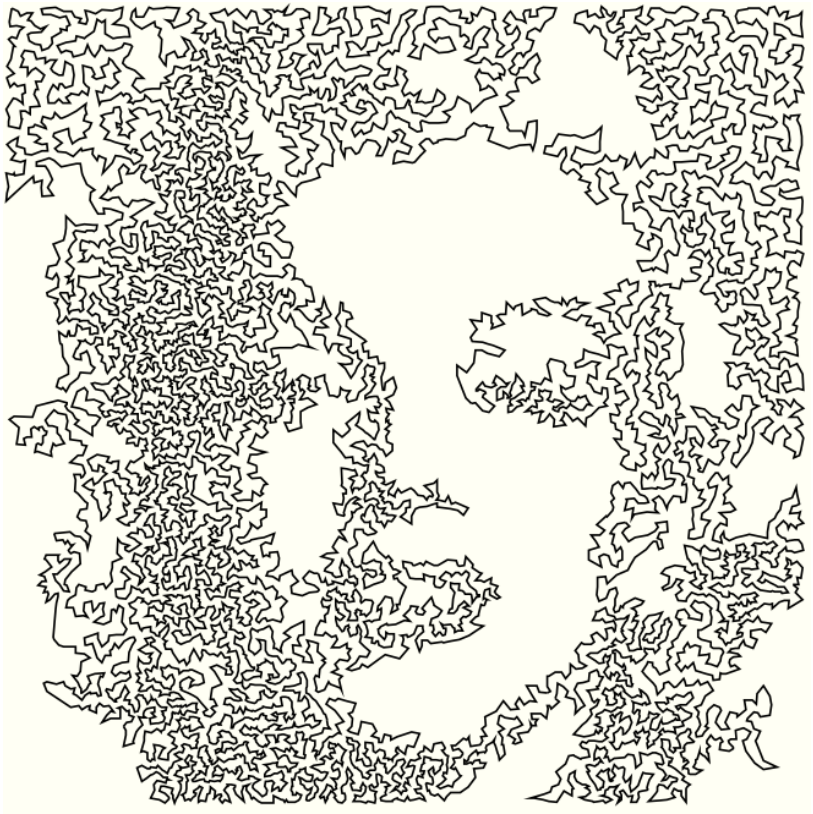

Figure 7. The one line drawing returned by the TSP method with 11,508 nodes (generated by Bosch and Herman (2003))