Learning Heuristics for the TSP by Policy Gradient

Michel Deudon[1], Pierre Cournut[1], Alexandre Lacoste[2], Yossiri Adulyasak[3(✉)],
and Louis-Martin Rousseau[4]

[1] Polytechnique (France), Palaiseau, France
{michel.deudon,pierre.cournut}@polytechnique.edu
[2] Element AI, Montreal, Canada
allac@elementai.ca
[3] HEC Montréal, Montreal, Canada
yossiri.adulyasak@hec.ca
[4] Polytechnique Montréal, Montreal, Canada
louis-martin.rousseau@cirrelt.ca

Abstract. The aim of the study is to provide interesting insights on how efficient machine learning algorithms could be adapted to solve combinatorial optimization problems in conjunction with existing heuristic procedures. More specifically, we extend the neural combinatorial optimization framework to solve the traveling salesman problem (TSP). In this framework, the city coordinates are used as inputs and the neural network is trained using reinforcement learning to predict a distribution over city permutations. Our proposed framework differs from the one in [1] since we do not make use of the Long Short-Term Memory (LSTM) architecture and we opted to design our own critic to compute a baseline for the tour length which results in more efficient learning. More importantly, we further enhance the solution approach with the well-known 2-opt heuristic. The results show that the performance of the proposed framework alone is generally as good as high performance heuristics (OR-Tools). When the framework is equipped with a simple 2-opt procedure, it could outperform such heuristics and achieve close to optimal results on 2D Euclidean graphs. This demonstrates that our approach based on machine learning techniques could learn good heuristics which, once being enhanced with a simple local search, yield promising results.

Keywords: Combinatorial optimization · Traveling salesman
Policy gradient · Neural networks · Reinforcement learning

1 Introduction

Combinatorial optimization is a topic that consists of finding an optimal object from a finite set of objects. Sequencing problems are those where the best order for performing a set of tasks must be determined, which in many cases leads to a NP-hard problem. Specific variations include single machine scheduling and the

W.-J. van Hoeve (Ed.): CPAIOR 2018, LNCS 10848, pp. 170–181, 2018.
https://doi.org/10.1007/978-3-319-93031-2_12

Traveling Salesman Problem (TSP). Sequencing problems are among the most widely studied problems in Operations Research (OR). They are prevalent in manufacturing and routing applications.

As in [2], our work is motivated by the fact that many real world problems arising from the OR community are solved daily from scratch with hand-crafted features and man-engineered heuristics. We propose a generic framework to learn heuristics for combinatorial tasks where the output is an ordering of the input. Our focus in this paper is a data-driven heuristic that can effectively solve the TSP, the well-known combinatorial problem (due to limited space, the review on optimization algorithms on TSP is provided in Appendix).

We first review some recent Reinforcement Learning (RL) approaches to solve the TSP in Sect. 2. We then present our proposed method in Sects. 3 and 4. Finally, we describe experiments and discuss results in Sect. 5.

2 Reinforcement Learning Perspective for the TSP

Reinforcement learning (RL) is a general-purpose framework for decision making in a scenario where a learner actively interacts with an environment to achieve a certain goal. In response to an action, the learner receives two types of information: his new state in the environment, and a real-valued reward, which is specific to the task and its corresponding goal. Successful examples include playing games at high level (Atari [8], Go [9,10]), navigating 3D worlds or labyrinths, controlling physical systems and interacting with users.

Combinatorial problems such as the TSP are often solved sequentially. Typically, a state is a partial solution (a sequence of visited cities) and an action is the next city to visit (among those not yet visited). In response to an action, the new state is the updated solution and the reward signal could either come when a tour is completed or be incremental. An RL agent builds on its own experience - sequences (state, action; reward, state) - to maximize future rewards. In practice, one could either learn directly a (deterministic or stochastic) mapping from state to action, called a policy $\pi(a|s)$, or learn an auxiliary evaluation function (Value or Q function) measuring the quality of a state and used to discriminate among actions based on their usefulness. In both cases, the combinatorial structure of the state space S is intractable and calls for the use of function approximators such as Deep Neural Networks. Deep Learning (DL) is a general-purpose framework for representation learning. Given an objective, a Neural Network learns the representation that is required to achieve the objective. Neural Networks compute hierarchical, abstract representations of the data (through linear transformations and non-linear activation functions) and learn features (at several levels of abstractions) by back-propagating gradients of the loss w.r.t. the parameters, using the chain rule and Stochastic Gradient Descent.

Recurrent Neural Networks (RNN) with Long Short Term Memory (LSTM) cells [11] have been successfully used for structured inputs and/or outputs with long term dependencies. More recently, attention based Neural Networks have significantly improved models in computer vision [12], image [13] or video

[14] captioning, machine translation [15], speech recognition [16] and question answering [17]. Rather than processing a signal once, attention allows to process step by step some regions or features of the signal at high resolution to acquire information when and where needed. At each step, next location is chosen based on past information and demands for the task. Google Brain's Pointer Network [18] is a neural architecture to learn the conditional probability of an output sequence with elements that are discrete tokens corresponding to positions in an input sequence. The neural network comprises a RNN encoder-decoder connected with hard attention. At each decoding step, a "pointer" is used to sample from the action space (in our case, a probability distribution over cities to visit). It overall parametrizes a stochastic policy over city permutations $p_\theta(\pi|s)$ and can be used for problems such as sorting variable sized sequences, and various combinatorial optimization problems. Google Brain's Pointer Network trained by Policy Gradient [1] could determine good quality solutions for 2D Euclidean TSP with up to 100 nodes.

In [2], the authors use a graph embedding network called structure2vec (S2V) to featurize nodes in the graph in the context of their neighbourhood. The learned greedy algorithm constructs solutions sequentially and is trained by fitted Q-learning to learn the policy together with the graph embedding network. For the TSP task, Google Brain's Pointer Network trained by Policy Gradient performs on par with the S2V network trained by fitted Q-learning.

Based on the recent work [1], we further enhance the approach in several ways. In particular, instead of relying on the LSTM architecture, our model is based solely on attention mechanisms. This result in a more efficient learning. The framework is further enhanced with a simple 2-opt procedure and the approach shows promising results on the TSP. We believe that the outcome of this study sheds light on a data-driven hybrid heuristic that makes use of ML and local search techniques to tackle combinatorial optimization.

3 Neural Architecture for TSP

Given a set of n cities s, the Traveling Salesman Problem (TSP) consists in finding a minimum cost tour visiting all n cities exactly once. The total cost of a tour is the total distance traveled in the tour. Following [1], we aim to learn the parameters θ of a stochastic policy over city permutations $p_\theta(\pi|s)$, using Neural Networks and Policy Gradient. Given an input set of points s, the key idea is to assign higher probability to "good" tours π^+ and lower probability to "undesirable" tours π^-.

We follow the general encoder-decoder perspective. The encoder maps an input set $I = (i_1, ..., i_n)$ to a set of continuous representations $Z = (z_1, ..., z_n)$. Given Z, the decoder then generates an output sequence $O = (o_1, ..., o_n)$ of symbols one element at a time. At each step the model is auto-regressive, using the previously generated symbols as additional input when generating the next.

3.1 TSP Setting and Input Preprocessing

In this paper, we focus on the 2D Euclidean TSP. Each $city_i$ is described by its $2D$ coordinates (x_i, y_i) in a Euclidean space. We use Principal Component Analysis (PCA) on the centered input coordinates to exploit spatial invariance by rotation of all cities. This way, the learned heuristic does not depend on the orientation of the input $s = ((x_i, y_i))_{i \in [1,n]}$.

3.2 Encoder

The purpose of our encoder is to obtain a representation for each action (city) given its context. The output of our encoder is a set of action vectors $A = (a_1, ..., a_n)$, each representing a city interacting with other cities. Our neural encoder takes inspiration from recent advances in Neural Machine Translation. Similarly to [19], our actor and our critic use neural attention mechanisms to encode cities as a set (rather than a sequence as in [1]).

TSP Encoder. We use the encoder proposed in [19], which relies on attention mechanisms in place of the traditional convolutions or recurrences. Our self attentive encoder takes as input an embedded and batch normalized [20] set of n cities $s = (city_i)_{i \in [1,n]}$ (d-dimensional space). It overall consists in a stack of N identical layers as shown in Fig. 1 in Appendix. Each layer has two sublayers. The first sublayer *Multi-head Attention* is detailed in the next paragraph. The second sublayer *Feed-Forward* consists of two position-wise linear transformations with a ReLU activation in between. The output of each sublayer is $LayerNorm(x + Sublayer(x))$, where $Sublayer(x)$ is the function implemented by the sublayer itself and $LayerNorm()$ stands for layer normalization [20].

Multi Head Attention. Neural attention mechanisms allow queries to interact with key-value pairs. For the TSP, queries and key-value pairs $q_i, k_i, v_i \in \mathbb{R}^d$ are obtained by linearly transforming each $city_i \in \mathbb{R}^d$ and applying a ReLu non linearity. Following [19], our attention mechanism is defined as

$$Attention(Q; K; V) = softmax(\frac{QK^T}{\sqrt{d}})V \tag{1}$$

where $Q = [q_1, ..., q_n]$, $K = [k_1, ..., k_n]$, $V = [v_1, ..., v_n]$. Our Multi Head Attention sublayer outputs a new representation for each city, computed as a weighted sum of city values, where the corresponding weights are defined by an affinity function between cities' queries and keys. As suggested in [19], queries, keys and values are linearly projected on h different learned subspaces (hence the name *Multi Head*). We then apply the attention mechanism on each of these new set of representations to obtain h d_h-dimensional output values for each city which are concatenated into the final values.

3.3 Decoder

Following [1], our neural network architecture uses the chain rule to factorize the probability of a tour as

$$p_\theta(\pi|s) = \prod_{t=1}^{n} p_\theta(\pi(t)|\pi(<t), s) \tag{2}$$

Each term on the right hand side of Eq. (2) is computed sequentially with softmax modules. As opposed to [1] which summarizes all previous actions in a fixed-length vector, our model explicitly forgets after $K = 3$ steps, dispensing with LSTM networks. At each output time t, we map the three last sampled actions (visited cities) to the following query vector:

$$q_t = ReLu(W_1 a_{\pi(t-1)} + W_2 a_{\pi(t-2)} + W_3 a_{\pi(t-3)}) \in \mathbb{R}^{d'} \tag{3}$$

Similar to [1], our query vector q_t interacts with a set of n vectors to define a pointing distribution over the action space. Once the next city is sampled, the trajectory q_{t+1} is updated with the selected action vector and the process ends when the tour is completed. See Fig. 2 in Appendix.

Pointing Mechanism. We use the same pointing mechanism as in [1] to predict a distribution over cities given encoded actions (cities) and a state representation (query vector). Pointing to a specific position in the input sequence allows to adapt the same framework to variable length tours. As in [1], our pointing mechanism is parameterized by two attention matrices $W_{ref} \in \mathbb{R}^{dd"}$, $W_q \in \mathbb{R}^{d'd"}$ and an attention vector $v \in \mathbb{R}^{d"}$ as follows:

$$\forall i \leq n, u_i^t = \begin{cases} v^T tanh(W_{ref} a_i + W_q q_t) \quad if \quad i \notin \{\pi(0), ..., \pi(t-1)\} \\ -\infty \quad \text{otherwise.} \end{cases} \tag{4}$$

$$p_\theta(\pi(t)|\pi(<t), s) = softmax(C\, tanh(u^t/T)) \tag{5}$$

$p_\theta(\pi(t)|\pi(<t), s)$ predicts a distribution over the set of n action vectors, given a query q_t. Following [1], we use a mask to set the logits (aka log-probabilities) of cities that already appeared in the tour to $-\infty$, as shown in Eq. (4). This ensures that our model outputs valid permutations of the input. As suggested in [1], clipping the logits in $[-C, +C]$ is a way to control the entropy. T is a temperature hyper-parameter used to control the certainty of the sampling. $T = 1$ during training and $T > 1$ during inference.

4 Training the Model

Supervised learning for NP-hard problems such as the TSP and its variants is undesirable because the performance of the model is tied to the quality of the supervised labels and getting supervised labels is expensive (and may be infeasible). By contrast, RL provides an appropriate and simple paradigm for training a Neural Network. An RL agent explores different tours and observes their corresponding rewards.

Following [1], we train our Neural Network by Policy Gradient using the REINFORCE learning rule [21] with a critic to reduce the variance of the gradients. For the TSP, we use the tour length as reward $r(\pi|s) = L(\pi|s)$ (which we seek to minimize).

Policy Gradient and REINFORCE: Our training objective is the expected reward, which given an input graph s is defined as:

$$J(\theta|s) = \mathbb{E}_{\pi \sim p_\theta(.|s)}[r(\pi|s)] \tag{6}$$

During training, our graphs are drawn from a distribution S and the total training objective is defined as:

$$J(\theta) = \mathbb{E}_{s \sim S}[J(\theta|s)] \tag{7}$$

To circumvent non-differentiability of hard-attention, we resort to the well-known REINFORCE learning rule [21] which provides an unbiased gradient of (6) w.r.t. the model's parameters θ:

$$\nabla_\theta J(\theta|s) = \mathbb{E}_{\pi \sim p_\theta(.|s)}[(r(\pi|s) - b_\phi(s))\nabla_\theta log(p_\theta(\pi|s))] \tag{8}$$

where $b_\phi(s)$ is a parametric baseline implemented by a *critic* network to reduce the variance of the gradients while keeping them unbiased. With Monte-Carlo sampling, the gradient of (7) is approximated by:

$$\nabla_\theta J(\theta) \approx \frac{1}{B}\sum_{k=1}^{B}(r(\pi_k|s_k) - b_\phi(s_k))\nabla_\theta log(p_\theta(\pi_k|s_k)) \tag{9}$$

We learn the actor's parameters θ by starting from a random policy and iteratively optimizing them with the REINFORCE learning rule and Stochastic Gradient Descent (SGD), on instances generated on the fly.

Critic: Our critic uses the same encoder as our actor. It uses once the pointing mechanism with $q = 0_{d'}$. The critic's pointing distribution over cities $p_\phi(s)$ defines a *glimpse* vector gl_s computed as a weighted sum of the action vectors $A = (a_1, ..., a_n)$

$$gl_s = \sum_{i=1}^{n} p_\phi(s)_i a_i \tag{10}$$

The glimpse vector gl_s is fed to a 2 fully connected layers with ReLu activations. The critic is trained by minimizing the Mean Square Error between its predictions and the actor's rewards.

5 Experiments and Results

We conduct experiments to investigate the behavior of the proposed method. We consider a benchmarked test set of 1,000 Euclidean TSP20, TSP50 and TSP100 graphs. Points are drawn uniformly at random in the 2D unit square. Experiments are conducted using Tensorflow 1.3.0. We use mini-batches of 256 sequences of length $n = 20$, $n = 50$ and $n = 100$. The actor and critic embed each city in a 128-dimensional space. Our self attentive encoder consists of 3 stacks with $h = 16$ parallel heads and $d = 128$ hidden dimensions. For each head we use $d_h = d/h = 8$. Our FFN sublayer has input and output dimension d and its inner-layer has dimension $4d = 512$. Queries for the pointing mechanisms are 360-dimensional vectors ($d' = 360$). The pointing mechanism is computed in a 256-dimensional space ($d" = 256$). The critic's feed forward layers consist in 256 and 1 hidden units. Parameters θ are initialized with xavier_initializer [22] to avoid saturating the non-linear activation functions and to keep the scale of the gradients roughly the same in all layers. We clip our tanh logits to $[-10,10]$ for the pointing mechanism. Temperature is set to 2.4 for TSP20 and 1.2 for TSP50 and TSP100 during inference. We use Adam [23] optimizer for SGD with $\beta_1 = 0.9$, $\beta_2 = 0.99$ and $\epsilon = 10^{-9}$. Our initial learning rate of 10^{-3} is decayed every 5000 steps by 0.96. Our model was trained for 20000 steps on two Tesla K80 (approximately 2h).

Results are compared in terms of solution quality to Google OR tools, a Vehicle Routing Problem (VRP) solver that combines local search algorithms (*cheapest insertion*) and meta-heuristics, the heuristic of Christofides [24], the well-known Lin-Kernighan heuristic (LK-H) [25] and Concorde exact TSP solver (which yields an optimal solution). Note that, even though LK-H is a heuristic, the average tour lengths of LK-H are very close to those of Concorde. Thus, the LK-H results are omitted from the Table. We refer to Google Brain's Pointer Network trained with RL [1] as *Ptr*. For the TSP, we run the actor on a batch of a single input graph. The more we sample, the more likely we will visit the optimal tour. Table 1 compares the different sampling methods with a batch of size 128. 2-opt is used to improve the best tour found by our model (model+2opt). For the instances TSP100, experiments were conducted with our model trained on TSP50. In terms of computing time, all the instances were solved within a fraction of second. For TSP50, the average computing time per instance are $0.05s$ for Concorde, $0.14s$ for LK-H and $0.02s$ for OR-Tools on a single CPU, as well as $0.30s$ for the pointer network and $0.06s$ for our model on a single GPU.

The results clearly show that our model is competitive with existing heuristics for the TSP both in terms of optimality and running time. We provide the following insights based on the experimental results.

- **LSTM vs. Explicitly forgetting:** Our results suggest that keeping in memory the last three sampled actions during decoding performs on par with [1] which uses a LSTM network. More generally, this raises the question of what information is useful to take optimal decisions.

Table 1. Average tour length (lower is better)

Task	Model	Model + 2opt	Ptr supervised	Ptr greedy	Ptr sample	Christofides	OR tools	Concorde (optimal)
TSP20	3.84	3.82	3.88	3.89	-	4.30	3.85	3.82
TSP50	5.81	5.77	6.09	5.95	5.80	6.62	5.80	5.68
TSP100	8.85*	8.16*	10.81	8.30	8.05	9.18	7.99	7.77

*N.B.: Results for TSP100 were obtained with our model trained on TSP50

- **Results for TSP100:** For TSP100 solved by our model pre-trained on TSP50 (see also Fig. 3 in Appendix), our approach performs relatively well even though it was not directly trained on the same instance size as in [1]. This suggests that our model can generalize heuristics to unseen instances.
- **AI-OR Hybridization:** As opposed to [1] which builds an end-to-end deep learning pipeline for the TSP, we combine heuristics learned by RL with local-search (2-opt) to quickly improve solutions sampled from our policy without increasing the running time during inference. Our actor together with this hybridization achieves approximately 5x speedup compared to the framework of [1].

6 Conclusion

Solving Combinatorial Optimization is difficult in general. Thanks to decades of research, solvers for the Traveling Salesman Problem (TSP) are highly efficient, able to solve large instances in a few computation time. With little engineering and no labels, Neural Networks trained with Reinforcement Learning are able to learn clever heuristics (or distribution over city permutations) for the TSP. Our code is made available on Github[1].

We plan to investigate how to extend our model to constrained variants of the TSP, such as the TSP with time windows, an important and practical TSP variant which is much more difficult to solve. We believe that Markov Decision Processes (MDP) provide a sound framework to address feasibility in general. One contribution we would like to emphasize here is that simple heuristics can be used in conjunction with Deep Reinforcement Learning, shedding light on interesting hybridization between Artificial Intelligence (AI) and Operations Research (OR). We encourage more contributions of this type.

Acknowledgment. We would like to thank Polytechnique Montreal and CIRRELT for financial and logistic support, Element AI for hosting weekly meetings as well as Compute Canada, Calcul Quebec and Telecom Paris-Tech for computational resources. We are also grateful to all the reviewers for their valuable and detailed feedback.

[1] https://github.com/MichelDeudon/encode-attend-navigate.

Appendix: supplementary materials

Literature review on optimization algorithms for the TSP

The best known exact dynamic programming algorithm for the TSP has a complexity of $O(2^n n^2)$, making it infeasible to scale up to large instances (e.g., 40 nodes). Nevertheless, state of the art TSP solvers, thanks to handcrafted heuristics that describe how to navigate the space of feasible solutions in an efficient manner, can provably solve to optimality symmetric TSP instances with thousands of nodes. Concorde [3], known as the best exact TSP solvers, makes use of cutting plane algorithms, iteratively solving linear programming relaxations of the TSP, in conjunction with a branch-and-bound approach that prunes parts of the search space that provably will not contain an optimal solution.

The MIP formulation of the TSP allows for tree search with Branch & Bound which partitions (Branch) and prunes (Bound) the search space by keeping track of upper and lower bounds for the objective of the optimal solution. Search strategies and selection of the variable to branch on influence the efficiency of the tree search and heavily depend on the application and the physical meaning of the variables. Machine Learning (ML) has been successfully used for variable branching in MIP by learning a supervised ranking function that mimics Strong Branching, a time-consuming strategy that produces small search trees [4]. The use of ML in branching decisions in MIP has also been studied in [5].

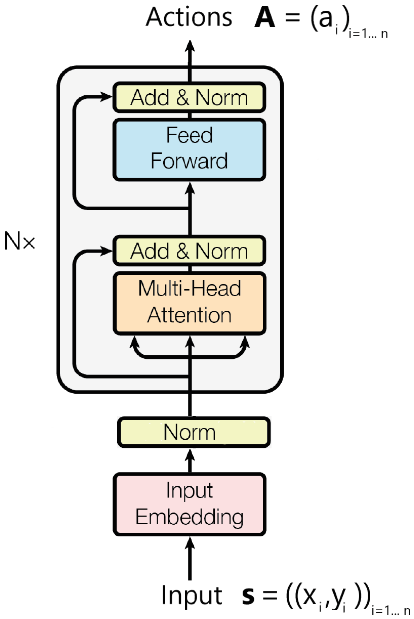
Actions $\mathbf{A} = (a_i)_{i=1\ldots n}$
Add & Norm
Feed Forward
Nx
Add & Norm
Multi-Head Attention
Norm
Input Embedding
Input $\mathbf{s} = ((x_i, y_i))_{i=1\ldots n}$

Fig. 1. Our neural encoder. Figure modified from [19].

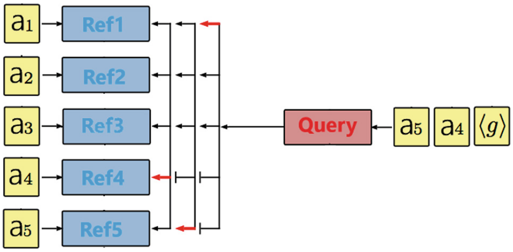

Fig. 2. Our neural decoder. Figure modified from [1].

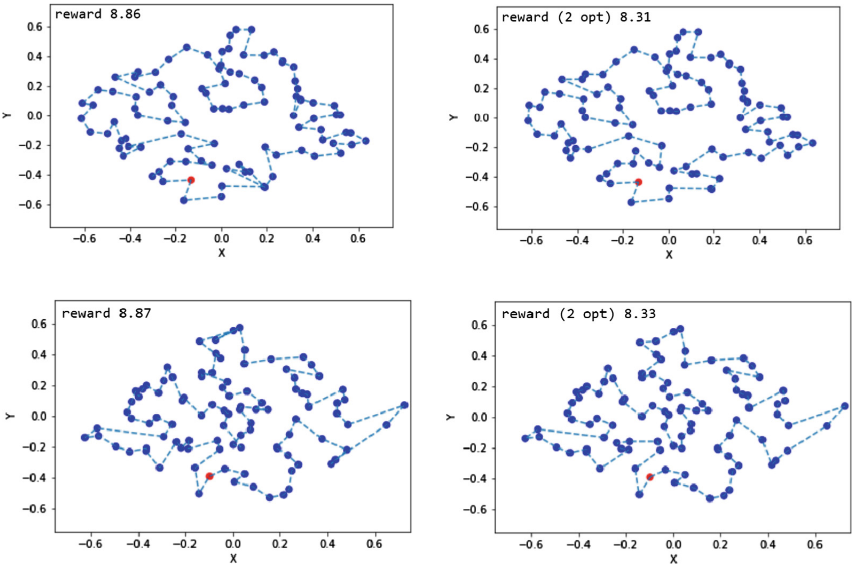

Fig. 3. 2D TSP100 instances sampled with our model trained on TSP50 (left) followed by a 2opt post processing (right)

For constrained based scheduling, filtering techniques from the OR community aim to drastically reduce the search space based on constraints and the objective. For instance, one could identify mandatory and undesirable edges and force edges based on degree constraint as in [6]. Another approach consists in building a relaxed Multivalued Decision Diagrams (MDD) that represents a superset of feasible orderings as an acyclic graph. Through a cycle of filtering and refinement, the relaxed MDD approximates an exact MDD, i.e., one that exactly represents the feasible orderings [7].

References

1. Bello, I., Pham, H., Le, Q.V., Norouzi, M., Bengio, S.: Neural combinatorial optimization with reinforcement learning. In: International Conference on Learning Representations (ICLR 2017) (2017)
2. Khalil, E., Dai, H., Zhang, Y., Dilkina, B., Song, L.: Learning combinatorial optimization algorithms over graphs. In: Advances in Neural Information Processing Systems, pp. 6351–6361 (2017)
3. Applegate, D., Bixby, R., Chvatal, V., Cook, W.: Concorde TSP solver (2006)
4. Khalil, E.B., Le Bodic, P., Song, L., Nemhauser, G.L., Dilkina, B.N.: Learning to branch in mixed integer programming. In: AAAI, pp. 724–731, February 2016
5. Di Liberto, G., Kadioglu, S., Leo, K., Malitsky, Y.: Dash: dynamic approach for switching heuristics. Eur. J. Oper. Res. **248**(3), 943–953 (2016)
6. Benchimol, P., Van Hoeve, W.J., Régin, J.C., Rousseau, L.M., Rueher, M.: Improved filtering for weighted circuit constraints. Constraints **17**(3), 205–233 (2012)
7. Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J.: Sequencing and single-machine scheduling. In: Bergman, D., Cire, A.A., van Hoeve, W.J., Hooker, J. (eds.) Decision Diagrams For Optimization, pp. 205–234. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42849-9_11
8. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Petersen, S.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529 (2015)
9. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
10. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Chen, Y.: Mastering the game of go without human knowledge. Nature **550**(7676), 354 (2017)
11. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
12. Mnih, V., Heess, N., Graves, A.: Recurrent models of visual attention. In: Advances in Neural Information Processing Systems, pp. 2204–2212 (2014)
13. Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., Zemel, R., Bengio, Y.: Show, attend and tell: neural image caption generation with visual attention. In: International Conference on Machine Learning, pp. 2048–2057, June 2015
14. Gao, L., Guo, Z., Zhang, H., Xu, X., Shen, H.T.: Video captioning with attention-based lstm and semantic consistency. IEEE Trans. Multimedia **19**(9), 2045–2055 (2017)
15. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: ICLR 2015 (2015)
16. Chan, W., Jaitly, N., Le, Q., Vinyals, O.: Listen, attend and spell: a neural network for large vocabulary conversational speech recognition. In: 2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 4960–4964. IEEE, March 2016
17. Xu, H., Saenko, K.: Ask, attend and answer: exploring question-guided spatial attention for visual question answering. In: Leibe, B., Matas, J., Sebe, N., Welling, M. (eds.) European Conference On Computer Vision. LNCS, pp. 451–466. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46478-7_28

18. Vinyals, O., Fortunato, M., Jaitly, N.: Pointer networks. In: Advances in Neural Information Processing Systems, pp. 2692–2700 (2015)
19. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. In: Advances in Neural Information Processing Systems, pp. 6000–6010 (2017)
20. Ioffe, S., Szegedy, C.: Batch normalization: accelerating deep network training by reducing internal covariate shift. In: International Conference on Machine Learning, pp. 448–456, June 2015
21. Williams, R.J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. In: Sutton, R.S. (ed.) Reinforcement Learning, pp. 5–32. Springer, Boston (1992). https://doi.org/10.1007/978-1-4615-3618-5_2
22. Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, pp. 249–256, March 2010
23. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: ICLR 2015 (2015)
24. Christofides, N.: Worst-case analysis of a new heuristic for the travelling salesman problem (No. RR-388). Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group (1976)
25. Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling-salesman problem. Oper. Res. **21**(2), 498–516 (1973)