Reinforcement Learning for Architectural Combinatorial Optimization

Leon Wietschorke
Digital Design Unit
Technische Universität Darmstadt
Darmstadt, Germany
leon.wietschorke@stud.tu-darmsatdt.de

Yuxi Liu
Digital Design Unit
Technische Universität Darmstadt
Darmstadt, Germany
de_liuyuxi@163.com

Jianpeng Chen
Digital Design Unit
Technische Universität Darmstadt
Darmstadt, Germany
jianpengchen92@gmail.com

***Abstract*—Creating an interface for architects to use reinforcement learning algorithms in their usual design software empowers architects and designers to come up with new solutions for old problems. In a capitalist society dealing with a climate crisis it is an interest to find optimized solutions to the problems of floor planning, material distribution and recycling elements. In this report we present an interface that creates a link between the design environment of Grasshopper and the RL algorithms of stable baselines 3.**

Keywords—Grasshopper3D, reinforcement learning, interface, architectural combinatorial problems

I. Introduction

The floor area of the world needs to be nearly doubled until 2060 and as building and construction today use 36% of the global final energy and produce 39% of the energy-related carbon dioxide, the construction process needs to change drastically. [1] It not only needs a lot of energy but also produces a lot of waste as falsework is often only used once. We think that robots in combination with AI and an efficient material distribution have the potential to revolutionize the building industry. This IP-Project is a first step into solving architectural combinatorial problems with the help of reinforcement learning (RL).

There are many combinatorial problems in the field of architecture. From optimizing floor plans [2, 3] to combining modular building elements to rearranging recycled building parts, represents a small part of combinatorial problems. These problems are not new to architecture but exist since the beginning of AI research in the 1960s [2]. With new and more powerful algorithms there comes a hope to solve the "wicked" design problems of architecture [4]. As architectural problems are always design problems there is no completely optimal solution to a problem, because there is nothing like a perfect design it always depends on subjective perception. This is a huge difference to engineering problems where the constraining factors form an optimal solution.

The method of reinforcement learning can be one option to solve complex design problems as it reacts to the environment. This report presents a general interface between the architectural design software of Rhino and Grasshopper and the scripting environment to run RL algorithms. In three case studies we show different approaches on how this interface can be used to solve combinatorial problems. As this is an early research on implementing RL algorithm in an design environment for architects it lays a stepstone for future research.

II. General interface

As architects use different environments to design buildings and structures, than computer scientists use for RL, it is necessary to create an interface between these environments.

Rhino is a 3D modeling software that allows users to create complex geometries. With Rhino, however, only a direct modeling is possible. Only with the built-in plug-in Grasshopper a parametric modeling is possible. Grasshopper is a visual programming language that uses connectable components to create 3D geometry. It is thereby a very powerful tool for architects and designers, because it allows to parametrically change the geometry without the need of totally rebuilding the object. In 2007, where Grasshopper was published for the first time, this was quite a novum in CAD applications and still today the parametrization of buildings is rarely done . Various projects show the potential of parametrized buildings. It not only provides a huge freedom of design but can also directly generate machine code for robots and CNC machines to manufacture the buildings.[5, 6] This opens up the question why not more projects a thought form design to production. Grasshopper provides an open platform that allows third party developers to create their own plugins for it. This enables us to implement RL algorithms in Grasshopper.
One essential plugin for the implementation of RL algorithms is the Hoopsnake plugin [7]. Hoopsnake allows the user to create self-referencing loops in Grasshopper, this is crucial because Grasshopper is from its initial point a linear program in that components can't refer to themself. Hoopsnake avoids this problem by creating a local copy of the input values and, when triggered, outputs this stored value. With this method the Hoopsnake component can avoid the recursive loop avoidance check of Grasshopper.

To call the RL scripts running outside of Grasshopper we used python scripting components to run a socket communication with the algorithms. The RL algorithms are part of stable baselines 3, that can run several different RL algorithms.

Grasshopper acts as the environment that receives actions and a reset flag and outputs a reward, a done flag, the observations and an info. These four information get sent to the RL algorithm that uses this information to send a new action. This general approach enables the interface to be used in various different applications. Grasshopper acts thereby only as the environment that interprets the sent

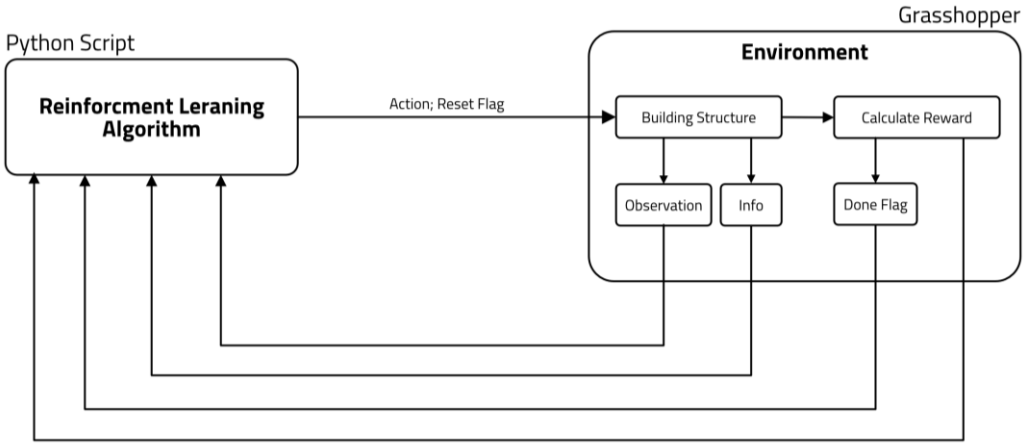

Fig.1: Flow diagram showing the principal communication of the interface

actions to calculate a reward. The algorithm does not know what is happening inside the environment. Fig. 1 shows an abstract flow diagram that showcases the communication between Grasshopper an the RL algorithm. Fig. 2 shows the translation to Grasshopper On the left side is the HoopSnake component that triggers the agent by every iteration. The

agent then sends an action to the Grasshopper environment. In this case the environment is a black box that interprets the action and gives the agent several information back. These information from the environment are used to send a new action. An installation guide for the interface can be found on github [8].

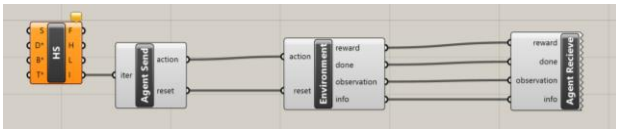

Fig.2: Grasshopper Canvas with main interface

III. Case Studies

To test the general interface and its use to solve architectural combinatorial problems we took a closer look on three problems. The first problem, the simple stacking of modules to reach a goal point, was used to set up the interface in order to then adapt it to the more complex problem of rebuilding structures out of a sequence of SL-Blocks and the reassembly of predefined parts to rebuild certain shapes.

A. Modular Stacking

The modules we used for the first test are based on existing modules that are used at the Digital Design Unit at the faculty of architecture at TU Darmstadt to build several aggregations with robots. [9] They are also the building elements of the IP-Project of Timm Schneider and Jan Schneider on the topic of "Architectural Assembly With Tactile Skills: Simulation and Optimization" that is part of the research project on Tactile Robotic Assembly. These modules have spikes on each side that enable them to interlock with each other but on the other hand enable a free movement in one direction.

In a first approach we used the plugin Wasp for Grasshopper to aggregate modules [10]. Wasp uses predefined connection points to aggregate modules. It can

either aggregate modules in a stochastic way by using a defined set of aggregation rules or with a field that provides a proximity to place modules at a certain position [11]. As both of these options don't provide a discrete way to aggregate modules we used the add-part component of wasp in combination with HoopSnake to build a discrete aggregation

in which we can define the position of every module in every step. With this method we were able to let the algorithm build aggregations. As an action we used three numbers that define the id of the parent part (PID), the connection point of the parent part (CID) and the connection point of the new part (NEXT). Within the Grasshopper environment we gave a reward if the center point of the new placed module comes closer to a given goal point and gave a negative reward if a collision was detected or the wrong PID was selected. The observation was thereby the recording of actions taken and was reset every time an episode was finished.

Wasp has a built in function that detects collisions between parts and automatically flips parts if certain connection points are selected. This is useful, because a separate collision control is not needed, but it resulted in slow computing times for every step.

Therefore we replaced the Wasp components by simple transformations components that used the information of the connection points, provided by the action, to place the new module at the right position. As the collision control of Wasp was missing we needed to do a separate collision control, and as Grasshopper is a geometry based program we needed to check for the physical collision of meshes, that is a slow process, but overall it was faster than the aggregation through Wasp.

To keep the action space for the CID and NEXT small and therefore the total amount of possible connections low we only used 14 connection points per module. There are 8 connection points on the upper side of the module and 6 on the lower side as seen in Fig. 4. In total there are 196 possible connections, from this 196 connections only 96 are feasible connections as only the lower 6 connection points of the parent part can connect to the upper 8 connection points of the new part as well as only the upper 8 connection points of

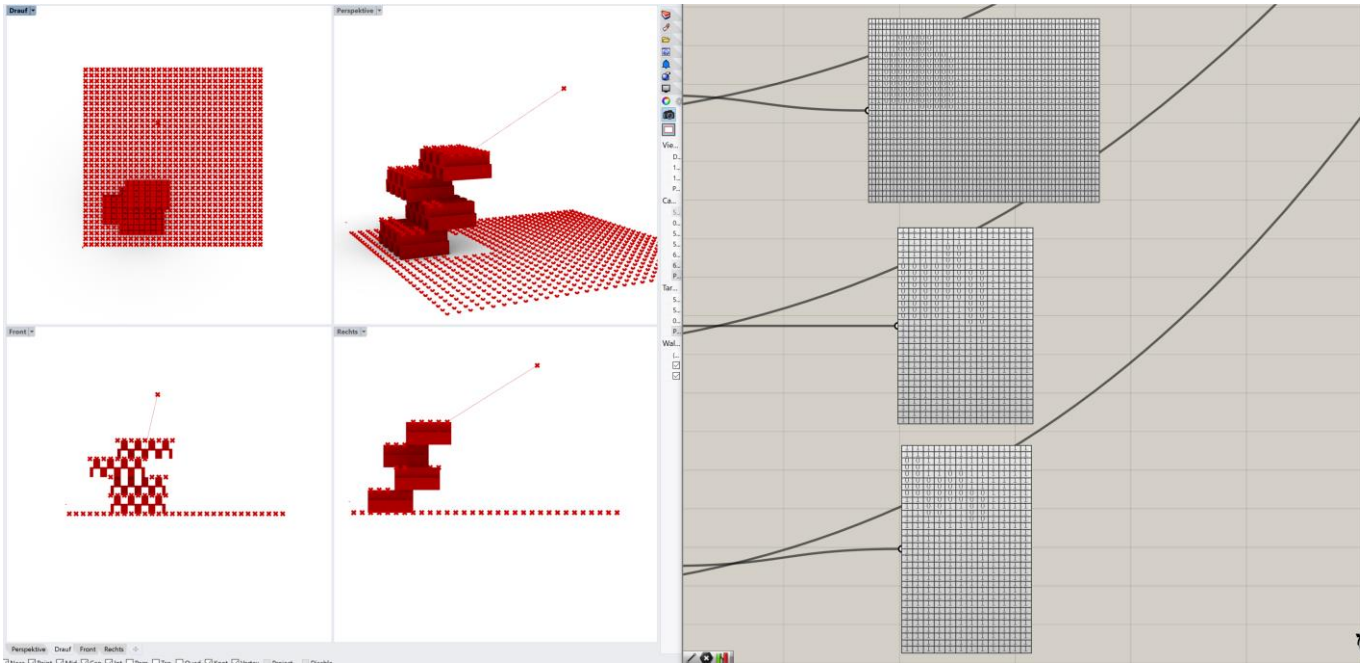

Fig. 3: Visualization of the aggregation in rhino and the corresponding observation matrix

the parent part can connect to the lower 6 connection points of the new part without causing a collision. This leads to a success rate of about 49%. If we now take into account that also the right index of the parent part needs to be picked, the success rate for every step drops significantly. Under the assumption that the action space for the PID is 4 the success rate for placing the second module without a collision drops to about 12%. For the next modules the success rate is even lower, because already placed modules decrease the amount of feasible connections to take for the next module.

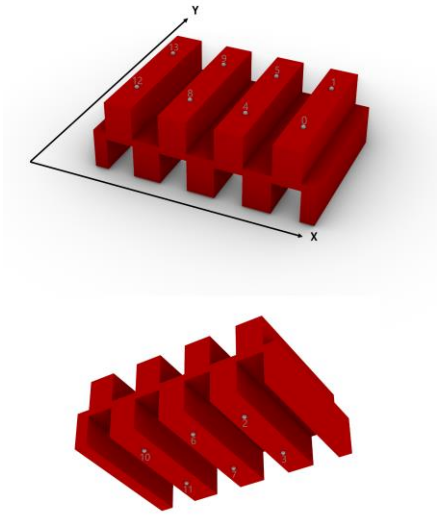

Fig. 4: Module with connection points and world axis

This problem was also observable in the learning process, as the algorithm took a lot of infeasible actions and therefore did not find enough feasible actions giving him a positive reward. The discrete action space, which made it difficult for the algorithm to place modules in a targeted manner to gradually approach a higher reward, increased the problem of a weak learning curve.

Therefore we decided to use a continuous action space to generate more feasible actions. The action from the algorithm is now a xy-coordinate that gets mapped to a predefined grid. The mapping is necessary to guarantee that the spike of the modules sit on top of each other or interlock in each other. The continuous movement along the y axis is still provided and not limited. This new action space allows the algorithm to place modules everywhere in the observable space. If a module is already placed at a position where the next module should be placed, the new module gets stacked on top of the previous placed module. In comparison to the discrete approach this enables a success rate for placing a module of 100%. This means that every action is possible and only the reward decides how good the action was. To test the functionality of this approach we implemented a simple reward function that would give a linear increasing reward, the closer the volumetric center point of the placed module comes to a certain goal point and only give a negative reward if the module is placed partially outside the observable space. The observation also shifted from a simple recording of the actions to an matrix-based observation that would measure the distance to the placed modules from three sides. This guarantees a precise and unique observation of the placed modules at every step. Fig. 3 shows this observation representation in comparison to the build structure in rhino.

First test with this new approach showed better results than with the discrete approach. As the time run short on testing this approach we can not give a well-founded conclusion to this approach, but this first test showed that the reward calculation is not optimal as the learning curve was weak.

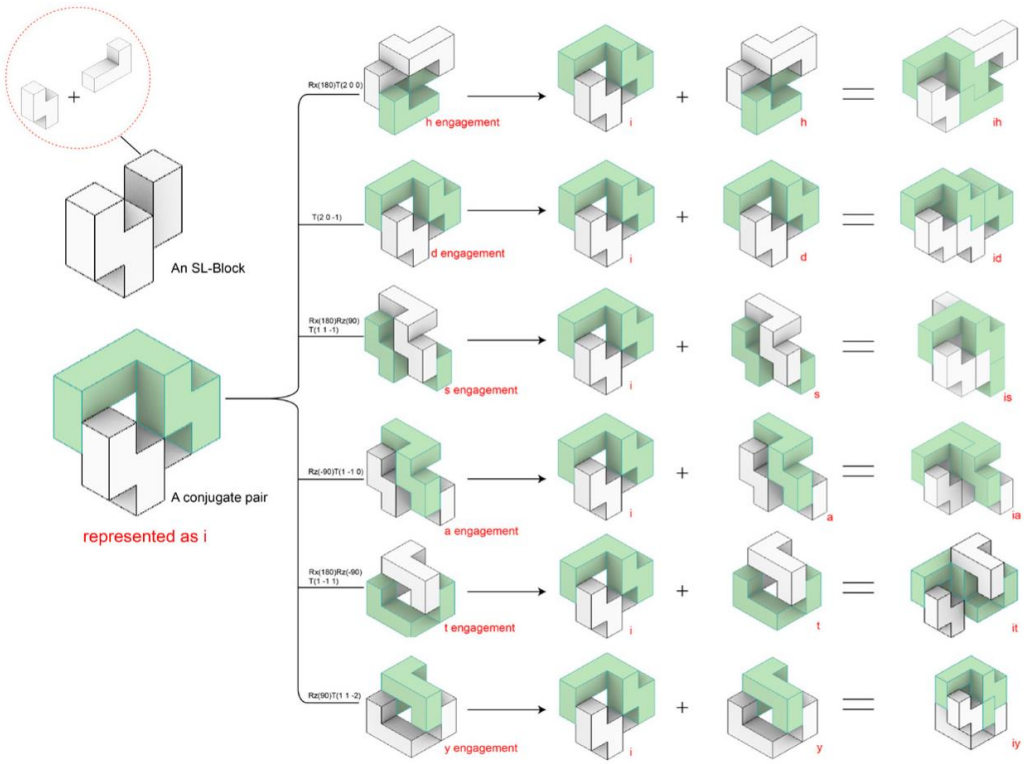

Fig. 5: A SL-Block, a conjugate pair and six types of engagements.

B. *SL-Blocks*

An SL-block is an octocube consisting of an S-shaped and an L-shaped tetracube that are connected to each other side by side. Two SL-blocks can be arranged into a conjugate pair, with each block rotated 180 degree to its counterpart. An SL Strand can be assembled by recursive engagements on one or both ends of a conjugate pair with additional pairs. Six types of engagements h, s, t, d, a, y are defined as geometric transformations that transform the host pair to the pair which is added on as seen in Fig. 5.

The aim in this case study is to rebuild given shapes with a sequence of SL-Blocks. As there are many ways to connect just two SL-Blocks the combinations for a sequence of blocks get so high that trying out every possible sequence, to find the best, would take too long. Therefore the algorithm should learn a policy on how to combine SL-Blocks to quickly rebuild a certain shape. To measure whether a shape is rebuild accurately, the goal form gets voxelized as seen in Fig. 6. The agent sends an action to the environment, in this case one number, that represents one of the conjugate pairs. The environment then tries to engage this new conjugate pair to the already existing structure. The engagements are added sequentially forming a string of conjugate pairs. The result of the learning process should be a sequence of SL-Blocks encoded in a string of conjugate pairs, that rebuilds the goal form in the best way.

Fig. 8 shows the use of the general interface to run the RL algorithm PPO in Grasshopper. Starting with one initial Block the learning process gets started. Therefore the action of the agent is translated into a conjugate pair. The conjugate

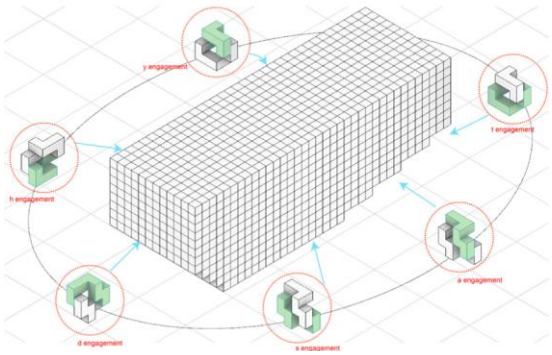

Fig. 6: Voxelized form that should be rebuild by a sequence of six engagements

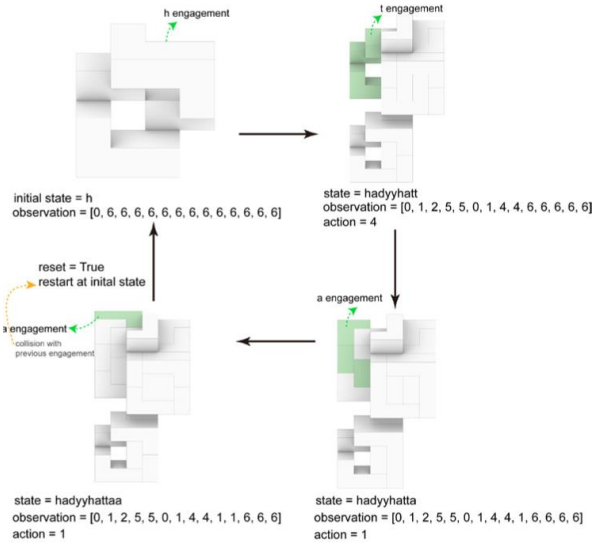

Fig. 7: Different states during training

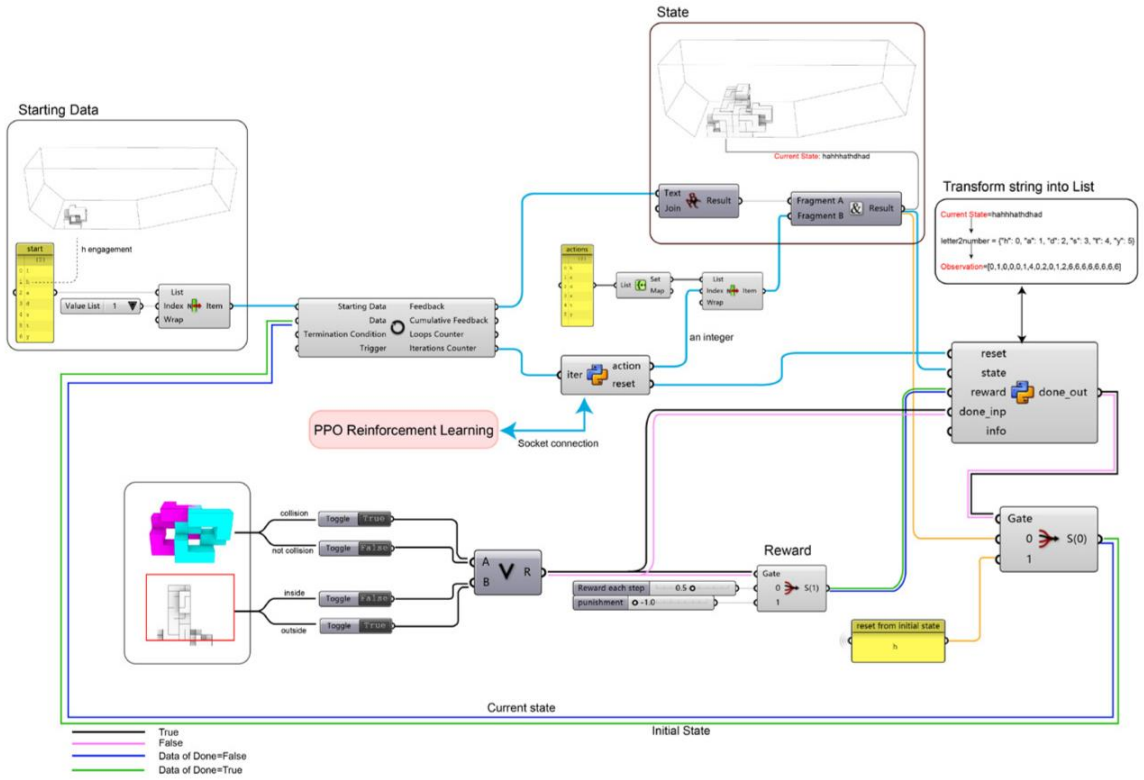

Fig. 8: Process of PPO reinforcement learning algorithm by using Grasshopper components and the socket connection

pair gets added to the aggregation and gets checked for collision with previous placed modules and if it is outside of the goal form. If one of these checks is true the agent receives a negative reward of -1. Otherwise he can add the conjugate pair to the sequence of placed modules and gets a positive reward of 0.5. The observation of the environment is encoded as a list of numbers (Fig. 7), describing the sequence of used conjugate pairs. The iteration of numerus episodes should lead to an optimal sequence to fill the given form.

We used several different RL algorithm from PPO to A2C and DQN that resulted in similar results using the same hyperparameters. The DQN algorithm was able to create longer engagements sequences but was therefore significantly slower. This leads us to the conclusion that the reward calculation is not optimal at the moment and that other parameters effect the results badly. This assumption gets support by the observation that there is not a huge difference between 1000 and 5000 total_timesteps. Although during the training process better results have been observed.

C. Recycling used parts

The third case study takes a deeper look into reassembling recycled building parts. Recycling and reusing building elements to build new buildings is an easy way to save carbon emissions. As the recycled and the new buildings can have different properties and dimensions it is important to re-distribute the building elements in the most efficient way.

In this example linear elements are used to rebuild different given shapes in the best way possible. As these linear elements are not made to handle bending moments compression only structures need to be created. The “Graphic Static” tool is capable of creating compression- and funicular-

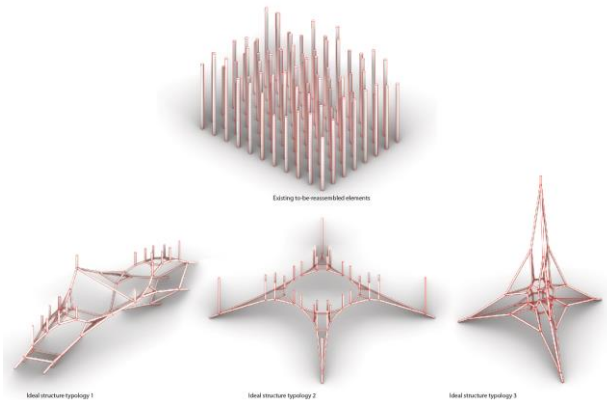

Fig. 9: Compression only structures and stock set of elements

only structures. These structures can be seen in Fig. 9. As the set of recycled elements is not the same as the set of elements used by the Graphic Static tool the recycled elements need to be re-distributed according to the goal structure.

Goal of the optimization is to select the optimal subset of stock elements to rebuild the goal structure as good as possible. Thereby some critical elements need to be as exact as possible. This leads to a prioritizing of elements in the structure. As there is a fixed amount of elements that can only connect at their ends, the action space is discrete. In combination with a high quantity of elements this leads to a high amount of combinations. Standard algorithm are not optimized for solving discrete problems. In order to get a solution to this problem we transfer the problem into an assignment problem. Therefore the elements are defined as agents and the line segments in the structure are the tasks. This enables us to use other techniques to solve this problem.

To evaluate the fitness of the assigned structure, the result of the assignment is quantified through the sum of the distances between each point before and after assignment. Therefore, with minimizing the addition, the optimal solution is inferred (Fig. 10). The Hungarian algorithm can be used as an combinatorial optimization algorithm that can solve this assignment problem.

During the IP-Project we realized that this case study can be solved much easier with other algorithms than with RL algorithms. In the future we would like to try solving this problem also with our general interface.

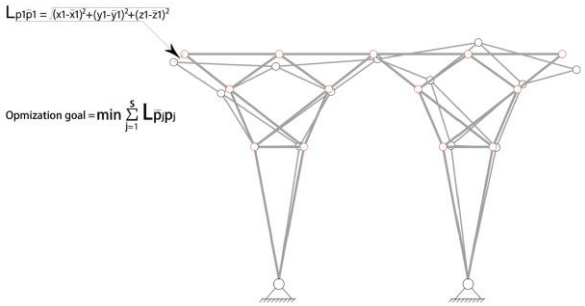

Fig. 10: Fitness evaluation of connection points

IV. Discussion

We were able to create a general interfaces that connects the design interface of Grasshopper with the RL algorithms of stable-baselines3. This is a huge step for bringing the power of RL into the world of architecture. As the agent only sends actions and needs to receive an reward this is a very general approach that allows for a high adaptability to other problems, then the three case studies presented in this report. In the process of setting up the interface a view different research problems came up that at the moment don't seem to be investigated. In the future work it might be an option to take a deeper look into those problems, as they are wildly common in the field of architecture.

A. Action Spaces

One major point is the changing action space. In standard algorithms the action space keeps throughout the hole learning process the same. In architectural combinatorial problems the action space often shifts from step to step as seen in the example of stacking modules. In the discrete approach the action space for the first module was 14 for CID and NEXT. The second module however hat a decreased action space as some connection points were blocked by the module that got placed before. In option to solve this problem would be to get an info about the action space for every iteration, as this is an information that can easily be extracted out of the environment embedded in Grasshopper. The problem hereby is that the algorithm can't handle this information. An implementation of an changing action space would increase the speed of the algorithm as there are less actions to take.

The discrete actions space has also some other disadvantages to the continuous action space. It makes it hard for the algorithm to approach the ideal value. As every action has a discrete reward value that has no direct correlation to other actions.

B. Simulation

With Tim Schneider and Jan Schneider form the IP-Project on "Architectural Assembly With Tactile Skills: Simulation and Optimization" we created an interface to run simulations in pybullet. This gives architects a fast and simple way to simulate discrete aggregations. As the information needed for the simulation are just simple transformations, it provides an easy accessibility for architects. In future projects the simulation could be integrated into the reward calculation for RL tasks. This would be a milestone on transferring the optimization process to real world applications as the simulation could predict the stability of the build systems.

C. Reward Optimisatzion

The next step to improve the results of the algorithms is to improve the reward calculations. At the moment the calculations are often just linear and interfere each other. This leads to non-useful rewards the can easily confuse the algorithms, as they don't give a clear indication on useful actions. With a proper reward adjustment we expect significantly better results.

D. Limitations

The current interface is not a plug and play version, that can be easily adapted to every use case. At the moment the action space needs to be set to the required format. As well as the observation needs to be adjusted to be understand by the algorithm. These adjustments need to be done on the python side of the interface. Architects therefore need to have good knowledge in scripting to adapt and run the algorithms for themselves.

We se a huge benefit in the implementation of RL algorithms into Grasshopper as it expends the designing possibilities to a new level. At the moment the start for non experienced designers is quiet hard, but we aim to overcome these adjusting challenges to provide an interface that can be easily adopted to every combinatorial problem.

References

[1] UN Environment and International Energy Agency, "Towards a zeroemission, efficient, and resilient buildings and construction sector. global status report 2017," https://www.worldgbc.org/, 2017.

[2] Gero, J.S.: Architectural Optimization - A Review. Engineering Optimization, 189-199 (1975)

[3] Ruiz-Montiel, M., Boned, J., Gvilanes, J., Jiménez, E., Mandow, L., Pérez-de-la-Cruz, J.: Design with shape grammars and reinforcement learning. Advanced Engineering Informatics 27, 230-245 (2013)

[4] Rittel, H.W.J., Webber, M.M.: Dilemmas in a general theory of planning. Policy Sci. 4(2), 155–169 (1973)

[5] Hartmann, V., Oguz, O., Driess, D., Toussaint, M., Menges, A. "Robust Task and Motion Planning for Long-Horizon Architectural Construction Planning", unpublished (2020)

[6] Schwinn, T.: 2016, Landesgartenschau Exhibition Hall, in Menges, A., Schwinn, T., Krieg, O. (eds.), Advancing Wood Architecture – A Computational Approach, Routledge, Oxford, pp. 111-124

[7] Available under: https://www.food4rhino.com/app/hoopsnake

[8] Interface available under: https://github.com/b4be1/gh_gym

[9] Wibranek, B., Wietschorke, L., Glaetzer, T., Tessmann, O. "Sequential Modular Assembly: Robotic Assembly of Cantilevering Structures through Differentiated Load Modules" CAADRIA 2020 vol. 2, pp.375-384 (2020)

[10] Wasp available under: https://www.food4rhino.com/app/wasp

[11] Rossi, A., Tessmann, O. "From Voxel to Parts: Hierarchical Discrete Modeling for Design and Assembly" In: Cocchiarella L. (eds) ICCG 2018, pp. 1001-1012 (2018)