Vol. 63 (2022) No. 4 December n. 214

ASSEMBLY SEQUENCE OPTIMIZATION OF SPATIAL TRUSSES USING GRAPH EMBEDDING AND REINFORCEMENT LEARNING

Kazuki HAYASHI[1], Makoto OHSAKI[2] and Masaya KOTERA[3]

[1]Assistant Professor, Kyoto University, Kyotodaigaku-katsura, Nishikyo, Kyoto 615-8540, Japan, hayashi.kazuki@archi.kyoto-u.ac.jp
[2]Professor, Kyoto University, Kyotodaigaku-katsura, Nishikyo, Kyoto 615-8540, Japan, ohsaki@archi.kyoto-u.ac.jp
[3]Graduate Student, Kyoto University, Kyotodaigaku-katsura, Nishikyo, Kyoto 615-8540, Japan, kotera.masaya.25n@st.kyoto-u.ac.jp

Editor's Note: The first author of this paper is one of the four winners of the 2022 Hangai Prize, awarded for outstanding papers that are submitted for presentation and publication at the annual IASS Symposium by younger members of the Association (under 30 years old). It is published here with permission of the editors of the proceedings of the IASS Symposium 2022 "Innovation, Sustainability and Legacy", that was held in September 2022 in Beijing, China.

DOI: https://doi.org/10.20898/j.iass.2022.016

ABSTRACT

We consider a truss as a graph consisting of nodes and edges, and combine graph embedding (GE) and reinforcement learning (RL) to develop an agent for generating a stable assembly path for a truss with arbitrary configuration. GE is a method of embedding the features of a graph into a vector space. By using GE, the agent can obtain numerical information on neighboring members and nodes considering their connectivity. Since the stability of a structure is strongly affected by the relative positions of members and nodes, feature extraction by GE should be effective in considering the stability of a truss. The proposed method not only can train agents using trusses with arbitrary connectivity but also can apply trained agents to trusses with arbitrary connectivity, ensuring the versatility of the trained agents' applicability. In the numerical examples, the trained agents are verified to find rational assembly sequences for various trusses more than 1000 times faster than metaheuristic approaches. The trained agent is further implemented as a user-friendly component compatible with 3D modeling software.

Keywords: *truss, assembly sequence optimization, machine learning, reinforcement learning, graph embedding*

1. INTRODUCTION

To deal with the increasing construction costs and ensure the safety of construction work, it is becoming more and more important to improve the efficiency of construction processes. There have been many attempts to introduce optimization methods to create a rational construction plan. In particular, the problem of optimizing the order of assembling discrete structural components is a combinatorial optimization problem with a huge number of patterns, and is also an area of recent interest in robot−assisted construction planning [1,2].

Most of the existing optimization approaches to assembly planning are based on mathematical programming and metaheuristics. When the structural stability changes in the middle of the assembly sequence, structural analysis is required at each step. Accordingly, mathematical programming and metaheuristics, which rely heavily on iterative calculations, require enormous computational time to obtain reasonable solutions of assembly sequence.

If there is a mathematical model that can capture the causal relationship between the assembly sequence and its performance, not only can the assembly sequence be searched more efficiently, but also human engineers modify the construction sequence while referring to the output of the mathematical model; in this sense, a new collaborative design process between humans and computers can also be expected.

To this end, we focus on reinforcement learning (RL), which is a type of machine learning, in order to build a mathematical model. RL is a method to learn a

sequential decision-making process to maximize rewards through a huge number of simulations. Although RL has been reported to perform well in tasks that are difficult to control by rule-based programming [3] and in solving optimization problems [4], RL has rarely been applied to the field of skeletal structures because of its complex connectivity [5].

Figure 1 illustrates the training scheme outlined in this paper. In this research, truss models are converted into a data structure called a graph consisting of nodes and edges, and a method combining graph embedding (GE) [6] and RL is applied to develop an agent that can efficiently generate a stable assembly path for arbitrary 3D trusses. Furthermore, the trained agent is successfully implemented within 3D modeling software as a user-friendly component.

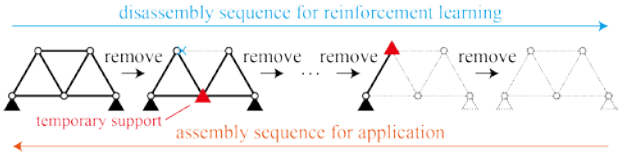


Figure 1: *Training scheme for assembly sequence optimization*

This paper is organized as follows. Section 2 formulates the assembly sequence optimization problem dealt with in this paper. Section 3 outlines the method of converting the optimization problem into an RL task and the combined method of GE and RL to train the agent. Section 4 presents numerical examples using various truss models. Section 5 describes the deployment of the trained agent and provides examples of visualization of the assembly sequence generated by the component. Finally, Section 6 presents the conclusions.

2. ASSEMBLY SEQUENCE OPTIMIZATION PROBLEM

A step of the assembly sequence of a truss shall be considered as a process of installing one member, and the assembly of a truss with n_{m} members is supposed to be completed in n_{m} steps. Let $n_{\mathrm{d}}^{(t_{\mathrm{c}})}$ be the total number of DOFs that are not constrained by permanent supports at step t_{c}, $\mathrm{rank}(\mathbf{K}^{(t_{\mathrm{c}})})$ be the rank of the global stiffness matrix $\mathbf{K}^{(t_{\mathrm{c}})} \in \mathrm{R}^{n_{\mathrm{d}}^{(t_{\mathrm{c}})} \times n_{\mathrm{d}}^{(t_{\mathrm{c}})}}$ at step t_{c}, and $n_{\mathrm{is}}^{(t_{\mathrm{c}})} = n_{\mathrm{d}}^{(t_{\mathrm{c}})} - \mathrm{rank}(\mathbf{K}^{(t_{\mathrm{c}})})$ be the degree of instability. At this step, there are $n_{\mathrm{is}}^{(t_{\mathrm{c}})}$ zero-eigenvalues for $\mathbf{K}^{(t_{\mathrm{c}})}$, and the eigenmodes corresponding to the zero eigenvalues are unstable deformations that should be constrained by temporary supports. To reduce the workload involved in the installation and removal of temporary supports, the temporary supports from the previous step are preferentially selected as a candidate for temporary supports.

Let $n_{\mathrm{ts}}^{(t_{\mathrm{c}})}$ denote the number of temporary supports at step t_{c}, the cost function F for minimizing the number of temporary supports in the entire construction process is defined as

$$F = \sum_{t_{\mathrm{c}}=1}^{n_{\mathrm{m}}} n_{\mathrm{ts}}^{(t_{\mathrm{c}})} \tag{1}$$

3. TRAINING METHOD

3.1. Conversion to a reinforcement learning task

Consider the process of reversely tracing the assembly sequence, i.e., removing a member at each step from the completed state until all members have been removed. The decision-making process in a typical RL task is assumed to be a Markov decision process (MDP), in which the transition to the next state s' and the observed reward r depend only on the current state s and action a. For this prerequisite, we first define the components s, a, and r of the disassembly process.

The action a is defined as the operation of removing one member from the existing members at the current step. The reward associated with the removal of the member at step t_{c} is determined in accordance with Eq. (1) as

$$r = -n_{\mathrm{ts}}^{(t_{\mathrm{c}})} \tag{2}$$

Let $n_{\mathrm{n}}^{(t_{\mathrm{c}})}$ and $n_{\mathrm{m}}^{(t_{\mathrm{c}})}$ denote the numbers of existing nodes and members at step t_{c}, respectively. The state s can be expressed as a tuple $\{\mathbf{C}, \hat{\mathbf{v}}\}$, in which $\mathbf{C} \in \mathrm{R}^{n_{\mathrm{m}}^{(t_{\mathrm{c}})} \times n_{\mathrm{n}}^{(t_{\mathrm{c}})}}$ is a connectivity matrix with respect to existing truss members at step t_{c}, and $\hat{\mathbf{v}} \in \mathrm{R}^{n_{\mathrm{fn}} \times n_{\mathrm{n}}^{(t_{\mathrm{c}})}}$ is the input matrix that concatenates node inputs $\mathbf{v}_k \in \mathrm{R}^{n_{\mathrm{fn}}}$ $(k = 1, \cdots, n_{\mathrm{n}}^{(t_{\mathrm{c}})})$. Here, $n_{\mathrm{fn}} = 2$ binary flags to distinguish permanent pin-supports and locally unstable nodes are adopted as node input. In particular, the binary flag for local instability is very important because nodal instability cannot be evaluated solely by member connectivity. For example, while the support condition and member connectivity between the two trusses in Fig. 2 are the same, the central node in the left figure is locally stable and that in the right figure is unstable.

Figure 2: *Stable (left) and unstable (right) two-bar trusses*

Local instability of nodes can be numerically determined according to the dimensionality of the truss model. For a 2D truss, if all the existing members connected to a certain node are on the same axis, the node is regarded as locally unstable. For a 3D truss, when all the existing members connected to a certain node are on the same plane, the node is regarded as locally unstable.

The node inputs do not include properties that require structural analysis, such as stresses, displacements, and eigenvalues of stiffness matrices. This implies that the agent can estimate the assembly order without performing any structural analysis after the training by RL is completed. From the above settings, the reward r and the next state s' are uniquely determined by the current state-action pair s and a, and this disassembly process is successfully formulated as an MDP.

However, the state $s=\{\mathbf{C},\hat{\mathbf{v}}\}$ is difficult to handle by RL because the sizes of $\mathbf{C}$ and $\hat{\mathbf{v}}$ change according to the numbers of existing nodes and members, which requires unrealistic computational costs to train the agent for each truss that is to be assembled. To address this problem, GE is introduced below to construct an RL agent trainable using various trusses with fewer parameters.

3.2. Extraction of member features using graph embedding

In this section, the feature vectors of each member with size n_f are extracted using GE to approximate the state $s=\{\mathbf{C},\hat{\mathbf{v}}\}$. Using fully connected neural network layers $\Phi_1,\cdots,\Phi_5$ with n_f units and trainable parameters $\boldsymbol{\theta}_1,\cdots,\boldsymbol{\theta}_5$, respectively, the member feature matrix $\hat{\boldsymbol{\mu}}_{(t_u)}\in\mathrm{R}^{n_\mathrm{f}\times n_\mathrm{m}^{(t_c)}}$ is updated by the following equation:

$$\hat{\boldsymbol{\mu}}_{(1)}=\varphi\left(\Phi_1\left(\varphi\left(\Phi_2\hat{\mathbf{v}}\right)\right)\mathbf{C}_\mathbf{A}^\mathrm{T}\right) \tag{3a}$$

$$\hat{\boldsymbol{\mu}}_{(t_u+1)}=\varphi\left(\Phi_3\hat{\boldsymbol{\mu}}_{(t_u)}+\Phi_4\sum_{k=1}^{2}\frac{\varphi\left(\Phi_5\left(\mathbf{C}_k\mathbf{C}_\mathrm{A}^\mathrm{T}\hat{\boldsymbol{\mu}}^{(t_u)\mathrm{T}}\right)^\mathrm{T}\right)}{\mathbf{n}_k^\mathrm{c}}\right) \tag{3b}$$

where $\mathbf{C}_\mathrm{A}$ is a matrix that takes an absolute value for each component of the connectivity matrix $\mathbf{C}$, $\mathbf{C}_1$ is a matrix whose components take 1 when the corresponding component of $\mathbf{C}$ is −1 and 0 otherwise, and $\mathbf{C}_2$ is a matrix whose components take 1 when the corresponding component of $\mathbf{C}$ is 1 and 0 otherwise. $\mathbf{n}_k^\mathrm{c}\in\mathrm{R}^{n_\mathrm{f}\times n_\mathrm{m}^{(t_c)}}$ is a matrix in which the row vector containing the number of members connected to their k th end is repeated in the column direction n_f times. φ is a Leaky Rectified Linear Unit (Leaky ReLU) function with a gradient for negative inputs of 0.2. See Ref. [5] for details.

The concepts of the formulation of Eq. (3) are summarized as follows:

- Convolutional operation using connectivity matrix

  In order to extract features taking into account the member connectivity, we have introduced a convolutional operation that propagates the numerical information of neighboring nodes and members using a connectivity matrix.

- Linear transformation using trainable parameters

  Trainable parameters $\boldsymbol{\theta}_1,\cdots,\boldsymbol{\theta}_5$ are introduced for calculating weighted linear sums from inputs and features. By adjusting these parameters during the learning process, meaningful features taking structural properties into consideration are extracted.

- Nonlinear transformation using activation function

  Since the nonlinear approximation cannot be performed only by the above linear transformation, the nonlinear function Leaky ReLU is introduced.

- Feature scaling

  Feature values are scaled using the number of connected members $\mathbf{n}_1^\mathrm{c}$ and $\mathbf{n}_2^\mathrm{c}$ so that their magnitude is uniform regardless of the number of members connected to the node.

Equation (3b) needs to be repeated multiple times in order to consider the contribution of nodes and members that are not directly adjacent to each other. In the following, Eq. (3b) is repeated twice, and the feature $\hat{\boldsymbol{\mu}}=\hat{\boldsymbol{\mu}}_{(3)}$ is regarded as the member feature matrix that expresses the current state.

3.3. Estimation of action values using extracted member features

In a value-based RL, the action value $Q(s,a) \in \mathrm{R}$ is defined as the expected return, a sum of rewards in this study, by taking action a in a certain state s and then following the current policy.

From the member feature matrix $\hat{\boldsymbol{\mu}}$, the action values of all members denoted by $\hat{\mathbf{Q}} \in \mathrm{R}^{n_{\mathrm{m}}^{(t_c)}}$ can be estimated by introducing another trainable parameters $\boldsymbol{\theta}_6 \in \mathrm{R}^{1\times 2n_{\mathrm{f}}}$ as

$$\mathbf{Q}(\hat{\boldsymbol{\mu}}) = \boldsymbol{\theta}_6 [\hat{\boldsymbol{\mu}}_\Sigma ; \hat{\boldsymbol{\mu}}] \tag{4}$$

where $[\bullet;\bullet]$ is a concatenation operator. $\hat{\boldsymbol{\mu}}_\Sigma \in \mathrm{R}^{n_{\mathrm{f}}\times n_{\mathrm{m}}^{(t_c)}}$ is a matrix in which the sums of each row of $\hat{\boldsymbol{\mu}}$ are arranged $n_{\mathrm{m}}^{(t_c)}$ times in the row direction, and can be regarded as the features of the entire truss. Once the action values have been estimated, the action with the highest expected return can be determined by the following equation, which is called the greedy policy.

$$\pi(s) = \underset{a}{\operatorname{argmax}}\, Q(s,a) \tag{5}$$

3.4. Loss minimization for training

Here, the learning problem can be defined as adjusting the trainable parameters $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_1, \cdots, \boldsymbol{\theta}_6\}$ so as to appropriately select the best actions and maximize the return. To this end, it is important to define the error that $\boldsymbol{\Theta}$ should minimize. When action a is taken in state s, and next state s' and reward r are observed, the estimation error δ of the action value is defined by

$$\delta = \left| \sum_{t=1}^{m} \gamma^{t-1} r_t + \gamma^m Q\left(s', \underset{a'}{\operatorname{argmax}}\, Q(s', a' \,|\, \boldsymbol{\Theta}) \,|\, \tilde{\boldsymbol{\Theta}} \right) - Q(s, a \,|\, \boldsymbol{\Theta}) \right| \tag{6}$$

where $\tilde{\boldsymbol{\Theta}}$ is the trainable parameters obtained at the previous learning step, and is synchronized with the current trainable parameter $\boldsymbol{\Theta}$ every 100 steps, which stabilizes the training [3]. The number m is the hyper-parameter of multi-step learning [7], in which the rewards from the current state to m steps ahead are memorized, and the loss is computed using those rewards and the estimated action value in the state m steps ahead. Since the gradient of δ becomes discontinuous at $\delta = 0$, the squared error $L = \delta^2$ shall be the loss to be minimized.

To stabilize the training process, we further introduce mini-batch training, in which trainable parameters are updated using multiple samples at the same time [5]. During the sampling phase, prioritized experience replay [8] is further employed to preferentially use samples that are highly unexpected to the agent. Specifically, the most recent error δ_i calculated for sample i is used to weight the sampling probability p_i as follows:

$$p_i = \frac{\nu_i}{\sum_{k=1}^{N_{\mathrm{B}}} \nu_k} \tag{7a}$$

$$\nu_i = (\delta_i + 0.001)^{0.6} \tag{7b}$$

The average of errors in the mini-batch is defined as the loss function to be minimized in mini-batch training. The trainable parameters are optimized using Adam [9], which is a gradient-based optimization algorithm frequently adopted for training neural networks.

4. NUMERICAL EXAMPLES

4.1. Training settings

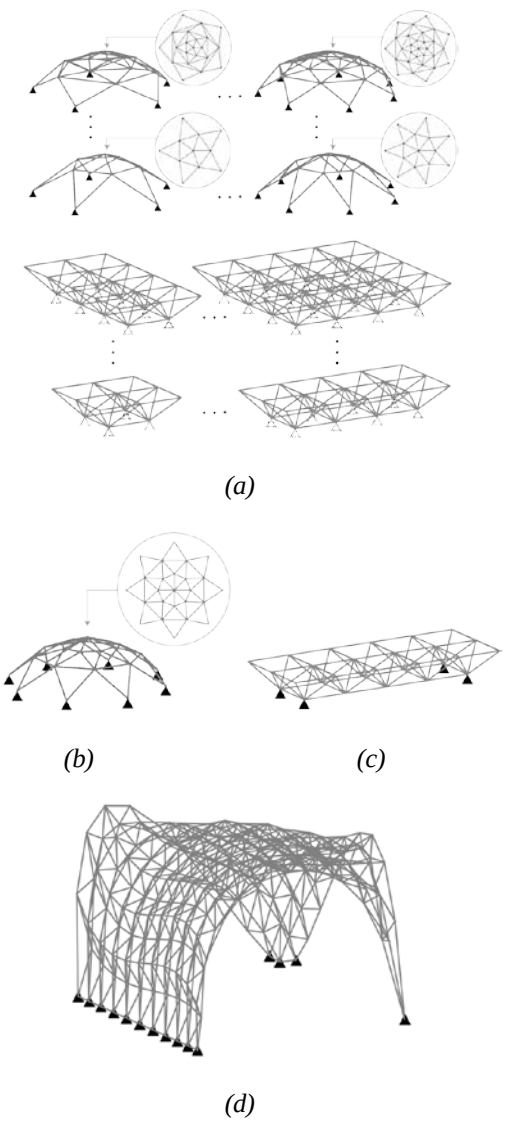
In the following, the units are omitted because they are not important. An episode is defined as the process of removing all members from the completed truss. At the beginning of the episode, the shape and connectivity of the truss is randomly generated from a set of trusses for training. At each step of the episode, the agent calculates the action value from the input based on Eqs. (3) and (4), selects an action, removes a member, and computes temporary support positions using eigenvalue analysis. Since the location of the temporary supports does not depend on the material or cross-sectional information, the Young's modulus and the cross-sectional area of the members are 1.0 for convenience.

As shown in Fig. 3(a), various trusses of different shape and connectivity are used for training so that the agent can demonstrate generalization performance. The flat roof trusses have four permanent pin-supports randomly selected from the bottom nodes to form a rectangle. The bottom nodes of the dome truss are always permanently pin-supported. During training, initial nodal positions are subjected to small irregularity with a probability of 0.05, so that the agent can properly learn the local instability.

In order to let the agent experience various states and actions during training, the agent acts based on the ε-greedy policy that selects random behavior with a low probability ε, while the agent adopts the greedy policy that eliminates randomness during verification. In order to reduce randomness as the training progresses, ε is calculated by the following equation:

$$\varepsilon = 0.2 \cdot \frac{\overline{n}_{\text{ep}} - n_{\text{ep}}}{\overline{n}_{\text{ep}}} \tag{8}$$

where n_{ep} is the number of trained episodes and $\overline{n}_{\text{ep}}$ is the total number of episodes, which is 5000 in this study.



(a)

(b) *(c)*

(d)

Figure 3: *Spatial trusses used during training. (a): for training. (b,c,d): for validation.*

4.2. Training results

It took 2.5 hours to complete the 5000-episode training. Figure 4 shows the history of cumulative rewards for the trusses of Figs. 3(b) and 3(c) recorded every 10 episodes. For both models, the cumulative reward sharply increases in the first 200 episodes and continues to maintain high values. This stable history is noteworthy considering that these shapes were not used during training. This implies that the agent exhibited generalization performance through limited learning materials.

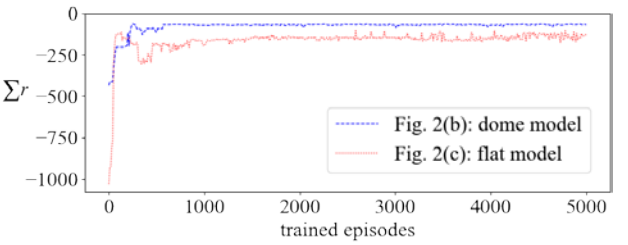


Figure 4: *History of cumulative rewards in each test*

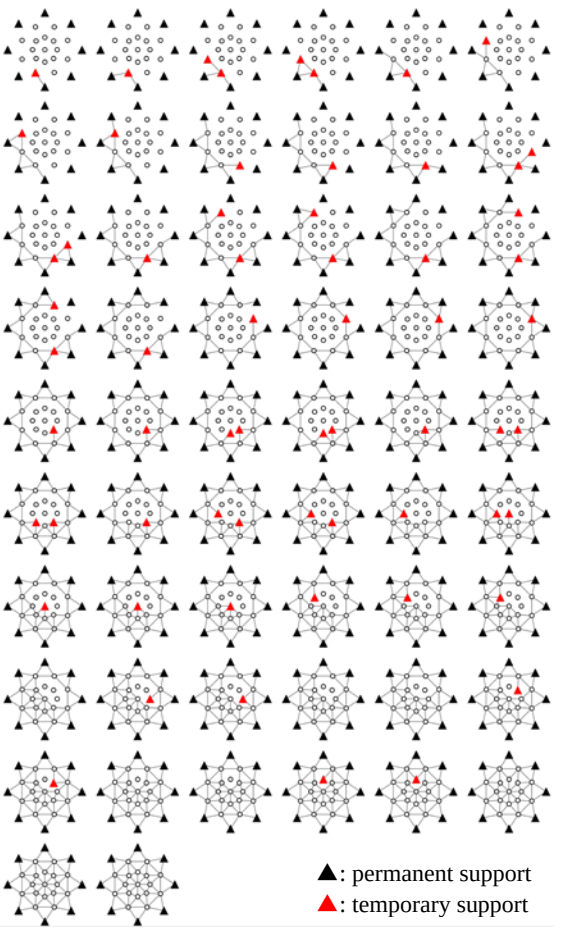


Figure 5: *Assembly sequence of the dome model (Fig. 2(b)) predicted by the best agent (plan view)*

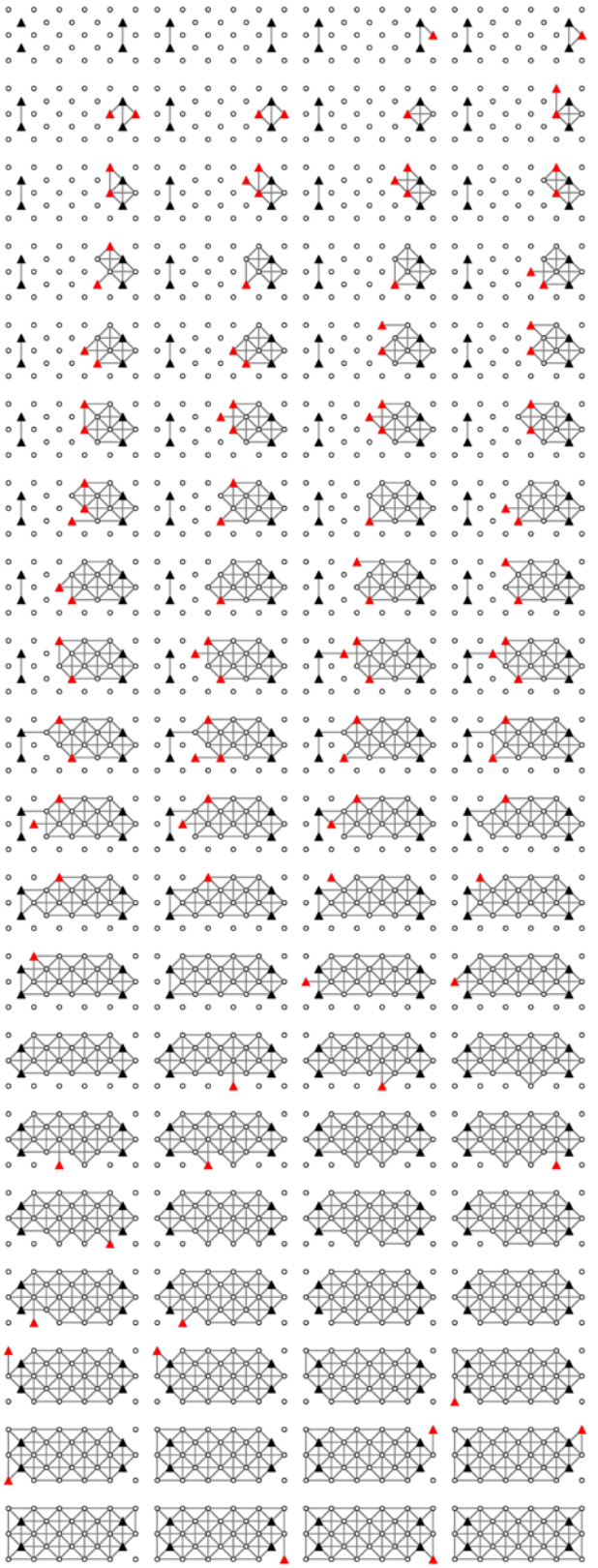


Figure 6: *Assembly sequence of the flat model (Fig. 2(c)) predicted by the best agent (plan view)*

Let the agent at 3470 trained episodes, which has the least product of cumulative rewards obtained for each of the two trusses in Figs. 3(b) and 3(c), be the best agent. The assembly sequences of the trusses for validation estimated by the best agent are shown in Figs. 5-7, respectively. In the dome model, the members were sequentially constructed from the outer supports to the inner nodes. In the flat model, members were sequentially constructed from a pair of supports to the other pair of supports, and then the outer peripheral members were finally constructed. In the latticed shell model, the arch shape was constructed from the support and then the members were constructed towards the remaining support.

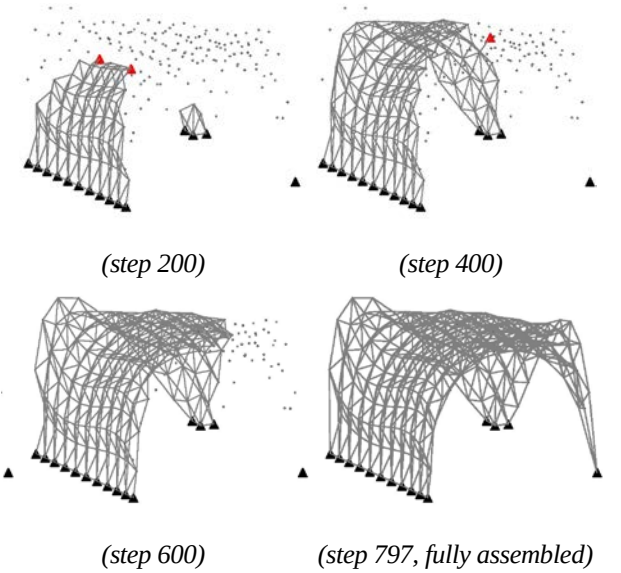


Figure 7: *Assembly sequence of the latticed shell model (Fig. 2(d)) predicted by the best agent*

4.3. Comparison to metaheuristic methods

In order to quantitatively evaluate the ability of the trained agent for searching solutions, we compare the trained agent with two metaheuristic methods, genetic algorithm (GA) and covariance matrix adaptation evolution strategy (CMA-ES) [10].

The GA-based method proposed by Kaneko et al. [11] is adopted as the first benchmark because this method deals with the similar problem of minimizing the number of temporary supports. The setting of GA hyperparameters conforms to Ref. [11] (population: 50, generation: 100, crossover rate: 0.4, mutation rate: 0.005, elite selection rate: 0.02, parent selection by roulette search, etc.). However, in the crossover process, the original method is not used, and a uniform order crossover that can efficiently obtain almost equivalent crossover results is used. For selection methods other than elite-selected individuals not described in Ref. [11], roulette selection based on fitness is employed.

Table 1: *Comparison of the optimization performance (t_e: average elapsed time for each optimization)*

Model	GE+RL		GA		CMA-ES	
Fig. 4(b) $n_n = 25$ $n_m = 56$	F	**63**	F_{median}	163	F_{median}	85
			F_{min}	117	F_{min}	77
	t_e [s]	**0.2**	t_e [s]	4936.2	t_e [s]	772.9
Fig. 4(c) $n_n = 28$ $n_m = 80$	F	**108**	F_{median}	605	F_{median}	290
			F_{min}	543	F_{min}	285
	t_e [s]	**0.3**	t_e [s]	50197.6	t_e [s]	5293.9
Fig. 4(d) $n_n = 220$ $n_m = 797$	F	**666**	F_{median}	--*	F_{median}	--*
			F_{min}	--*	F_{min}	--*
	t_e [s]	**47.5**	t_e [s]	--*	t_e [s]	--*

∗ Not implemented due to huge computational cost

CMA-ES introduces covariance matrix adaptation (CMA) into a stochastic algorithm, evolution strategy (ES). CMA-ES is chosen as the second benchmark method because CMA-ES-based methods outperform other metaheuristics for functions that are difficult to optimize [12], and because it has relatively few hyperparameters. Since CMA-ES is a method that deals with continuous variables, the sorted index of the continuous variables is used as the assembly order. The number of individuals and the number of generations are set to 50 and 100, respectively.

GA and CMA-ES algorithms are implemented using DEAP [13], a Python library for evolutionary algorithms. Since the quality of solutions obtained by metaheuristics strongly depends on the initial solution, GA and CMA-ES algorithms are implemented 5 times for randomly generated initial solutions.

Table 1 shows the results of optimizing the assembly sequence using the trained agent (denoted as GE+RL), GA, and CMA-ES. Note again that the smaller F is, the more rational the assembly order is with fewer temporary supports. It can be seen that GE+RL efficiently obtains better solutions than those obtained by GA and CMA-ES, which required a huge computational time.

The superior computational efficiency of GE+RL is due not only to the small computational load of using the trained agent, but also to the saving computational cost for poor solutions, which is wasted in GA and CMA-ES. Since GA and CMA-ES are stochastic approaches, poor solutions that require a large number of temporary supports occur during optimization. In such cases, more repetitive eigenvalue analysis is required to determine the temporary support locations compared with superior solutions. On the other hand, in GE + RL, the agent attempts to generate superior solutions based on experience, thus avoiding an increase in the computational cost for the eigenvalue analysis during training. This is the reason why the time required to train the agent was relatively short.

5. DEPLOYMENT

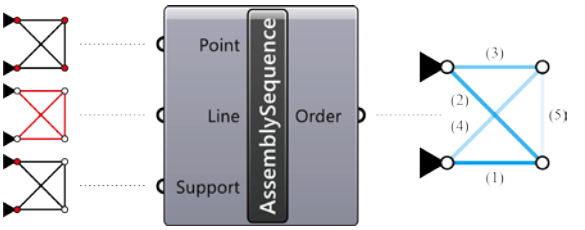


Figure 8: *Developed Grasshopper component that packages the trained RL agent*

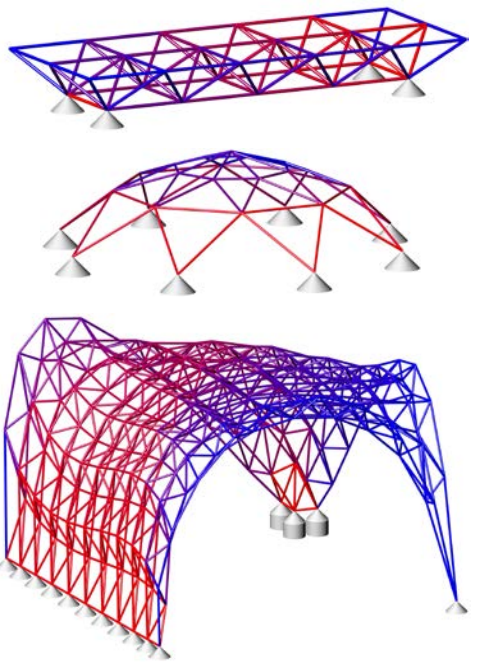


Figure 9: *Colormaps of predicted assembly sequences visualized in Rhino and Grasshopper*

Since the process of calculating the action values by this method is a sequence of simple matrix operations, the trained agent can be easily used on different computers once the trainable parameters are imported. Taking advantage of this, we packaged the trained agent as a component compatible with Grasshopper, a visual programming interface of the Rhino 3D modelling software, as shown in Fig. 8.

This component automatically calculates the assembly order using the trained agent by specifying the nodal positions, the line segments representing the member connectivity, and the positions of the supports. This component is freely available in Food4Rhino, plug-in community service for Rhino. See more details at https://www.food4rhino.com/en/app/assembly-sequence-predictor.

Figure 9 shows examples of displaying the assembly sequence predicted by the developed component in a colormap. The red members are assembled in the early stage, and the blue members are assembled in the final stage. In this way, packaging the agent for use within 3D modeling software has made it possible to easily visualize complex sequence data in an easy-to-see format.

6. CONCLUSION

We propose a machine learning method that combines GE and RL for the assembly sequence optimization for spatial trusses to minimize the number of temporary supports in the assembly process. Representing the state of the trusses by GE enables to extract the feature vectors of the same size and consider the connectivity of nodes and members. Since the sizes of trainable parameters do not depend on the numbers of nodes and members, the same calculation procedure can be applied to trusses of different configuration. Owing to this property, various trusses can be used during training and the trained agent can also be applied to various trusses.

The trained agent successfully generated superior solutions with much less computational cost compared with metaheuristic approaches. The trained agent is further implemented as a Grasshopper component for easy use. These achievements may accelerate the use of machine learning for discrete structures, which have been difficult to apply machine learning methods.

ACKNOWLEDGMENTS

This research was sponsored by JSPS KAKENHI No. JP20H04467 and JSPS Grant-in-Aid for Young Scientists (Start-up) No. JP21K20461.

DATA AVAILABILITY

All data and code used for this publication are publicly available from the following repository: https://github.com/kazukihayashi/AssemblySequenceOptimization.git. The developed Grasshopper component is available at https://www.food4rhino.com/en/app/assembly-sequence-predictor.

REFERENCES

[1] M. McEvoy, E. Komendera and N. Correll, “Assembly path planning for stable robotic construction,” in 2014 IEEE International Conference on Technologies for Practical Robot Applications (TePRA), Boston, MA, USA, April 14-15, 2014, IEEE, pp. 1-6, 2014. (DOI: 10.1109/TePRA.2014.6869152)

[2] L. Brodbeck and F. Iida, “Automatic real-world assembly of machine-designed structures,” in 2014 IEEE International Conference on Robotics and Automation (ICRA), Hong Kong, China, May 31-June 5, 2014,pp.1221-1226,2014. (DOI: 10.1109/ICRA.2014.6907009)

[3] V. Mnih et al., “Human-level control through deep reinforcement learning,” Nature, vol. 518,pp.529–533,2015. (DOI: 10.1038/nature14236)

[4] K. M. Powell, D. Machalek and T. Quah, “Real-time optimization using reinforcement learning,” Computers & Chemical Engineering, vol. 143, no. 107077, 2020. (DOI: 10.1016/j.compchemeng.2020.107077)

[5] K. Hayashi and M. Ohsaki, “Graph-based reinforcement learning for discrete cross-section optimization of planar steel frames,” Advanced Engineering Informatics, vol. 51, no.101512,2021. (DOI:10.1016/j.aei.2021.101512)

[6] H. Cai, V. W. Zheng and K. Chang, “A comprehensive survey of graph embedding: Problems, techniques, and applications”, IEEE Transactions on Knowledge and Data Engineering, vol. 30, no. 9, pp. 1616–1637, 2018. (DOI: 10.1109/TKDE.2018.2807452)

[7] R. S. Sutton and A. G. Barto, Introduction to Reinforcement Learning, MIT Press, Cambridge, MA, USA, 1st edition, 1998. (ISBN: 978-0262039246)

[8] T. Schaul, J. Quan, I. Antonoglou and D. Silver, “Prioritized experience replay,” arXiv, no.1511.05952,2015. (DOI: 10.48550/arXiv.1511.05952)

[9] D. P. Kingma and J. Ba, “Adam: A Method for Stochastic Optimization,” arXiv, no. 1412.6980,2014. (DOI: 10.48550/arXiv.1412.6980)

[10] H. Nikolaus and O. Andreas, “Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation,” in the 1996 IEEE International Conference on Evolutionary Computation, Nagoya, Japan, May 20-22, 1996, IEEE, pp. 312-317,1996.
(DOI: 10.1109/ICEC.1996.542381)

[11] Y. Kaneko et al., “Construction process optimization for truss structures by genetic algorithms,” J. Struct. Constr. Eng., AIJ, vol. 63, no. 508, pp. 87–92, 1998. (in Japanese).
(DOI: 10.3130/aijs.63.87_3)

[12] N. Hansen, A. Auger, R. Ros, S. Finck and P. Pošík, “Comparing results of 31 algorithms from the black-box optimization benchmarking BBOB-2009,” in the 12th annual conference companion on Genetic and Evolutionary computation, Portland, OR, USA, July 7-11, 2010, Association for Computing Machinery, pp. 1689-1696, 2010.
(DOI: 10.1145/1830761.1830790)

[13] F. Fortin et al., “DEAP: Evolutionary algorithms made easy,” Journal of Machine Learning Research, vol. 13, pp. 2171-2175, 2012.