Learn to Optimize the Constrained Shortest Path on Large Dynamic Graphs

Yin, Jiaming; Rao, Weixiong; Zhao, Qinpei; Zhang, Chenxi; Hui, Pan

2024-03

Institute of Electrical and Electronics Engineers Inc.

http://hdl.handle.net/10138/591343

Yin, J, Rao, W, Zhao, Q, Zhang, C & Hui, P 2024, 'Learn to Optimize the Constrained Shortest Path on Large Dynamic Graphs', IEEE Transactions on Mobile Computing, vol. 23, no. 3, pp. 2456-2469. https://doi.org/10.1109/TMC.2023.3258974

Learn to Optimize the Constrained Shortest Path on Large Dynamic Graphs

Jiaming Yin, Weixiong Rao, *Member, IEEE*, Qinpei Zhao, Chenxi Zhang, and Pan Hui, *Fellow, IEEE*

***Abstract*—The constrained shortest path (CSP) problem has wide applications in travel path planning, mobile video broadcasting and network routing. Existing works do not work well on large dynamic graphs and suffer from either ineffectiveness or low scalability issues. To overcome these issues, in this paper, we propose an efficient and effective solution framework, namely CSP_GS. The solution framework includes two key components: (1) the techniques to decompose a large CSP instance into multiple small sub-instances and (2) the developed learning model CSP_DQN to solve small CSP instances. The evaluation result on real road network graphs indicates that our approach CSP_GS performs well on large dynamic graphs by rather high quality and reasonable running time, and particularly adapt to significant graph changes even with broken edges. To the best of our knowledge, this is the first learning-based model to well solve the CSP problem on large dynamic graphs.**

***Index Terms*—Constrained shortest path, combinatorial optimization, reinforcement learning, dynamic graphs.**

I. Introduction

THE constrained shortest path (CSP) problem is to find the cheapest path between two input nodes, meanwhile the found path should satisfy a given constraint condition. The CSP problem has been widely applied in travel path planning, mobile video broadcasting and network routing [7], [12], [44], [46]. For example, in daily travel, drivers would like to minimize the travel time and also expect that the travel expense (e.g., fuel cost or road toll) should not exceed a certain threshold.

The CSP problem has been proven to be NP-hard [17], and is much harder and more general than the simple shortest path problem without any constrained condition. The classic approaches to solve CSP, such as the Pareto-optimal labeling algorithms [7], [17], [41] and A* heuristic algorithm [14], unfortunately do not work well on dynamic graphs. For example, it is rather common that the traffic is congested in some peak period (e.g., 4:30-5:30 PM) and yet empty in midnight [10]. The plan of shortest travel routes in urban road networks suffers from graph changes (e.g., dynamic route time on road segments or even broken road segments). Using the CSP path on static graphs and ignoring graph changes could lead to the ineffectiveness issue. Alternatively, we could re-optimize from scratch the classic approaches above whenever graphs change. Yet, due to frequent graph changes, such re-optimization is inefficient especially if graph size is large.

Recently machine learning becomes popular to solve hard graph problems. Particularly when graphs change in any way, learning approaches are robust and can adapt their solution. For example, to solve the Traveling Salesman Problem (TSP), the works [5], [45] exploited recurrent neural networks (RNN) to learn a probability distribution over various vertex permutations. Next, the works [22], [31] are rather effective to solve the Vehicle Routing Problem (VRP), almost comparable to the highly optimized and specialized operational research (OR) approaches. However, these learning approaches work well only on small graphs with tens or at most hundreds nodes. How to make these learning approaches become scalable to large graphs is challenging.

Until now, the literature works above suffer from the ineffectiveness issue caused by graph changes or work poorly on large graphs. To tackle the issues above, in this paper, we propose an efficient and effective learning framework to solve the CSP problem on large dynamic graphs. Specifically,

1) In terms of *learning efficiency* on large graphs, we decompose a large CSP instance into multiple small sub-instances. After solving the CSP sub-instances, we merge the sub-paths found on such sub-instances into a full feasible path on the original instance. In this way, we have chance to solve a large instance efficiently. The key of the decomposition is to sample important candidate paths and select crucial nodes that are likely to appear within CSP solution paths.
2) In terms of *learning effectiveness* on dynamic graphs, we develop a Deep Q-learning network (DQN) to learn feasible solutions for CSP sub-instances. By exploiting Graph Convolution Networks (GCNs) to learn embedding vectors from dynamic graphs as input state, our DQN policy chooses a 2-hop neighbour node as the action to maximize the Q-value. Unlike the simple approach using a one-hop neighbour as an action, our work reduces DQN iterations to efficiently solve the CSP sub-instance. Meanwhile, after pre-trained on static graphs, the DQN model has chance to work well on dynamic graphs.

Manuscript received 29 October 2022; revised 19 February 2023; accepted 7 March 2023. Date of publication 20 March 2023; date of current version 5 February 2024. This work was supported by the NSFC under Grant 61972286. Recommended for acceptance by M. Rossi. *(Corresponding author: Weixiong Rao.)*

Jiaming Yin, Weixiong Rao, Qinpei Zhao, and Chenxi Zhang are with Tongji University, Shanghai 200070, China (e-mail: 14jiamingyin@tongji.edu.cn; wxrao@tongji.edu.cn; qinpeizhao@tongji.edu.cn; xzhang2000@163.com).

Pan Hui is with the System and Media Laboratory (SyMLab), Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, and also with the Department of Computer Science, University of Helsinki 00100 Helsinki, Finland (e-mail: panhui@cse.ust.hk).

Digital Object Identifier 10.1109/TMC.2023.3258974

As a summary, we make the following contributions.

- To the best of our knowledge, this is the first learning-based model[1] to well solve the CSP problem on large dynamic graphs to balance the tradeoff between solution CSP path quality, solution efficiency and constraint consumption.
- We propose to decompose large graphs via the developed path sampling and node score estimation techniques to efficiently solve large CSP instances. Meanwhile, the developed CSP_DQN model, pre-trained on static graphs, can effectively learn feasible paths on dynamic graphs for CSP sub-instances.
- Evaluation on real road network graph data in Shanghai China and Koln Germany indicates that our approach, namely CSP_GS, can find rather high quality CSP paths with fast running time. For example, on a large Shanghai graph with 10889 nodes, CSP_GS exhibits very high scalablity, about 587 $\times$ faster than a labeling approach (namely LA_3-dg), meanwhile with an approximation ratio only 3.214, and on the dynamic graphs with broken edges, CSP_GS demonstrates the adaptive ability to significant graph changes.

The rest of the paper is organized as follows. Section II first reviews background and related works, and Section III next formulates our problem definition. After Section IV introduces our framework, Section V and Section VI give the details of graph decomposition and DQN model, respectively. Section VII then evaluates our solution and Section VIII finally concludes the paper.

II. Background and Related Works

A. Background

Deep Q Learning: As a topic of machine learning, reinforcement Learning (RL) [40] has been applied in decision making problems by the mapping from states to actions so as to maximize a numerical reward signal. Among various RL models, Q-learning [40] is a value-based method. In the traditional Q-learning, a table is used to store the Q values for any state s and action $a = \pi(s)$, where π is a policy function. However, the traditional method fails in high-dimensional state space. To solve this issue, the previous works [28], [29] use either a linear function or a neural network function (i.e., Deep Q-Network, DQN) to estimate the action-value function. To reduce the overestimation issue of Q-learning in certain conditions, the work [43] develops the Double Q-learning algorithm, which evaluates the greedy policy with the *online network* and meanwhile uses a *target network* to estimate the value. Dueling network [47] is another improvement to DQN, by using two separate estimators: one for the state value function and one for the state-dependent action advantage function.

Graph Convolution Networks (GCNs)*:* Convolution Neural Network (CNN) can effectively extract local patterns of regular grid data but is inapplicable to general graphs. GCNs generalize the convolution operation to graph structures. GCNs are divided into two main categories: (1) *Spatial-based approaches* formulate graph convolutions as aggregating feature information from neighbors, and (2) *Spectral-based approaches* define graph convolutions by introducing filters from the perspective of graph signal processing. Spectral CNN [6], ChebNet [9] and 1stChebNet [21] are the representative approaches on spectral-based GCN.

The work [6] proposes the first spectral convolution neural network (Spectral CNN) and defines the convolution operation in the Fourier domain by computing the eigendecomposition of the graph Laplacian. However, this operation needs expensive cost to compute the eigendecomposition. Next, to avoid the expensive computation of eigendecomposition, ChebNet [9] defines a filter as Chebyshev polynomials of the diagonal matrix of eigenvalues with the complexity linear to the number of edges. After that, the work [21] proposes a first-order approximation of ChebNet. For a signal with C channels, (i.e., a C-dimensional feature vector for each node), the operation is defined as the multiplication of a signal $X \in \mathbb{R}^{N\times C}$ with a filter g_Θ

$$g_\Theta * X = (I_N + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})X\Theta = \left(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}\right)X\Theta, \tag{1}$$

where $\Theta \in \mathbb{R}^{C\times F}$ is a matrix of filter parameters. $\widetilde{A} = A + I_N$ and $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$ are the renormalized adjacent matrix and degree matrix. This work shows impressive performance in many node classification tasks and is considered as a strong baseline in the research community.

GCN offers the power to extract local features from graph structures and has been applied in many graph combinatorial optimization problems like Minimum Vertex Cover (MVC), TSP [19], Maximal Independent Set (MIS) [23] and Influence Maximization (IM) [25]. In this work, we leverage GCN and DRL for dynamic graphs.

B. Related Works

The Constrained Shortest Path (CSP) Problem is a well-known NP-hard problem [17]. Researchers have proposed many approaches to solve the CSP problem, such as path ranking [37], [38] dynamic programming [7], [18], [35] and labeling algorithm [17], [34], [41].

First, the path ranking approaches rank candidate paths in non-decreasing path weight until a feasible solution has been found. They are inefficient when the graph size grows and more paths exist from source to destination. [38] improves the path ranking algorithms with a binary partition and pruning strategy to tackle the inefficiency issue but the scalability issue still exists. The dynamic programming approach, such as Joksch's algorithm [18] does not work well for the general CSP problem with *continuous numeric* constraint consumption, suffering from discretization errors. Thus, the work [7] uses the randomized path discretization technique to mitigate the error.

Labeling algorithms are predominant algorithms to provide Pareto-optimal solutions for the CSP problems. They use a label to represent a partial path from source node to some node in graph and store the information about the resource consumption and weight along the partial path. By expanding the partial paths recursively, the labeling algorithms can finally obtain the

[1]This is an extension work to the conference paper [48].

feasible path with minimum weight. One issue is that the number of labels explodes as the graph size grows. Bidirectional labeling strategies [34], [41] mitigate the explosion labels. Nevertheless, such improvement is still inefficient to solve the large-scale CSP problems due to high memory cost required to maintain the labels of sub-paths. Some works combine the traditional approaches with heuristic search strategies like bidirectional A* search [2], [27]. The recent work [2] introduces several heuristics in the resource constrained path finding context on the top of a bidirectional A* algorithm. However, such an improvement requires problem-specific domain knowledge and significant trial-and-error efforts.

However, these works above do not take into account graph changes and all assume that graphs are fully static. Given the CSP problem on dynamic graphs, they will suffer from either the ineffectiveness issue or the inefficiency problem that have been introduced in Section I, and these issues motivate our work in this paper.

In addition, on dynamic graphs (with the changes of edge weights, insertion or deletion of vertices and/or edges), the previous work [3] solves the simple shortest path problem with no constraints. However, this paper implicitly assumes that the changes of the input graph are available before running the proposed algorithm, and it can construct a shortest path tree rooted at the source to maintain the changes. Yet, in our case, the input graphs are changed in an online manner and hard to maintain such a tree structure. Thus, we employ the DQN to in almost real time fashion select an action to adapt to graph dynamics.

Learning Combinatorial Optimization Problems: Due to the success of deep learning, the literature has developed many learning-based approaches to solve combinatorial optimization problems, including the supervised Pointer Network [45] and RL-based approaches [1], [5], [19], [22], [31]. However, these previous approaches usually work well on small TSP instances, typically with graph size equal to *tens and at most hundreds* of graph vertices. To address the scalability issue, GCOMB [25] first exploits GCN to prune poor nodes, then learns the embeddings of good nodes by supervised learning, and next exploits a Q-Learning component to select the solution set from the good nodes. The recent work [11] generalizes a pre-trained small-scale model to large TSP instances by repetitively using the small-scale model to build heat maps for large TSP instances, and performs Monte Carlo Tree Search (MCTS) on the heat maps to search final solutions. In addition, some works target the hard optimization problems in such scenarios as routing [13], [30], scheduling [26] and planning [16], [32].

Practical optimization problems usually involve certain constraint conditions, and very few previous learning-based works focus on constraint-based combinatorial optimization problems. The work [39] adds penalty signals from constraint dissatisfaction to the reward signals when solving the constrained Job Shop and Resource Allocation problem, and the work [24] learns a hierarchical policy to find the feasible solution to TSP with time windows (TSPTW). In addition, the aforementioned works [5], [19], [22], [31] do not give specific solutions to tackle the constraints and yet simply exploit a masking scheme to avoid infeasible actions during the decoding process.

The CSP problem studied in this paper significantly differs from the graph combinatorial optimization problems above. For example, a TSP instance on an input graph involves an associated TSP tour connecting entire graph vertices. Yet in our case, an input graph is with various shortest feasible paths, depending upon the input source-destination pairs. Moreover, a shortest feasible path involves a very small number of graph vertices. Thus, we cannot directly apply the approaches above to solve the CSP problem. Nevertheless, inspired by the success of DRL to solve such combinatorial optimization problems, we develop a DQN model to solve the CSP problem.

Learning Shortest Path Problem: In terms of learning-based approaches, very few works are developed to solve either the CSP problem or the simple shortest path problem with no constraint condition. For example, our previous work [13] proposes a dynamically adjustable route planning algorithm to plan the route which minimizes the travel time from the given source to destination. This work models the path-finding strategy as a discrete Markov decision process and learns the strategy with a dueling DQN. Yet, by dividing an entire 2D area into multiple grid cells, [13] does not take into account the graph topology structure constraint, which is yet our main focus.

In addition, the work [30] studies the packet routing problem in computer networks and develops a RL-based routing algorithm. However, when the network size grows, this work suffers from high dimensional space of the state. The previous work [4] aims to solve the navigation problem in 3D environments on uncertain topological maps. This work trains a neural graph-based planner in a supervised way and treats path planning as a classification problem. For a given target, the planner will predict the subsequent node on the optimal path to the target for the current node. Instead, our work does not require training labels and learns a RL model for next-hop node selection. To the best of our knowledge, there haven't been learning-based methods to solve CSP problem on large dynamic graphs.

III. Problem Statement and Challenges

In this section, we first introduce some definitions, next define the problem, and then highlight the challenges.

A. Problem Statement

Definition 1: [Dynamic Graph] In an directed graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, we denote $\mathcal{V} = \{v_i\}_{i=1}^{N}$ to be a set of N nodes v_i, $\mathcal{E}$ a set of edges, and $A \in \mathbb{R}^{N \times N}$ a binary adjacency matrix. The element $a_{v_i \to v_j} = 1$ indicates an edge from v_i to v_j and otherwise $a_{v_i \to v_j} = 0$ no edge.

Definition 2: [Path] Given a source-destination vertex pair $\langle v_s, v_d \rangle$ with $v_s \neq v_d \in \mathcal{V}$, a path $P(v_s, v_d)$ is a node sequence $\{v_j\}_{j=1}^{L}$, where $v_1 = v_s, v_L = v_d$ and $a_{v_j \to v_{j+1}} = 1$ indicating that the two sequential nodes v_j and v_{j+1} are connected by an edge.

Definition 3: [Edge/Path Weight] Each edge $e \in \mathcal{E}$ is with a certain weight $W(e)$. For a path p on a graph $\mathcal{G}$ with totally L nodes between the source v_1 and the destination v_L, we define the

path weight as $W(P) = \sum_{j=1}^{L-1} W(v_j \to v_{j+1})$. Here, the item $W(v_j \to v_{j+1})$ indicates the weight on the edge $v_j \to v_{j+1}$.

An edge (and a path) may involve multiple weights. Among these weights, some are fixed (e.g., edge distance) and yet others change over time (e.g., travel time on an edge in various time periods). We thus define the following dynamic weight.

Definition 4: [Dynamic Weight] When a route arrives at the source v_j of an edge $v_j \to v_{j+1}$ at the time step t_j, we define $W_{t_j}(v_j \to v_{j+1})$ as the *dynamic edge weight* at time t_j. Moreover, we assume that $W_{t_j}(v_j \to v_{j+1})$ is available when and only when the route arrives at v_j. Similarly, we define the *dynamic path weight* of P as $W_{t_1}(P) = \sum_{j=1}^{L-1} W_{t_j}(v_j \to v_{j+1})$ at the time step t_1 starting from the source v_1 until the final destination v_L. Intuitively, the path weight is the accumulation of dynamic edge weights on the time step t_j arriving at the starting node v_j of each intermediate edge $v_j \to v_{j+1}$.

Note that the definition above means that the dynamic weight $W_{t_j}(v_j \to v_{j+1})$ are available only at the time t_j when the CSP route starts from v_j to v_{j+1}, but unavailable or hard to predict at time $t' < t_j$. Meanwhile, it implicitly assumes that edge weight remains fixed during the route within the edge. That is, Such an assumption makes sense in many applications. For example in the travel route in road networks, even if road weights change within the route of a road segment, we do not re-schedule the route in the mid-point of the road segment and avoid turning back to the starting node of the road segment for driving safety. Instead, a safer way is to plan new routes at the arrival at the endpoint of the road segment.

Definition 5: [Feasible Path]: Given a path p and a certain weight type C, we say that the path P is feasible if the associated path weight of P is not greater than a given constraint threshold $\mathbf{C}$, i.e., $C(P) \leq \mathbf{C}$.

Problem 1: For a CSP instance $\langle \mathcal{G}, v_s, v_d, \mathbf{C} \rangle$ with four elements: a dynamic graph $\mathcal{G}$, the source-destination pair v_s and v_d, and the constraint threshold $\mathbf{C}$, our optimization objective is to find one feasible path P, which minimizes the path weight $W_{t_0}(P)$, i.e.,

$$\min_{P \in \mathcal{P}_{v_s, v_t}} W_{t_0}(P)$$
$$\text{s.t.} \qquad C(P) \leq \mathbf{C}, \tag{2}$$

where $\mathcal{P}_{v_s, v_t}$ is the set of all feasible paths on graph $\mathcal{G}$ from v_s to v_d and t_0 is the time step when the route starts from the source v_s.

B. Challenges

Solving the CSP problem is hard due to the fact that the dynamic edge weight above depend upon not only the edge itself but also the arrival time. As a result, at time step t_1, when a route arrives at the source v_1 of the path P above, we may have the current graph snapshot and find the current weight $W_{t_1}(v_j \to v_{j+1})$ for an intermediate edge $v_j \to v_{j+1}$. Nevertheless, this weight $W_{t_1}(v_j \to v_{j+1})$ may differ from the true one $W_{t_j}(v_j \to v_{j+1})$ at the time step t_j when the route really arrives at v_j. It means that the true weight $W_{t_j}(v_j \to v_{j+1})$ and path weight $W_{t_1}(p) = \sum_{j=1}^{L-1} W_{t_j}(v_j \to v_{j+1})$ are unavailable even at time t_1. Particularly when the prediction of $W_{t_j}(v_j \to v_{j+1})$ is hard at time step t_1, e.g, due to unexpected traffic accident, we have to make nearly real-time decision only based on the current graph information with help of a RL model.

We now introduce a *baseline solution* to solve CSP. That is, we repeatedly perform an existing CSP solver (i.e., the classic Pareto-optimal labeling algorithm) on every intermediately met node during the route from v_s to v_d. Specifically, we first find the shortest feasible path from v_s to v_d on the current graph snapshot $\mathcal{G}_{t_0}$ at time step t_0. With help of the found path, we have the next-hop neighbour node, say v_t, the corresponding edge weight $W_{t_0}(v_s \to v_t)$ and constraint consumption $C(v_s \to v_t)$. When arriving at the node v_t at time step t, we again perform the CSP solver on the new graph snapshot $\mathcal{G}_t$ of time step t to find the shortest feasible path from v_t to v_d and the next hop neighbour node $v_{t'}$. Again we have the edge weights $W_t(v_t \to v_{t'})$ and $C(v_t \to v_{t'})$. We repeat the steps above until the route reaches the final destination v_d. Now, we can find the shortest feasible path $P = \{v_s \to v_t \to v_{t'} \to \ldots \to v_d\}$.

It is not hard to find that the baseline solution above works in a greedy fashion and leads to the Pareto-optimal shortest path under the aforementioned assumption (i.e., edge weights do not change within the route of road edges and future dynamic weights W_{t_j} are difficult to predict at the time $t' < t_j$ and only available at time t_j when the route arrives at the starting nodes of road segments). It is mainly because, at every time step $t \to t' \to \ldots$ above, the CSP solver in the baseline solution can find an optimal next-hop neighbour node. However, the baseline solution suffers from *inefficiency* issue: we have to repeat the CSP solver by $|P|$ times where $|P|$ is the node count of path P. Given a CSP instance on large graphs, we have a great number $|P|$ and high running time.

To tackle the inefficiency issue, our work in general involves the trade-off between faster running time and a sub-optimal path. That is, we expect that our learning-based solution can achieve faster running time and produce an alternative path P' from v_s to v_d almost comparable to the optimal one P.

IV. CSP Solution Framework

In this section, we first give the key observation to efficiently and effectively solve CSP. Unlike the TSP tour to visit the entire graph nodes, a certain CSP route from v_s to v_d requires only a small number of graph nodes. Moreover, for a given dynamic graph, the CSP path varies depending upon the input pair $\langle v_s, v_d \rangle$. Thus, among the entire graph nodes, only a small number of graph nodes are crucial for a given CSP instance. Following this observation, the key of the CSP path is how to select those *crucial nodes*, which are more likely to appear within the CSP path than others.

Fig. 1 illustrates the pipeline of our proposed approach, which mainly consists of two steps.

1). For a given CSP instance, we first estimate a node score to measure the possibility of such a node to be within the CSP path. According to node scores, we select crucial nodes and then decompose an input graph into multiple sub-graphs and have corresponding small CSP instances.

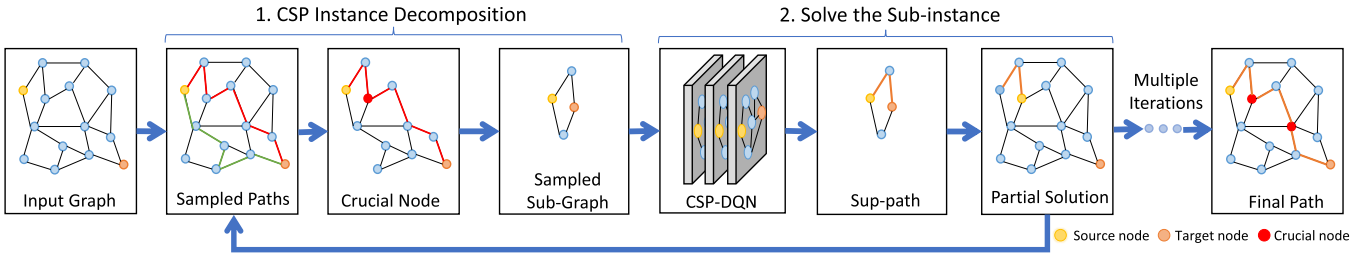


Fig. 1. Pipeline of the proposed approach.

Algorithm 1: CSP_GS.

input : CSP instance $\langle \mathcal{G}, v_s, v_d, \mathbf{C}, t_0 \rangle$
output: Path P

1 $t \leftarrow t_0, \mathbf{C}_t \leftarrow 0, v_t \leftarrow v_s, P \leftarrow [v_s], \mathcal{G}' \leftarrow \mathcal{G}$;
2 **while** $v_t \neq v_d$ **do**
3 $P^*, v^*, \mathcal{G}', \mathcal{G}'' \leftarrow$ DecomposeG$(\mathcal{G}', v_t, v_d, \mathbf{C} - \mathbf{C}_t, t)$;
4 $P^\dagger \leftarrow$ CSP_DQN$(\mathcal{G}'', v_t, v^*, \mathbf{C}_{v^*}, t)$;
5 **if** the path score of $P^\dagger$ is higher than P^* **then**
6 append $P^\dagger$ to P
7 **else** append P^* to P ;
8 update time step t and constraint $\mathbf{C}_t$, $v_t \leftarrow v^*$;
9 **end**
10 **return** P

2). Next, we exploit a pre-trained CSP_DQN model on the decomposed sub-graphs to learn sub-paths for an input source-destination pair. After that, we merge all learned sub-paths to find a complete path as the final solution to the original CSP instance.

Algorithm 1 outlines the steps of our solution CSP_GS. This algorithm takes an input CSP instance $\langle \mathcal{G}, v_s, v_d, \mathbf{C} \rangle$ and returns a path P. In line 1 of this algorithm, the three variables t, $\mathbf{C}_t$ and v_t, indicate the current time step, constraint consumption and currently visited node by the CSP route, respectively.

Before the CSP route arrives at the destination v_d, within the **while** loop (lines 2-9), we first decompose the current input graph $\mathcal{G}'$ into a small subgraph $\mathcal{G}''$ and the remaining one $\mathcal{G}'$ (line 3). Here, the graph decomposition module DecomposeG will be given by Algorithm 2 in Section V. The variable v^*, returned by DecomposeG, indicates a crucial node having the high possibility to be within the CSP path. Given the v^*, we take the source-destination pair $\langle v_t, v^* \rangle$ and associated constraint consumption $\mathbf{C}_{v^*}$, together with the decomposed subgraph $\mathcal{G}''$, as the small CSP sub-instance (see line 4). In this way, the pre-trained CSP_DQN model can find a sub-path $P^\dagger$ from v_t to v^* on the sub-graph $\mathcal{G}''$ with the constraint $\mathbf{C}_{v^*}$ The CSP_DQN model will be given by Algorithm 4 in Section VI.

Now, in lines 5-8, given the found sub-path $P^\dagger$ and another sub-path P^* sampled by DecomposeG, we choose the one with a higher path score (the path score will given in Section V). After that, we append the chosen sub-path to P and update the time step t, constraint consumption $\mathbf{C}_t$ and current node v_t. When the route finally arrives at the destination v_d, line 10 returns P as the CSP path.

TABLE I
MAIN SYMBOLS AND ASSOCIATED DESCRIPTION

Component	Sym.	Description
Node estimator	P_i	The i-th sampled path
	P^*	The optimal path in m sampled paths
	v^*	The crucial node of an CSP instance
	ps_i	Score of p_i via path weight
	s_i^c	Score of p_i via constraint consumption
	ns_v	The score of v via graph sampling
Graph Embedding	v_c	Current node
	x_v	The input feature of node v
	μ_v	The node embedding of node v
	$\mathcal{N}(v)$	The set of neighbors of node v
	$\widetilde{A}$	The normalized weighted adjacent matrix
	K	The # of GCN layers
	q_l	Output dim. of the l-th layer in GCN
	$\Theta^{(l)}$	The parameters of the l-th layer in GCN
	$H^{(l)}$	The embedding of the l-th layer in GCN
DQN	Θ	Parameters of the Q-network
	Θ^-	Parameters of the target Q-network
	T	The max. # of iterations
	M	The # of steps to update target networks
	L	The length of the action sequence

Time Complexity: Recall that in Section III-B, the baseline solution repeats a CSP solver by $(|P| - 1)$ times. Unlike that, CSP_GS performs the CSP_DQN model on average by $\frac{|P|}{|P'|}$ times, where $|P'|$ is the average number of nodes within the found sub-paths. It is not hard to find that the running time of CSP_GS strongly depends upon $|P'|$ and we will carefully tune $|P'| > 1$ for faster running time of Algorithm 1.

Table I summarizes the mainly used symbols and associated description.

V. Node Score-Based Graph Decomposition

Given a CSP instance $\langle \mathcal{G}, v_s, v_d, \mathbf{C} \rangle$, the decomposition in Algorithm 2 involves the following steps. It first samples m paths from v_s to v_d in the graph $\mathcal{G}$, and next estimates a *path score* for every sampled path and a *node score* for every node appearing within the sampled paths. Here, the path score (resp. node score) indicates the possibility of such a path (resp. node) to become (resp. appear within) the shortest feasible path for the CSP instance. With the estimated path scores (resp. node scores), we choose the crucial path p^* (resp. node v^*) with the highest score. Finally, with help of the chosen path and node, we decompose the input CSP graph into a small sub-graphs $\mathcal{G}''$ and remaining one $\mathcal{G}'$. In the rest of the section, we first

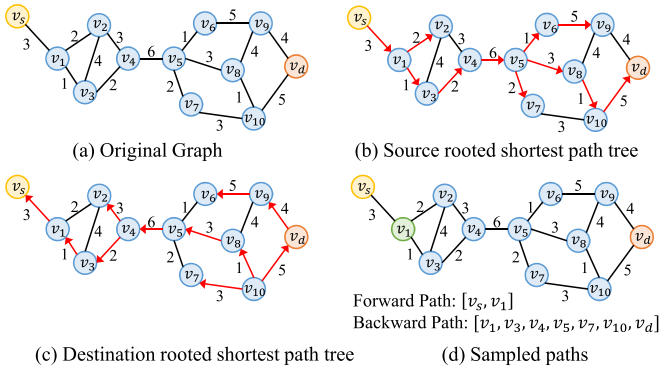


Fig. 2. An example for m-path sampling.

give the techniques of m-Path sampling and the node/path score estimation, and next present the detail of graph decomposition.

A. m-Path Sampling Technique

In this section, as shown in Fig. 2, we give the steps to efficiently sample m paths on the graph $\mathcal{G}$ from the source v_s to the destination v_d. First, we perform the single-source shortest path algorithm (i.e., one-to-all Dijkstra algorithm) twice in the graph $\mathcal{G}$, such that we can build two single-source shortest path trees rooted at v_s and v_d, respectively. The shortest-path tree rooted at the source v_s, denoted by $\mathcal{T}_{v_s}$, records the predecessors on the path with shortest length from v_s to each graph nodes. The shortest-path tree rooted at v_d, denoted by $\mathcal{T}_{v_d}$, records the successors on the shortest length path to v_d. By locating the descendant nodes of v_s within $\mathcal{T}_{v_s}$, we can comfortably combine the forward and backward sub-paths to obtain a complete path from source v_s to destination v_d via the located descendant nodes.

Fig. 2 illustrates an example to sample m paths from v_s to v_d. After building the two single source shortest path trees rooted at v_s and v_d, we first locate the descendant node (e.g., v_1) of the source v_s. By combining the forward path to v_1 (i.e., $v_s \rightarrow v_1$) in $\mathcal{T}_{v_s}$ and the reversed backward path in $\mathcal{T}_{v_d}$ (i.e., $v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_8 \rightarrow v_{10} \rightarrow v_d$), we obtain a complete path ($v_s \rightarrow v_1 \rightarrow v_3 \rightarrow \ldots \rightarrow v_d$). By locating other descendant nodes of v_s, we finally sample all m paths from v_s to v_d.

Note that the shortest path computation during the m-path sampling step is based on static edge length. Yet, we on time evaluate the path weight on the snapshot graph $\mathcal{G}_t$, where t is the current time step. In this way, the path weight is adaptive to graph changes. In the case that some edges are broken at time step t, we purposely set very large weights for broken edges, such that those path containing broken edges have little chance to become CSP paths.

Time Complexity: Given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, the time complexity to calculate the shortest path trees is $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}|log|\mathcal{V}|)$. The cost of combining the complete paths heavily depends upon the traversal of $\mathcal{T}_{v_d}$ to locate the descendant nodes of v_s within $\mathcal{T}_{v_d}$. Since the traversal of $\mathcal{T}_{v_d}$ requires the time cost $\mathcal{O}(|\mathcal{V}|)$, we finally have the overall cost $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}|(m + log|\mathcal{V}|))$ to sample m paths.

B. Node Score Estimation

Recall that a desirable CSP route is the path with the least path weight and the constraint consumption smaller than the given threshold $\mathbf{C}$. Thus, a certain path with lower path weight and constraint consumption is more likely to be the optimal one. To this end, we define a *path score* as the probability of a given path to be the CSP route. With help of the path score, we next define a *node score* as the probability of a given node to appear within the CSP route.

In Algorithm 2, the `Estimate` function first sample m paths from v_s to v_d on the graph snapshot $\mathcal{G}_t$ and chooses a path with the greatest score as the *crucial path* P^* (lines 1-2). Here, in (3), we need to compute the path scores for such m sampled paths by taking into account both edge weights and constraint consumption.

$$ps_i = \beta \cdot e^{-\frac{(pw_i - pw_{min})^2}{{\sigma_{pw}}^2}} + (1-\beta) \cdot e^{-\frac{max\{0, c_i - \mathbf{C}\}}{{\sigma_c}^2}}. \quad (3)$$

In the equation above, the first item indicates the score of edge weights, where pw_i denotes the sum of edge weights within a path p_i, pw_{min} is the minimum path weight among the m sampled paths and ${\sigma_{pw}}^2$ is the variance of such m path weights. The second item $e^{-\frac{max\{0, c_i - \mathbf{C}\}}{{\sigma_c}^2}}$ indicates the score of the constraint consumption c_i on path p_i. If the consumed consumption c_i is smaller than the constraint condition $\mathbf{C}$, we have a feasible path, indicating the second item having a greatest score 1.0. Otherwise, we have an infeasible path having a score smaller than 1.0. By the parameter $0.0 \leq \beta \leq 1.0$, (3) balances the two items and the final score is between 0.0 and 1.0.

After that, in lines 3-4, we compute a node score ns_v for every node $v \in \mathcal{V}$, and next choose those nodes $v \in \mathcal{V}' \subseteq \mathcal{V}$ with $ns_v > \delta$ where δ is a predefined threshold.

$$ns_v = \sum_{i=1}^{m} \widetilde{ps}_i,$$
$$\text{where } \widetilde{ps}_i = \begin{cases} ps_i, & \text{if } v \in p_i \\ 0, & \text{otherwise} \end{cases}. \quad (4)$$

As shown in (4), if a node v appears within more sampled paths, the associated node score ns_v is greater. It is particularly true that v appears in those paths with the greater path scores ps_i. In this way, if those poor nodes (with scores smaller than δ) are pruned, we can determine that the poor nodes have little chance to be within the shortest path.

After choosing those nodes $\mathcal{V}'$, in line 5, we extract a subgraph $\mathcal{G}' \subseteq \mathcal{G}$, where the vertex set of $\mathcal{G}'$ is just equal to $\mathcal{V}'$, and the edges $\mathcal{E}' \subseteq \mathcal{E}$ are those with all endpoints belonging to $\mathcal{V}'$. In this way, the remaining graph $\mathcal{G}'$ contains a much smaller number of nodes than $\mathcal{G}$, and the proposed solution CSP_GS can efficiently solve a large CSP instance.

C. Graph Decomposition

Given two techniques above, we now give the details of graph decomposition. Since graph changes with time, we perform the crucial node selection and graph decomposition in *chronological order*. That is, we sequentially select a crucial node every l

Algorithm 2: `Estimate`.

Input: Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, node pair $\langle v_s, v_d \rangle$, constraint $\mathbf{C}$, time step t
Output: Path P^*, remaining graph $\mathcal{G}'$

1: Sample m paths from v_s to v_d on the graph snapshot $\mathcal{G}_t$;
2: $P^* \leftarrow$ the path with the greatest score ps_i;
3: Compute ns_v for each node $v \in \mathcal{V}$ by (4);
4: $\mathcal{V}'$ is a set of nodes $v \in \mathcal{V}' \subseteq \mathcal{V}$ with $ns_v > \delta$;
5: $\mathcal{G}' \leftarrow (\mathcal{V}', \mathcal{E}')$: $\forall e_{v_i \to v_j} \in \mathcal{E}'$, $e_{v_i \to v_j} \in \mathcal{E}$ also holds;
6: **return** $P^*, \mathcal{G}'$

Algorithm 3: `DecomposeG`.

input : Graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, node pair $\langle v_s, v_d \rangle$, constraint $\mathbf{C}$, time step t
output: Path p^*, crucial node v^*, sub-graphs $\mathcal{G}'$ and $\mathcal{G}''$

1 $P^*, \mathcal{G}' \leftarrow$ `Estimate`$(\mathcal{G}, v_s, v_d, \mathbf{C}, t)$;
2 **if** $|P^*| > l$ **then**
3 $\quad v^* \leftarrow$ the l-th node of P^*;
4 $\quad p^*, \mathcal{G}'' \leftarrow$ `Estimate`$(\mathcal{G}', v_s, v^*, \mathbf{C}_{v^*}, t)$
5 **else** $v^* \leftarrow v_d$, $p^* \leftarrow P^*$, $\mathcal{G}'' \leftarrow \mathcal{G}'$;
6 **return** $p^*, v^*, \mathcal{G}', \mathcal{G}''$

nodes along the CSP path P from source to destination, where l is a predefined parameter and can be intuitively treated as the size of decomposed sub-graphs. Thus, we perform graph decomposition by $\left\lceil \frac{|P|}{l} \right\rceil$ iterations.

Following the idea above, Algorithm 3 first exploits `Estimate`$(\cdot)$ to find the crucial path P^* from v_s to v_d and the remaining sub-graph $\mathcal{G}'$ (line 1). In lines 2-4, when the path size $|P^*|$ is greater than the aforementioned parameter l, within the crucial path P^* starting from v_s to v_d, we select the lth node as the crucial node v^*. By again applying the `Estimate` function with the new pair $\langle v_s, v^* \rangle$ on the sub-graph $\mathcal{G}'$, we then find a new crucial path p^* from v_s to v^* and a sub-graph $\mathcal{G}''$. Otherwise if $|P^*| \leq l$, the decomposition is unnecessary and line 5 directly returns the found path P^* and sub-graph $\mathcal{G}'$. Finally, we have a CSP sub-instance $\langle \mathcal{G}'', v_s, v^*, \mathbf{C}_{v^*} \rangle$, and will employ the `CSP_DQN` model to effectively solve the sub-instance.

VI. CSP_DQN Model

Given a small CSP instance, our 2nd component `CSP_DQN` learns a shortest feasible path. As shown in Fig. 3, `CSP_DQN` involves the multi-iterative selection of graph nodes starting from a source node. In each iteration, `CSP_DQN` employs GCN to learn graph embedding vectors, and next exploits a n-step Q-learning network [40] on the embedding vectors to choose a *k-hop node* by an evaluation function $\widehat{Q}$. Typically we set $k = 2$ to balance the trade-off between high action space and small number of iterations to perform the DRL model (and thus for higher efficiency).

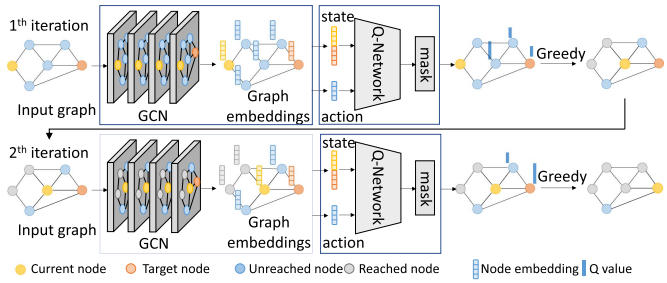


Fig. 3. Architecture of the CSP-DQN model.

A. Graph Embedding

GCN is powerful to learn graph node representation. Specifically, given an input graph $\mathcal{G}$, we design the following features x_v for each graph node $v \in \mathcal{V}$. First for the edges connected by v, we find the statistics of the edge features (e.g., the maximal, minimal and average of the edge weights and constraint consumption). Next, the two other features include the GPS location of v and a binary value to indicate whether or not the node v is the destination node v_d. Now given the aforementioned features x_v, we employ GCN [21] to compute a q_K-dimensional feature embedding μ_v. The GCN architecture consists of K layers. At each layer l, the node embeddings are updated as follows:

$$H^{(l+1)} = \sigma(\widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}H^{(l)}\Theta^{(l)}), \tag{5}$$

where $\widetilde{D}_{ii} = \sum_j \widetilde{A}_{ij}$, $\widetilde{A}$ is the normalized weighted adjacent matrix, $\Theta^{(l)}$ is the trainable weight matrix of the lth layer, and $H^{(l)} \in \mathbb{R}^{|\mathcal{V}| \times q_l}$ is the embedding vector of the lth layer. Here, $H^{(l=1)}$ in the first layer is initialized by the input features x_v. The final layer outputs the embedding vector $\mu_v \in \mathbb{R}^{q_K}$ for each node v in the graph where q_K is the size of the embedding vector.

By (5), with the help of the matrix $\widetilde{A}$, the node embedding vector takes as input the node features from direct neighbors, and is next propagated to more distant neighbor nodes when more update process is performed. Finally, each node embedding will contain the K-hop neighborhood feature vectors. In this way, the embedding vector is propagated according to graph topology. Thanks for the powerful node representation ability offered by GCN, after the parameter $\Theta^{(l)}$ is learned, we can compute node embeddings even for unseen dynamic graphs.

B. Learning Q-Function

Reinforcement Learning Formulation. To learn the shortest feasible path, we exploit the Q-learning [8] and define the following reinforcement learning model.

- *State*: We define the state $S_t = (\mu_t, \mu_d)$, where μ_t and μ_d are the node embedding vectors of the current node v_t and destination node v_d, respectively. The state S_t concatenates the two vectors and offers the power to generalize to various CSP instances such as unseen graphs and source-destination pairs.
- *Action*: An action corresponds to a k-hop neighbour node $v_k \in \mathcal{N}^{(k)}(v_t)$ to the current node v_t, where $\mathcal{N}^{(k)}(v_t)$ is the set of k-hop neighbours of v_t in $\mathcal{G}$ and $k = 2$ by default.

Algorithm 4: Finding SP by Double DQN (CSP_DQN).

input : A CSP instances $\langle \mathcal{G}, v_s, v_d, \mathbf{C} \rangle$
output: Path P

1 Initialize the current node v_t by source v_s;
2 Initialize the path $P = [v_s]$, time step $t = 0$;
3 **while** $v_t \neq v_d$ **do**
4 Embed the state $\mathrm{S_t} = (\mu_c, \mu_d)$;
5 $v_2 \leftarrow \arg\max_{v \in \mathcal{N}^{(2)}(v_t) \cap v \notin P} \widehat{Q}(S_t, v; \Theta)$;
6 $v_1 \leftarrow \arg\min_{v \in \mathcal{N}^{(1)}(v_t)} W_t(v_t \rightarrow v) + W_t(v \rightarrow v_2)$;
7 add v_1, v_2 to P;
8 $v_t \leftarrow v_2$, update time step t ;
9 **end**
10 **return** P

When one 2-hop node, denoted by v_2, is chosen, we can then determine a certain 1-hop node v_1 among all 1-hop neighbours, such that the sub-path $v_t \rightarrow v_1 \rightarrow v_2$ is with the least path weight. Compared to a 1-hop neighbour as an action, a 2-hop node offers the benefit of using a half number of iterations to perform the DQN model. However, using a k-hop node suffers from higher action space. We will empirically study the effect of k.

- *Transition*: When a node v is selected as an action, the state is transited to $S_{t+1} = (\mu_v, \mu_d)$ at time $(t+1)$.
- *Reward*: We define an immediate reward of selecting a k-hop node (as the action a_t) by $r_t = -w_{a_t} - \lambda J_C(c_{a_t})$, where w_{a_t} and c_{a_t} are the sum weight and resource consumption of adding new edges (e.g., the aforementioned $v_t \rightarrow v_1 \rightarrow v_2$) to the solution path by the action a_t and $J_C(a_t)$ is the penalty to the constraint which will be given very soon.
- *Policy*: By learning the parameter set $\Theta_{\widehat{Q}}$, we want to find a deterministic greedy policy $\pi(v|S)$ satisfying $\pi(v|S) := \mathrm{argmax}_{v' \in \mathcal{N}(k)(v_c) \cap v' \notin P} \widehat{Q}(S,\ v')$, where P is the set of already selected nodes.

Following the main idea above, Algorithm 4 learns a shortest path by the DQN model. Here, we assume that the Q function has been well trained. After the initialization (lines 1-2), within the **while** loop (lines 3-9), we first embed the DQN state with help of the current graph snapshot $\mathcal{G}_t$ and next use the Q network to select a 2-hop neighbour v_2 as the node. Then, we choose a direct neighbour v_1 such that the subpath $v_t \rightarrow v_1 \rightarrow v_2$ is with the least weight. After that, We add the two nodes v_1 and v_2 to P and update the current node and time step. We repeat the loop until the arrival of the destination v_d.

Optimization Objective: We first illustrate the challenge of using a base DQN model [28], [29] to learn the shortest feasible path for a small CSP instance. Given a CSP instance, the DQN model generates an entire sequence of actions $(a_1, \ldots, a_L)$ to learn a feasible path. For the tth action a_t with $1 \leq t \leq L$, the accumulated constraint consumption of the t actions from a_1 to a_t may exceed the constraint threshold $\mathbf{C}$ and yet the one of $a_1, \ldots, a_{t-1}$ may still satisfy the threshold $\mathbf{C}$. It is not hard to find that the action a_t alone cannot lead to the constraint dissatisfaction. Instead, the dissatisfaction might be caused by the false selection of a_t and those prior to a_t. Thus, we cannot tackle this issue by simply masking the action a_t alone. To this end, we develop the flowing improved CSP_DQN model.

The CSP_DQN model defines a new optimization objective. Given the action sequence $(a_1, \ldots, a_L)$, we have to take into account both the weight sum and feasibility of the sequence to solve CSP. Thus, following the work [39], we exploit the Lagrange technique to tackle CSP. That is, we reformulate CSP as an unconstrained version by relaxing the constraints as a penalty term into the objective function.

$$\min_{a_t} \mathbb{E}\left[\sum_{t=1}^{L} w_{a_t} + \lambda \left(\sum_{t=1}^{L} c_{a_t} - \mathbf{C}\right)^{+}\right], \tag{6}$$

where λ is for the penalty coefficient, and $(x)^+$ is an operation to compare 0 and x and outputs the larger one. The second term in (6) stands for the penalization for the constraint dissatisfaction. If the solution is feasible, the penalty is 0. The value of the penalty is related to how much the constraints are dissatisfied.

Consider that the contribution of each action to the final reward is based on the associated weight and constraint consumption. We thus define the following reward function.

$$r_t = -w_{a_t} - \lambda \cdot \frac{c_{a_t}}{\sum_{i=1}^{L} c_{a_i}} \cdot \left(\sum_{i=1}^{L} c_{a_i} - \mathbf{C}\right)^{+}. \tag{7}$$

By (7), to train the DQN model, we collect multiple training samples when an episode ends.

Learning the Parameter Set $\Theta_{\widehat{Q}}$. After the node embedding vectors are computed by GCN, we can define the parameter set $\widehat{Q}(S_t, v; \Theta)$ by the following state and action formulation:

$$\widehat{Q}(S_t, v; \Theta) = \theta_1^{\top} relu\left(\left[\theta_2 \left[\mu_c, \mu_d\right]^{(\top)}, \theta_3 \mu_v^{(\top)}\right]\right), \tag{8}$$

where $\theta_1 \in \mathbb{R}^{2q_K}$, $\theta_2 \in \mathbb{R}^{q_K \times 2q_K}$ and $\theta_3 \in \mathbb{R}^{q_K \times q_K}$ are the model parameters, $[\cdot, \cdot]$ is the concatenation operator, $[\mu_c, \mu_d]$ represents the state, and μ_v is the action vector.

Algorithm 5 learns the parameter set $\Theta = \{\theta_1, \theta_2, \theta_3\}$ by an end-to-end fashion via the double Q-learning [43] and n-step Q-learning [40]. Among the initialization stage (lines 1-3), Θ and Θ^- denote the parameters of the Q-network and target Q-network, respectively. Within the 1st **for** loop in lines 4-16, in one *episode*, we construct a complete path from the source v_s to the destination v_d. Within the 2nd **for** loop in lines 6-12, each *step* corresponds to an action (i.e., selecting a 2-hop neighbour). By the Q-network, in line 8, we select a 2-hop neighbour v_t which minimizes the state-action function $\widehat{Q}(S_t, v; \Theta)$. In line 14, the n-step Q-learning updates the parameters Θ via the Adam optimizer [20] to minimize the squared loss

$$J(\Theta_{\widehat{Q}}) = \left(y - \widehat{Q}\left(S_t, v_t; \Theta\right)\right)^2, \tag{9}$$

In the equation above, the target Q-network computes $y = \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i}) + \gamma \widehat{Q}(S_{t+n}, v'; \Theta^-)$, where the former item indicates the sum of n-step immediate rewards and γ is the discount factor to balance the importance of the immediate reward with the predicted future reward. The second item estimates the future reward with the target network, where

Algorithm 5: Learning the Parameter Set $\Theta_{\widehat{Q}}$.

input : A set of CSP instances $\{\langle \mathcal{G}, v_s^{(i)}, v_d^{(i)}, \mathbf{C}^{(i)} \rangle\}_{i=1}^{E}$
output: The parameterized Q-network

1 Initialize experience replay memory $\mathcal{M}$;
2 Initialize a Q-network with random weights Θ;
3 Initialize a target Q-network with $\Theta^- = \Theta$;
4 **for** *episode=1 to E* **do**
5 Initialize the current node v_c by source $v_s^{(i)}$;
6 **for** *step t=1 to T* **do**
7 Embed the state $\mathrm{S_t} = (\mu_c, \mu_d)$.;
8 $v_t \leftarrow \begin{cases} \text{rand. node } v \in \mathcal{N}^{(2)}(v_c), & \text{with prob. } \varepsilon \\ \text{argmax}_{\mathrm{v} \in \mathcal{N}^{(2)}(\mathrm{v_c})} \widehat{\mathrm{Q}}\,(\mathrm{S_t}, \mathrm{v}; \Theta)\,, & \text{otherwise} \end{cases}$;
9 $v_c \leftarrow v_t$;
10 **if** $t \geq n$ **then** add $(S_{(t-n)}, v_{(t-n)}, R_{(t-n),t}, S_t)$ to $\mathcal{M}$;
11 **if** $v_c = v_d$ **then** break;
12 **end**
13 Sample a batch of experiences;
14 Update Θ by ADAM according to Equation 9;
15 Update target Q-network every M steps: $\Theta^- = \Theta$;
16 **end**
17 **return** The parameterized Q-network

$v' = \arg\max_v \widehat{Q}(S_{t+n}, v; \Theta)$ is the action selected by the Q network.

In the algorithm above, we can alternatively exploit the fitted Q-iteration [33] for better efficiency. The fitted Q-iteration leverages experience replay $\mathcal{M}$ to update $\Theta_{\widehat{Q}}$ with a batch of samples, rather than a single sample being currently experienced. By an experience replay memory $\mathcal{M}$ and at step $t+n$, in line 10, we add a tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to $\mathcal{M}$ to the memory, where $R_{t,t+n} = \sum_{i=0}^{n-1} r(S_{t+i}, a_{t+i})$.

To accelerate the learning process, we pre-train the model with the DQfD [15] before the normal RL training. Moreover, with the help of those paths on small static graphs e.g., found by a labeling algorithm, we can achieve a training data set for the CSP instances, and generate the demonstrations by such found paths for better result.

VII. Experiments

A. Experimental Setting

#1: Data Sets: As a fundamental problem, CSP has been applied in many applications. We consider the daily scenario of vehicle routing in urban road networks to evaluate the effectiveness and efficiency of CSP_GS. Specifically, on a road network, we define the CSP problem with the *optimization objective* to minimize the path travel time from a source to a destination and the *constraint condition* that the travel path length should be smaller than a given threshold. For each road segment (edge) in a road network, the edge length is static and yet the travel time on road segments varies dynamically. We give the detail of two following datasets in terms of graph data (i.e., road networks) and travel time on road segments (i.e., dynamic edge weights).

Shanghai Dataset: To generate large dynamic graphs, we use real road network graphs of shanghai, China extracted from OpenStreetMap.[2] Table II gives the statistics of the six

[2]www.openstreetmap.org

TABLE II
Statistics of Shanghai and Koln Datasets

Data set	Graph	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\#\langle v_s, v_d\rangle$
Shanghai	graph-140	140	348	100
	graph-308	308	926	100
	graph-603	603	1484	100
	graph-1108	1108	2844	100
	graph-2030	2030	5320	100
	graph-10889	10889	28290	100
Koln	graph-1889	1889	3942	100

TABLE III
Speed Range of Three Road Types and Speed Levels (km/h)

Speed Level	Primary Road	Secondary Road	Other Road
high	80 ± 10	65 ± 7.5	40 ± 5
medium	60 ± 10	50 ± 7.5	30 ± 5
low	40 ± 10	35 ± 7.5	20 ± 5

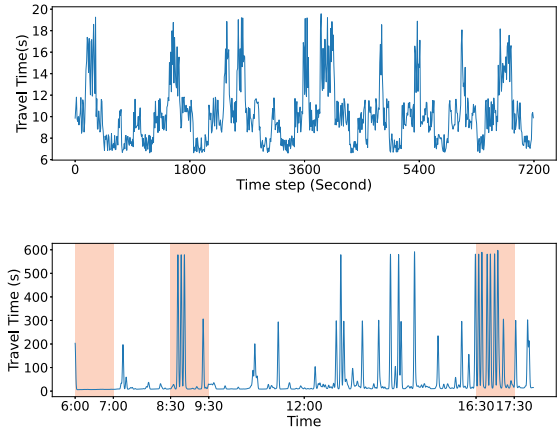


Fig. 4. Example data of Shanghai (top) and Koln (bottom) data sets.

randomly chosen road network graphs in Shanghai. To simulate CSP routes, we evenly split graph nodes in every road network into 9 regions according to their GPS coordinates, and select source-destination pairs $\langle v_s, v_d\rangle$ randomly chosen from two non-adjacent regions. The $\#\langle v_s, v_d\rangle$ in Table II indicates the number of source-destination pairs and thus the number of CSP instances in this graph.

Since OpenStreetMap offers actual length of road segments, we can comfortably use it to setup constraint conditions. Instead, to the best of our knowledge, OpenStreetMap does not offer fine-grained travel time on each road segment at various time periods. We thus simulate dynamic travel time on road segments as edge weights. In our setting, the travel time of a road segment depends upon three factors: road length, speed limit and road traffic dynamics. In terms of speed limit, we acquire three types of primary, secondary and other roads from OpenStreetMap. Table III gives three speed levels and associated speed ranges for each road type.

Now, we simulate the averaged travel time (i.e., dynamic weight) as follows. For an example primary road segment, Fig. 4 (top) illustrates dynamic travel time in y-axis for every time step t in x-axis. To simulate such dynamic travel time, starting from $t = 0$, we randomly choose a speed level (say

"high") for this road and next have the associated speed range (i.e., 70 – 90 km/h) according to Table III. Next, for a predefined interval $[30, 240]$, we generate a random time period (say 170 steps) evenly between the interval, indicating that the travel speed range 70 – 90 km/h remains by 170 time steps. For each time step t from 0 to 170, we generate a specific speed between the range 70 – 90 km/h, and compute the average travel time (i.e., the y-axis of Fig. 4) by the ratio between the actual road length and the generated speed. Such an average makes sense because Section III-A assumes that the travel speed remains unchanged during the route within the road segment. When this time period of 170 steps ends, we transit to a next randomly chosen speed level, and repeat the simulation until the CSP route arrives at its destination. In this way, we can comfortably simulate the average travel time on each road segment on every time step t, i.e., a time series of average travel time on each road, i.e., a time series of average travel time on each road as shown in Fig. 4 (top).

Besides, to simulate graph dynamics, we by default choose 1% broken road segments that purposely appear within CSP solution paths. After that, we make such randomly chosen roads broken for a time period, which is randomly generated between 30 and 240 time steps. During the occurrence of such accidents, in case that a CSP solver falsely selects one broken edge to the solution path, we give a travel time penalty of 500 seconds.

Koln Dataset: We use a traffic simulation tool offered by the previous work [42] to generate a traffic data set simulated in Koln, Germany, for 24 hours. In the data set, we select a downtown area of 25 km^2 and extract a road network graph with 3942 edges and 1889 nodes. To study our work on various traffic conditions, in Fig. 4 bottom, we highlight three time slots, early morning (6:00-7:00), morning peak (8:30-9:30) and evening peak (17:00-18:00), and will evaluate our approach on such three time slots.

As shown in Fig. 4, for the Shanghai dataset, the edge weight (travel time) on an example road varies by following three speed levels. Yet for the Koln dataset, most travel time changes between 5 to 50 seconds and several outliers are significantly greater than 200 seconds during congestion peak periods (e.g., morning peak and evening peak). In this way, we mainly use the two datasets to study the tradeoff between the efficiency and effectiveness of CSP solvers.

Until now, for a generated dynamic graph with T time steps, we then have a graph time series $\mathcal{G}_t$ from $t = 1$ to $t = T$. For every vertex $v \in \mathcal{G}_t$, we define 9 features $x_v = [lon_v, lat_v, w_v^{max}, w_v^{min}, w_v^{avg}, c_v^{max}, c_v^{min}, c_v^{avg}, d_v]$. Here, lon_v and lat_v are the longitude and latitude, $(w_v^{max}, w_v^{min}, w_v^{avg})$ and $(c_v^{max}, c_v^{min}, c_v^{avg})$ are the maximum, minimum and average edge weights and constraint consumption of those neighbour road edges connected to v, and d_v is a binary value indicating whether or not v is the destination. In a CSP problem, the optimization objective is to minimize the total travel time of a path from source v_s to destination v_d and meanwhile satisfy the resource constraint. Next, we set the CSP constraint threshold $\mathbf{C} = c \cdot len(SP)$, where $c > 1.0$ and $len(SP)$ denotes the path length of the shortest path SP from v_s to v_d (solved by Dijkstra algorithm). A larger c means a looser constraint condition and we set $c = 1.5$ by default in our experiments.

TABLE IV
PARAMETER SETTING

Parameter	Value
q_K: Node embedding size in §6.1	64
K: The number of GCN layers in §6.1	3
n-step Q learning in §6.2	3
λ: Penalty coefficient of disobeyed constraint in §6.2	1.0
l: Average step number of the sampled paths	5
δ: Node score threshold to prune poor nodes in §5	0.05
m: The number of sampled paths in §5	150

#2: Counterparts: To solve the CSP problem, we compare our CSP_GS against four competitors. We use the classic labeling algorithms and its variants (the implementation refers to the Python source code in [36]) for comparison.

- *LA-sg:* the LA-sg approach directly performs the labeling algorithm [41] to find a feasible path on the static graph $\mathcal{G}_0$.
- *LAx-dg:* we repeatedly exploit a labeling algorithm to select a next-hop node on a snapshot graph $\mathcal{G}_t$ at each time step, and implement three variants of the mono-directional forward labeling algorithm (*LA1*), the bidirectional labeling algorithm with static halfway point (*LA2*) and the bidirectional labeling algorithm with dynamic halfway point (*LA3*). For a certain CSP instance, the three variants find the same paths but with very different running time.

#3: Metrics: We evaluate a CSP solver mainly in terms of effectiveness (measured by the *approximation ratio* against the best approach and the *constraint consumption*) and efficiency (measured by *running time*).

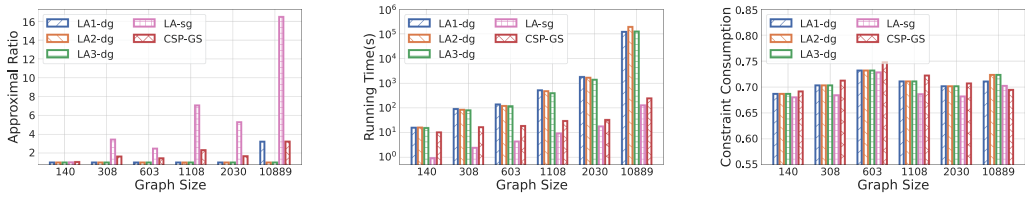
We take the path P found by LA_3-dg as the best-known one (see Section III-B), and compute the *approximation ratio* and *constraint consumption*. The previous work [17] has demonstrated that the labeling algorithm can obtain Pareto-optimal solutions to the CSP problem on static graphs and among the three variants of labeling algorithms, LA3 [41] is the most efficient. That is, for a path P' found by a certain CSP solver, we define an *approximation ratio* to be the ratio between the travel time of path P' and the one obtained by LA_3-dg. An approximation ratio, greater than 1.0, indicates that the solver to find the path P' leads to higher travel time and low effectiveness. Note that the three variants of LAx-dg algorithms find the exactly same CSP paths, and thus lead to the approximation ratio 1.0.

Besides the approximation ratio, we meanwhile measure the *constraint consumption* of a CSP solver by the ratio against the given constraint bound $\mathbf{C}$.

#4: Parameters: Table IV lists the total 7 hyper-parameters in our model. The first four parameters (q_K, K, n, λ) are for the DQN model and the last three parameters (l, δ, m) are about the graph sampling procedure.

B. Baseline Study

Shanghai Dataset: Fig. 5 plots the evaluation result on Shanghai dataset. First, LA_1-dg, LA_2-dg and LA_3-dg find the exactly



(a) Approximation Ratio (b) Running Time (c) Constraint Usage Rate

Fig. 5. Baseline results on Shanghai dataset (a, b, c) and Time consumption of DQN and Estimator (d).

TABLE V
BASELINE RESULTS ON KOLN DATA SET (μ AND σ REPRESENT MEAN VALUE AND STANDARD DEVIATION OF TRAVEL TIME IN THREE TIME SLOTS)

Time slot	Early Morn. 6:00-7:00 ($\mu = 10.5, \sigma = 13.4$)			Morn. Peak 8:30-9:30 ($\mu = 34.5, \sigma = 165.9$)			Even. Peak 16:30-17:30 ($\mu = 79.9, \sigma = 443.4$)		
Metric	Approx. Ratio	Constraint Usage Rate	Running Time (s)	Approx. Ratio	Constraint Usage Rate	Running Time (s)	Approx. Ratio	Constraint Usage Rate	Running Time (s)
LA1-dg	1.000	0.671	619.3	1.000	0.803	1023.2	1.000	0.859	1110.7
LA2-dg	1.000	0.671	593.4	1.000	0.803	801.3	1.000	0.859	837.4
LA3-dg	1.000	0.671	580.4	1.000	0.803	763.5	1.000	0.859	813.9
LA-sg	1.002	0.669	14.5	1.715	0.723	16.6	4.863	0.734	15.9
CSP-GS	1.002	0.670	28.5	1.538	0.723	30.7	2.419	0.754	33.3

same CSP paths and thus have the approximation ratios 1.0, but at cost of much higher running time than our work CSP_GS. Such high running time is due to the fact that these labeling approaches repeatedly perform the next-hop node selection on every intermediately met node. Since the original LA_1-dg suffers from the highest running time, significantly greater than 1,000 seconds, we slightly improve the LA_1-dg: we limit the running time of each iteration of LA_1-dg by 1,000 seconds. As a result, the improved LA_1-dg version leads to faster running time than $LA_{2,3}$-dg, but with higher approximation ratios. Differing from the three dynamic algorithms, the static LA-sg approach performs the labeling algorithm only once on the initial graph snapshot and leads to much faster running time, with the highest approximation ratio among all CSP solvers.

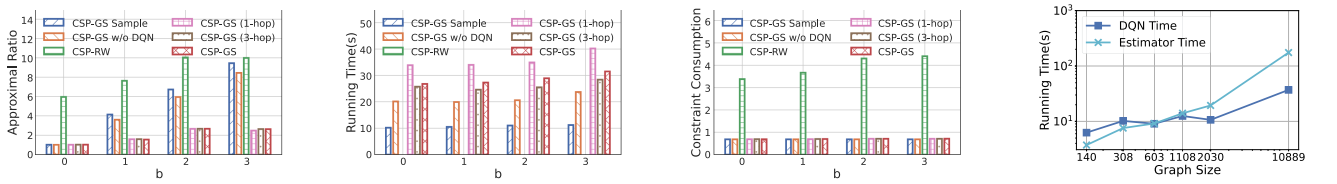
Instead, our work CSP_GS achieves reasonably low approximation ratio (comparable to the improved version of LA_1-dg) and rather fast running time. For example, on graph-10889, CSP_GS is about 587 $\times$ faster than LA_3-dg. Meanwhile the approximation ratio of CSP_GS is only 3.214, much better than the ratio 14.355 of LA-sg. In addition, the accuracy gap between LA-sg and CSP_GS grows significantly with graph size. It is mainly because more dynamic and larger graphs lead to more travel time to arrive at the destination. Such results demonstrate that our approach can adapt to such graph dynamics and thus outperform LA-sg on the graphs with high dynamics. Besides, in Fig. 5(c), the four LA approaches suffer from higher constraint consumption than our work CSP_GS, indicating that our work CSP_GS can best balance the running time, approximation ratio and constraint consumption.

Koln Dataset: In Table V, we evaluate the five CSP approaches on three time periods in Koln dataset. The result is roughly consistent with Fig. 5(a)–5(c), and our work CSP_GS can still well adapt to various graph dynamics in all three time periods. Moreover, by comparing the result of the three time periods, we have the following interesting finding. Since Fig. 4 has illustrated such three periods on an example road, the early morning period is with stable travel time and the evening peak period rather dynamic pattern. Given the three periods, our approach CSP_GS consistently achieves the best trade-off among the approximation ratio, running time and constraint consumption. Particularly in the evening peak period, CSP_GS can lead to a reasonably small approximation ratio 2.419, constraint rate 0.754 and rather fast running time 33.3 seconds.

C. Ablation Study

To study the performance of each individual component in our work CSP_GS, we conduct an ablation study on graph-2030. In this experiment, we select the number b of broken edges on each shortest path on a CSP instance. In this way, we can evaluate how CSP_GS performs on such dynamic graphs. (1) To study the benefit of the proposed graph decomposition technique, we simply choose the best one among the m sampled paths as the final solution path. We denote this approach as CSP_Sample. (2) After the removal of the DQN component, on each decomposed CSP sub-instance, we choose a best one among the m-sampled paths, and then connect such sub-paths as the final path. We name this method as CSP_GS w/o DQN. (3) To validate the benefit of the proposed m-path sampling technique, we alternatively adopt a random walk-based sampling technique, i.e., starting from the source node and select a neighbour node randomly until finally arriving at the target node. We denote this method as CSP_RW. (4) To evaluate the DQN policy to choose a 2-hop neighbor as the action, we implement two alternatives by using the 1-hop and 3-hop neighbor as the action, respectively. Here, after the DQN model chooses a 3-hop neighbor v_3 as the action, we then search choose a direct neighbor v_1 and a 2-hop neighbor v_2, such that a sub-path $v_c \rightarrow v_1 \rightarrow v_2 \rightarrow v_3$ is with the least sub-path weight among all candidate sub-paths from the current source v_c to the chosen neighbor v_3.

Fig. 6(a)–6(c) plots the performance of 6 version of CSP_GS with different number of broken edges with the following findings.



(a) Approximation Ratio (b) Running Time (c) Constraint Usage Rate (d) Run. Time of Two Components

Fig. 6. Ablation results on graph-2030 (a, b, c) and Time consumption of DQN and Estimator (d). The running time of CSP_RW in (b) is omitted because its running time is much longer than others (more than 2000 seconds).

- The three variants of CSP_GS with 1-hop, 2-hop and 3-hop neighbors obtain the best approximation ratio and rather low constraint consumption ratio. Due to the adopted m-path sampling technique and the DQN component, they require more running time than CSP_Sample and CSP_GS w/o DQN, but much faster than CSP_GS_RW. Such result demonstrates the effectiveness and efficiency of CSP_GS.
- Though with faster running time, CSP_Sample and CSP_GS w/o DQN suffer from much higher approximation ratios. It is mainly because the m paths sampled on static graphs by the two approaches cannot adapt to graph dynamics with broken edges. As a result, the paths found by the two approaches could falsely contain broken edges, leading to high penalty. Instead, the three variants of CSP_GS can observe the state and choose an action in real time fashion to adapt to graph dynamics. It is particularly true when graph dynamics are more significant with a greater number b. In addition, CSP_RW suffers from the rather high approximation ratio and running time due to the random walk manner.
- Among the three variants, CSP_GS (1-hop) leads to the smallest approximation ratio but suffers from the highest running time. Instead, CSP_GS (3-hop) achieves the fastest time but with the worst approximation ratio and yet our work CSP_GS can balance a smaller number of iterations of performing the DQN model and larger action space for reasonable approximation ratio and fast running time with the 2-hop neighbor as the action.
- We also evaluate LA-sg when the various number of broken edges (we do not plot the result of LA-sg in Fig. 6 for better readability of this figure). When there is no broken edge, the approximation ratio of LA-sg is 1.893. However, when the number of broken edges is three, the approximation ratio of LA-sg is 7.559, much higher than that of CSP_GS. This result illustrates that the approach LA-sg fails to work well on highly dynamic graphs.

In addition, Fig. 7.3(d) plots the running time of two key components (DQN and graph decomposition used in CSP_GS). We can find that the running time of the two components grows with graph size. For a large graph size, e.g., graph-10889, the growth trend of node score estimator is more significant than the one of DQN, mainly because the running time of node score estimator, $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}|(m + log|\mathcal{V}|))$, directly depends upon graph size.

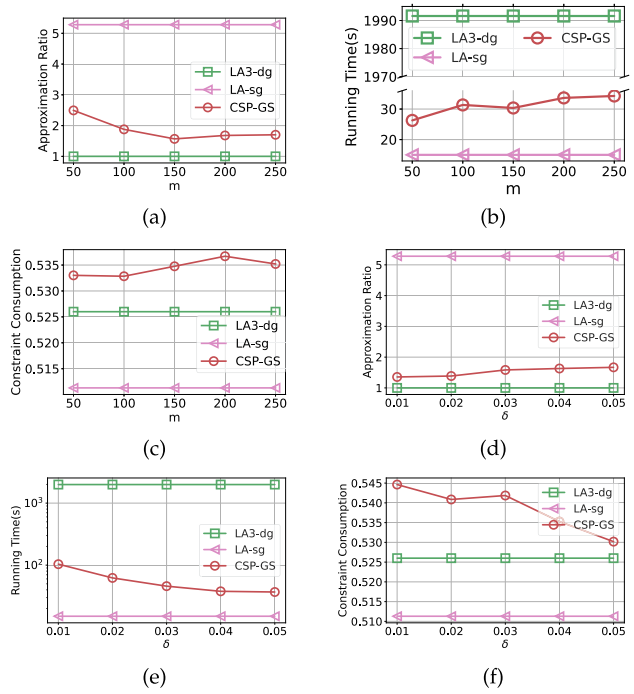


Fig. 7. Effect of sampled paths (a-c); Effect of node score threshold (d-f).

D. Sensitivity Study

To perform the sensitivity study of some key parameters, we conduct experiments on the Shanghai graph-2030 data.

#1. Effect of Sampled Paths: Fig. 7(a)–7(c) studies the effect of the number m of samples paths. When given more sampled paths, the approximation ratios of CSP_GS decrease due to more chance to find better paths. Moreover, when the number m is greater than a certain value, e.g., 150, the approximation ratio of CSP_GS remains relatively stable. Fig. 7(b) shows that more sampled paths lead to higher running time due to more efforts used to sample such paths.

#2. Effect of Node Score Threshold δ: Finally, we study the effect of node score threshold δ (see Algorithm 2 in Section V-B) to select sub-graph nodes. In Fig. 7(e)–7(f), a greater δ leads to a higher approximation ratio because a smaller number of nodes are selected and there is less chance to a potentially better CSP path. Meanwhile, more nodes mean a greater graph size, and lead to longer running time.

#3. Effect of Sub-Graph Size l: Recall that we use the parameter l to limit the size of decomposed subgraphs. In Fig. 8, a greater

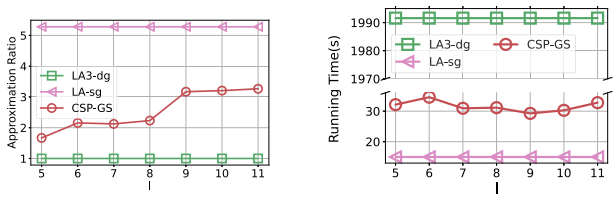


Fig. 8. Effect of sub-graph size: Approximation Ratio and Running Time.

l leads to the growth of the approximation ratio, because the proposed DQN model works better on smaller decomposition sub-graphs. Meanwhile, the running time decreases first then increases. It make sense: since a larger l leads to a smaller number of decomposed sub-graphs and larger size of sub-graphs (which is just the input of CSP-DQN model), CSP_GS requires the trade-off between the decreased time of graph decomposition and the increased time of the DQN solver. As for the constraint consumption, the resource usage ratio of CSP_GS also increases with larger l (which isn't plotted due to space limitation) because less paths with better constraint consumption will be found.

VIII. Conclusion and Future Works

To study the CSP problem on large dynamic graphs, we propose an efficient and effective framework to learn a CSP route path. The framework consists of two key components: 1) a graph decomposition technique to divide the input CSP instance into multiple small sub-instances and 2) a DQN model to learn shortest feasible sub-paths on small sub-instances. Our extensive evaluation on the Shanghai and Koln datasets demonstrates that our work CSP_GS works very well by the trade-off among rather fast running time and reasonable high quality on large dynamic graphs.

As future work, we are interested in two following directions. 1) We continue to study the more general CSP problem with multiple optimization objectives and hard/soft constraint conditions. 2) We are interested in the CSP problem with mobile destination nodes.

References

[1] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, "Solving NP-hard problems on graphs by reinforcement learning without domain knowledge," 2019, *arXiv:1905.11623*.

[2] S. Ahmadi, G. Tack, D. D. Harabor, and P. Kilby, "A fast exact algorithm for the resource constrained shortest path problem," in *Proc. AAAI Conf. Artif. Intell.*, 2021, pp. 12217–12224.

[3] M. Alshammari and A. Rezgui, "A single-source shortest path algorithm for dynamic graphs," *AKCE Int. J. Graphs Combinatorics*, vol. 17, no. 3, pp. 1063–1068, 2020.

[4] E. Beeching, J. Dibangoye, O. Simonin, and C. Wolf, "Learning to plan with uncertain topological maps," in *Proc. Eur. Conf. Comput. Vis.*, Springer, 2020, pp. 473–490.

[5] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," in *Proc. Int. Conf. Learn. Representations*, Toulon, France, 2017, pp. 1–5.

[6] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," in *Proc. Int. Conf. Learn. Representations*, Banff, AB, Canada, 2014, pp. 1–14.

[7] S. Chen, M. Song, and S. Sahni, "Two techniques for fast computation of constrained shortest paths," *IEEE/ACM Trans. Netw.*, vol. 16, no. 1, pp. 105–115, Feb. 2008.

[8] J. H. Connell and S. Mahadevan, "Introduction to robot learning," in *Robot Learning*, Berlin, Germany: Springer, 1993, pp. 1–17.

[9] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *Proc. Adv. Neural Inf. Process. Syst.*, Barcelona, Spain, 2016, pp. 3837–3845.

[10] X. Di, Y. Xiao, C. Zhu, Y. Deng, Q. Zhao, and W. Rao, "Traffic congestion prediction by spatiotemporal propagation patterns," in *Proc. IEEE 20th Int. Conf. Mobile Data Manage.*, Hong Kong, SAR, China, 2019, pp. 298–303.

[11] Z. Fu, K. Qiu, and H. Zha, "Generalize a small pre-trained model to arbitrarily large TSP instances," in *Proc. 35th AAAI Conf. Artif. Intell.*, AAAI Press, 2021, pp. 7474–7482.

[12] S. Gao, H. Zhang, and S. K. Das, "Efficient data collection in wireless sensor networks with path-constrained mobile sinks," *IEEE Trans. Mobile Comput.*, vol. 10, no. 4, pp. 592–608, Apr. 2011.

[13] Y. Geng et al., "Deep reinforcement learning based dynamic route planning for minimizing travel time," in *Proc. IEEE Int. Conf. Commun. Workshops*, Montreal, QC, Canada, 2021, pp. 1–6.

[14] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.

[15] T. Hester et al., "Deep Q-learning from demonstrations," in *Proc. AAAI Conf. Artif. Intell.*, New Orleans, Louisiana, USA, AAAI Press, 2018, pp. 3223–3230.

[16] H. Hu, X. Zhang, X. Yan, L. Wang, and Y. Xu, "Solving a new 3D bin packing problem with deep reinforcement learning method," 2017, *arXiv:1708.05930*.

[17] S. Irnich and G. Desaulniers, "Shortest path problems with resource constraints," in *Column Generation*, Berlin, Germany: Springer, 2005, pp. 33–65.

[18] H. Joksch, "The shortest route problem with constraints," *J. Math. Anal. Appl.*, vol. 14, no. 2, pp. 191–197, 1966.

[19] E. B. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, Long Beach, CA, USA, 2017, pp. 6348–6358.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Representations*, San Diego, CA, USA, 2015, pp. 1–11.

[21] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proc. 5th Int. Conf. Learn. Representations*, Toulon, France, 2017, pp. 1–14.

[22] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!," in *Proc. 7th Int. Conf. Learn. Representations*, New Orleans, LA, USA, 2019, pp. 1–12.

[23] Z. Li, Q. Chen, and V. Koltun, "Combinatorial optimization with graph convolutional networks and guided tree search," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, Montréal, Canada, 2018, pp. 537–546.

[24] Q. Ma, S. Ge, D. He, D. Thaker, and I. Drori, "Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning," 2019, *arXiv:1911.04936*.

[25] S. Manchanda, A. Mittal, A. Dhawan, S. Medya, S. Ranu, and A. K. Singh, "GCOMB: Learning budget-constrained combinatorial algorithms over billion-sized graphs," in *Proc. 34th Int. Conf. Neural Inf. Process. Syst.*, 2020, Art. no. 1679.

[26] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM Special Int. Group Data Commun.*, Beijing, China, 2019, pp. 270–288.

[27] Y. Marinakis, A. Migdalas, and A. Sifaleras, "A hybrid particle swarm optimization - variable neighborhood search algorithm for constrained shortest path problems," *Eur. J. Oper. Res.*, vol. 261, no. 3, pp. 819–834, 2017.

[28] F. S. Melo and M. I. Ribeiro, "Q -learning with linear function approximation," in *Proc. 20th Annu. Conf. Learn. Theory*, San Diego, CA, USA, Springer, 2007, pp. 308–322.

[29] V. Mnih et al., "Playing Atari with deep reinforcement learning," 2013, *arXiv:1312.5602*.

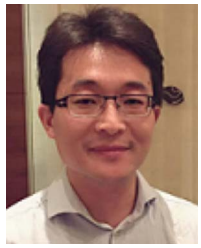
[30] D. Mukhutdinov, A. Filchenkov, A. Shalyto, and V. Vyatkin, "Multi-agent deep learning for simultaneous optimization for time and energy in distributed routing system," *Future Gener. Comput. Syst.*, vol. 94, pp. 587–600, 2019.

[31] M. Nazari, A. Oroojlooy, L. V. Snyder, and M. Takác, "Reinforcement learning for solving the vehicle routing problem," in *Proc. 32nd Int. Conf. Neural Inf. Process. Syst.*, Montréal, Canada, 2018, pp. 9861–9871.

[32] A. Oroojlooyjadid, M. Nazari, L. V. Snyder, and M. Takác, "A deep q-network for the beer game with partial information," 2017, *arXiv:1708.05924*.

[33] M. A. Riedmiller, "Neural fitted Q iteration - first experiences with a data efficient neural reinforcement learning method," in *Proc. Eur. Conf. Mach. Learn.*, Porto, Portugal, Springer, 2005, pp. 317–328.
[34] G. Righini and M. Salani, "Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints," *Discret. Optim.*, vol. 3, no. 3, pp. 255–273, 2006.
[35] G. Righini and M. Salani, "New dynamic programming algorithms for the resource constrained elementary shortest path problem," *Networks*, vol. 51, no. 3, pp. 155–170, 2008.
[36] D. T. Sanchez, "cspy: A Python package with a collection of algorithms for the (resource) constrained shortest path problem," *J. Open Source Softw.*, vol. 5, no. 49, pp. 1655–1658, 2020.
[37] L. Santos, J. Coutinho-Rodrigues, and J. R. Current, "An improved solution algorithm for the constrained shortest path problem," *Transp. Res. Part B: Methodol.*, vol. 41, no. 7, pp. 756–771, 2007.
[38] A. Sedeño-Noda and S. Alonso-Rodríguez, "An enhanced K-SP algorithm with pruning strategies to solve the constrained shortest path problem," *Appl. Math. Comput.*, vol. 265, pp. 602–618, 2015.
[39] R. Solozabal, J. Ceberio, and M. Takác, "Constrained combinatorial optimization with reinforcement learning," 2020, *arXiv:2006.11984*.
[40] R. Sutton and A. Barto, *Reinforcement Learning, second edition: An Introduction*. Cambridge, MA, USA: MIT Press, 2018.
[41] C. Tilk, A. Rothenbächer, T. Gschwind, and S. Irnich, "Asymmetry matters: Dynamic half-way points in bidirectional labeling for solving shortest path problems with resource constraints faster," *Eur. J. Oper. Res.*, vol. 261, no. 2, pp. 530–539, 2017.
[42] S. Uppoor, O. Trullols-Cruces, M. Fiore, and J. M. Barcelo-Ordinas, "Generation and analysis of a large-scale urban vehicular mobility dataset," *IEEE Trans. Mobile Comput.*, vol. 13, no. 5, pp. 1061–1075, May 2014.
[43] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *Proc. 30th AAAI Conf. Artif. Intell.*, Phoenix, Arizona, USA, AAAI Press, 2016, pp. 2094–2100.
[44] A. Vera, S. Banerjee, and S. Samaranayake, "Computing constrained shortest-paths at scale," *Operations Res.*, vol. 70, pp. 160–178, 2020.
[45] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst.*, Montreal, Quebec, Canada, 2015, pp. 2692–2700.
[46] C. Wang, J. Li, F. Ye, and Y. Yang, "NETWRAP: An NDN based realtimewireless recharging framework for wireless sensor networks," *IEEE Trans. Mobile Comput.*, vol. 13, no. 6, pp. 1283–1297, Jun. 2014.
[47] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," in *Proc. 33rd Int. Conf. Mach. Learn.*, New York City, NY, USA, 2016, pp. 1995–2003.
[48] J. Yin, W. Rao, and C. Zhang, "Learning shortest paths on large dynamic graphs," in *Proc. IEEE 22nd Int. Conf. Mobile Data Manage.*, Toronto, ON, Canada, 2021, pp. 201–208.

Jiaming Yin received the BE degree from Tongji University, Shanghai, China, in 2019, where she is currently working toward the PhD degree with the School of Software Engineering. Her research interests include time series, reinforcement learning, and mobile computing.



Weixiong Rao (Member, IEEE) received the PhD degree from The Chinese University of Hong Kong, Hong Kong, in 2009. After that, he worked for the Hong Kong University of Science and Technology (2010), University of Helsinki (2011–2012), and University of Cambridge Computer Laboratory Systems Research Group (2013) as postdoctor researchers. He is currently a full professor with the School of Software Engineering, Tongji University, China (since 2014). His research interests include mobile computing and spatiotemporal data science, and is a member of the CCF and ACM.

Qinpei Zhao received the BSc degree in automation technology from Xi'dian University, Xi'an, China, in 2004, the MSc degree in pattern recognition and image processing from Shanghai Jiaotong University, Shanghai, China, in 2007, and the PhD degree in computer science with the University of Eastern Finland, in 2012. She is currently with the School of Software Engineering, Tongji University, China. Her current researches are focused on clustering algorithm and multimedia processing.

Chenxi Zhang received the degree from the National University of Defense Technology, in 1988. He is currently a professor of computer science with the School of Software Engineering, Tongji University. He has written or edited two books, including *New Generation Computing: Recent Research* (North-Holland Press, 1990) and *Research on Frontiers in Computing* (Tsinghua University Press, 1989). His research interests include methods and theory in distributed systems, peer-to-peer networks, and next generation networks. He received the National Science Fund for Distinguished Young Scholars, in 1996.

Pan Hui (Fellow, IEEE) received the both bachelor's and MPhil degrees from the University of Hong Kong, and the PhD degree from the Computer Laboratory, University of Cambridge. He is the Nokia chair professor in data science and professor of computer science with the University of Helsinki. He is also the director of the HKUST-DT Systems and Media Lab with the Hong Kong University of Science and Technology. He was a senior research scientist and then a distinguished scientist for Telekom Innovation Laboratories (T-labs) Germany and an adjunct professor of social computing and networking with Aalto University. His industrial profile also includes his research with Intel Research Cambridge and Thomson Research Paris. He has published more than 300 research papers and with more than 17,500 citations. He has 30 granted and filed European and US patents in the areas of augmented reality, data science, and mobile computing. He has been serving on the organising and technical program committee of numerous top international conferences including ACM SIGCOMM, MobiSys, IEEE Infocom, ICNP, SECON, IJCAI, AAAI, ICWSM and WWW. He is an associate editor for the leading journals *IEEE Transactions on Mobile Computing*. He is an ACM distinguished scientist, and a member of the Academia Europaea.