Multi-agent Reinforcement Learning for Assembly of a Spanning Structure

Gabriel Vallat

Master thesis

Under the supervision of:
Prof. Maryam Kamgarpour and Prof. Stefana Parascho

20.01.2023

École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

Abstract

In this master thesis, multi-agent reinforcement learning is used to teach robots to build a self-supporting structure connecting two points. To accomplish this task, a physics simulator is first designed using linear programming. Then, the task of building a self-supporting structure is modeled as a Markov game, where the robot arms correspond to the agents of the game. This formalism is then used to design learning agents and train them using deep reinforcement learning. Two different types of deep neural network models, based on image analysis and graph theory, respectively, are used to develop their policy. The agents are then trained either centrally or distributively to compare their learning processes and weaknesses. In a final experiment, the efficiency of the learning algorithm Soft Actor-Critic, is compared to Advantage Actor-Critic, highlighting the effectiveness of using Shannon entropy to search through the policy space. Finally, the training procedure allows agents to successfully build a structure that spans ten times the width of the building blocks without the need to use any binding between them or a removable scaffold during assembly.

Contents

1 Introduction

1.1 Motivation

The idea of using robots in construction can be traced back to the beginning of the 20th century [1]. However, their use in large-scale projects, apart from their novelty, was not very attractive due to the shift from masonry to concrete structures, which can hardly benefit from automation during the construction phase. In the last decade, the environmental challenges caused by the use of concrete (e.g., the impact of sand mining on biodiversity or the high amount of greenhouse gasses generated by clinker production) have increased the trend towards alternative construction methods. Although these issues cannot be overlooked, it would be a mistake to focus only on concrete production: The end-of-life of concrete is as problematic as its production. Due to its almost indestructible resistance, concrete is extremely difficult to recycle, and doing so can almost be reduced to grinding it up and using it as an aggregate for road surfaces (in the best cases).

Masonry was not affected by this last drawback, since the building components can be reused after the destruction of a building. The most extreme example of such reusability is a dry stone wall, where no cement is needed to fix the blocks in place, as gravity and friction alone hold the structure together. This makes such structures completely disassemblable. However, these methods cannot match the incredible scalability of concrete. Concrete indeed allows for reliable construction in a much smaller time frame than masonry. Massive structures like Notre Dame Cathedral in Paris were built with masonry, but their construction took hundreds of years.

However, new developments in robotics and control could provide a solution to speeding up old construction methods: larger blocks could be used because they would not need to be carried by human or animal force. Moreover, the use of optimization methods could enable the construction of more stable and efficient structures.

While most previous work has focused on the fine control of robots to precisely pick and place blocks to obtain a given design [2], this thesis takes an orthogonal approach: the goal is to understand

where and in what order blocks should be placed to obtain a stable structure. To obtain a scalable solution, in this thesis we propose to use multi-agent reinforcement learning (MARL) on simulated robots. In our approach, we consider a discrete 2D environment in which blocks are placed to simplify the development of learning algorithms by not using a continuous 3D environment. Although this is a strong deviation from reality due to the limited number of block orientations and shapes that a discrete environment allows, it is a first step in addressing the difficult problem of learning to build a self-supporting structure. It also allows different learning approaches to be compared, as is done in this thesis.

1.2 Literature Review

In the literature review, we will cover two different topics of related work: The first part deals with self-supporting structures and the second with the application of reinforcement learning (RL) in the field of construction.

1.2.1 Self-Supporting Structures

So-called self-supporting structures are widely used in the domain of architecture. Constructions such as bridges, church roofs, and even most types of camping tents fall under this designation.

From an assembly perspective, a self-supporting structure is an assembly of rigid components that does not require any binder to connect the parts and is held in static equilibrium by friction and gravity-induced compressive forces that immobilize all the parts [3]. These structures tend to be neglected in favor of reinforced concrete in modern times: This last material allows for large constructions to be built in a monolithic way, where each building component is solidly attached to the others. On the other hand, Self-supporting structures have the advantage of being very light and disassemblable and they can therefore still find application, e.g. the Armadillo vault [4], where the fragile floor could not support a heavy structure. Since self-supporting structures are well understood, most modern research relates to shaping the parts to create a structure with the desired shape: either to find straight cuts in an existing structure [5], or to optimize the curvature of the building blocks to the forces [4]. To the best of our knowledge, no simulations of spanning structures have been performed with a definite set of building blocks. On the other hand, this task has been addressed in the context of interlocking assembly by using SL-blocks, a common interlocking shape consisting of 8 cubes (see Figure 1), to build a structure that follows a line [6].

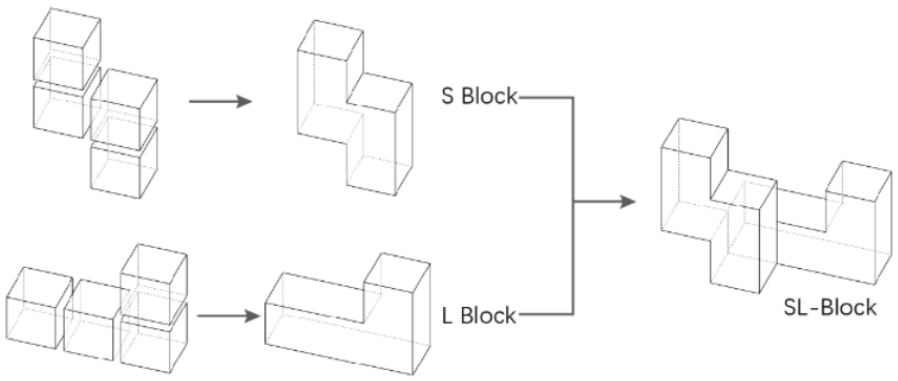


Figure 1: SL-block composition, adapted with permission from [6]

The main difference between self-supporting and interlocking assemblies is that interlocking ones not only resist gravity, but can actually withhold any force, requiring only 2 conditions to hold: The force is not able to break the building blocks apart, and a particular block, called the key, is held in place by other means. This type of structure and its assembly have received much more attention in recent years than self-supporting structures [3, 7], and they have a number of common challenges, such

as the need to choose the position and orientation of the parts, as well as the need for the structure to be physically stable or otherwise externally held during construction.

In this work, we consider a free structure, which is a different problem than assembly. The main difference between these two problems is that in the latter the shape of the finished object or the position of the individual components is already known, whereas in a free structure, fewer specifications are given, like only the starting and end point of the structure.

Despite the attention paid to the interlocking framework, only the paper on SL blocks [6] deals with free structures, while most papers consider the design process as an assembly problem [8].

1.2.2 Reinforcement Learning in Construction

In Bapst et. al. [9], the use of reinforcement learning is tested with a single agent for a set of construction tasks. One of the tasks given to the agent is the placement of blocks to connect some points while avoiding obstacles. In this work, it is found that representing the construction as a graph of interconnected blocks is very beneficial, as is defining the actions in a relative reference rather than an absolute one, i.e., defining actions such as "place block A on block B" rather than "place block A at (x,y)".

More broadly, the block stacking task is a common theme in hierarchical reinforcement learning [10], where the agent learns to perform a simple subtask, such as placing a single block on top of another block. Then, the agent uses a planning algorithm to combine the subtasks into complete sequences. This method of model-based RL only uses learning to fit a model, and then uses a classical tree search to find the optimal way of reaching the target state. The key element of this method is the assumption that stacking one block on top of another is the same task whether the first block is on the ground or already at the top of a tower. While this is true in the case of a vertical tower where the physical stability conditions are almost the same for each block, this assumption fails our case. Moreover, even in the experiment done in [10], where only up to four blocks are placed, some problems due to a mismatch between reality and the human-made state reduction leads to some failures.

RL methods also find some applications in real buildings, especially in assembly tasks to exploit feedback from rich tactile sensors [11, 12]. In both of these works, the focus is on the precise placement of each part given a target pose, which is a complementary approach to the objective of this thesis, where the goal is to find the target pose of the blocks.

1.3 Contribution

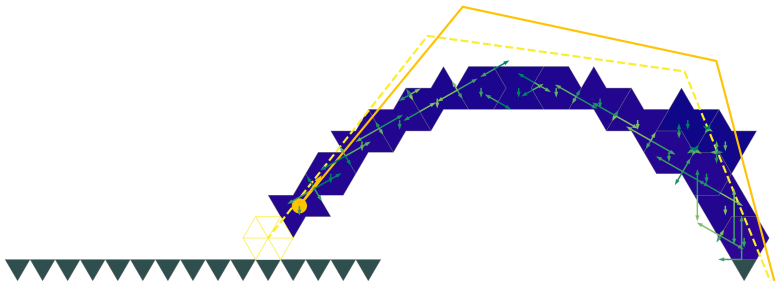


Figure 2: Last step in the construction of a wide structure: The aim of the agents controlling the robot arms is to connect both grounds (in grey) by using blocks (in blue) of different shapes

The main contributions of this thesis are:

- The design of a discrete simulator using linear programming. This simulator is able to efficiently check the stability of a structure and directly determine the forces that the robots must apply to maintain it.
- The formalisation of the construction process as a Markov game with a state-dependent action set.

- The use of two deep policy models, one based on convolutional neural network and the other on graph neural network.
- The implementation of a state-of-the-art learning algorithm, called Soft Actor-Critic, to train the agents, and its benchmarking against an older algorithm, called Advantage Actor-Critic.
- The training of the agents, either in a centralised way, where they share the same internal model, or in a distributed way, where each agent is completely independent.
- The design of macro-actions to help the learning process by directly enforcing the cooperation of the agents.
- The successful construction of a structure that spans a large distance, a snapshot of which is shown in Figure 2.
- The construction of prototype blocks, similar to those used in the simulator, to verify that the structures created by the agents can be replicated in real life.

The different design choices that lead to successful training of the agents are also motivated by three different experiments that compare the different models and learning approaches.

2 Setup

2.1 Discrete Setup

The main environment used during the experiments is a 2D isometric grid (see Figure 3), in which the blocks visible in Figure 4 are placed. Compared to a classical square grid, this type of grid allows more surface orientations: horizontal, 60 and 120 degrees, as opposed to horizontal or vertical for a square grid. Although square blocks are more commonly used than triangular (or hexagonal) ones in practice, it is not possible to build an arch-shaped structure: Non-parallel sides are required to dissipate the vertical force horizontally. A continuous setup is also defined in the appendix A.3, but was not used because of the added complexity of training, the computational cost of detecting collisions between blocks, and the additional assumption that are needed in a hyper-static structure[1].

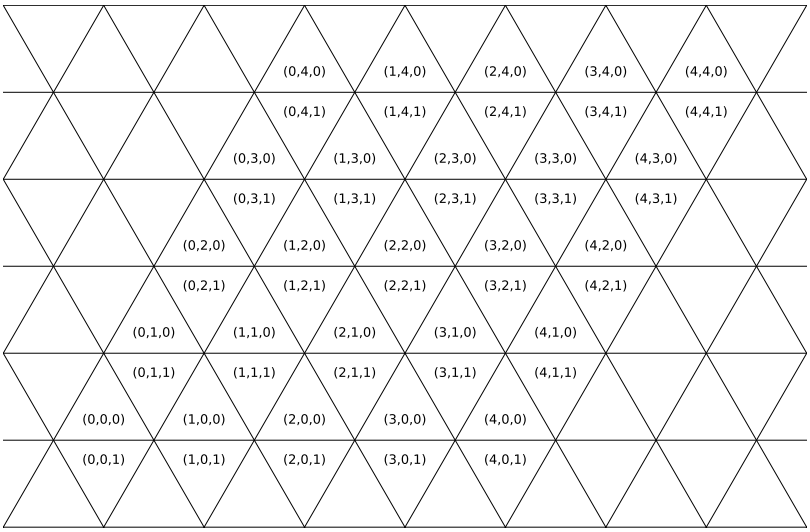


Figure 3: Isometric grid: the third coordinate describe whether the triangle points up or down

Although this environment could use any type of blocks made out of triangles, only two types were used in the experiments: Hexagons and links (see figure 4).

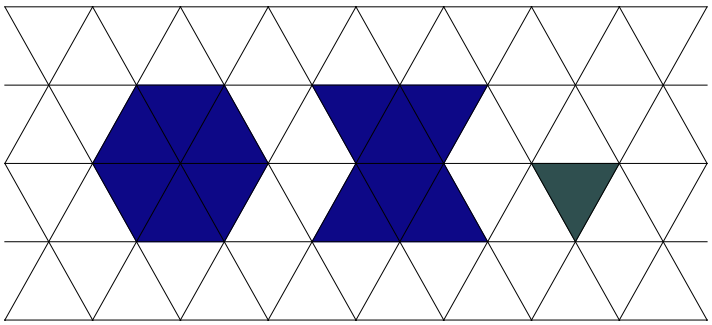
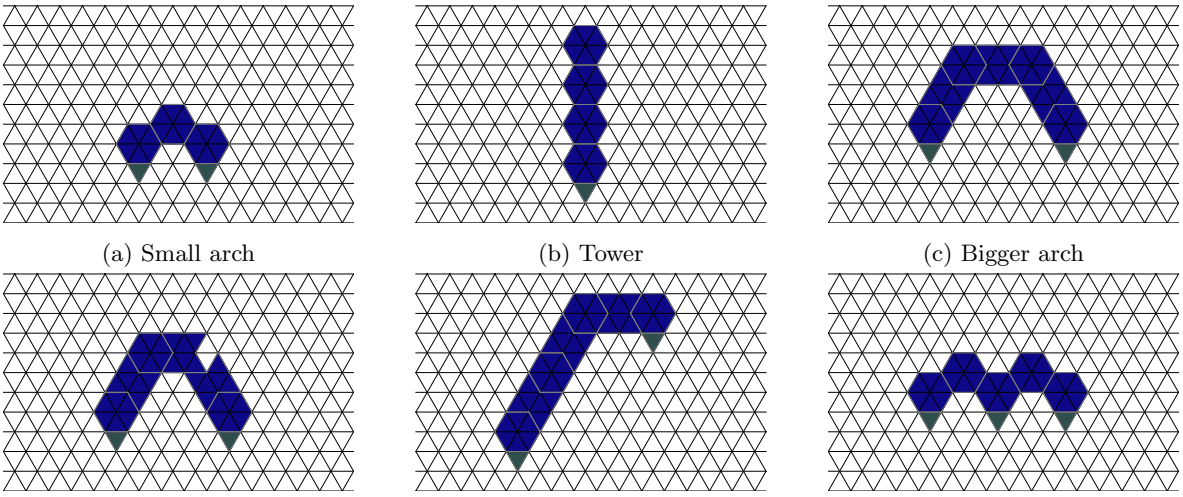


Figure 4: From left to right: hexagon, link and ground

The hexagon is chosen to allow the construction of basic structures (see Figure 5a, 5b), and the shape of the link was motivated by the wide variety of structures that can be built without relying on friction of the blocks (see Figure 5c): The use of a non-convex polygon called links, such as the middle one in Figure 4, allows to match them with two sides of the hexagon, and the resulting contact force can be oriented arbitrarily within a range of 120 degrees. In addition, these links can create

[1]The discussion about what is a hyper-static structure is out of the scope of this thesis, but they represent structures where the position is over-constrained whereas the forces applied are under-constrained.



(a) Small arch (b) Tower (c) Bigger arch

(d) Dislocation creation (e) Different levels (f) Double arch

Figure 5: Different types of structures. Each block is drawn in blue and is subject to gravity, while the ground is drawn in gray and can absorb any force applied to it

dislocations in the network of hexagons necessary to achieve specific goals (see Figure 5d). Depending on the task, either a single triangle or multiple aligned triangles are chosen to represent the ground. Using a single triangle (as in Figure 5) reduces the number of possible actions at the beginning of the construction, while using multiple triangles allows for small adjustments in the placement of blocks while eliminating the need to create defects.

In the experiments in Section 5, we use two different targets that must be connected. However, the environment allows the use of more targets (as in the structure 5f) and can also be modified to deal with obstacles.

The robotic arms are sometimes represented, as on Figure 6, but they were generally not depicted in the graphical interface to lighten the figures, and only their respective intentions and effects on the blocks were represented by their specific colors, as shown in Figure 7. The color of a particular robot was then used for all of its actions:

- When a robot hold statically a block in a structure, a dot is drawn in the robot's color, as shown in Figure 6b
- When a robot intends to place a block, its skeleton is drawn in the robot's color at the wanted position, as in Figure 6c

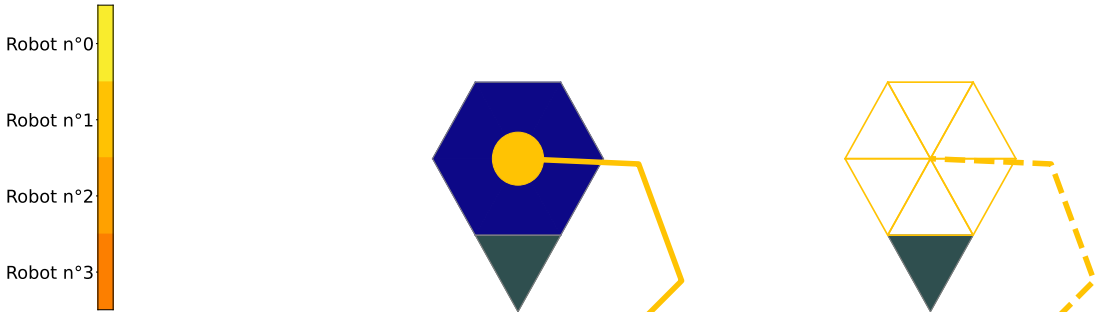
A small scenario where two robots are holding a structure and want to place a new block can be seen in 9. Any number $N > 1$ of robots could be used to build a self-supporting structure, but for simplicity and because this is the minimum number of robots needed to build an arch without removable supports, we consider two robots during the experiments in Section 5.

The color of the blocks depends on the strength of the forces acting on them. It ranges from dark blue when no forces are acting to yellow when a robot would not be strong enough to counteract the maximum force acting on the block, as shown in Figure 8. To simplify the figures, the grid is generally not shown in the following sections of this paper.

2.2 Physics

The complete simulator has to handle two tasks: the collision detection and the static equilibrium calculation. The task of collision detection is to make sure that no blocks (or grounds) are intersecting one another, and can be trivially solved using the grid. On the other hand, the static equilibrium computation is more subtle, so this section of the thesis describe precisely how it is computed.

A structure is in static equilibrium when it is not accelerating and is in a state of balance. According to Newton's first law, this is the case when the net forces and moments are zero.



(a) Color map used when up to $N = 4$ robots are used

(b) Robot 1 is holding this block

(c) Robot 1 tries to place this block there

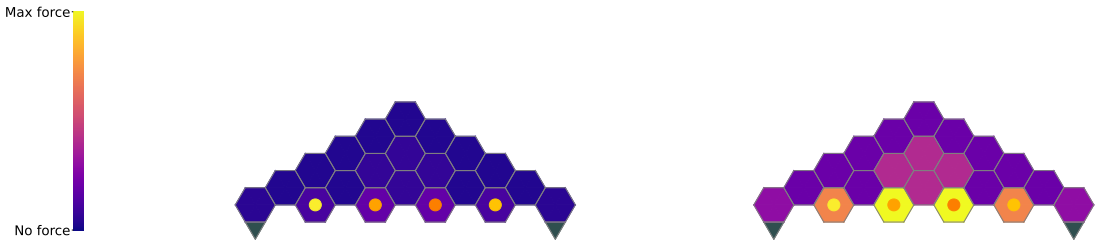
Figure 6: Graphical depiction of the robots' actions

(a) Robot 1 is holding this block

(b) Robot 1 tries to place this block there

Figure 7: Graphical depiction of the robots' effect and intent

The simulator only considers the statics of the structure, and although most physics engines (such as PhysX or pybullet) and optimization libraries (Gekko, Gurobi, ...) are able to handle this easily, these programs are designed for more complex tasks, such as multi-body dynamics, and the time required to send messages describing the structure to the external solver often exceeds the time required to actually calculate the stability of the structure. Therefore, it was decided to calculate the physical equilibrium directly using linear programming.

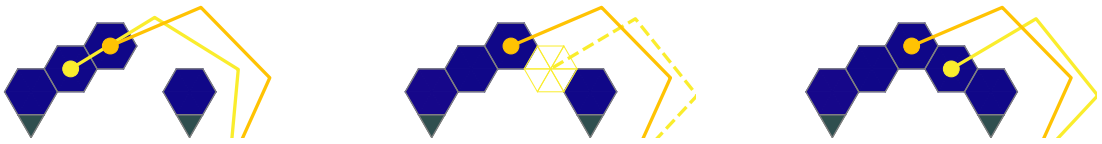


(a) Force color map

(b) Example with strong robots

(c) Example with weak robots

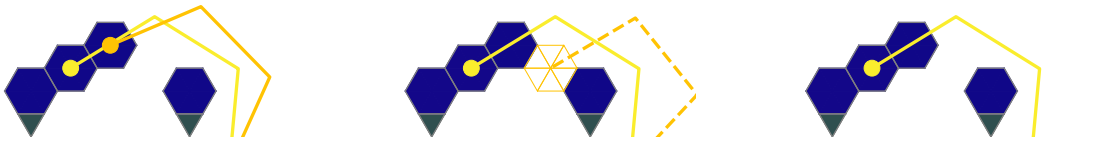
Figure 8: In 8b, each block is colored according to the force applied to it, but since the weight of the blocks is negligible compared to the force of the robot, the color change is hardly noticeable. For 8c on the other hand, the weight of the blocks is close to the maximum force of the robot and the colors of the blocks span the entire color map



(a) Initial situation

(b) The robot 0 wants to place a new block

(c) Result



(d) Initial situation

(e) The robot 1 wants to place a new block

(f) The action cannot be performed

Figure 9: Sequences that occur in the simulator during a typical scenario. On the first line, the robot 0 acts, resulting in a successful action. On the second line, the robot 1 acts, which leads to a failure because it cannot let go of the block it was holding

2.2.1 Static equilibrium

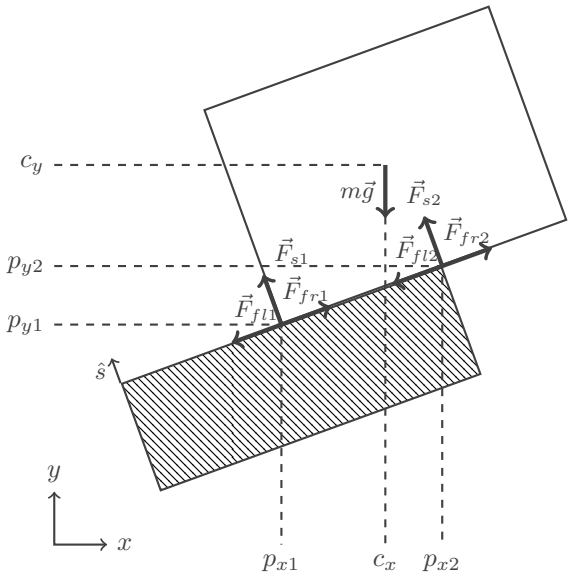


Figure 10: A single block on the ground: in this simplest case, 6 distinct forces are needed to compensate the weight of the block

As mentioned earlier, the static equilibrium conditions require that the sums of the forces and moments acting on each block equal 0. When the block has only one interface oriented with a direction $\hat{s}$, as in the Figure 10, the equilibrium can be written as $A_{eq}^{(s)}\mathbf{x}^{(s)} = \mathbf{b}_{eq}^{(s)}$, where $\mathbf{x}$ is the amplitude of all the reaction forces, and where the first two rows of $A_{eq}^{(s)}$ are respectively tasked to calculate the sum of the reaction forces along the x- and y-directions, and the third (and last) row is tasked to calculate the sum of the moments. This last step justifies the choice of two separate sets of forces applied at each end of the interface: by combining them linearly, the resulting force can be applied at any point along the interface without the need for a nonlinear operation at any time. Finally, the vector $\mathbf{b}_{eq}^{block}$ is simply composed of the weight of the block and its induced moment:

$$A_{eq}^{(s)\mathrm{T}} = \begin{pmatrix} s_x & s_y & p_{y1}s_{xi} + p_{x1}s_{yi} \\ -s_y & s_x & p_{y1}s_{yi} + p_{x1}s_{xi} \\ s_y & -s_x & -p_{y1}s_{yi} - p_{x1}s_{xi} \\ s_x & s_y & p_{y2}s_{xi} + p_{x2}s_{yi} \\ -s_y & s_x & p_{y2}s_{yi} + p_{x2}s_{xi} \\ s_y & -s_x & -p_{y2}s_{yi} - p_{x2}s_{xi} \end{pmatrix} \mathbf{x}^{(s)} = \begin{pmatrix} F_{s1} \\ F_{fl1} \\ F_{fr1} \\ F_{s2} \\ F_{fl2} \\ F_{fr2} \end{pmatrix} \qquad \mathbf{b}_{eq}^{block} = \begin{pmatrix} 0 \\ mg \\ c_x mg \end{pmatrix},$$

where the component of the matrix A_{eq}^{s} are the x and y component of the vector normal to the surface $\hat{s}$ and the coordinate of the application point of each force (as seen on Figure 10).

2.2.2 Compression constraints

An important factor in our setup is that no mortar is used between the blocks, requiring that all of the contact forces are compressive. Doing so creates a non-negativity constraints on the support forces:

$$F_{s1} \geq 0,\ F_{s2} \geq 0$$

To simplify the calculation, the frictional forces were also constrained to be non-negative, but it can be seen that the combination of the two forces acting in opposite directions already allows them to bypass this constraint. One can then use the formulation

$$\mathbf{x}^{(s)} \geq \mathbf{0}$$

2.2.3 Friction constraints

In solid physics, the maximum friction force that can be applied is equal to a coefficient μ_c times the support force (in the static case), i.e $F_{fr1} + F_{fl1} \leq \mu_c F_{s1}$ and $F_{fr2} + F_{fl2} \leq \mu_c F_{s2}$. This constraint can then be written in matrix form, $A^{inter}\mathbf{x} \leq 0$, by using

$$A^{(s)} = \begin{pmatrix} -\mu_c & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -\mu_c & 1 & 1 \end{pmatrix}.$$

This coefficient depends on the material of both the block and the ground, as well as other factors such as the roughness of the surface. Its value in real building is generally around 0.7, but with the prototype blocks used in the appendix A.1 to validate the model, a value of 0.5 better matched the reality.

2.2.4 Combination of several interfaces

In a real construction, each block is generally not in contact with only one other, but with several (as in Figure 11)

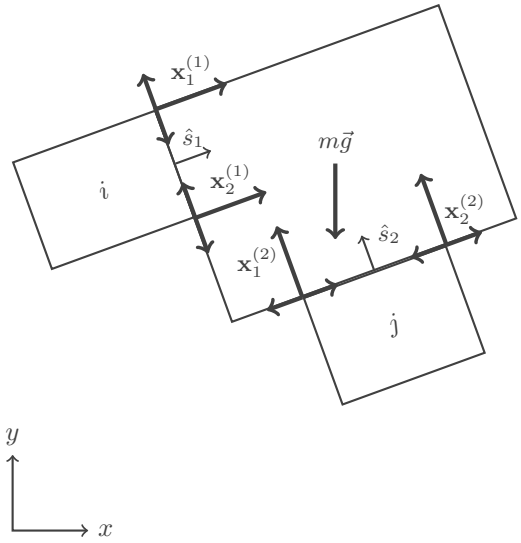


Figure 11: A block added to an existing structure. It shares its interface (1) with the block i and its interface (2) with the block j

In this case, the matrices defined above can simply be stacked together, creating the system of equations and inequation

$$\begin{aligned} A_{eq}^{block}\mathbf{x}^{block} &= \mathbf{b}^{block} \\ A^{block}\mathbf{x}^{block} &\leq \mathbf{0} \\ \mathbf{x}^{block} &\geq \mathbf{0} \end{aligned}$$

In this case, the matrices defined above can simply be stacked together, creating the system of equations and inequation x=y. If the block has n_s different interfaces, the composition of each matrix and vector

is:

$$A_{eq}^{block} = \begin{pmatrix} A_{eq}^{(s_1)} & A_{eq}^{(s_2)} & ... & A_{eq}^{(s_{n_s})} \end{pmatrix}$$
$$A^{block} = \begin{pmatrix} A^{(s_1)} & \mathbb{0} & \cdots & \mathbb{0} \\ \mathbb{0} & A^{(s_2)} & \cdots & \mathbb{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbb{0} & \mathbb{0} & \cdots & F^{(s_{n_s})} \end{pmatrix}$$
$$\mathbf{x}^{block} = \begin{pmatrix} \mathbf{x}^{(s_1)} \\ \mathbf{x}^{(s_2)} \\ \vdots \\ \mathbf{x}^{(s_{n_s})} \end{pmatrix},$$

where $\mathbb{0}$ is filler matrix composed exclusively of zeros.

2.2.5 Combination of several blocks

These constraints do not need to be satisfied only for a block, but for each block present in the structure. The complete constraint system of a structure can then be built iteratively by adding a block at each time step:

$$A_{eq}^{(t)} = \begin{pmatrix} A_{eq}^{(t-1)} & A_{eq}^{col_t} \\ \mathbb{0} & A_{eq}^{block_t} \end{pmatrix}$$
$$A^{(t)} = \begin{pmatrix} A^{(t-1)} & \mathbb{0} \\ \mathbb{0} & A^{block_t} \end{pmatrix}$$
$$\mathbf{x}^{(t)} = \begin{pmatrix} \mathbf{x}^{(t-1)} \\ \mathbf{x}^{block_t} \end{pmatrix}$$
$$\mathbf{b}_{eq}^{(t)} = \begin{pmatrix} \mathbf{b}_{eq}^{(t-1)} \\ \mathbf{b}_{eq}^{block_t} \end{pmatrix}$$
$$\mathbf{b}^{(t)} = \begin{pmatrix} \mathbf{b}^{(t-1)} \\ \mathbf{0} \end{pmatrix},$$

where $A_{eq}^{col_t}$ is taken care of the reactive forces according to Newton's third law by placing the opposite of the matrix $A_{eq}^{(s_i)}$ in the rows corresponding to the support block:

$$A_{eq\ jx,jy,jm}^{col_t} = \begin{cases} \mathbb{0} & \text{if the block j does not share an interface with the new block} \\ -A_{eq}^{(s_i)} & \text{if the block j shares the interface i with the new block} \end{cases},$$

where the indices jx, jy, jm represent the three lines of the matrix $A_{eq}^{col_t}$ that compute the equilibrium of the block j.

2.2.6 Robot forces

The problem must also deal with a robot holding a block. This is done by using the initial matrices and vector $A^{(0)}$, $\mathbf{b}^{(0)}$,$\mathbf{x}^{(0)}$, (while $A_{eq}^{(0)}$ and $\mathbf{b}_{eq}^{0}$ are initialised empty). The vector

$$\mathbf{x}^{(0)} = \begin{pmatrix} \mathbf{F}_0 \\ \vdots \\ \mathbf{F}_{N-1} \end{pmatrix}$$

contains the forces applied by each robot (numbered from 0 to $N-1$). Each F_n, $n = 0, \ldots, N-1$, contains 6 components

$$\mathbf{F}_n = \begin{pmatrix} F_{x+} \\ F_{x-} \\ F_{y+} \\ F_{y-} \\ M_+ \\ M_- \end{pmatrix},$$

each representing the force (or moment) acting in a particular direction. If the robot n holds the block i at a certain time during the simulation, A_{eq} is changed in the following way:

$$A_{eq\ (ix,iy,im)\times\mathbf{n}} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ -p_{ry} & p_{ry} & p_{rx} & -p_{rx} & 1 & -1 \end{pmatrix},$$

where the indices $(ix, iy, im) \times \mathbf{n}$ represent the three lines corresponding to the block i and the six columns corresponding to the robot n.

2.2.7 LP formulation of the physics module

A linear program (LP) has a linear objective and several linear equality and inequality constraints. In matrix form, the problem can be formulated as follows:

$$\begin{aligned} &\min_{\mathbf{x}}(\mathbf{c} \cdot \mathbf{x}) \\ &s.t.\ A_{eq}\mathbf{x} = \mathbf{b}_{eq}, \\ &\qquad A\mathbf{x} \leq \mathbf{b}, \\ &\qquad \mathbf{x} \geq \mathbf{0}, \end{aligned}$$

As can be seen, the constraints defined above can be used directly in the LP, and only the cost vector $\mathbf{c}$ remains to be defined. As no preference is given to any of the forces, all forces were then given a constant weight of 1.

2.2.8 Soft constraints

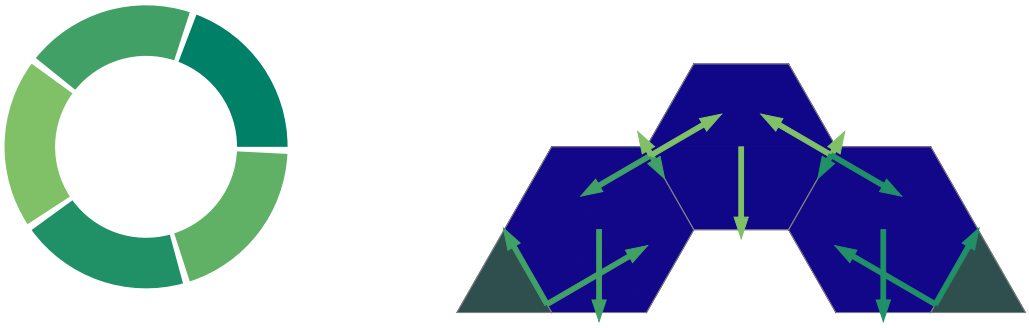
In the next section, the forces applied to each blocks are added to the structures. Another helpful aid that can be added is the failure points of the structures, allowing to know whether a collapse originates from a robot that is not able to sustain the force, a friction that is too large or an out of balance block. To find and indicate these failure points, the LP is extended with unbounded sticking forces that would act on the mortar or binding if some was used:

$$
\begin{aligned}
A^{(t)}_{eqsoft} &= \begin{pmatrix} A^{(t)}_{eq} & -A^{(t)}_{eq} \end{pmatrix} \\
A^{(t)}_{soft} &= \begin{pmatrix} A^{(t)} & \mathbb{0} \end{pmatrix} \\
\mathbf{x}^{(t)}_{soft} &= \begin{pmatrix} \mathbf{x}^{(t)} \\ \mathbf{x}_{stick} \end{pmatrix} \\
\mathbf{b}^{(t)}_{eqsoft} &= \mathbf{b}^{(t)}_{eq} \\
\mathbf{b}^{(t)}_{soft} &= \mathbf{b}^{(t)} \\
\mathbf{c}^{(t)}_{soft} &= \begin{pmatrix} \mathbf{c}^{(t)} \\ \beta \mathbf{c}^{(t)} \end{pmatrix}
\end{aligned}
$$

where β is an arbitrarily large constant (fixed at 10^6 in the experiments). This softening of the equality constraints allows a solution to always be found, and the nature of the linear cost tends to minimize the number of soft constraints violated rather than the magnitude of those violations, leading to fewer failure points and an easier-to-understand failure mode: The collapse of a structure is often caused by a single frictional force that is too large or could be fixed by using mortar in only a few locations. The result of using such soft constraints can be seen on the second line of Figure 13, where the red arrows show the failures points.

2.2.9 Visualisation

To illustrate the resulting forces when used in the setup, the forces acting on the corners of the same interface are added together, and only the resulting force is then displayed. The application point of the resulting forces is computed so the moment it creates equals the one created by all corner forces[2].



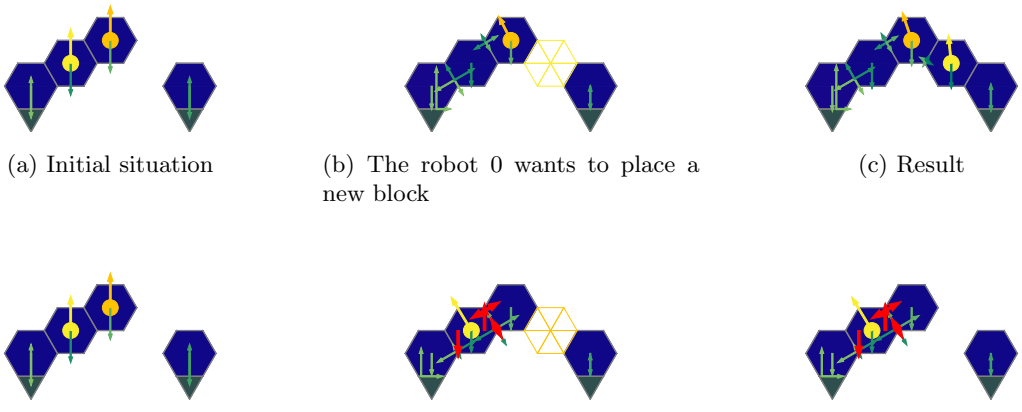
(a) Colors used sequentially to draw the arrows

(b) Example: without using different colors, it would not be possible to determine whether the friction forces applied on the top block are pointing upward or downward

Figure 12: Colors used to draw the arrows when no robot is holding the structure

The color rule used to visualize forces is shown in Figure 12, except for forces exerted by any robot. Whenever the arrows were drawn, the force exerted by a robot was represented in its corresponding color. With this notation, the first scenario in Figure 9 looks like the first scenario in Figure 13.

Thick red arrows are used to highlight failure points when soft constraints are not met, as shown at the end of the second scenario in Figure 13



(a) Initial situation

(b) The robot 0 wants to place a new block

(c) Result

(d) Initial situation

(e) The robot 1 wants to place a new block

(f) The action cannot be performed

Figure 13: Same sequences as in figure 9, with the force arrows replacing the colored grid when a robot is holding a block and a red arrows at the failure points of the structure: In the second scenario, the robot would need to apply a torque and the third block would need to be stuck to the second one.

[2]This position can easily be computed by taking an amplitude-weighted average of the application points of each force

To increase visibility when a larger number of block are used, the amplitude of each force is normalized with respect to the largest one. Note that this information is still qualitatively visible by the color of the blocks, as in Figure 14.

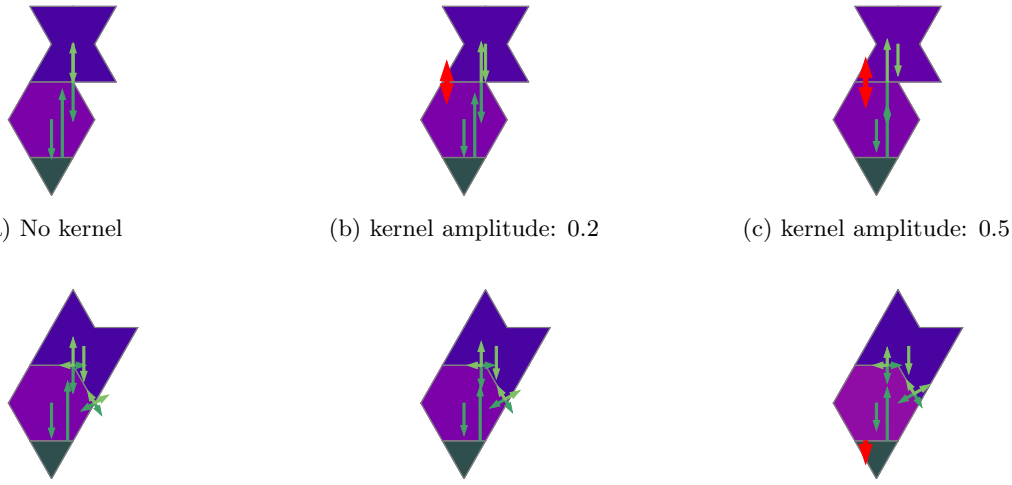
(a) When only three blocks need to be held, the force received by each is quite small compared to the maximum force of the robot, so the color remains uniform.

(b) While the actual force is greater, the size of the oblique arrows does not change when more blocks are used. However, the color of the blocks is modified to indicate the increase in force, and the weight are relatively smaller

Figure 14: To hold the structure together, the robot needs to apply a force whose amplitude and direction changes with the number of blocks.

2.2.10 Safety kernel

To give the physics solver a sense of stability, the notion of a safety kernel is introduced. The concept of safety kernel is to modify directly the application points of the forces (p_x and p_y on Figure 10), to create a safe region in which the forces can be applied. This method allows to eliminate some unstable equilibria that occur when forces are applied too close to the corners of the block (see Figure 15a: in practice, the stability of this structure is unreliable, as a slight shift of the top block to the left would be enough to make it collapse). Instead, the point of application of each force is shifted toward the center of the side by a fraction of its length. This simple trick eliminates most unrealistic equilibria, and the resulting structures could always be tested with real blocks (see the appendix A.1). Note that the implementation of this safety does not requires to modify the equation used above, but rather modify the geometry of the blocks.



(a) No kernel (b) kernel amplitude: 0.2 (c) kernel amplitude: 0.5

(d) No kernel (e) kernel amplitude: 0.2 (f) kernel amplitude: 0.5

Figure 15: On the top line, a small deviation from the position would result in a collapse of the structure, and the introduction of a kernel makes this structure always unstable, leading to safer and more realistic constructions. The structure on the bottom line was consistently stable in the prototype, justifying a kernel size of 0.2.

3 Agent

Several modeling methods and frameworks could be used to model multi-agent tasks such as the self-supporting construction. We chose to use a Markov Decision process (MDP) as a framework, as this method is extremely flexible and commonly used in reinforcement learning. The following section formally defines a Markov process, a Markov decision process and a Markov game, while the one thereafter describes how these concepts were adapted to our setup.

3.1 Framework

3.1.1 Markov Chain

A Markov chain or Markov process is a common way to describe a stochastic process. It consists of a tuple $(\mathcal{S}, P)$ that models a set of states $\mathcal{S}$ and their evolution through time. The evolution through time is modeled by a transition function $P : \mathcal{S} \to \Delta\mathcal{S}$, where $\Delta\mathcal{S}$ represents the probability simplex over $\mathcal{S}$ and the next state depends only on the current state. The state space can be either finite or infinite, and the transitions can be either continuous or discrete, but in this master thesis, only the finite-state discrete time Markov process is discussed. In this context, the transition function P can be viewed as a matrix whose components p_{ij} are:

$$p_{ij} = P(s_{t+1} = i | s_t = j)$$

A Markov process can also be described as a graph, where the nodes represent the states and the edges represent all nonzero transition probabilities. For example, the transition matrix

$$P = \begin{pmatrix} 0.5 & 0 & 0.1 \\ 0.5 & 0.5 & 0.1 \\ 0 & 0.5 & 0.8 \end{pmatrix}$$

can be equivalently modeled as the graph in Figure 16

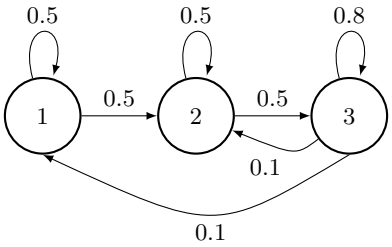


Figure 16: Example of Markov process

3.1.2 General formulation of Markov decision process

A Markov decision process (MDP) is an extension of the Markov process that includes an agent that can influence the transition function: To do so, three elements are added to the Markov chain tuple: a set of actions $\mathcal{A}$ from which an agent can choose, a reward function $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ and a discount factor $\gamma \in [0, 1]$. The transition function must also be modified to take an action as input: $P : \mathcal{S} \times \mathcal{A} \to \Delta\mathcal{S}$, and the MDP tuple can finally be written as $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$

The agent's goal is then to find a policy $\pi : \mathcal{S} \to \Delta\mathcal{A}$, where $\Delta\mathcal{A}$ represents the probability simplex over $\mathcal{A}$ such that π maximizes the expected discounted cumulative reward V:

$$V(s_0) = \mathbb{E}_{a_0 \sim \pi(s_0)}[r(s_t, a_0)] + \sum_{t=1}^{T} \gamma^t \mathbb{E}_{s_t \sim P(s_{t-1}, a_{t-1}), a_t \sim \pi(s_t)}[r(s_t, a_t)],$$

where s_0 is an arbitrary initial state and T is the time horizon. To calculate this value, it is possible to bootstrap it in the so-called Bellman equation:

$$V(s_t) = \mathbb{E}_{a \sim \pi(s_t)}[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s_{t+1} = s' | s_t = s, a) V(s_{t+1})]$$

This equation, which underlies optimal control, model predictive control, and reinforcement learning, simply states that the expected cumulative reward of a state is the average reward obtained in that state added to the expected cumulative reward of the next state. $V(s_t)$ can then be interpreted as the value of state s_t when using a policy π.

Note that MDPs generally use a static action set (common to all states), while in the later cases of this work a dynamic action set is often used where the available actions are state-dependent. Such an action set is referred to as $\mathcal{A}_{s \in \mathcal{S}}$. While this has an important impact on the learning process, it makes the notation more cumbersome. Alternatively, we define a base action set $\mathcal{A}$ such that $\mathcal{A}_{s \in \mathcal{S}} \subset \mathcal{A}$, and whether an action in the base set is available in state $s \in \mathcal{S}$ is integrated into the policy π by setting $\pi(s, a) = 0, \forall a \in \mathcal{A} \setminus \mathcal{A}_s$. This process will be referred to as masking in the following sections.

3.1.3 General formulation of a Markov game

A Markov game is a further extension of the Markov process that allows several different agents to participate (compete or cooperate) in the same environment to achieve their own goals. Formally, a Markov game is a tuple $(N, \mathcal{S}, \{\mathcal{A}^i\}_{i \in 1,...,N}, P, \{R^i\}_{i \in \{1,...,N\}}, \gamma)$, where N denotes the number of agents, $\mathcal{S}$ is the set of states, $\mathcal{A}^i$ is the action set of agent i, where $\mathcal{A}$ denotes the Cartesian product $\mathcal{A}^1 \times ... \times \mathcal{A}^N$, $P : \mathcal{S} \times \mathcal{A} \to \Delta\mathcal{S}$ is the transition function, $R^i : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward received by agent i, and $\gamma \in [0, 1]$ is a discount factor. The action vector $\mathbf{a}_t \in \mathcal{A}$ denotes the action of each agent at time t.

Note that a Markov game is not only an extension of MDP, but also a specific kind of repeated game, so concepts from game theory can be applied.

A common way to train agents to solve a Markov game is to use a centralized training approach: In the training phase, a single model is used to create the policy of all agents. Then, during execution, each agent can either use the model as is, which is called centralized execution, or use only its own observations of the state, which is called distributed execution. This type of execution is especially useful when agents cannot communicate with each other outside of the training setup and cannot

observe the entire state[3]. For example, centralized execution can be used in board games such as chess, since both agents have access to the entire situation on the board, while distributed execution can be used in games such as poker, where each agent has a portion of the state (its cards) that is available only to it.

In the task of coordinating multiple robot arms across a construction site, it can be assumed that all robots have the same state description, allowing for centralized training and centralized execution. The decentralized training, decentralized execution framework is also used and compared to in experiment 5.2, but with less success.

3.2 Stable structure construction

To adapt the above formulation to the task of building a self-supporting spanning structure, we consider two approaches: In the first one, the system is modeled as a Markov game solved by distributed training, where each robot is an agent with its own model. In this approach, the agents only see the result of the actions of the others, and the actions themselves as well as the policy are kept private. The rewards can then be shared between agents or kept separate in a more individualistic perspective.

In the second approach, a supervisor or architect controls all robots sequentially in a centralized training, centralized execution fashion, and the Markov game is reduced to an MDP, preserving the theoretical guarantees of single-agent RL.

In the following section, we define the state space, action space, transition matrix, and reward function for the task of building a spanning structure in both centralized and distributed training. The reduction step from a Markov game to an MDP is discussed in 3.2.4, and the model used to build and learn the optimal policy is finally discussed in Section 4.5.2

3.2.1 States

The state s_t either contains the structure s present in the simulator at time t or corresponds to the terminal state s_T. A structure is a set composed of n_g grounds (g_1, ... g_{n_g}) and n_b blocks (b_1,b_2,..., b_{n_b}) which contains information about their types (links, hexagons, triangles, ...) , positions (in the isometric grid), and whether or not they are held in the case of blocks.

3.2.2 Action set and transition matrix

Before defining the action set and transition matrix, we define three logical operators: *leave*, *collision* and *connection*. Let the *leave* operator $L : \mathcal{S} \times N \to \{0, 1\}$, where $\mathcal{S}$ is the set of structures and N is the set of robots, represent whether or not the physical constraints defined in section 2.2 are respected when a given robot leaves the block it was holding[4], and let the *collision* operator $Col : \mathcal{S} \to \{0, 1\}$, represent whether or not some blocks intersect one-another in the structure. Finally The *connection* operator $Con : \mathcal{S} \to \{0, 1\}$ requires that we first define a distance between the different elements of the structure. Let $d : B \cup G \times B \cup G \to \mathbb{R}$, where B is the set of possible blocks and G is the set of possible grounds, represent the smallest euclidean distance between the corners of two blocks or grounds. Then, the distance in free air between two elements of a structure $d^* : B \cup G \times B \cup G \to \mathbb{R}$ is defined recursively by:

$$d^*(x, y) = \min_{z \in s}(d(x, y), d^*(x, z) + d^*(z, y)),$$

where both x and y are elements (grounds or blocks) of the structure s. The connection operator Con can then represent whether or not $d^*(x, y) = 0 \ \forall x, y \in s$. $Con(s)$ would then be equal to 1 if a path made of blocks links all of the grounds in s. Visually, the minimum distance in free air d^* between the two grounds (g_1 and g_2) of the structure in Figure 17 is shown in blue, whereas the distance d between them is shown in black. It can also be noted that $d^*(b_i, b_j) = 0$ if b_i and b_j are blocks of the same color, and $d^*(b_i, b_j) = d^*(g_1, g_2)$ otherwise. The distance d^* is then perfect to represent the shortest distance between two disconnected regions.

[3]The notion of observation relies on partially observable MDPs, that are not covered in this master thesis

[4]If the robot r is not holding a block in the structure s, then $L(s, r) = 1$

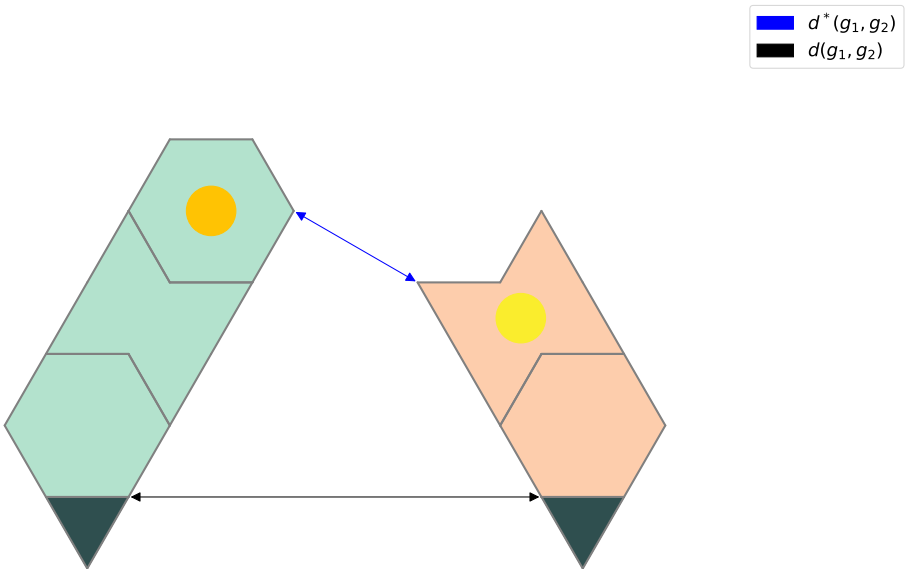


Figure 17: Comparison between d and d^*.

The actions that can generally be taken by a robot are either to stay in place and keep holding a block or to add a new block to the structure[5]. The action of the agent i is then denoted as H_i if the agent chose to keep holding the block it was holding and as $P_i(b_i)$ if the robot i tries to place a new block b_i. The transition matrix can then be defined by either adding the blocks to the structure ($s_{t+1} = \hat{s}_t = s_t \cup \{b_i : a_i = P_i(b_i)\ \forall i \in N\}$), or go to the terminal state ($s_{t+1} = s_T$). Three conditions can potentially terminate the episode, i.e., set $s_{t+1} = s_T$:

- The structure s_t is not stable once the robots try to leave, i.e. $\exists\ i \in N\ s.t.\ L(s_t, i) = 0\ and\ a_i = P_i(b_i)$
- The potential new structure $\hat{s}_t$ contains some collisions: $Col(\hat{s}_t) = 1$. The decision to terminate the episode as soon as a collision occurs is motivated by the fact that attempting to place a block at this location results in undefined behaviour in real life: Whether the structure collapses or the new block is simply added at a different position than specified depends on the setting of the real robot arms. So a conservative approach is taken and the collisions lead to the end of the episode.
- The structure successfully connects all the grounds, and all robots are able to leave: $Con(\hat{s}_t) = 1\ and\ L(\hat{s}_t, i) = 1 \forall i \in N$

The final question regarding the action is how to choose the position of the new block. For this we use the base action set $\mathcal{A}$, whose elements are either the *stay* action or describe a way to place a block given any state. Two ways to describe them are relative and absolute positioning: in absolute positioning, a block is defined by its position in the grid and its rotation, and the feasible action set of a state can then be defined as all possible ways to place a new block such that at least one of its sides touches a side of the existing structure. On the other hand, this condition is always satisfied if relative positioning is used: With this base action set, an action is directly defined as placing the side of a new block against one of the structure. Since this positioning has the pleasant property of being translation invariant, and since it was more effective in the literature [9], it was chosen in the experiments.

Note that importantly the framework taken here is sub-optimal: We know the transition matrix beforehand, and we know that it is deterministic. However, a MDP consider a stochastic transition and

[5]It was also tried to add other actions, such as *leave*, that results in the robot simply leaving the block it was holding without adding a new one or *remove*, that allowed the robot to remove some blocks that were already placed. These actions were later removed as they were unnecessary at best, and detrimental at worse.

the aim of a RL algorithm is precisely to learn what is the transition matrix. The matrix coefficients are then not used by the agent, that is tasked to learn them by itself.

By taking advantage of this prior knowledge, one could then design a model-based RL algorithm, that directly uses the information and thus converges faster to the optimal policy.

3.2.3 Reward choices

As reward design is still an open question, a modular reward was chosen: Different parameters could be tuned to observe the resulting changes. This framework could accept several different inputs and assign a positive or negative reward for each of them:

- **Success**: A binary flag that is set to true whenever all targets are connected and the robots do not need to exert any force to maintain the structure.
- **Failure**: A binary flag set to true if the selected action causes the structure to collapse or causes a collision.
- **Closer**: A binary flag set to true if a robot has set a block that reduces the distance between two targets
- **Number of sides**: The number of interfaces created by placing the new block.
- **Opposing sides**: The number of *opposing* sides created by placing a new block.

The success and failure flags alone allow the construction of a self-supporting spanning structure, but such a sparse policy is generally worse than a more dense one, that systematically gives a reward for good actions and punishes bad ones. The other parameters allow for different types of helpers that incentivise different structure shapes when the policy is learned: A high bonus for putting a block closer leads to more direct constructions, while a smaller bonus leads to more convoluted constructions. Adding a fixed cost for each action speeds up the learning, but sometimes the agent learns that it is better to fail early than to try anything. Adding a bonus to the number of new interfaces leads to more intricate and stable structures. However, since the simulator ignores assembly dynamics, like the friction forces applied when the robot adjust the position of the new block, the structures created would most likely be impossible to build without penalizing the number of opposite sides: When a block is in contact with two opposing sides, it must perfectly fit the gap between the two. In the real world, this would lead to two separate problems: Firstly the tolerances on the pieces would have to be much higher (one block cannot be one centimeter thicker than the others, otherwise it would not fit). Secondly, the new block would forcefully have to be slided between the two faces, creating friction forces on the blocks that could destabilize the whole structure.

Several sets weights for each components were tested, and the best ones can be found in the appendix 8.

A subtle trick with the *closer* flag is that it does not change depending on how far the new block is closer to the target. By using this trick, the cumulative reward is maximized when the shape of the structure tends to a semicircle. Indeed, a semicircle is the longest structure that can be achieved with doing infinitesimal changes in the distance, maximizing the number of rewarded actions that the agent can do in an episode. As the semicircle is also a great way of obtaining a self-supporting structures, the reward obtained by this flag is positive in all actions that tends to build an arch, reducing the number of local minima in which the algorithm could get stuck. As an example, the agents generally quickly find the structure consisting of a straight line to connect two grounds. This shape allows to span a gap of up to 4 triangles, but cannot be used to build bigger structures. Doing so need to find another kind of spanning shape, going toward the target in a more indirect fashion.

On the distributed training setup, two different reward aggregation could also compared: fully shared or fully individual. The fully shared aggregation simply forms an homogeneous reward for all agents by summing together the rewards for each individual agent, while the individual aggregation keep them separate. The huge advantage of the shared reward is that no agent is encouraged to act deceptively toward another, such as blocking the other agent to be the only one able to place new blocks, thus accumulating more rewards. A full cooperation is then more easily obtained when using a shared reward. Its main drawback, on the other hand, is that it adds up uncertainties in the training process: If an agent does nothing and the others build the structure without it, the agent simply learn

that doing nothing is the right way to build the structure. As the task of building a self-supporting structure requires a tight coordination between the agents, the fully shared reward was used in the experiment 5.2

3.2.4 Centralized training

In this setup, the robots are forced to act sequentially, alternating between a placing a block and holding it while the other robots place their own blocks. Since only one action had to be chosen at each time step (where to place the new block), only one reward was given. Combined with the centralized training approach, this allows the transition from the perspective of a Markov game, where each agent tries to maximize its own reward, to the perspective of an MDP, where the overall reward is maximized and only one agent is considered. One could compare this approach to a supervisor who, like an architect on a construction site, tells each worker where to place the blocks. The training task would then no longer be to train each worker to place a block in the right place, but only to train the supervisor.

Note that the centralized training would also be possible using simultaneous actions, like in the previous approach, but doing so quadratically augment the size of the action set, resulting in a harder problem to solve.

A final heuristic can be used to constrain the policy space: By exploiting the sequential placement of blocks, each robot can be forced to place a block only on the last block while it is still held by the previous robot and thus directly release it [6].

Removing the *keep holding*, H_i, actions highlight a powerful way to help agents learn an optimal policy: If two actions must be executed sequentially in all cases, it is better to replace them with a single action that executes both actions sequentially (which is then called a macro-action). The *place* actions of the decentralized training could already be considered a macro-action, since it consists of three steps:

- Leave the block that the robot is holding
- Place a new block at a given location
- Hold it

Using such macro-actions was shown to be highly beneficial in term of training time, to the point where some learning architectures tried to find some when no human-made macro-action were easily described [13].

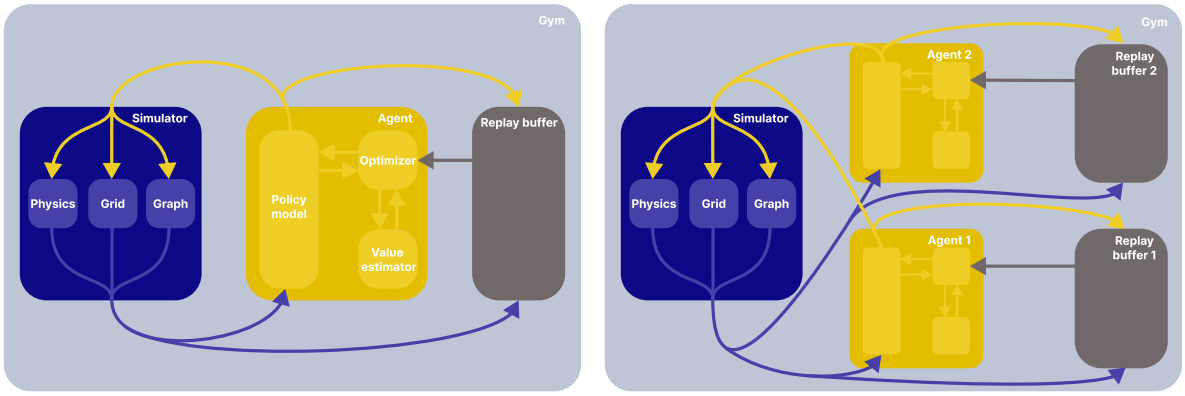
4 Learning

With the task modeled as an MDP, the learning process combines three different modules: The simulator that takes the actions generated by the agents and outputs a new state, the agent that takes the state and outputs the action thanks to its policy model, and finally a replay buffer that stores together both the state, the action, the next state and the reward received. An illustration of the learning setup, called gym, can be seen in Figure 18.

As can be seen in Figure 18a, the simulator and agents are divided into different sub-parts. The agent architecture used in this thesis is called actor-critic: given a state, the agent can either use its policy model (the actor) to generate an action, or its value estimator (the critic) to generate an estimate of the expected discounted cumulative reward of the state. However, this second model is used only to optimize the actor, using different algorithms described in Section 4.4. In this work, both models are deep neural networks described in section 4.2, but any type of function approximator can be used as long as it is differentiable and contains enough parameters to store the optimal solution.

It is also interesting to note that the optimizer does not directly use the state generated by the simulator, but instead uses the output of the replay buffer. This family of learning algorithms, called off-policy, does not rely on the current policy to optimize their models. This allows the optimizer to use a random sample of past transitions stored in the replay buffer instead of the current state, which is highly correlated with the previous one, and learn from older failures or successes.

[6] This heuristic can also not be used, as is the case in the experiment 5.2



(a) Centralized gym

(b) Distributed gym

Figure 18: Architecture of the gym for centralized and decentralized training

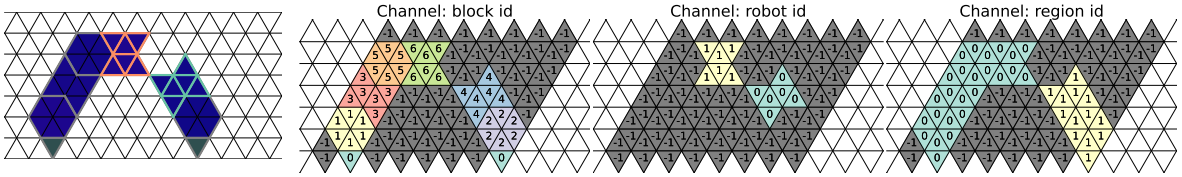
On the simulator side, one can see that in addition to the physics module (described in Section 2.2) and the grid containing the blocks, it also creates a graph. The first learning step is indeed to find a representation that can be used efficiently to train a model, and the use of a graph encoding has been shown in the literature to be one of these representations. For more details on the various state encoders, see Section 4.1.

Finally, it can be seen in Figure 18b that the distributed agents are optimized independently by using a separate replay buffer that does not contain the actions of the other agent. It can also be seen that the simulator receives both actions to update itself and produces the same state for both agents[7].

4.1 State visualisation

In both setups, the number of possible states is combinatorially huge. Using the grid defined in Section 2.1, the number of possible states would be on the order of $n_{blocks}^{2\times x_{max}\times y_{max}}$, but this is equivalent to saying that the number of possible binary images is equal to $2^{n_{pixels}}$: this is true, but most of the possible images are never seen and have no meaning. The agent's first task would then be to encode the state more compactly.

A first method is to use a convolutional neural network (CNN) on the grid, considering it as an image with multiple channels, and already different ways of representing the state can be used and compared. A first possible method is to use a channel containing the identifier of each block, a channel containing the ground to which the blocks are connected, and a channel indicating whether a block is held or not (and by which robot). Such a representation can be seen in Figure 19.



(a) Human readable version

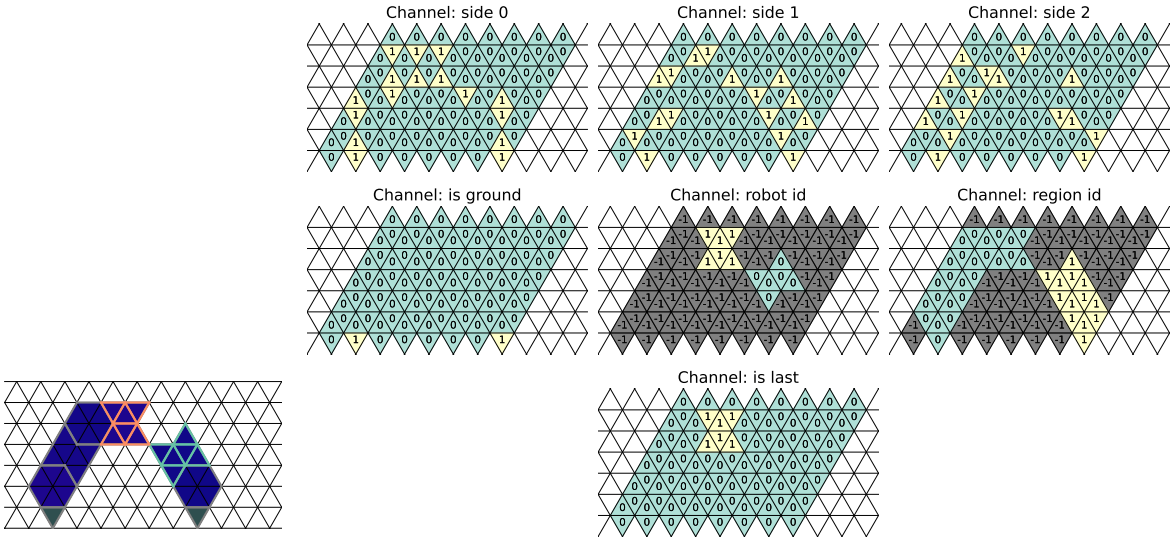
(b) Computer version

Figure 19: Order dependent state representation

In the centralized training setup, a different method can be used: Since the IDs of all but the last block are irrelevant, omitting them reduces the amount of useless information. However, the agent must still be able to distinguish whether or not a part of the structure consists of two different blocks or only one: if a part of the structure consists of only one block, it cannot break internally. To describe the state well and provide all the information needed to build the structure, the sides of each block are

[7]There is actually a small difference between the states of the two agents, since they do not have the same feasible action set $\mathcal{A}_s$

used, as shown in Figure 20. As one can see on the first row of the computer visualization, the center triangles of the link are not shown as they do not contribute to the perimeter of the whole block.



(a) Human readable version

(b) Computer version

Figure 20: Order independent state representation. Note that this structure is only there as an illustration, and could not be built as the agent is required to only place a block on the last one

The resulting channel decomposition is much harder for humans to read, but it implicitly eliminates the encoding step of determining the edges of the blocks based on their IDs. Finally, since this encoding cannot distinguish the different blocks, a single support block (the last one) against which the new block have to be placed is marked with a separate channel.

4.1.1 Graph visualization

An even more optimised representation of the states, as found in [9], would be to use a graph representation of the construction. In this thesis, we have chosen to use a heterogeneous, sparse graph with different nodes and edges to represent the different objects and feasible actions in the simulation. This is because it is more informative to describe a structure as "One block is placed on top of another, which in turn is placed on top of the ground" than to use a more image-based description such as "A blue block on top of a red block".

The set of nodes of the graph is the union of several sets. The first ones are the set of grounds and blocks in the structure $G_s \subset G$ and $B_s \subset B$, where G is the set of all possible grounds and B is the set of possible blocks. As an example, the nodes corresponding to the blocks and grounds of the structure in Figure 21a are shown in Figure 21b. In contrast to the previous representations, the set of robots $R = \{r_1, ...r_N\}$ is also added (as seen in Figure 21c), and the feasible action set A_s is embedded directly into the state. To do this, each action is decomposed into two parts: the support side against which the new block is placed (in grey in Figure 21e) and the potential new block that would be placed against it (in yellow)[8]. The set of nodes of the graph representation can then be described as $\mathcal{N} = G_s \cup B_s \cup R \cup \Psi_s \cup P_s$, where Ψ_s and P_s represent the set of support sides and potential new blocks, respectively.

Each of the node types have a set of attributes: The grounds and the blocks have their types (one hot-encoded) and their positions, the robots have the forces that they apply, the support sides have their orientation and a unique ID used to differentiate between two parallel sides, and finally the potential new blocks have the type of the new block and the ID of the side that would be connected to the one of the support.

The set of (undirected) edges of the graph is also the union of several type of edges.

[8]When the robots can act simultaneously, the *stay* action is modelled as a specific type of new block that is not connected to anything other than the robot

(a) Human readable structure s (b) B_s (in blue) and G_s (in gray) (c) R (d) Ψ_s (in gray) and P_s (in yellow) (e) Complete set of nodes $\mathcal{N}$

Figure 21: Nodes of the graph state representation

- Each robot is connected to each block with an edge labelled as *reaches*: $\mathcal{E}_{r1} = \{(b_i, r_j), \forall b_i, r_j \in B_s, R\}$. These edges are visible in blue on Figure 22a
- Each robot is also connected to each ground, and as the type of the nodes is different, the type of the edges also needs to be: $\mathcal{E}_{r2} = \{(g_i, r_j), \forall g_i, r_j \in G_s, R\}$. While these edges are also labelled as *reaches*, they are colored in gray on Figure 22a
- All of the robots are densely connected by a type of edge labelled as *communicate*: $\mathcal{E}_c = \{(r_i, r_j), \forall r_i, r_j \in R\}$. The only edge of this type is visible in orange on Figure 22a.
- Whenever a block is held, another type of robot-block edge is added to the graph, labelled as *holds*. As the name suggests, this kind of edge links the robot to the block it is holding: $\mathcal{E}_h = \{(H(r_i), r_i), s.t\ H(r_i) \neq \emptyset\ \forall r_j \in R\}$, where $H(r_i)$ returns the block that the robot r_i is holding or the empty set if it is not holding a block. These edges are visible in red on Figure 22b.
- The support sides are linked to the block they are a part of by using a type of edge labelled as *put against*. When the agents have to place a block against the last one, the set of these actions is $\mathcal{E}_\Psi = \{(\text{argmax}_i(b_i), \psi_j), \forall \psi_j \in \Psi_s\}$. These edges are visible in gray on Figure 22c.
- Each of the support sides is then connected to the action nodes that would consist in placing a new block against it, with an edge type labelled as *action description*: $\mathcal{E}_a = \{(\psi_i, p_j), \forall \psi_i, p_j \in \Psi_s, f(\psi_i)\}$, where $f(\psi_i)$ maps a support side to the set of all potential new blocks that could be connected to it.
- Each of the potential new block is connected to the robot that would place it with an edge labelled as *chooses*: $\mathcal{E}_\beta = \{(p_i, r_j), \forall p_i \in P_s\}$, where r_i is the robot that has to act. These edges are visible in orange on Figure 22d.
- The blocks are connected to their neighboring blocks by an edge labelled as *touches*: $\mathcal{E}_{t1} = \{(b_i, b_j), \forall b_i, b_j \in B_s,\ s.t. d(b_i, b_j) = 0\}$. These edges are visible in blue on Figure 22e
- The blocks are finally connected to the neighboring grounds with an edge sharing the same *touches* label: $\mathcal{E}_{t2} = \{(b_i, g_j), \forall b_i \in B_s, g_j \in G_s\ s.t.\ d(b_i, g_j) = 0\}$. This edges are visible in gray on Figure 22e

As said previously, the whole set of edges is the union of all the different types: $\mathcal{E} = \mathcal{E}_{r1} \cup \mathcal{E}_{r2} \cup \mathcal{E}_c \cup \mathcal{E}_h \cup \mathcal{E}_\Psi \cup \mathcal{E}_a \cup \mathcal{E}_\beta \cup \mathcal{E}_{t1} \cup \mathcal{E}_{t2}$, and the graph obtained by using the node-edge tuple $(\mathcal{N}, \mathcal{E})$ can then be visible on Figure 23

4.2 Models

4.2.1 Convolutional neural networks

As mentioned earlier, a convolutional neural network is used to encode the structures. The resulting abstract vector then passes through a few fully connected layers to finally output a list of values or probabilities, depending on what the network is used for. The length of this final list is equal to the size of the base action set, using the feasible action set of the state to ensure that the probability of choosing an unfeasible action is equal to 0.

(a) *reaches* and *communicates* (b) *holds* (c) *put against* and *action description* (d) *chooses* (e) *touches*

Figure 22: Edges of the graph of the structure in Figure 21a. The node types that are not used in each subplots are not shown for clarity

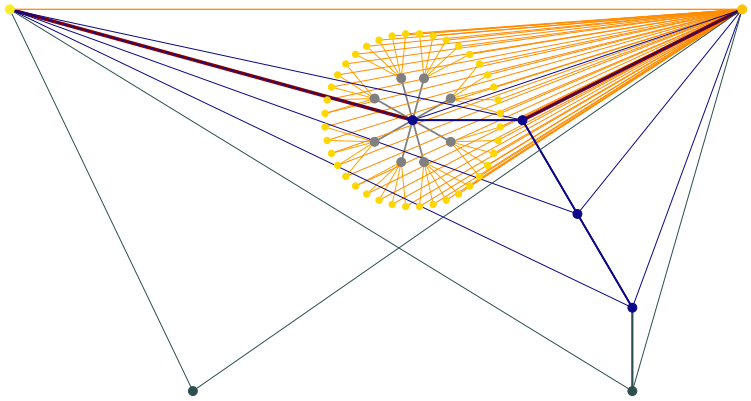


Figure 23: Full graph representing the structure on Figure 21a

An important design decision motivated by the learning behaviour of the neural network and the use of relative positioning is to remove rotations from the simulator. The reason for this is that neural networks do not behave well when a single change in input leads to a drastically different result[9], and that a rotation of the block, i.e., the support side, actually greatly affects the result of placing a block against it. If we rotate a block 180 degrees, the action that previously placed a new block on the block will now place it under the support.

Special care was also taken to design the base action set: Because of the properties of fully connected neural networks, augmenting the base action set can be a very effective way to add an implicit bias to the model. The reasoning is as follows: When the same action is performed in two different states, the top layer parameters of the network are updated each time. This behaviour allows the network to generalise the model and provide good results for unseen states, but arbitrarily choosing which action to update in which state can be a big help. Actions that generally change the state transition and receive a similar reward should be the same, but they should be different if the state transition (or reward) changes greatly. Let us take an example: If the new blocks must touch the last one in the single agent setup, connecting a hexagon to side 1 of a horizontal link could use the same action as connecting to side 1 of another hexagon: The problem, however, is that the two actions could result in the hexagon being placed either to the left or right of the block, depending on the relative position of side 1 on the support block, so a different action was used for each type of block, as in Figure 24. As a result, it augmented the size of the base action set without changing the size of the state-dependent feasible action sets: Only the action related to attaching a new block to the hexagon would be available if the last block is actually a hexagon.

[9]While this is true for neural networks, other models such as decision trees handle such changes much more easily

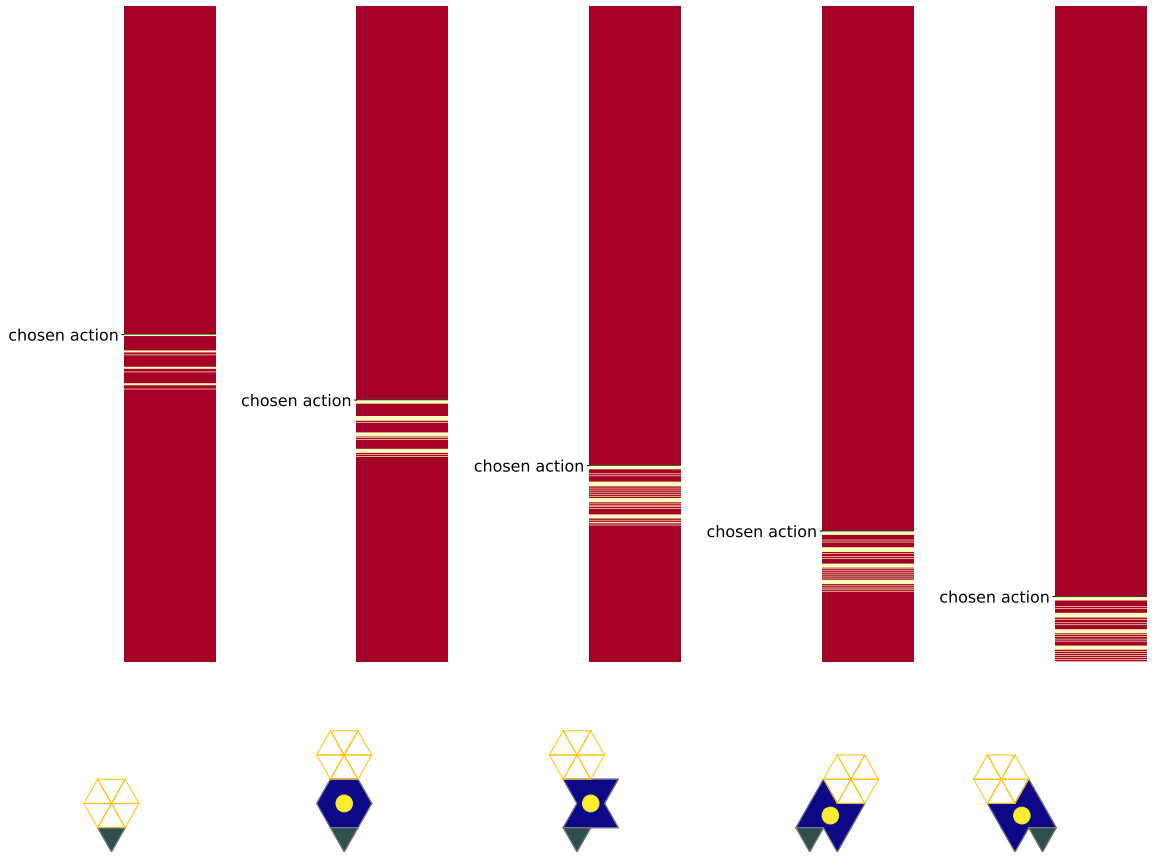


Figure 24: Illustration of the use of a base action set: each of the bars represents the base action set $\mathcal{A}$. The feasible action set $\mathcal{A}_s$ in the state below is shown in yellow, while the infeasible actions are in red. The effect of the single labelled action, shown in green, is shown on each of the states below. One can also notice that the top half of the action set is never feasible for the orange agent: they represent the same action, but performed by the yellow agent.

4.3 Graph neural networks

Graph representation has the great advantage that the model can be based on the specific topology of the input: Graph Neural Networks (GNN), a type of neural network that has become increasingly popular in the last decade [14, 15], directly use the input graph to share parameters between different steps: each block is processed with the same parameters, each ground is processed with the same parameters, etc. GNNs are actually a generalization of the various effective neural network architectures in use today: If the graph used is a square grid, the resulting GNN would be a convolutional neural network, while if the graph is fully connected, it would be a transformer. In the case of any sparse graph like ours, the resulting architecture is called message passing networks: In each layer, a node can use the information contained in the neighbouring nodes to update itself in the following layer. Thus, information travels from node to node and retains a sense of locality. This, of course, makes it extremely suitable for a task where forces follow the same pattern, and blurs the difference between model-free and model-based reinforcement learning.

To process the graph of the structure, the approach of [16] is adopted: In a first step, the heterogeneous graph is mapped to a typed homogeneous graph by projecting each of its features onto a basis. The homogeneous graph then goes through several graph convolution layers: The weight of each node is updated based on its own value and the value of the nodes connected to it. The particular type of graph convolution used in graph models is a Residual Gated Graph Convolution [17], which allows values to flow directly through the network without being updated, and thus has an output that can be based more directly on the input of the node in question. This is particularly useful for distinguishing between different actions, since they are all near the same node.

At the end of processing, each node contains a single value. The values of the action nodes are then collected and either used directly or to create the probability distribution of the policy.

One can also notice that in this model the base action set is not used: Instead, the feasible action set is embedded directly into the graph.

4.4 Training algorithm

The training loop performed in the gym is defined by the algorithm 1. As mentioned earlier, the agent is trained off-policy,using a buffer of past transitions: After each action of the agent, the state, the action, the reward and the next state are stored and only later used for training the policy. This method has the advantage of having less correlated samples to train on than the on-policy counterpart. The training loop consists of a fixed number of episodes in which some actions are executed up to a maximum number of time steps or until the terminal state is reached (see algorithm 2 for the centralized training and algorithm 3 for the distributed version). The method used to place the grounds could be either random or fixed, depending on how much the agent should generalize, but was always random in the experiments of Section 5.

Algorithm 1 Training loop with replay buffer

Require: agents **A**, simulator sim
  Initialise an empty replay buffer B
  Initialise **A**
  Initialise sim
  $ep \leftarrow 0$
  while $ep < ep_{max}$ **do**
    reset sim
    $B \leftarrow$ run episode(A,B,sim)
    $ep \leftarrow ep + 1$
  end while

At each time step of an episode, the agent first defines what is the feasible action set and produce a mask m such that:

$$m_i = \begin{cases} 0 & \text{if } a_i \in \mathcal{A} \setminus \mathcal{A}_s \\ 1 & \text{if } a_i \in \mathcal{A}_s \end{cases} \forall a_i \in \mathcal{A}$$

Algorithm 2 Episode with random initial state and sequential actions, centralized training

Require: a supervisor A ,a replay buffer B, a simulator sim
  $t \leftarrow 0$
  $s \leftarrow$ place grounds
  while $t < t_{max}$ and $s \neq s_T$ **do**
    $m \leftarrow$ mask of feasible action produced by A given s and t
    $a \leftarrow$ action chosen by A
    $s' \leftarrow sim(s, a)$
    $r \leftarrow R(s, a)$
    Store (s, m, a, r, s', m') in B
    update A with B
    $s \leftarrow s'$
    $t \leftarrow t + 1$
  end while
  return B

Algorithm 3 Episode with random initial state and simultaneous actions, distributed training

Require: a set of agents **A** ,a replay buffer for each agent **B**, a simulator sim
  $t \leftarrow 0$
  $s \leftarrow$ place grounds
  while $t < t_{max}$ and $s \neq s_T$ **do**
    for $A_i \in A$ **do**
      $m \leftarrow$ mask of feasible action produced by A_i given s
      $a_i \leftarrow$ action chosen by A_i
    end for
    $s' \leftarrow sim(s, \mathbf{a})$
    $\mathbf{r} \leftarrow R(s, \mathbf{a})$
    Store (s, m, a_i, r_i, s', m') in B_i ▷ When using a shared reward, all rewards r_i are the same
    for $A_i \in A$ **do**
      update A_i with B_i
    end for
    $s \leftarrow s'$
    $t \leftarrow t + 1$
  end while
  return B

And then randomly choose an action with either probability $\pi(s, m)$ (see algorithm 4[10]) or $\epsilon - greedy(\pi(s, m))$ (see algorithm 5).

Algorithm 4 Choose action: softmax

Require: state s, mask m, base set of actions $\mathcal{A}$ and intermediate policy model $\hat{f}$
Ensure: $m_i = 0$ if $a_i \in \mathcal{A} \setminus \mathcal{A}_s$ and $m_i = 1$ if $a_i \in \mathcal{A}_s, \forall a_i \in \mathcal{A}$
$val_a \leftarrow \hat{f}(s) + \kappa(m - 1)$ ▷ $\kappa \gg \max val_a$
$p \leftarrow softmax(val_a)$
$a \leftarrow$ sample $\mathcal{A}$ with probability distribution p
Return a

Algorithm 5 Choose action: ϵ-greedy

Require: state s, feasible action set $\mathcal{A}_s$, and policy model f
$p \leftarrow f(s, m)$
$r \leftarrow \mathcal{U}([0, 1])$ ▷ Sample r uniformly between 0 and 1
if $r \leq \epsilon$ **then**
$a \leftarrow \mathcal{U}(\mathcal{A}_f)$ ▷ Sample a feasible action uniformly
else
$a \leftarrow \text{argmax}(p)$ ▷ Pick the best action
end if
Return a

While the choosing step is fairly straightforward for both methods, updating the parameters of the policy parameter update is more subtle. Two different training algorithms are tested: Advantage Actor-Critic (A2C) and Soft Actor-Critic (SAC). As the names suggest, both methods are based on an actor-critic architecture, where two different networks generate a policy and an estimate of the state value, respectively. The main difference between the two methods, highlighted in the next section, is that A2C uses *surprise* to update its policy, while SAC acts as randomly as possible by maximizing its *entropy*. The aim of using two different algorithms is to use the older A2C as a baseline and compare the structures that each algorithm converges to.

4.4.1 Advantage Actor-Critic

The A2C model is a synchronous version of the Asynchronous Advantage Actor-Critique (A3C) presented in [18]. According to the authors, the asynchronous part of the algorithm is only useful when trained on CPU, and better performance is obtained with A2C when a graphics card is used. The algorithm uses three different networks: a policy network f, also called actor, and two value networks g and g', which are initialized with the same parameters but follow different updating rules. The updating of the weights is performed as follows:

After each action, the agent samples a mini-batch $\mathbf{b}$ of N transitions (s, m, a, r, s', m') from the replay buffer, respectively referring to a state, a mask of feasible actions, the action performed, the reward received, the next state and the next mask.

In a first step, network g generates an estimate of the discounted cumulative expected reward of state s, $V(s)$, while target network g' generates the estimate of $V(s')$. The objective for networks g and g' is then to approximate the Bellman equation:

$$V(s) = r + \gamma V(s')$$

To do so, gradient descent can be used to minimize a mean square error loss over the mini-batch

$$l_g = \frac{1}{N} \sum_{i=0}^{N} (g(s_i) - r_i - \gamma g'(s'_i))^2$$

[10] In theory, the masking operation (chosen for its effectiveness on a GPU) can produce infeasible actions with nonzero probability ($P(a \in \mathcal{A} \setminus \mathcal{A}_s | s, \pi) > 0$), but with a κ that is sufficiently large (i.e. 10^{10}), this probability is smaller than the computer's floating point error and is therefore rounded up to exactly 0

The reason to use two separate networks can be highlighted here: if only one network h was used to compute both values of $V(s)$ and $V(s')$, updating its weight θ_h by gradient descent would result in

$$\nabla_{\theta_h} l = \frac{1}{N} \sum_{i=0}^{N} (h(s_i) - r_i - \gamma h(s'_i))(\nabla_{\theta_h} h(s_i) - \gamma \nabla_{\theta_h} h(s'_i))$$

, meaning that the update step would optimize not only the value of the current state s, but also that of the next state s', without regard to the action that would have been taken in that new state. This approach has been shown to severely destabilize the network, and several methods are used to avoid this problem. The original algorithm copies the parameters of g into g' every few episodes, but the algorithm used in this thesis is instead using a soft update, introduced with the Deep Deterministic Gradient Descent (DDPG) algorithm [19]. In this method, the network g' is not updated by gradient descent, but by a weighted average of its current parameters and the parameters of g, slowly and gradually changing the value of the next state.

On the actor side, the network f uses the *advantage* (sometimes called surprise) $g'(s') - g(s)$ and multiplies it by the negative log-likelihood (nll) of performing the action a with the current policy to obtain its loss. This is equivalent to saying that the probability of performing action a in state s increases if the outcome is better than expected, and decreases otherwise. The algorithm 6 summarizes the different steps.

Algorithm 6 Update A2C

Require: buffer B of transitions $T = (s, m, a, r, s', m')$, discount factor γ, policy network f parameterised by θ_f, value network g parameterised by θ_g , and target network g' parameterized by θ'_g
  Sample a minibatch $\mathbf{b} = (\mathbf{s}, \mathbf{m}, \mathbf{a}, \mathbf{s}', \mathbf{m}')$ of lenght L from B
  for all $s'_i \in \mathbf{s}'$ **do**
    if $s'_i = s_T$ **then**
      $v'_i \leftarrow r_i$
    else
      $v'_i \leftarrow r_i + \gamma g'(s'_i)$
    end if
  end for
  $\mathbf{v} \leftarrow g(\mathbf{s})$
  $l_v \leftarrow ||\mathbf{v} - \mathbf{v}'||_2$
  $\theta_g \leftarrow \theta_g - \alpha \nabla_{\theta_g} l_v$ ▷ Update the parameters by using stochastic gradient descent
  $\pi \leftarrow f(\mathbf{s})$
  $l_p \leftarrow \frac{1}{L} \sum_{i=1}^{L} (v_i - v'_i) \log(\pi_{ai})$ ▷ Multiply the advantage by the nll of action a
  $\theta_f \leftarrow \theta_f - \alpha \nabla_{\theta_f} l_p$ ▷ Update the parameters by using stochastic gradient descent
  $\theta'_g \leftarrow (1 - \tau)\theta'_g + \tau \theta_g$ ▷ Soft-update the target network

4.5 Soft Actor-Critic

This model, which is newer than A2C [20, 21], uses five separate neural networks f, g_1, g_2, g'_1, and g'_2. As in A2C, the task of the actor network f is to generate the policy, that of the critic networks g_1 and g_2 is to generate an estimate of the state value, and the task of the target critic networks g'_1 and g'_2 is to generate the estimate of the value of the next state (see algorithm 7). The reason that led to the use of double-Q tables [22], in which two separate critic networks are used, is that a neural network tends to overestimate the value of a state. Using two networks, both initialised independently, and using the minimum value of their estimates effectively reduces this overestimation.

The main innovation of SAC is the use of an entropy bonus added to the reward. The entropy H of a discrete stochastic policy π is defined as

$$H(\pi) = -\sum_{p_a \in \pi} p_a \log(p_a)$$

where p_a represents the probability of performing action a The maximum entropy is then equal to $\log(n)$, where n is the number of actions, and occurs when π is a uniform distribution over all actions.

On the other hand, the minimum entropy is equal to 0 and occurs when one action is executed with probability 1 and all others are never executed. A higher policy entropy would then mean that the agent acts more randomly, leading to more exploratory behaviour.

To implement this bonus, the architecture SAC uses a Q-value system instead of a V-value system: instead of computing $V(s) \in \mathbb{R}$ directly, the critic networks actually generate a Q-table of n_{action} values: $g_k : \mathcal{S} \to \mathbb{R}^{n_{actions}}$. The V value of a state can then be calculated back by multiplying the value of each action in the Q-table by the probability of its execution and adding the entropy bonus:

$$V'_k(s) = \pi(s)^T g'_k(s) + \alpha H(\pi(s))$$
$$V^-(s) = \min(V'_1(s), V'_2(s)),$$

where α is the amplitude of the bonus. The networks g_k can then be optimized using the mean squared error or, as proposed in recent work, using the Huber loss [23], which uses an L1 norm instead of the L2 norm when the error is large, stabilizing the learning [11].

$$l_{gk} = \begin{cases} 0.5(g_k(s)^{(a)} - V^-(s))^2 \text{ if } g_k(s)^{(a)} - V^-(s) < \delta \\ \delta(|g_k(s)^{(a)} - V^-(s)| - 0.5\delta) \text{ else} \end{cases}$$

where the superscript (a) means that the element with index a is selected, and $\delta = 1$ was chosen as it matches well with the range of the reward function.

The policy network then directly uses the Q-value and the entropy bonus to update its parameters, as the optimal policy obtain with SAC is actually a compromise between the obtained rewards and the entropy of the policy:

$$Q^- = \min_{k \in \{1,2\}} g_k(s)$$
$$l_f = \pi(s) \cdot Q^- - \alpha H(s)$$

Algorithm 7 Update SAC

Require: buffer B of transitions $T = (s, m, a, r, s', m')$, entropy bonus α, discount factor γ, policy network f parameterised by ϕ, value networks g_1 and g_2 parameterised by θ_1 and θ_2 respectively, and target networks g'_1 and g'_2 parameterized by θ'_1 and θ_2

Sample a mini-batch $\mathbf{b} = (\mathbf{s}, \mathbf{m}, \mathbf{a}, \mathbf{s}', \mathbf{m}')$ of length L from B

for all $s'_i \in \mathbf{s}'$ **do**
  if $s'_i = s_T$ **then**
    $v'_i \leftarrow r_i$
  else
    $v'_i \leftarrow r_i + \gamma \min_{k=1,2}(f(s'_i) \cdot g'_k(s'_i)) + \alpha H(f(s'_i))$
  end if
end for
for all $k \in \{1, 2\}$ **do**
  $\mathbf{v}_k \leftarrow g_k(\mathbf{s})^{(\mathbf{a})} + \alpha \sum_{i=0}^{L} H(f(s_i))$
  $l_{vk} \leftarrow ||\mathbf{v}_k - \mathbf{v}'||_2$
  $\theta_{gk} \leftarrow \theta_{gk} - \lambda \nabla_{\theta_{gk}} l_{vk}$ ▷ Update the parameters by using stochastic gradient descent
end for
$Q^- \leftarrow \min(g_1(\mathbf{s}), g_2(\mathbf{s}))$ ▷ The minimum is taken between g_1 and g_2 action-wise
$\pi \leftarrow f(\mathbf{s})$ ▷ The tensor π has shape $(L, n_{actions})$
$l_p \leftarrow \frac{1}{L} \sum_{i=1}^{L} \pi^{(i)} \cdot (Q_i^-) + \alpha H(\pi)$ ▷ Multiply the value by the probability of taking action a
$\theta_f \leftarrow \theta_f - \lambda \nabla_{\theta_f} l_p$ ▷ Update the parameters by using stochastic gradient descent
$\theta_{g'_k} \leftarrow (1 - \tau)\theta_{g'_k} + \tau \theta_{gk}$ for $k \in \{1, 2\}$

[11] To add further stabilization, the target value was also pruned at the failure reward

4.5.1 Dueling Q-tables

Since this learning method uses Q-tables, the critic networks could use the dueling architecture: In some cases, due to a previous error in the policy, all actions lead to a failure of the task (e.g., when a robot has to act when it actually holds the whole structure). In such cases, it can be very beneficial to punish the state directly rather than the chosen action. The Q-values are then decomposed into two different channels: a V-value common to all actions, and an advantage A that is specific to each action this time. To ensure stability, it is still necessary to ensure that the advantage has a mean value of zero (a detailed explanation can be found in the paper presenting this architecture [24]). The two values are then simply added to obtain a complete Q-table, so this method is actually self-contained and does not interact with the rest of the program.

4.5.2 Dynamic exploration

To remove the hyper-parameter α, the method SAC was upgraded to instead use a target entropy in [25] and optimize α such that the optimal policy stays above this value. This method adds a loss function

$$l_\alpha = \alpha(H(\mathbf{s}) - H_t)$$

and use it to dynamically fit α using gradient descent. Since H_t is a lower bound, it is not desirable to actually "pull" the entropy towards this value, so a simple non-negativity constraint is added

$$\alpha^+ = max(\alpha, 0)$$

and the loss function is slightly modified so the gradient does not disappear whenever α is smaller than 0, and reduces its decay to keep some reactivity in the learning:

$$l_{\alpha^+} = ELU(\alpha)(H(\mathbf{s}) - H_t)$$

, where ELU is the exponential linear unit function. According to [25], using this method ensure that the policy π^* that is obtained once the training converged is

$$\pi^* = \underset{\pi}{\text{argmax}}(\mathbb{E}_{a_0 \sim \pi(s_0)}[r(s_t, a_0)] + \sum_{t=1}^{T} \gamma^t \mathbb{E}_{s_t \sim P(s_{t-1}, a_{t-1}), a_t \sim \pi(s_t)}[r(s_t, a_t)])$$
$$s.t.\ H(\pi^*) \geq H_t$$

One final detail that needed to be implemented is that the terminal cost or value of the terminal state should not be set to 0: Since the value of α has no upper bound, it is not possible to ensure that the optimal policy π^* finishes to link the different targets by granting a large reward in case of success. This problem is solved by scaling the terminal value V_T as a function of α:

$$V_T = H_t \alpha$$

Adding this dynamic entropy bonus turn the algorithm 7 into the algorithm 8

Algorithm 8 Update SAC with target entropy

Require: buffer B of transitions $T = (s, m, a, r, s', m')$, entropy bonus α, target entropy H_t, discount factor γ, policy network f parameterised by ϕ, value networks g_1 and g_2 parameterised by θ_1 and θ_2 respectively, and target networks g'_1 and g'_2 parameterized by θ'_1 and θ_2

Sample a mini-batch $\mathbf{b} = (\mathbf{s}, \mathbf{m}, \mathbf{a}, \mathbf{s}', \mathbf{m}')$ of length L from B

for all $s'_i \in \mathbf{s}'$ **do**

  if $s'_i = s_T$ **then**

    $v'_i \leftarrow r_i + \gamma\alpha^+ H_t$

  else

    $v'_i \leftarrow r_i + \gamma \min_{k=1,2}(f(s'_i) \cdot g'_k(s'_i)) + \alpha^+ H(f(s'_i))$

  end if

end for

for all $k \in \{1, 2\}$ **do**

  $\mathbf{v}_k \leftarrow g_k(\mathbf{s})^{(\mathbf{a})} + \alpha^+ \sum_{i=0}^{L} H(f(s_i))$

  $l_{vk} \leftarrow ||\mathbf{v}_k - \mathbf{v}'||_2$

  $\theta_{gk} \leftarrow \theta_{gk} - \lambda\nabla_{\theta_{gk}} l_{vk}$ ▷ Update the parameters by using stochastic gradient descent

end for

$Q^- \leftarrow \min(g_1(\mathbf{s}), g_2(\mathbf{s}))$ ▷ The minimum is taken between g_1 and g_2 action-wise

$\pi \leftarrow f(\mathbf{s})$ ▷ The tensor π has shape (L, $n_{actions}$)

$l_p \leftarrow \frac{1}{L} \sum_{i=1}^{L} \pi^{(i)} \cdot (Q_i^-) + \alpha H(\pi)$ ▷ Multiply the value by the probability of taking action a

$l_{\alpha^+} \leftarrow ELU(\alpha)(H(\mathbf{s}) - H_t)$

$\alpha \leftarrow \alpha - \lambda\nabla l_{\alpha^+}$

$\theta_f \leftarrow \theta_f - \lambda\nabla_{\theta_f} l_p$ ▷ Update the parameters by using stochastic gradient descent

$\theta_{g'_k} \leftarrow (1 - \tau)\theta_{g'_k} + \tau\theta_{gk}$ for $k \in \{1, 2\}$

5 Experiments

5.1 Different state representations

In this experiment, the three types of encoders described in section 4.1 were compared in a centralized training setup. The experimental setup consist of two robots with infinite force, but unable to apply a torque[12]. At the beginning of each episode, a single triangle ground is placed at the bottom right of the simulator as a starting point, and a target that needs to be connected is placed at the bottom left. The width of the target was randomly chosen so that the gap between it and the starting point ranges from 1 to 10, creating tasks of varying difficulty to obtain a smoother learning curve and allow for better measurement of performance. The training phase consists of only 10000 episodes. This amount is too small to achieve convergence of the policy, but allows to highlight the different rate at which the algorithm improves. The hyper-parameters of the setup and of all three encoders can be found in the tables 1, 4 and 5.

5.1.1 Results

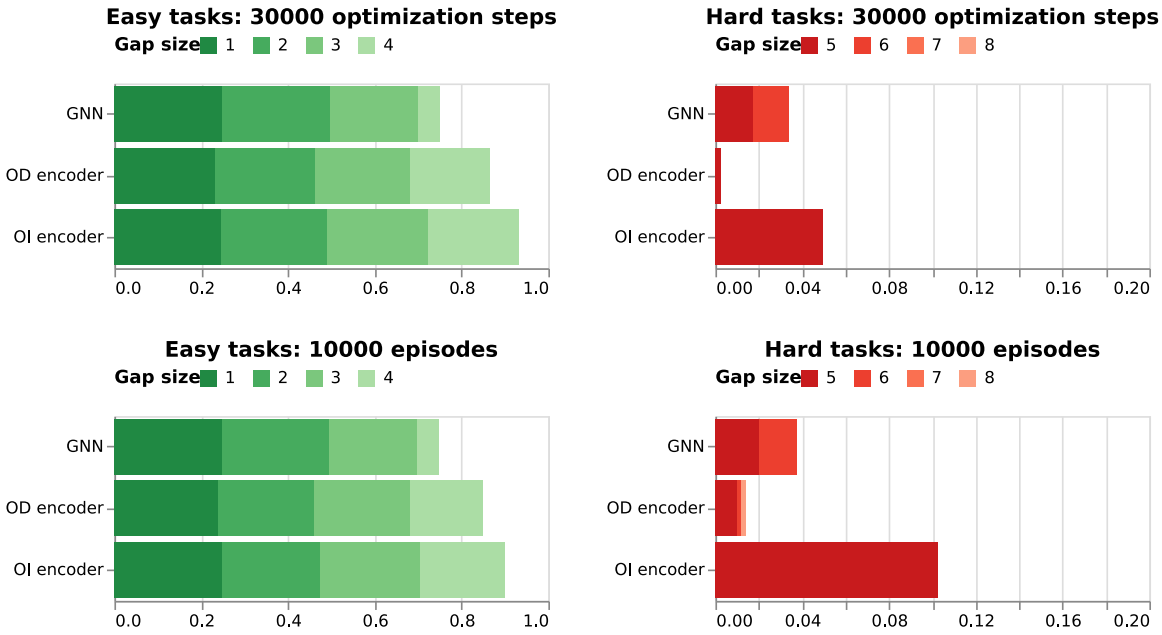


Figure 25: Success rate of the different models on specific difficulties, after 30000 optimization steps (top) or 10000 episodes (bottom).The colors represent the proportion of the success rate attributed to a specific gap

Because the convolutional encoders (either order-dependent (OD) or order-independent (OI)) used a different policy search than the Graph Neural Network Processor (GNN), they were able to use the fixed number of episodes to train on around 43000 optimization steps, while the GNN could only perform 31000. To allow a fair comparison, Figure 25 shows the results after 10000 episodes as well as after 30000 optimization steps in the top and bottom rows, respectively.

The tasks are then divided into 3 categories: The easy tasks, where the agent has to connect two regions separated by a gap from 1 to 4, allow a naive solution consisting of a straight line (examples of this solution can be seen in the first line of Figure 26). The hard tasks, i.e., the tasks with a gap of 5 to 7, require the agent to build arch-shaped structures (examples of this can be seen in Figure 27). Finally, the extreme tasks (which are not shown in Figure 25) were not achieved once in this experiment.

[12]In practice, the maximum force the agent could apply was set to 1000, as this was more than enough to never be a problem

The simplest tasks are not difficult for any of the encodings, although the graph-based agent struggles with the gap size of 4. It can also be observed that the difference between the two convolutional encoders is small for the easy tasks, while the OI encoder is much more successful for the hard tasks. After the same number of optimization steps, the difference between the GNN and the OI encoder is small, but the GNN is able to build slightly larger structures.

Throughout the training, the construction process was recorded once every 100 episodes, allowing us to highlight the areas where each type of encoding had more difficulty than the others. A selection of successful structures is shown in Figures 26 and 27, and the last three failures of each model are shown in Figures 28, 29, and 30.

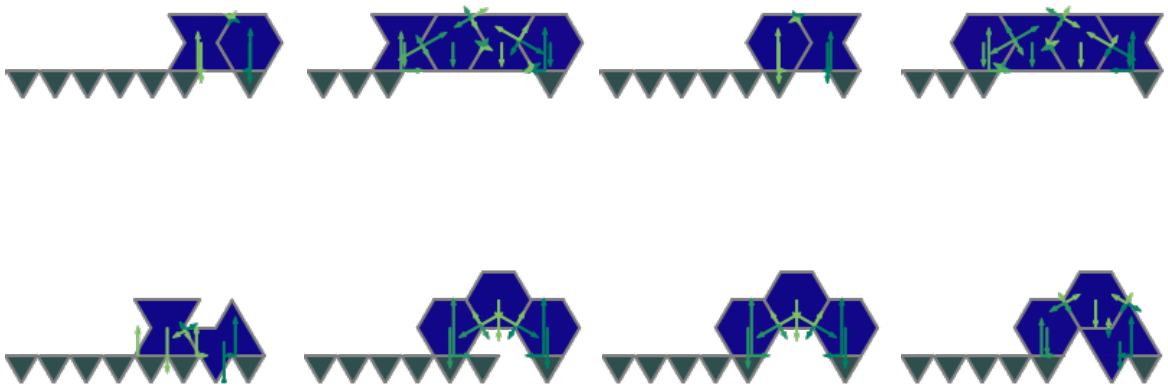


Figure 26: Types of structure that all the agents were all able to build

(a) Structure made by the GNN (b) Structure made by the OI CNN (c) Structure made by the OI CNN

Figure 27: Bigger structures, allowing to break past the gap limit of 4

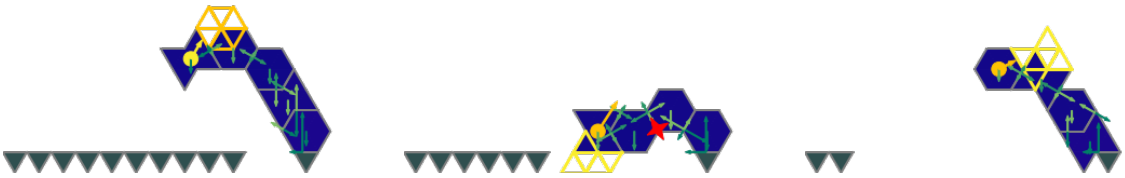


Figure 29: Last 3 failures of the OD CNN. The last action tried by the agent is depicted

Figure 30: Last 3 failures of the GNN. The last action tried by the agent is depicted

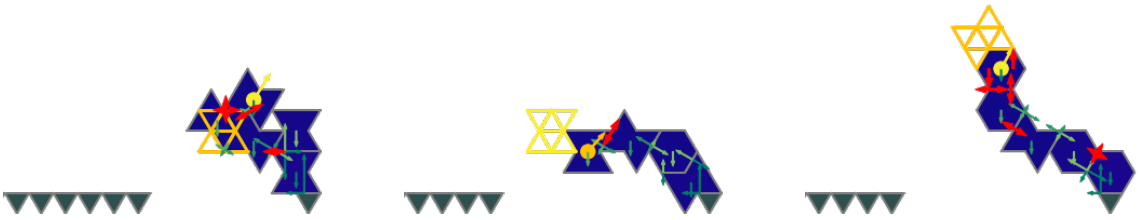


Figure 28: Last 3 failures of the OI CNN. The last action tried by the agent is depicted

5.1.2 Discussion

As can be clearly seen in Figure 25, the OI encoder seems to be the most efficient encoding. The fact that the OD encoder can build a bridge of size 8 is remarkable, but the success rate is too low to be significant. The different performances can be explained by the last failures of each model: The OI encoder, by using its expanded base action set, is able to easily learn which actions lead to a collision between the new block and its support, so its failures are always (at least partially) due to the collapse of the structure (see Figure 28). The OD encoder also relies on the same base of actions, but must learn that the IDs of the blocks are irrelevant. As can be seen in Figure 29, this causes the agent to attempt to place blocks that do not directly intersect with their support, but collide with previous blocks in the structure. Finally, the graph processor does not rely on such base action state and must

learn the shape of each block, distinguishing whether the failure is due to a constraint non-satisfaction or to a collision, which sometimes results in a new block colliding with its support (as in the third example in Figure 30). Since these errors occur early in the episode, the number of optimization steps that the GNN can perform in its limited number of episodes is reduced. Graph processing can then be considered when the shape of the block is simple, as was the case in [9], but not for more complex polygons

In addition, it can be noted that the number of episodes was the same for all encoders, but the time required to train the graph processor was eight times greater than that for the CNNs (24 hours for the graph processor versus 3 hours for the CNNs). For this reason, the GNNs are no longer used in the remaining experiments.

5.2 Centralized versus decentralized

The purpose of this experiment is to compare the centralized training, where the Markov game is reduced to an MDP with a single agent, to the decentralized training setup. To reduce the difficulty of the problem, only the hexagonal blocks are available to the agents, and all actions that lead to a collision are removed from the feasible action set. One small difference between this setup and that of the previous experiment is that the two grounds that the agents must connect are now of similar size, and since the agents are not forced to place the blocks on the last floor placed, they can start on either of the grounds.

In the decentralized setup, two agents, each controlling a single robot arm, can either place a new block or stay put. They then have the option of placing two blocks at a time, and must learn to avoid placing a block in the same position as the other agent, or it would result in a dynamic collision (a special kind of collision, where the agent cannot use the state to know if it would happen as it results directly from the action of the other robot) and a failure of the episode. As described in Section **??**, the agents do not have any information about the action of the other except its concrete result on the state. Finally, the fully-shared reward is used to help the agent cooperate.

In order to use the centralized training as a fairer baseline, the condition of placing a block only on the last placed block is removed. However, it still has the advantage that both robots have a single model and that their actions automatically switch between *place* and *stay*.

Since the order-independent encoder cannot be used when the agents can place a new block against any other and not just the last, and since the graph processor is too slow to be efficient, the order-dependent encoder is used. The agents are trained on 40000 episodes, and the gap between the two grounds was randomly set between 1 and 7 at the beginning of each episode. The remaining hyperparameters can be found in the tables 6 and 2.

5.2.1 Results

As in the first experiment, the tasks were divided into 3 categories: Easy, hard and extreme. The easy tasks, i.e., those with a gap between the grounds of 1 to 3, do not require active cooperation between the agents, whereas the hard tasks require the agents to purposefully alternate between placing a new block supporting the one held by the other robot and remaining in place until the now released robot comes to place another new block. The extreme tasks additionally require the construction of a support pillar to increase the frictional force on the ground, and have never been achieved

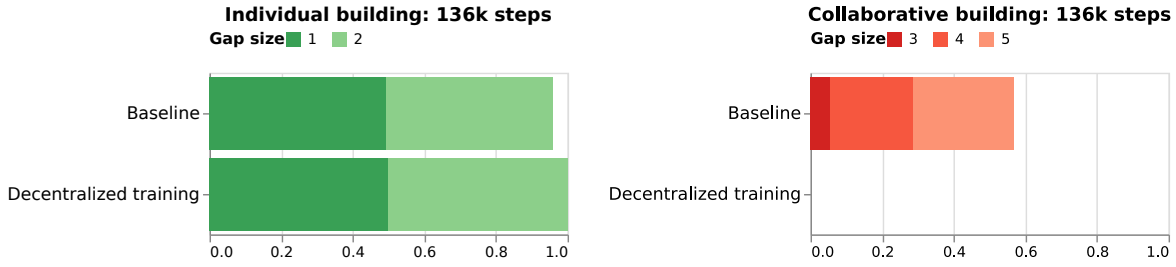


Figure 31: Success rate of the different setup on specific difficulties

Figure 32: Successful structures. The sample were all taken from the single agent setup, but the successful structures made by the multi-agent one look similar

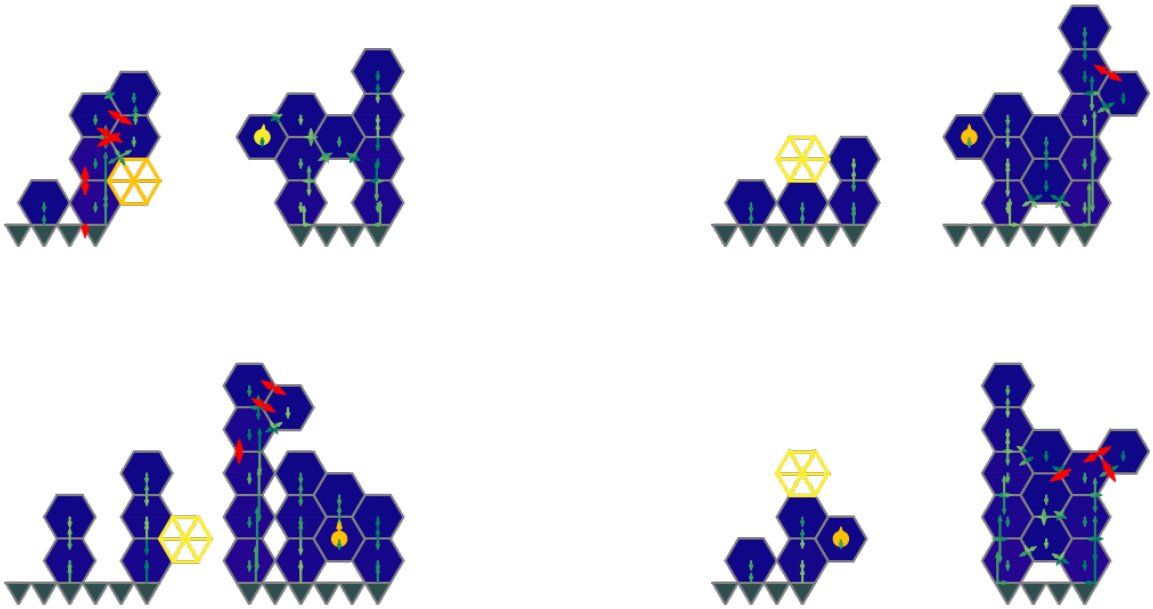


Figure 33: Failed attempts made by the single agent

As before, a list of the successful structures is presented in Figure 32, and a sample of the last failures of the two models is depicted in Figure 33 and 34

In addition, the sequence of action taken by the two agents to link the two grounds are also depicted in Figure 35

5.2.2 Discussion

As can be seen, the distributed agents were unable to cooperate sufficiently to compete with the centralized training. However, looking at the failures of the multi-agent setup, we see that there is no dynamic collision, indicating that both agents have specialized in their own set of location so that they are not at risk of colliding. This underscores the importance of macro-actions: By forcing the robots to hold its block for exactly one turn in the centralized training setup, their ability to cooperate is greatly enhanced, as an error would systematically occur if the second robot was instructed to place a block in a location that did not support the previous one.

It is also interesting to note that the success rate of the centralized training is lower for gaps of size 3 and 4 than for the gap of size 5. This is indeed due to a local minima: When the gap size is 5, the agent must place its blocks on the edge of the each ground to successfully build the structure, whereas when the gap size is 3 or 4, the agent must anticipate the subsequent reward and not simply perform the action that yields the largest reward in that state. This local minima is particularly viscous in the case of the gap size of 3: If the agent decides to place a block on the edge of each ground, the structure

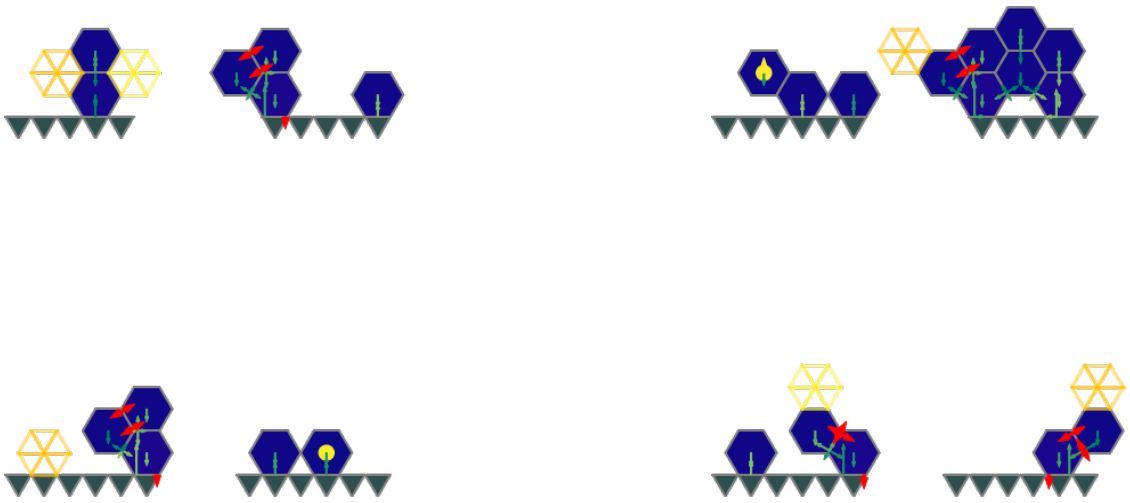


Figure 34: Failed attempts made by the multi-agent

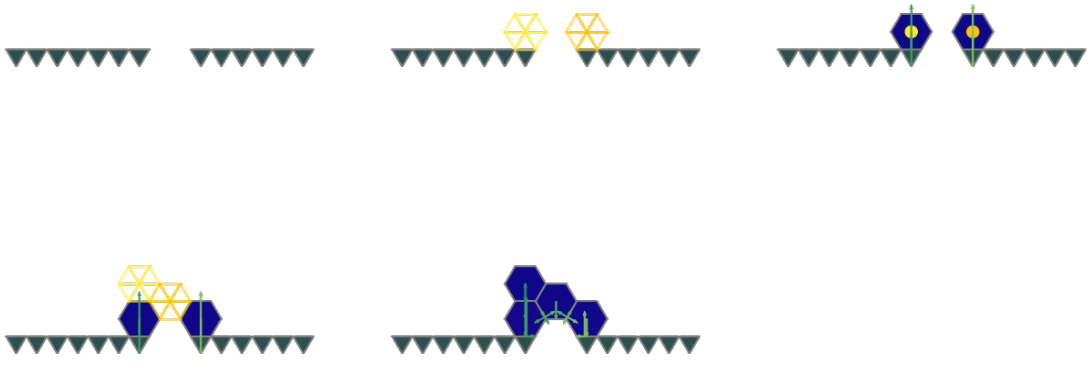


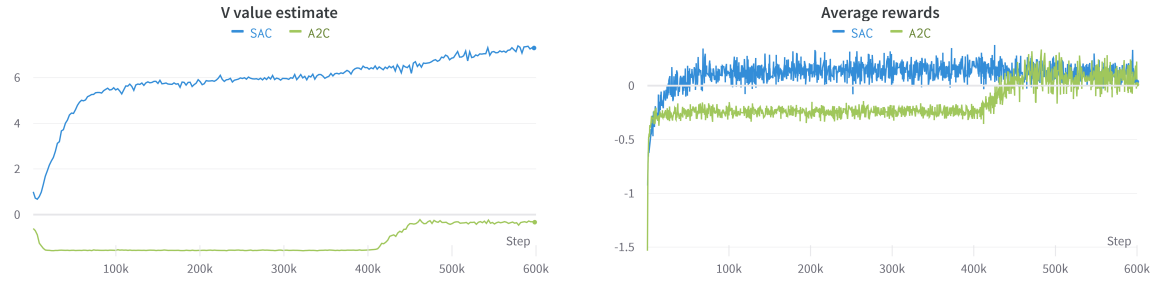
Figure 35: Sequence of actions resulting in a success using the simultaneous actions

cannot be built (as in the third situation in Figure 33). Another observation is that the helper reward concerning the number of sides in contact, supposed to help the agent build a stable buttress against which a longer bridge can be built is well maximized by the agents trained centrally trying to fill the available space with blocks when no bridge building solution is found.

5.3 Dynamics of learning on harder tasks

This experiment attempts to illustrate the learning dynamics that occur when training on a large setup, and how the simple tasks are used to support exploration when using SAC learning algorithm. To compare this exploration method, the A2C algorithm is used as a baseline. A single agent is centrally trained on a large setup with, once more, the goal of connecting two regions separated by a gap. As in experiment 5.1, a single triangle is placed at the right of the simulator and a target of random width is placed at the left, so that a gap of 1 to 19 triangles is created. On the SAC agents, the training is performed for 70000 episodes and generally last around one day and 8 hours. The A2C agent is then trained on 100000 episodes to match the same number of optimization steps and allow a fair comparison, and also took roughly the same training time. The hyperparameters of the model can be found in the table 7 in the appendix, and are the same for both learning agents.

5.3.1 Results



(a) Estimated V value of SAC and A2C

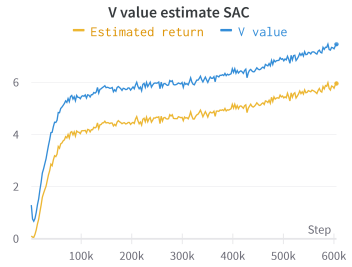
(b) Average rewards sampled to optimize the model

Figure 36: Rewards over time

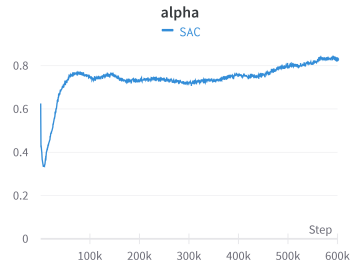
Figure 36 shows two different metrics for rewards achieved in an episode. As can be seen in Figure 36b, the average reward plateaus early on for both agents, and the A2C agent achieves a breakthrough, obtaining the a similar average value as the SAC agent at around 400k steps. On the other hand, the expected cumulative reward of the SAC agent (in blue on Figure 36a) increases continuously, and is much higher than the one of the A2C agent (in green). Note that the metric on Figure 36 is the average reward of the mini-batch sampled from the replay buffer, and the V-value of Figure 36a is produced by the target value neural networks on the same mini-batch.

The entropy of the policy produced by each of the agents is shown on Figure 37c. The low entropy of the A2C agent (in green) is not representative of its actual policy, as it is choosing its action using an ϵ-greedy policy, thus the entropy of the agent is always around 0.25 (depending on the number of feasible actions, this number can change slightly). The small increase in α seen in Figure 37b has negligible impact on the dynamics of the V value of SAC after 100k steps: the blue curve on Figure 37a represents the V value as calculated in algorithm 8, whereas the entropy bonus is subtracted in the yellow curve, only keeping the actual expected returns. On the other hand, this entropy bonus still allows the entropy of the policy to remain at its target value of 1.8 (as seen in Figure 37c).

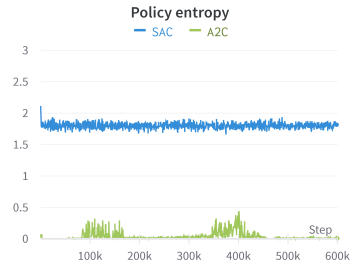
The success rate of the agents was divided into four categories according to the size of the gap between the two grounds, and the mean, minimum, and maximum are shown in Figure 38. As one can see on Figure 38a, the A2C agent is not able to build a single structure before training for around 400k steps, and is then only able to build the easiest structures: Its success rate stays at 0 for the harder tasks. On the SAC agent side, there is a clear overshoot early on for the easiest tasks on Figure 38a, followed by a plateau and finally a decay. Its success rate for the intermediate tasks, shown in Figure 38b, does not show the same kind of overshoot as with the easy tasks, but still follows the same



(a) Comparison between the V value estimated by SAC and its actual estimated return



(b) Entropy bonus used in SAC

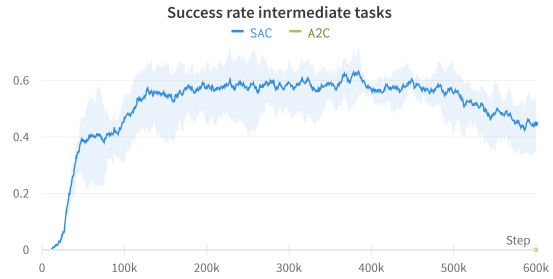


(c) Entropy of the policy for both SAC and A2C

Figure 37: Entropy bonus metrics

downward trend after 400k steps. The success for the hard and extreme tasks, shown in Figure 38c and 38d, show no sign of this decrease, and the extreme tasks even show a significant increase only after the other categories begin to decay.

(a) Success rate obtained on gaps of size 1, 2, 3 and 4



(b) Success rate obtained on gaps of size 5, 6 and 7

(c) Success rate obtained on gaps of size 8, 9 and 10

(d) Success rate obtained on gaps of size 11 to 19

Figure 38: Success rate: The solid line represent the mean of the category, and the shaded area goes from its minimum to its maximum. On the intermediate tasks and harder, the A2C agent could not find a spanning structure a single time in the whole training

5.3.2 Discussion

Looking at Figure 36, the difference between the average reward and the cumulative reward can be explained by the following argument: As the policy gets better, the episodes get longer. Consequently, the average reward, which aggregates rewards based on a state-action pair, is quickly dominated by the helper reward. On the other hand, the cumulative reward aggregates the rewards on an episode basis, so it does not overrepresent the non-terminating actions in the same way. Since the A2C policy generally had a negative helper reward prior to its breakthrough, the average reward also increases at 400k steps. One can also notice that before the plateau, the average reward of the SAC agent grows faster than the cumulative reward, and this difference can be easily explained by the exponential

average used to stabilize the target critic. The A2C agent's cumulative reward begins by decreasing, even as the average reward increases. This is due to the fact that the initial values of the neural network parameters are randomly initialized, which leads to biased estimation in the initial steps. To measure the agent's performance on the task for which it is optimizing (i.e., maximizing the V-value), one could first observe the average reward and then gradually move to the estimated cumulative reward as it becomes less biased as training progresses. Indeed, using the V-value gives a much better idea of the success rate of the agents than the average reward: the average rewards of the A2C and SAC agents are extremely similar once the A2C agent is able to perform the easiest tasks, but the V-value allows one to see directly that the SAC learning algorithm actually has much better performance on the more difficult tasks, as shown in 38.

To explain the sudden increase in performance of the A2C algorithm after 400k steps, one can look at the structure it was able to build before and after this change:

(a) Failure caused by reaching the maximum height

(b) Failure caused by trying to place a block to far to the right

Figure 39: Structures produced by the A2C learning algorithm after 350k steps

As can be seen in Figure 39, the A2C algorithm tends to produce a similar structure in each episode, regardless of the gap. The worst thing about this policy is that by going toward the right, any deviation from the structure in Figure 39a that occurs after the placement of the third block would result in a direct failure (as in Figure 39b), and the straight line going in the wrong direction would appear optimal to the algorithm. This is a good example of how easily local minima arise and how difficult it is to overcome them.

(a) Success for gaps up to 4 triangle wide

(b) Failure for bigger gaps: the friction force with the ground is not big enough to sustain the structure

Figure 40: Structures produced by the A2C learning algorithm after 450k steps

Once the starting block is randomly changed to the horizontal link of Figure 40, the agent quickly learns that this solution is better and changes its policy to systematically build the structure shown in Figure 40a. However, this policy is still not optimal, since only the simple tasks can be solved with this structure.

To explain the overshoot and decay of the success rate of the SAC in Figure 38, it is necessary to recall the main result of the SAC learning algorithm: It converges to the policy that maximizes the discounted cumulative reward, with the constraint that the entropy of the policy is higher than a given target seen in Figure 37c. The curves in Figure 38 can then be explained as follows: In a first phase, the agent acts randomly on most tasks, so it can reduce its entropy on the simplest tasks, acting more deterministically and achieving a high success rate. This is the effect of averaging entropy

across batches: if the entropy of the policy is high in most states, it may be 0 in a few states. However, the agent cannot increase its success rate on more difficult tasks without increasing the entropy of the easiest tasks, which naturally lowers the success rate. Moreover, the extreme tasks require a higher degree of determinism: if the correct action is performed with probability 0.9 and it takes 20 actions to build the entire structure, the success rate is only 12%. The entropy of all other tasks must then be higher to allow the success rate of the extreme tasks to grow. This phenomenon explains why in the Figure 38d, the success rate increases only when the success rate in the Figure 38a and 38b decreases.

One might think that reducing the target value over time would fix this problem, but there is actually no need to do so. A success rate of 1, like the one obtained with the A2C algorithm, means that the agent simply fully exploits the easy tasks and does not use them for exploration, and structures such as the one of Figure 41b would not be tried. Using the easy tasks to learn to perform the harder, more punitive, ones is a massive advantage of the SAC algorithm, and the diversity of structures created allows it to make a better use of its sample: instead of having the same structure over and over again like it is the case with the A2C algorithm, the entropy bonus forces a more varied set of state to train on.

(a) The same strategy as A2C is sometimes used

(b) Different structures are also tried

Figure 41: Structures produced by the SAC learning algorithm after 20k steps. Note that unlike the structures created by A2C, these structures are different even though the initial state is the same, which is an effect of the bigger entropy of the policy

As long as the probability of the best action is slightly higher, one could choose to select it using a greedy policy in a later, exploitative phase. This approach is demonstrated in Figure 42, which shows various structures generated in this way. To demonstrate the usefulness of the slightly reduced success rate of the smaller structures, the agent was forced to take sub-optimal actions at the beginning of the building in Figure 43 and was still able to build the structures, while it always result in a failure when using the policy obtained with A2C. This shows that in addition to being more sample efficient, the SAC algorithm is also more noise resistant, and has better chances to be able to go from the simulation to the reality: If an action is unfeasible in real life for any reason such as the robot kinematics or the friction dynamics, the agent could adapt on the go and simply change plan mid-way.

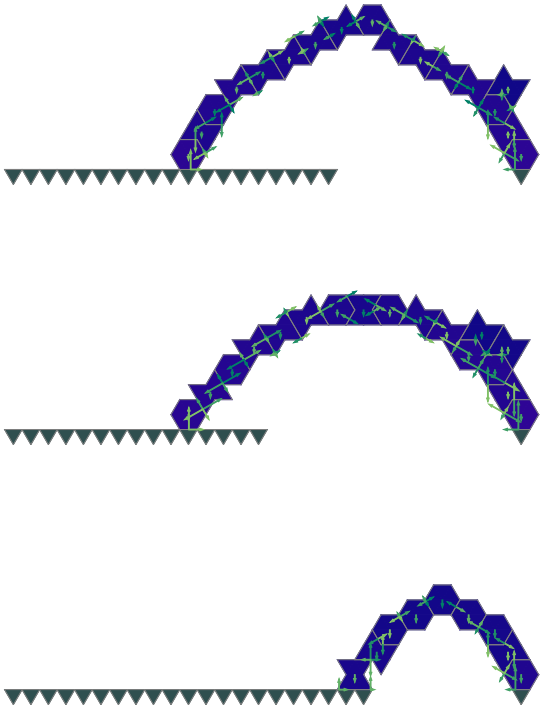


Figure 42: Structures obtained when using a greedy policy on the SAC agent

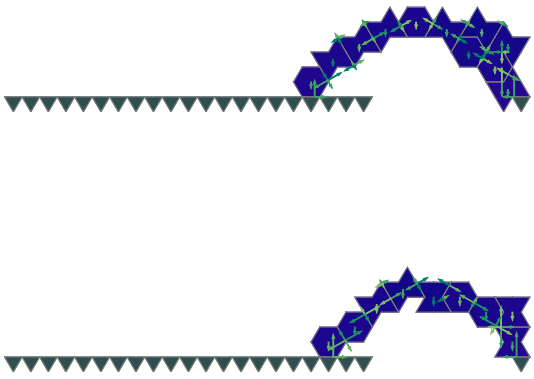


Figure 43: Structures obtained when using a greedy policy on the SAC agent and forcing it to take a different start

6 Conclusion

In conclusion, two tasks were successfully accomplished in this work: The first one was to develop an efficient simulator for static constructions, which allows to build structures exposed to gravity and to check their stability. The resulting simulator was able to achieve good performance in terms of realism: All structures that were found to be stable by the simulator were also stable in reality. The conservative approach of the simulator, which uses a safety kernel and neglects the point forces at the corners, resulted in some structures being incorrectly classified as unstable, but this was necessary to allow coarser tolerances on the blocks. The computation time required by the simulator was optimised so that one-third of the total exploitation time was spent on collision checking and calculating the equilibrium of the structure. The remaining two-thirds was used to evaluate the policy, which was performed on a GPU and without the optimization step.

The second task was to use this simulator to train agents, using reinforcement learning, to build arch-shaped structures that could connect two grounds separated by different distances from each other. Three experiments were conducted to test different approaches, and the following points could be highlighted: To learn how to build a structure, the agent(s) must learn three subtasks. The first is to learn what components make up the structure and how they interact with each other. When comparing different models in experiment 5.2, this subtask was well solved by combining convolutional neural networks and augmented base action sets, which use visual input to learn whether a new block would collide with the structure if placed in a particular location. Graph neural networks were unable to do the same thing efficiently and were abandoned because they were struggling to handle difficult objects like the links. The second subtask was to coordinate the agents to act sequentially when needed. This subtask proved extremely difficult in experiment 5.2, where the distributed agents were unable to learn it, and was successfully solved by implementing macro actions that force the agent to act sequentially, although the ability to build structures in parallel was removed. A third subtask was for the agent to learn the sequence of actions that would lead to a stable structure. This required the agent to efficiently search the policy space, which was successfully accomplished in experiment 5.3 using the discrete actor-critic algorithm (SAC). Indeed, this algorithm allowed efficient use of average entropy to search the policy space, and the resulting structures were able to reconfigure themselves during construction to allow for unexpected outcomes.

7 Outlook

7.1 Challenges

The task of building a self-supporting structure, compared to common RL benchmarks such as the Atary games, have a significant amount of different actions, and requires long term planning. These two factors are generally hard to manage by RL algorithms, and an intensive amount of work had to be done, both in this thesis and in the literature, to handle such setup.

However, the biggest challenge in this task is the time required for the algorithm to converge. This parameter alone limits the policy models that could be used to the simplest ones, since trying to use models that better fit the results, such as GNNs, inevitably increases the time it takes the algorithm to learn. Most of the best examples from MARL, such as the hiding game in [26], require hardware accelerators that could not be matched. In this last work, they were able to run a model for hundreds of millions of steps in just 16 hours, using a batch size hundreds of times larger than that used in the experiments.

The successful construction of a self-supporting structure in a simulator is already a step in the direction of mechanised building process. These methods could be helpful in the context of sustainable building, where the aim is to use novel construction techniques to reduce the use of concrete and enable the reuse of old building components. However, to reduce the environmental impact of building, it is important to consider the huge energy cost of reinforcement learning, and few-shot learning methods, such as imitation learning or model-based RL, should rather be used.

7.2 Next steps

While this work has shown that it is possible to build a self-supporting structure using MARL, some improvements can be made. They fall into three categories: Improving the setup, improving the

models, or improving the learning algorithm.

7.2.1 Setup

This category of next steps mainly focuses on adapting the setup and simulator to reality. An obvious next step would be to use a continuous 3D environment where the blocks could be polygons of different shapes or sizes, but one could also consider other directions. Including noise, for example, would already be a step toward realism. It could easily be implemented by adding an uncontrolled force to each action of the robot or even to each building step, but it would complicate the learning process. The idea of modelling the static equilibrium of the structure as LP was motivated by the fast computation time, but it may have been too zealous as the time required to check the stability of the structure is still a factor of 2 less than the time required to select the action, even without the optimization step and running the model on a GPU. While this in itself is not a problem, it definitely allows for more latitude in the simulator, and using a classic training simulator such as Grasshoper, Mujojo, or Webots would allow the noise of the action and the continuous 3D environment to be incorporated directly. Moreover, the use of one of these setup would allow to train not only the statics but also the dynamics of the robots and to include their sensory feedback.

7.2.2 Model

Although the GNN compares poorly to the simple CNN, this first network type has great potential: testing different architectures, e.g., graph gated recurrent units, could potentially increase the quality of the output policy significantly. Another advantage of these models is that they can handle many more actions than their CNN counterparts, since they share more parameters. However, their weakness in understanding the shape of the blocks should be addressed first, especially for use with recuperated blocks, which would all have a unique shape.

Model-based RL or model-predictive control could also be interesting prospects, since they generally require fewer training steps than the model-free approaches and since the actual transition function of the MDP is known (but costly to compute). A final improvement that could definitely make a difference is the use of a planning algorithm such as a Monte Carlo search tree. Performing a search step in addition to the value estimation and policy optimization was effectively proven to be highly beneficial in MDP with large action space, such as the game of Go.

7.2.3 Algorithm

While the learning algorithm itself achieves acceptable performance and currently represents the state of the art, a deeper look at distributed training could be interesting to effectively use all robots simultaneously. For example, centralized training with distributed execution would greatly improve the chances of efficiently training robots simultaneously, but would only be useful if more than two robots are used, as this would create a less constrained system that could better utilize all robots. However, with only two robots, the sequential actions of the robots are too close to the optimal solution to make a simultaneous solution worthwhile. This setup is also a good sandbox for learning different types of general sum games: one could easily design orthogonal, cooperative, or adversarial reward functions and test different algorithms to find the optimal strategy for these different games.

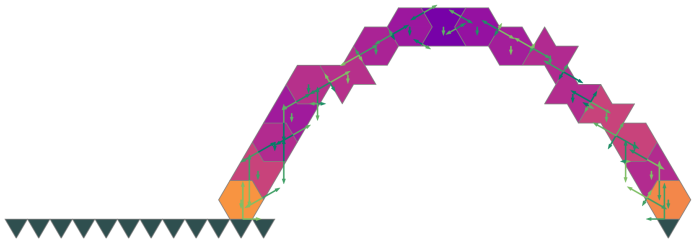


Figure 44: Structure completed by robots having a smaller maximum force

References

[1] T. J, "Brick laying machine," US772191A, Oct. 11, 1904.

[2] S. Parascho, I. X. Han, S. Walker, A. Beghini, E. P. G. Bruun, and S. Adriaenssens, "Robotic vault: a cooperative robotic assembly method for brick vault construction," *Construction Robotics*, vol. 4, pp. 117–126, Nov. 2020.

[3] Z. Wang, P. Song, and M. Pauly, "Desia: A general framework for designing interlocking assemblies," *ACM Trans. Graph.*, vol. 37, dec 2018.

[4] "Advances in achitectural geometry," (Zürich, Switzerland), vdf Hochschulverlag AG, an der ETH Zurich, Sept. 2016.

[5] U. Frick, T. Mele, and P. Block, "Decomposing three-dimensional shapes into self-supporting, discrete-element assemblies," 10 2015.

[6] B. Wibranek, Y. Liu, N. Funk, B. Belousov, J. Peters, and O. Tessmann, "Reinforcement learning for sequential assembly of sl-blocks," 09 2021.

[7] P. Song, "Interlocking assemblies: Applications and methods," *Materials Today: Proceedings*, 2022.

[8] S. Ghandi and E. Masehian, "Review and taxonomies of assembly and disassembly path planning problems and approaches," *Computer-Aided Design*, vol. 67-68, pp. 58–86, 2015.

[9] V. Bapst, A. Sanchez-Gonzalez, C. Doersch, K. L. Stachenfeld, P. Kohli, P. W. Battaglia, and J. B. Hamrick, "Structured agents for physical construction," in *ICML*, 2019.

[10] A. Zhang, A. Lerer, S. Sukhbaatar, R. Fergus, and A. Szlam, "Composable planning with attributes," 2018.

[11] B. Belousov, B. Wibranek, J. Schneider, T. Schneider, G. Chalvatzaki, J. Peters, and O. Tessmann, "Robotic architectural assembly with tactile skills: Simulation and optimization," *Automation in Construction*, vol. 133, p. 104006, 2022.

[12] A. A. Apolinarska, M. Pacher, H. Li, N. Cote, R. Pastrana, F. Gramazio, and M. Kohler, "Robotic assembly of timber joints using reinforcement learning," *Automation in Construction*, vol. 125, p. 103569, 2021.

[13] J. Randlov, "Learning macro-actions in reinforcement learning," in *Advances in Neural Information Processing Systems* (M. Kearns, S. Solla, and D. Cohn, eds.), vol. 11, MIT Press, 1998.

[14] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković, "Geometric deep learning: Grids, groups, graphs, geodesics, and gauges," 2021.

[15] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, C. Gulcehre, F. Song, A. Ballard, J. Gilmer, G. Dahl, A. Vaswani, K. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu, "Relational inductive biases, deep learning, and graph networks," 2018.

[16] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, "Modeling relational data with graph convolutional networks," 2017.

[17] X. Bresson and T. Laurent, "Residual gated graph convnets," 2017.

[18] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," 2015.

[20] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

[21] P. Christodoulou, "Soft actor-critic for discrete action settings," 2019.

[22] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018.

[23] K. Gokcesu and H. Gokcesu, "Generalized huber loss for robust learning and its efficient minimization for a robust statistics," 2021.

[24] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, "Dueling network architectures for deep reinforcement learning," 2015.

[25] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications," 2018.

[26] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent autocurricula," 2019.

A Annexes

A.1 Simulator validation and prototype

To validate the results, sometimes counter-intuitive, of the simulator, wooden pieces were laser-cut (see Figure 45). They were then placed against an inclined plane, and held in place using magnets. These

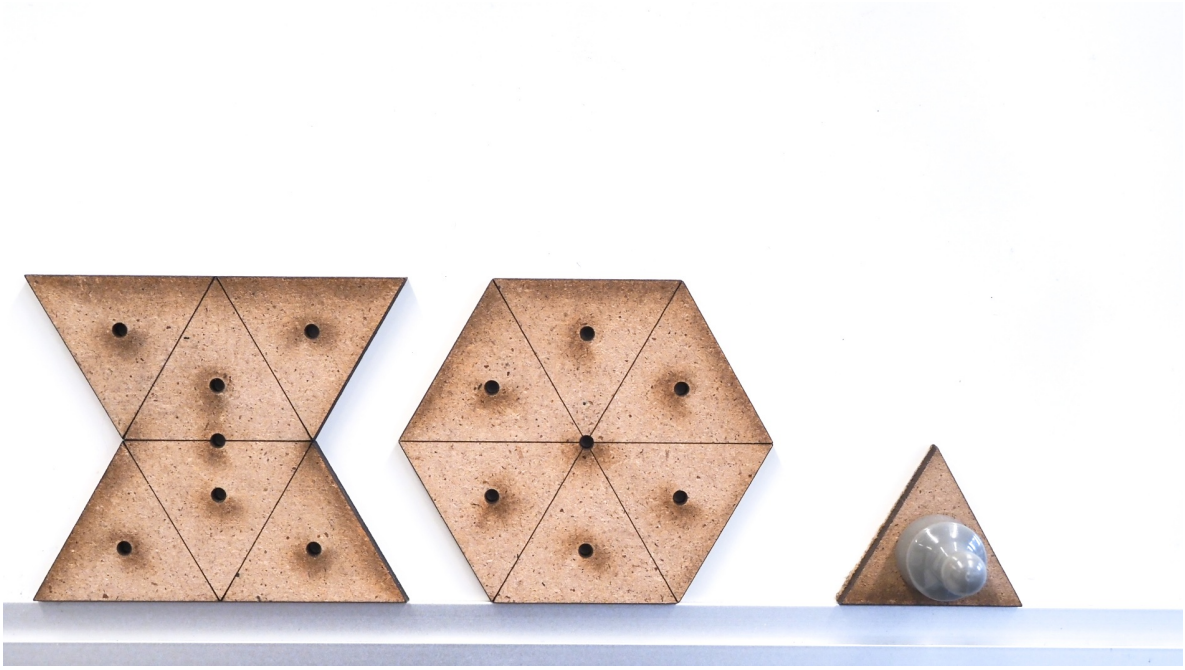


Figure 45: Prototype of blocks

prototypes were first used to estimate the friction coefficient between the blocks. As one can see on Figure 46, the friction coefficient of these experimental blocks is between 0.5 and 0.7. After thorough testing, all structures that are considered as stable in the simulator were possible to build using the real blocks. Some of the structures that broke the constraints of the simulator were still possible to build (see Figure 47), relying on 2 different reasons that were neglected by the simulator:

- Corner to corner forces: These forces were not considered in the simulator, as they are not reliable enough in real life. Some of them, however, were able to maintain the structure (see Figure 47a)

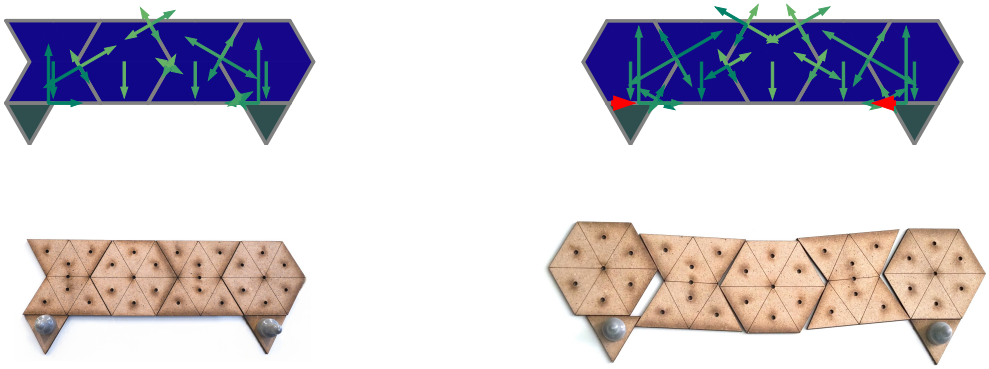


Figure 46: On the top line, equilibrium calculations from the simulator. On the bottom line, reproduction of the structures using the prototype blocks. Note that in the bottom right image, the blocks get a stable configuration by breaking the alignment

- Torque in the robot: Due to technical issues with real robots, the torque that they are applying is hard to control. To mitigate this effect, the simulator set a maximum torque of 0. The prototype, however, was perfectly able to hold of the rotation of the held blocks(see Figure 47b).

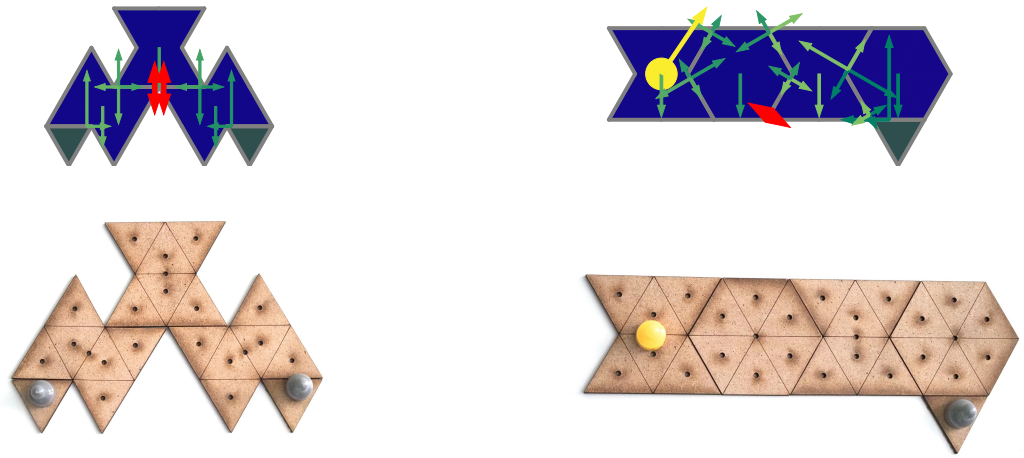
These conservative approaches used in the simulator can be justified by the fact that a real block would have its sides not as well defined as a laser cut piece, thus relying on them would be inconvenient in the long run.

A.2 Code architecture

To design the whole gym setup, an object-oriented approach was chosen. The full setup could be divided into several modules[13]:

- Block module: the aim of this module is to provide a block object to the other ones, as well as other objects used to store an existing structure. The task of verifying whether a new block would collide with some part of the structure was also done in this module.
- Physics module: the aim of this module is to compute the static equilibrium of a structure.
- Graphics module: All low level drawing task were described there, so that some high level drawing functions, like "add_frame" or "animate" could be used instead in the rest of the project
- Simulator module: this module is used to put together the three previous ones: if a block is added to the simulator, it is both added to the physics simulation, the storage object (such as a grid or a graph) and drawn without any concern. It also allows for a clear separation between the agent and its environment, and could return whether or not an action was successful or not
- Internal model module: this module was used to keep the neural network and optimizer separated from the rest of the agent. The main task of this module is to encapsulate all the pytorch functions
- Agent module: The task of this module is to allow the agent to interact with the environment, to chose an action and to update its policy.
- Gym module: This final module is used to combine the agent and the environment, and to implement the training loop.

[13]the code is available on github: https://github.com/Fllask/MARL



(a) Structure holding by using corner to corner forces

(b) Structure holding by applying a torque

Figure 47: On the top line, equilibrium calculations from the simulator. On the bottom line, reproduction of the structures using the prototype blocks. Note how despite being judged as unstable by the simulator, the real structures hold correctly

The gym object (described in the module of the same name) is tasked to initialize all of the other components. To do so while keeping a trace of all of the different changes in the setup, a configuration dictionary is used at initialisation time to give all information relevant to a given part: which learning algorithm to use, how many neurons should be used, and so on.

A.3 Continuous setup definition

A continuous setup was tested in some early trials, but was later on discarded due to the time needed to detect collisions, as well as the higher complexity of the problem. In this setup, a block could be defined as an arbitrary convex quadrilateral. A new block could be added to a structure by creating a new interface and specifying a value for its degree of freedom. Two kinds of interfaces could be created: slides and hangs. A slide defined an interface where a side of the new block was put against an existing side, and a hang is an interface where two sides of the new block each touches a different corner of the existing structure. In both cases, the degree of freedom was a number between 0 and 1 defining the relative distance of the starting corner of the new block to the end corner. In practice, the hang interfaces was disabled as its stability was lower, and only the sliding interfaces were kept

A first difficulty appears at this moment: In a stable construction, at least a block is required to create 2 interfaces. This is possible only by fixing the degree of freedom of a first interface to an exact value: if the value is different, the block would either collide with the structure or not touch it at all. An easy fix to this problem could be to use a tolerance around the exact value in which the blocks are touching without colliding, but the drawback is that this specific action would be extremely hard to find. The method chosen was instead to add a separate action simulating a force sensor. The action choice would then become:

- Place side S_b of block of type T against side S_s of the structure, with its degree of freedom at value x
- Place side S_b of block of type T against side S_s of the structure, with its degree of freedom chosen so that the new block creates a another interface

In the end, this setup was not used to train agents as the task would have been harder than on the discrete setup. As it is by nature closer to reality, and could use rectangular blocks that are more

Grid size	20×15
Blocks available	hexagons, links
Collision check	No
Gap range	1 to 10
Gap position	right
friction coefficient	0.5
Robot max torque	0
Robot max force	1000
Max blocks	30
train episodes	10000
length of replay buffer	5000

Table 1: Hyper parameters used in the setup of experiment 5.1

Grid size	15×15
Blocks available	hexagons
Collision check	Yes
Gap range	1 to 7
Gap position	center
friction coefficient	0.5
Robot max torque	0
Robot max force	1000
Max blocks	15
train episodes	40000
length of replay buffer	500

Table 2: Hyper parameters used in the setup of experiment 5.2

common in the construction industry, its usage could be considered in future works.

A.4 Hyper parameters

Grid size	30×20
Blocks available	hexagons, links
Collision check	No
Gap range	1 to 10
Gap position	right
friction coefficient	0.7
Robot max torque	0
Robot max force	1000
Max blocks	30

Table 3: Hyper parameters used in the setup of experiment 5.3

Dueling Q table	Yes
Fully connected layers	3
Number of neurons in the fully connected layer	64
Convolution layers	4
Kernel size	3
Number of internal channels (CNN)	64
Convolution stride	1
last block only	Yes
gamma	0.9
batch size	512
learning rate	0.0001
target entropy	1.8
tau	0.0005
weight decay	0.0001
initial alpha	1
learning rate alpha	0.001
Lower bound on V	-2
Optimizer	NAdam

Table 4: Hyper parameters of both CNN agents in experiment 1

Convolution layers	5
Number of internal channels (GNN)	64
last block only	Yes
gamma	0.9
batch size	512
learning rate	0.0001
target entropy	1.8
tau	0.0005
weight decay	0.0001
initial alpha	0.01
learning rate alpha	0.0001
Lower bound on V	-2
Optimizer	NAdam

Table 5: Hyper parameters of GNN agent in experiment 1

Dueling Q table	Yes
Fully connected layers	3
Number of neurons in the fully connected layer	64
Convolution layers	4
Kernel size	3
Number of internal channels (CNN)	64
Convolution stride	1
last block only	Yes
gamma	0.9
batch size	512
learning rate	0.0001
target entropy	1.8
tau	0.0005
weight decay	0.0001
initial alpha	1
learning rate alpha	0.001
Lower bound on V	-2
Optimizer	NAdam

Table 6: Hyper parameters of OD CNN agents in experiment 5.2 (both single agent and multi-agent)

Dueling Q table	Yes
Fully connected layers	3
Number of neurons in the fully connected layer	128
Convolution layers	6
Kernel size	3
Number of internal channels (CNN)	64
Convolution stride	1
last block only	Yes
gamma	0.9
batch size	256
learning rate	0.0001
target entropy (SAC)	1.8
ϵ (A2C)	0.03
tau	0.0005
weight decay	0.0001
initial alpha (SAC)	1
learning rate alpha (SAC)	0.001
Lower bound on V	-2
Optimizer	NAdam

Table 7: Hyper parameters of both OI CNN agents in experiment 5.3. The parameters used by only one of the learning algorithm have it written in parenthesis next to its name

Success	5
Failure	-2
Closer	0.4
Number of sides	0.05 per sides
Number of opposing sides	0
Place action	-0.2
Stay action	-0.3

Table 8: Reward parameter used in all experiments