Reinforcement Learning for Delay-Constrained Network Optimization

Wenjie Xu
SID: 1155118056
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
xw018@ie.cuhk.edu.hk

Gongpu Chen
SID: 1155136642
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
gpchen@link.cuhk.edu.hk

Abstract

Delay-constrained network optimization is an NP-hard optimization problem. Its objective is minimizing the total cost of transporting one agent from a node to another over a graph subject to a hard deadline constraint. Existing approximation algorithms, e.g. FPTAS (8), still incur high computational time complexity and can not scale up. We propose a reinforcement learning method to solve this problem. First, to fully reflect the graph structure, we design a graph embedding network to learn state features. On this basis, we design our Q-value approximator and use Q-learning to train our model, the learned policy incrementally construct a solution according to the output of graph embedding network. The experiment results show that our algorithm outperforms fastest agent/random agent by $24\%/30\%$ in terms of empirical approximation ratio.[1]

1 Introduction

Delay-constrained network optimization (DNO) refers to the problem that minimizing the cost of delivering perishable goods or information with hard deadlines over multi-hop networks. This problem arises from many practical applications, such as communication network scheduling, transportation and logistics. Take truck transportation as an example, on the one hand, more than 70% of freight tonnage in the U.S. are delivered by trucks (3), while about 34% of truck operation cost comes from the cost of fuel (15). Therefore, how to reduce fuel consumption is crucial for cost effectiveness. On the other hand, delay constraint is an important concern in such a network optimization. For example, Amazon has goods delivery time guarantee for customers (1), Uber Freight also guarantee truck transportation delay for shippers (2). In this case, network optimization, e.g., route planning over the complex transportation network, may significantly reduce the fuel cost on the premise of satisfying the constraint of delivery deadline. Similar problems also widely exist in communication networks, information bits incur transmission delay when traversing links, and it is often critical to constrain the end-to-end delay while minimize transmission costs over the network.

Despite the importance and ubiquity of delay-constrained network optimization, how to solve the delay-constrained network optimization problem is still facing two fundamental challenges. The first one is complexity, in fact, even a simplest form of delay-constrained network optimization, *i.e.*, restricted shortest path (RSP) problem, is NP-hard (9). Existing approximation algorithms and heuristics either lose optimality or are difficult to scale. The second challenge is robustness, the parameters in the delay-constrained network optimization problem are uncertain in many cases. For example, in energy-efficient timely truck transportation problem, truck's speed is subject to uncertain and dynamic speed range constraints (17). However, such uncertain and possibly dynamic parameters

[1] Code is available in https://github.com/JackieXuw/ierg6130_course_project.

are assumed to be known in existing solutions. Thus it is very likely that the solution violates the uncertain constraints.

Recent years, reinforcement learning (RL) has shown great power in solving sequential decision making problems, and many research effort has been devoted to develop RL method for complex optimization problems. For example, recently reinforcement learning has achieved some progress in combinatorial optimization problems like traveling salesman problem (TSP) (4) and vehicle routing problem (VRP) (13). Compared with the canonical TSP problem, delay-constrained network optimization is more sophisticated because of the deadline constraint. On the one had, it introduces a continuous element into the environment state. On the other hand, it requires a better model to formulate the deadline into the RL framework.

In this paper, we propose a deep Q-learning approach to address delay-constrained network optimization problems. The decision-making of DNO problem deeply depends on the structure of the underlying network, including its topology, cost and time consumption of each edge. Therefore, how to represent state features to better reflect the network structure is crucial for this problem. We first construct a graph embedding network to learning state features, on this basis, the learning agent behaves like a greedy algorithm to obtain a solution incrementally, and the action is determined according to the current state and the next state.

2 Related work

Recently, many research effort has been devoted on investigating learning-based heuristics for combinatorial problems. For example, the Pointer network(Ptr-net) proposed in (16) uses a softmax probability distribution as a "pointer" to address the fundamental problem of representing variable length dictionaries. This modified recurrent neural network can be applied to solve combinatorial optimization problems, but the training of this model relies on supervised signals given by an approximate solver. Based on Ptr-net, (4) proposes an architecture that using Ptr-net as a policy model to parameterize a stochastic policy to solve the TSP problem. The work in (13) develops a RL method for VRP problems, a parameterized stochastic policy that can handle different instances with the same size is trained to find near-optimal solutions. However, the architectures used in these works are generic, not yet effectively reflecting the combinatorial structure of graph problems.

To improve the performance of learning algorithms for graph problems, some graph embedding techniques have been developed (5; 10). A node2vec technique that can learn continuous feature representations for nodes in network was proposed in (11). Using a random walk strategy to search neighborhood for each node, this algorithm can learn a mapping function which can transform network nodes to feature vectors. Nodes2vec adpots short random walk strategy which focus only on the local structure and ignores global relationships, this may cause its non-convex objective function to be stuck in local optima. To address this problem, HARP (6) compresses the input graph before embedding, it creates several hierarchical levels for a graph, and then embeds them to obtain the graph embedding. The work in (7) put forward a light weight neural network that can better utilize graph structure. In particular, contrasting to recent approaches that adopt a graph-agnostic sequence-to-sequence mapping, they used a graph embedding network, called *structure2vec* (S2V) to represent the policy. In addition, they use Q-Learning to train their model, which is more sample efficient than policy gradient.

Dai et al. (12) proposed a learning algorithm for combinatorial optimization over graphs, which combines deep Q-learning and graph embedding to address the challenge. Their algorithm uses a graph embedding network called structure2Vec to parameterize the evaluation function, and then applies Q-learning to learn a greedy policy. Although many existing studies have proposed RL approaches for combinatorial problems, to the best of our knowledge, the network optimization problems with deadline constraint have not been well-solved, we thus design a algorithm to fill this gap in this paper.

3 Problem Formulation

3.1 Problem Statement

Consider a network that can be modeled as a directed graph $G(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the vertex set, and $\mathcal{E}$ is the edge set. Define $V \triangleq |\mathcal{V}|$ as the number of vertexes, and $E \triangleq |\mathcal{E}|$ as the number of edges. An edge from vertex u to vertex v is denoted by (u, v) or e_{uv}. For each edge $e \in \mathcal{E}$, let $c(e) > 0$ denote the cost of this edge, and $t(e) > 0$ denote the time consumption of going through this edge. For a transmission task from source node $s \in \mathcal{V}$ to destination node $d \in \mathcal{V}$, assume the deadline constraint for this task is D, which means that the time interval of departing from s and arriving at d cannot exceed D. Let $\mathcal{X}_{sd}$ denote the set of all paths that connects s and d, and each path $\mathbf{x} \in \mathcal{X}_{sd}$ is an edge set, where $\mathbf{x} = \{x_e : e \in \mathcal{E}\}$ is an E-dimensional binary vector defined as follows:

$$x_e = \begin{cases} 1, & \text{if edge } e \text{ is selected into this set,} \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Assume $\mathcal{I}(u) = \{(v, u) : (v, u) \in \mathcal{E}\}$ denotes the incoming edge set of node u, and $\mathcal{O}(u) = \{(u, v) : (u, v) \in \mathcal{E}\}$ is the set of out coming edges. Then, any $\mathbf{x} \in \mathcal{X}_{sd}$ satisfies

$$\sum_{e \in \mathcal{O}(u)} x_e - \sum_{e \in \mathcal{I}(u)} x_e = \mathbf{1}_{(u=s)} - \mathbf{1}_{(u=d)}, \quad \forall u \in \mathcal{V} \tag{2}$$

where $\mathbf{1}_{(\cdot)}$ is an indicator function.

Given a source-destination pair $s, d \in \mathcal{V}$, the delay-constrained network optimization problem aims to find one path that minimize the delivery cost while satisfying the deadline constraint, and we can formulate the problem as follows:

$$\text{DNO:} \quad \min_{\mathbf{x} \in \mathcal{X}_{sd}} \quad \sum_{e \in \mathcal{E}} x_e c(e) \tag{3}$$

$$s.t. \quad \sum_{e \in \mathcal{E}} x_e t(e) \leq D \tag{4}$$

As aforementioned, such an integer programming is NP-hard, and we will develop a reinforcement learning method to solve it.

3.2 Reinforcement Learning Formulation

To build a reinforcement learning framework, we reformulate the DNO problem in (3)-(4) as a Markov decision process (MDP). Basically, starting from the source node, each time we need to select an out coming edge to move to another node, until arriving at the destination node. In particular, we define the states, actions and rewards as follows:

1. States: let $S = \{s_i\}$ denote the state space, and each state is a tuple defined as $s_i = (v_i, t_i, d_i)$, where $v_i \in \mathcal{V}$ is the current vertex, $t_i \leq D$ is the remaining time of deadline, and d_i is the destination. There are two kinds of terminated states, 1) success states, if $v_i = d_i$ and $t_i \geq 0$, then the delivery task is completed successfully; 2) fail states, if $t_i < 0$, the task is failed because time is exhausted.
2. Actions: the action space is denoted by $A = \{a_i\}$. Given any non-terminated state $s_i = (v_i, t_i, d_i)$, an available action for this state is an out coming edge of node v_i, i.e., $a_i = (v_i, u_i) \in \mathcal{O}(v_i)$.
3. Transition: the transition is deterministic in this environment, at a non-terminated state $s_i = (v_i, t_i, d_i)$, taking an action $a_i = (v_i, u_i)$ will transfer the state to $s_i' = (u_i, t_i - \tau_i, d_i)$, where $\tau_i = t(v_i, u_i)$ is the time consumption of traveling through edge (v_i, u_i).
4. Rewards: the DNO problem aims to minimize the delivery cost, therefore we define the reward function of taking an action $a_i = (v_i, u_i)$ at state s_i and transitioning to a new state s_i' as follows:

   $$R_i = -c(a_i) \tag{5}$$

   By this way, maximizing the cumulative reward of a success state is equivalent to minimize the objective of DNO problem. In addition, for the fail states, we put a large negative penalty on the reward such that the cumulative reward would be small enough.

5. Policy: based on the Q-value function, a greedy algorithm will be used to determine an action at each state.

4 Proposed Approach

As aforementioned, we expect a Q-value function that can evaluate the current partial solution with the network structure being considered. Intuitively, how good is a state depends on the connection between the current vertex and the destination. Therefore, we introduce a network embedding technique, i.e., *Structure2Vec*, to represent state features, and on this basis, we can parameterize the Q-value function, Fig. 1 shows the diagram of our design.

4.1 Feature construction and Q-Learning

Structure2Vec was first proposed in (7), which can compute a multi-dimensional feature for each node with a partial solution. We proposed a modified Structure2Vec to accommodate the DNO problem, in which the cost and time consumption of all out coming edges of the current vertex are taken into account. In particular, with an input network $G(\mathcal{V}, \mathcal{E})$, a p-dimensional feature of a state $s = (v, t_r, d)$ is computed by an iterative equation as follows:

$$\begin{aligned} f_s^{(t+1)} =& \theta_1 \sum_{e \in \mathcal{O}(v)} \frac{f_e^{(t)}}{|\mathcal{O}(v)|} + \theta_2 \sum_{e \in \mathcal{O}(v)} \frac{\theta_3 c(e)}{|\mathcal{O}(v)|} + \theta_4 \sum_{e \in \mathcal{O}(v)} \frac{\theta_5 t(e)}{|\mathcal{O}(v)|} + \theta_6 t_r + \theta_7 t_f \\ &+ \theta_8 \big[c(e_1); t(e_1)\big] + \theta_9 \big[c(e_2); t(e_2)\big] \end{aligned} \tag{6}$$

where θ_1-θ_9 are model parameters with suitable dimensions, $\mathcal{O}(v)$ is the set of out coming edges of the current vertex v. We design three summation terms over all out coming edges of v to aggregate the information of achievable neighbors that are invariant to permutations over neighbors. Among which, $f_e^{(t)}$ is the feature of new state after taking action e, we initialize $f_e^{(0)}$ at each node as 0; $c(e)$ and $t(e)$ are the cost and time consumption of edge e. Depending on the network topology, these three summation terms give an average evaluation on the current vertex. In addition, we also consider the best action that can be applied in the current vertex by the last tow terms, with

$$e_1 = \arg \min_{e \in \mathcal{O}(v)} \{c(e)\}, \quad e_2 = \arg \min_{e \in \mathcal{O}(v)} \{t(e)\}$$

which reflect the minimum cost and time consumption to move to the next vertex from the current vertex. Further, t_r is the remaining time of deadline in the current state, and t_f is the fastest time to arrive at the destination from the current vertex, which is calculated by a near optimal minimum weight path algorithm, these two terms evaluate the state from the perspective of time, which is associated with the deadline constraint.

Based on (6), the features are aggregated recursively according to G's network topology. At round $t + 1$, we will calculate all features of relevant vertexes simultaneously based on the existing features of round t. *Relevant* means the vertexes and their adjacent vertexes of round t. After T steps of recursion, the final feature of current vertex v will be obtained by taking into account both network characteristics and long-range interactions between these node features.

After we get the embeddings from *structure2Vec*, we can apply them to parameterize the Q-value function. More specifically, we use a linear approximator and the input are features of the current state and the next state (after taking action a), i.e.,

$$\hat{Q}(s, a) = \lambda_1^\top ReLU\big([\lambda_2^\top f_s^{(T)}; \lambda_3^\top f_a^{(T)}]\big) \tag{7}$$

where λ_1, λ_2 and λ_3 are model parameters, $f_a^{(T)}$ is the feature of new state after taking action a, ReLU is the rectified linear unit, i.e., $ReLU(z) = \max(0, z)$. Since the features $f_s^{(T)}$ and $f_a^{(T)}$ are computed from the *structure2Vec* on the basis of parameters $\Theta \triangleq \{\theta_i\}_{i=1}^9$, $\hat{Q}(s, a)$ will depends on a collection of 12 parameters Θ and $\Lambda \triangleq \{\lambda_i\}_{i=1}^3$, and all these parameters will be learned.

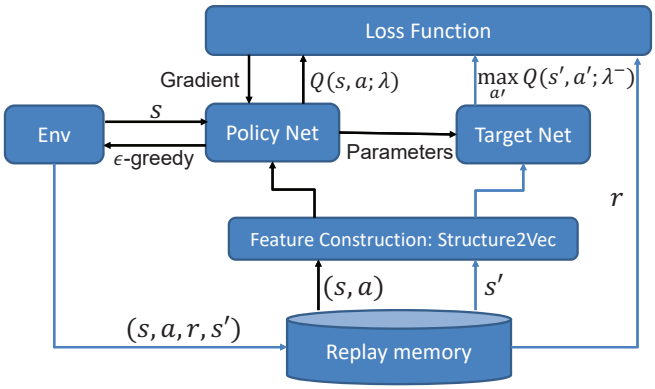

Figure 1: Deep Q-learning combined with Structure2Vec.

4.2 Training

The DNO problem is generalized from the shortest path problem, of which we have the well known polynomial solution. In order to take advantage of it, we use supervised learning to train our network on instances generated by the shortest path algorithm, as shown in Algorithm 1.

Algorithm 1: Supervised Learning for Initialization

Input: Graph $G(\mathcal{V}, \mathcal{E})$;
D=generateInstances(G);
S=generateSolution(D);
for *iteration* $i = 1$ *to* L **do**
  Randomly sample instances batch B from S ;
  for $j = 0$ *to* $|B|$ **do**
    Initialize time budget t_d, source s, destination d, solution path P and target T according to $B[j]$.
    for *node* $n \in P$ **do**
      Calculating $\hat{Q}(G, s, n, t_d, d)$
      Accumulate gradient according to loss function $(q - T[s][n])^2$
      Update s, t_d according to G.
    end
  end
  Perform gradient descent over accumulated gradients and clean gradients.
end

In order to perform and end-to-end learning of the parameters in $\hat{Q}(s, a)$ and $f_s^{(T)}$, we use a combination of n-step Q-learning and fitted Q-learning (14), as illustrated in Algorithm 2. We use the term iteration to refer to a complete sequence of action making starting from the source node, and until termination. Standard (1-step) Q-learning updates the function approximator's parameters at each step of an episode by performing a gradient step to minimize the squared loss:

$$Loss = (y - \hat{Q}(s_t, s_{t+1}; \Theta, \Lambda))^2 \tag{8}$$

where $s_t = (v_t, t_r, d)$ is non-terminal state, which will transit to s_{t+1} after taking action a; while $y = \gamma \max_{u \in \mathcal{O}(v_{t+1})} \hat{Q}(s_{t+1}, u; \Theta, \Lambda) + r(s_t, s_{t+1})$.

A natural extension of 1-step Q-learning is to wait n steps before updating the approximator's parameters, so as to collect a more accurate estimate of the future rewards. Formally, the update is over the same squared loss, but with a different target:

$$y = \sum_{t=0}^{n-1} r(s_{t+i}, s_{t+i+1}) + \gamma \max_{u \in \mathcal{O}(v_{t+n})} \hat{Q}(s_{t+n}, u; \Theta, \Lambda) \tag{9}$$

Instead of updating the Q-function sample-by-sample as in Equation (8), we use experience replay to update the function approximator with a batch of samples from a dataset E, rather than the single sample being currently experienced. In addition, we use off-policy reinforcement learning in this paper, because it is more sample efficient.

Algorithm 2: Q-learning for the DNO Problem

Input: Graph $G(\mathcal{V}, \mathcal{E})$;
Initialize experience replay memory E to capacity N;
Generate an instance from $G(\mathcal{V}, \mathcal{E})$ to get source s, destination d, and deadline D;
for *episode* $i = 1$ *to* L **do**
  Initial state $s_0 \leftarrow (s, D, d)$, solution set $I \leftarrow \emptyset$;
  for *step* $k = 0$ *to* $|\mathcal{V}|$ **do**
    $a_{k+1} = \begin{cases} \text{Random edge } e \in \mathcal{O}(v_k), & \text{w.p. } \epsilon, \\ \arg\max_{e \in \mathcal{O}(v_k)} \hat{Q}(G, s_k, e), & \text{otherwise.} \end{cases}$
    $t_{k+1} = t_k - t(a_{k+1})$;
    $r_k = -c(a_{k+1})$;
    if $t_{k+1} < 0$ **then**
      $r_k = -P_e$, where penalty P_e is a large positive number;
    end
    $I \leftarrow I \cup (s_k, a_{k+1}, r_k)$, where $s_k = (v_k, t_k, d)$;
    if $k \geq n$ **then**
      Add $(s_{k-n}, a_{k+1-n}, r_{k-n})$ to E
    end
    if $(v_{k+1} = d$ *and* $t_{k+1} \geq 0)$ *or* $(t_{k+1} < 0)$ **then**
      Break;
    end
  end
  Sample random batch B from E;
  Update Θ and Λ by SGD over (8) for B
end

5 Experiments

5.1 Experiment Set-up

We use one Intel(R) Core(TM) i5-4210M CPU @ 2.60GHz cpu to do experiments.

Network Topology. Since there is no commonly used DNO/RSP problem instances, we use the US national highway network (8) and extract a sub-network with 63 nodes and 143 edges as our network. And we randomly assign a driving time and corresponding fuel cost to each edge.

Source-destination pairs. We randomly select two nodes as source and destination from the graph. To make sure that the problem instance is feasible, we need to make destination reachable from source. If not reachable, we abandon this source-destination pair and randomly sample source-destination pair again.

Deadline. To make our problem instance feasible, we need to set the deadline large enough so that at least fastest path solution can arrive at the destination in time. So for one instance, we first calculate the fastest path's time t_f from s to d. Then we randomly set the deadline to be between $1.0t_f$ and $2.0t_f$.

Deadline satisfaction issue. Since all pairs' fastest path time is easy to derive in our setting using Floyd-Warshall algorithms. We first derive all pairs' fastest time. And before our agent takes action, we abandon the actions that leads to late arrival by comparing remaining time and fastest path time to destination. Thus we can guarantee any agents output feasible solution.

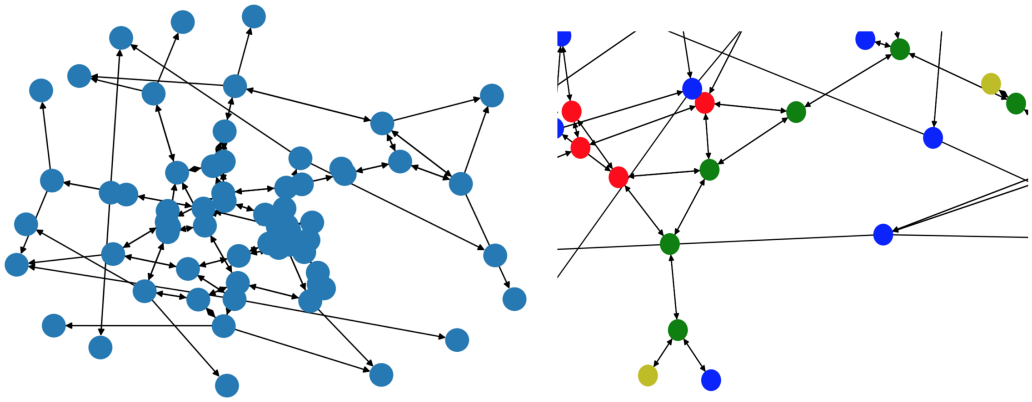

Figure 2: Modified American Highway Network. **Left:** The overall structure of this graph. **Right:** Yellow nodes are the source and the destination. When time is sufficient, our agent will select the most economical path(red) instead of fastest path(green).

5.2 Adaptive paths selection with different deadlines

For real world applications, we created a graph by modifying the American Highway Network(Figure (2)). Results show that our approach can effectively switch between different paths regarding time budget. For instance, when time is sufficient, our agent will select the most economical path(red) instead of fastest path(green).

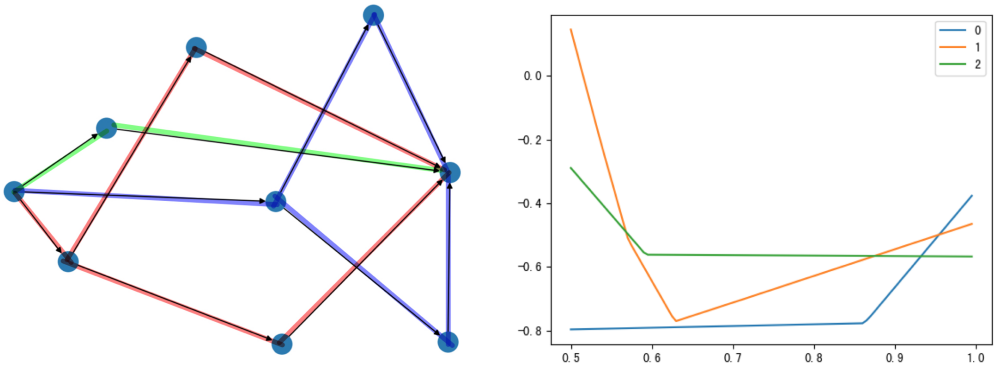

Figure 3: Solution on a simple graph **Left:** At the leftmost node, there are three feasible state-action pairs –the green, the red and the blue. **Right:** Q values of the three state-action pairs. The x-axis is the time budget at the leftmost node. The y-axis is the Q value(bigger is better.) When time budget is increasing, the rankings of the three Q values are changing, which shows that our algorithm will switch to different actions with regarding to the time budget.

To illustrate our works better, we designed a simple graph with 9 nodes and plot the Q-value of our approximator $\hat{Q}$ with regarding to time budget(Figure (3)).

Due to the limitation of express power of our model, the q-value function does not approximate the real one exactly. However, the rankings are true in most circumstance.

5.3 Performance comparison to baselines

We run our training algorithms for 200 iterations. Fig. 4 shows the average reward over 100 episodes of our agents and random agents in training process. It can be observed our agent can achieve

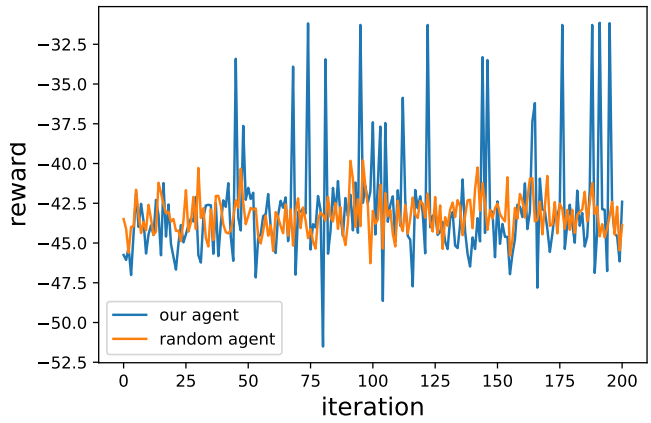

Figure 4: The evolution of average reward over 100 episodes of our agents in training process.

significantly larger reward than the random agent. However, we find that our agent can be unstable in the training process.

To evaluate the solution quality on test instances, we use the approximation ratio of each method relative to the optimal solution, averaged over the set of test instances. The approximation ratio of a solution S to a problem instance G is defined as,

$$R(S,G) = \frac{c(S)}{c(OPT)}, \tag{10}$$

where $c(S)$ is the total cost of the path selected by S and $OPT(G)$ is the optimal cost derived by enumerating all the feasible paths. Tab. 1 gives different agents' empirical approximation ratio.

Agents	Random	Fastest	Ours
Empirical approximation ratio	1.67	1.54	1.17

Table 1: Different algorithms' empirical approximation ratio

We can observe that our agents' approximation ratio is very close to 1, 30%/24% smaller than the random/fastest agent.

6 Conclusion

In this paper, we proposed a reinforcement learning approach for delayed constrained network optimization problems. Central to our design is the combination of *structure2Vec* with deep Q-learning. We leveraged *structure2Vec* as a graph embedding network to represent state features, based on which the Q-value function was constructed. Extensive experiments demonstrate the effectiveness of our algorithm. Specifically, the experiment results show that our algorithm outperforms fastest agent/random agent by $24\%/30\%$ in terms of empirical approximation ratio.

References

[1] Place an order with guaranteed delivery. `https://www.amazon.com/gp/help/customer/display.html/ref=hp_left_v4_sib?ie=UTF8&nodeId=201117390`.

[2] Uber freight. `https://www.uberfreight.com/`.

[3] Improving the fuel efficiency of american trucks– bolstering energy security, cutting carbon pollution, saving money and supporting manufacturing innovation. The White House, 2014.

[4] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *CoRR*, abs/1611.09940, 2016.

[5] H. Cai, V. W. Zheng, and K. C. Chang. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1616–1637, 2018.

[6] H. Chen, B. Perozzi, Y. Hu, and S. Skiena. Harp: Hierarchical representation learning for networks. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2702–2711. JMLR.org, 2016.

[8] L. Deng, M. H. Hajiesmaili, M. Chen, and H. Zeng. Energy-efficient timely transportation of long-haul heavy-duty trucks. *IEEE Transactions on Intelligent Transportation Systems*, 19(7):2099–2113, 2018.

[9] M. R. Garey and D. S. Johnson. Computers and intractability: A guide to the theory of np-completeness. 1979.

[10] P. Goyal and E. Ferrara. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

[11] A. Grover and J. Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[12] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[13] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, pages 9839–9849, 2018.

[14] M. Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328.

[15] W. F. Torrey and D. Murray. An analysis of the operational costs of trucking: A 2015 update. 2015.

[16] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015.

[17] W. Xu, Q. Liu, M. Chen, and H. Zeng. Ride the tide of traffic conditions: Opportunistic driving improves energy efficiency of timely truck transportation. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, pages 169–178, 2019.