Multi-agent learning of incremental housing development strategies for solar utilisation in Peru

Sergio E.M. Poco-Aguilar[1], Parag Wate[1], Darren Robinson[1]
[1]The University of Sheffield, Sheffield, United Kingdom

Abstract

Incremental housing, whereby an initial core unit is constructed and occupied by a family, who then progressively enlarge it during its lifetime, is a common form of housing provision in Peru, and many other countries in the Global South. However, these complex enlargement decisions, which place a significant financial burden on homeowners, are likely to be sub-optimal in terms of energy and cost effectiveness. To address this, we have developed a new computational workflow, combining geometry generation, multi-agent reinforcement learning and energy modelling to support incremental housing owners to optimise their enlargement decisions. In this paper, we describe this new prototypical workflow and its application to two use cases: a single housing unit, surrounded by a static neighbouring scene and a single housing unit that dynamically interacts with a changing neighbouring scene. In both cases, we arrive at stable solution that maximise solar energy availability at least cost.

Key Innovations

- Multi-agent reinforcement learning to optimise incremental housing enlargement decisions.
- At present, simply maximising solar energy availability at least cost.
- In the future, minimising operational and embodied energy use.
- Employing a computational workflow that supports a cloud-based implementation.

Practical Implications

This new computational workflow, combining geometry generation, multi-agent reinforcement learning and energy modelling has the potential to support incremental housing owners to make better informed decisions of where and how to enlarge their homes, to maximise their energy performance at least cost.

Introduction

For more than half a century now, assisted self-help has been promoted as a set of viable strategies for massive housing provision in low and middle-income countries. Incremental housing, one of those strategies, consists in providing an initial housing core to be later extended as household space needs increase and financial means become available. The case of incremental housing in Peru is particularly relevant as their assisted self-help housing policy is one of the oldest on record (Fernández-Maldonado, 2014) and to date, is still officially promoted through the *Techo Propio* programme.

During the useful lifetime of the building, most households owning an incremental house engage in various, continuous and independent self-construction processes. This form of building development is not exclusive to assisted self-help programmes, but is widely common in the global south. In fact, as of 2016, self-construction represented more than half of the total investment in the construction sector in Peru (Luna Victoria Vereau, 2016). Despite this, and even when this practice is promoted via governmental programmes, little is done to support the provision of novel, energy-efficient and low-cost technologies and techniques to reduce the environmental impact of daily living and permanent construction. This might be due to, on one side, the enormous amount of actors (when compared to industrialized practices) involved in self-construction and on the other, the complexity in time that this process involves. Construction works take decades to deliver a fully finished product and on the way deliver various semi-finished ones that might remain as such for several years. It seems clear that a bottom-up approach is needed.

Figure 1: A Street in Urbanizacion Santa Margarita, in Piura, north of Peru. In the centre-left, in white, various housing cores, the initial state of all buildings on sight.

On the other end, dwellers also face challenges if they decide to contribute by independently implementing environmentally conscious practices. This can be exemplified in the case of solar accessibility. As any household willing to invest in housing extension while maximizing their solar accessibility (for energy savings and conversion), one located in an incremental housing neighbourhood would need to look for specialized help to make an informed and profitable decision. Nevertheless, they would face two challenges particular to this type of development. On one hand, as a low-cost housing solution, the dwellers are less likely to have the necessary financial means to access this specialized help. On the other, a highly dynamic and unpredictable built environment (in which neighbours amass savings, extend their houses and create new occlusions at different moments) surrounds them. These dynamics may render sub-optimal any previously optimal solution, thus further disincentivising an environmentally wise investment.

Given these challenges, we have conceived a tool to promote environmentally wise incremental micro-investments in the residential sector in the global south, with a particular focus on incremental housing in Peru in the first instance. To guide its development, we identified three key requirements for such a tool. We believed that this tool should:

- Account for households' changing socioeconomic circumstances and how these trigger a decision to enlarge their home (R1).
- Minimise the joint objectives of financial cost and carbon emissions (and comfort) whilst satisfying the demand for additional space (R2).
- Account for the impacts of changes in neighbours' homes, as they too incrementally develop, on the target home's performance (R3)

Based on these requisites, we pose a simulation-optimization-generation loop for energy-driven incremental building development, which relies on *Multi-agent reinforcement learning* (MARL) for the optimization tasks. This algorithm couples Social Simulation and a Geometry Generation tool with a building performance evaluator. Although the final goal is for the workflow to handle the simulation of operational and embodied energy use at the neighbourhood scale, in the form of an *Urban Building Energy Model* (Reinhart & Cerezo-Davila, 2016), the present paper introduces an initial version that simply uses solar accessibility as the objective function and deals with up to two interacting household agents.

Methods

Our initial premise is to consider households involved in incremental development as agents looking to cost-effectively maximize the form of their dwellings for solar utilization. The timing and dimensions of the house extension are constrained by the available money to invest; while their solar accessibility is affected by the occlusion caused by the volumetric changes of their neighbours, who are competing for the same resource given the same limitations. Our objective then is to find the best combination of individual actions to maximize the solar exploitation of each household during the time range evaluated. This is compatible with a decentralized system in which each household has its own optimization mechanism with access to the performance values of the whole ensemble. Such a system should allow the agent to follow an optimal sequence of actions considering its own predicted socioeconomic situation while becoming aware of the physical changes in the environment.

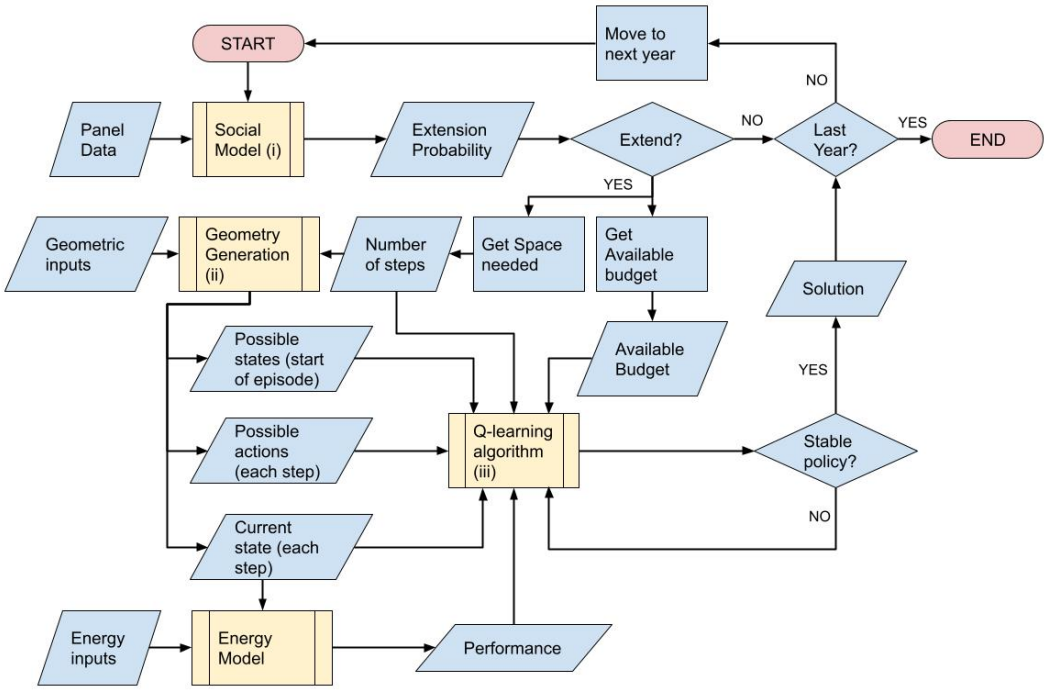

Figure 2: Proposed workflow. In yellow, main sub-processes. Next to lower-case romans, those coded specifically for the project.

The above description of our problem is compatible with a prototypical reinforcement learning paradigm (an agent, an environment and a set of states and actions). As such we decided to use *Q-learning* (Watkins & Dayan, 1992), a model-free reinforcement-learning algorithm able to find the optimal policy for any finite Markov-decision process. Being model-free, this method enables our agents to explicitly learn by trial-and-error how they can best meet their objectives.

Based on this, we proposed a workflow or loop (Figure 2) consisting of four main sub-processes or models (*shown in yellow background*), three of which are to be coded specifically for this project (*shown with lower case roman numerals on the side*) and one which is to be used as a "black box" (the energy model). The social model's (*i*) main task is to provide the probabilities of expansion that will trigger a change in the building's envelope. This will be done based on panel socio-economic data from Peru's National Household Survey (*ENAHO*). The geometry generation tool (*ii*) has two main tasks: Generate an initial geometry and transform it according to the instructions given by the *Q-learning* algorithm (*iii*). The latter (*iii*) is the core of our workflow, as the outcomes of all the other sub-process converge here. Its task is to coordinate the actions of the other components by creating, reading and updating a *Q*-table (Figure 3). Finally, the energy model receives the resulting envelope, simulates its solar exposure and produces performance results that will be input to the *Q-learning* algorithm.

In this paper, we present the integration between the Geometry generation (*ii*), the *Q-learning* algorithm (*iii*) and the energy model, leaving their interactions with the social model (*i*) for future development.

MARL algorithm

As in most reinforcement-learning algorithms, in *Q-learning* transitions between states are controlled by a set of actions. The learning process meanwhile, is iterative and composed of episodes, which end abruptly as the agent either reaches a pre-defined objective or hits a constraint. When an action is executed on a state, the agent "moves" to the next one and receives a numerical score, also known as a reward. At each time step, the agent evaluates its next action by seeking to maximize its reward, so after a period of exploration, the agent is able to determine the optimal (reward-maximising) policy.

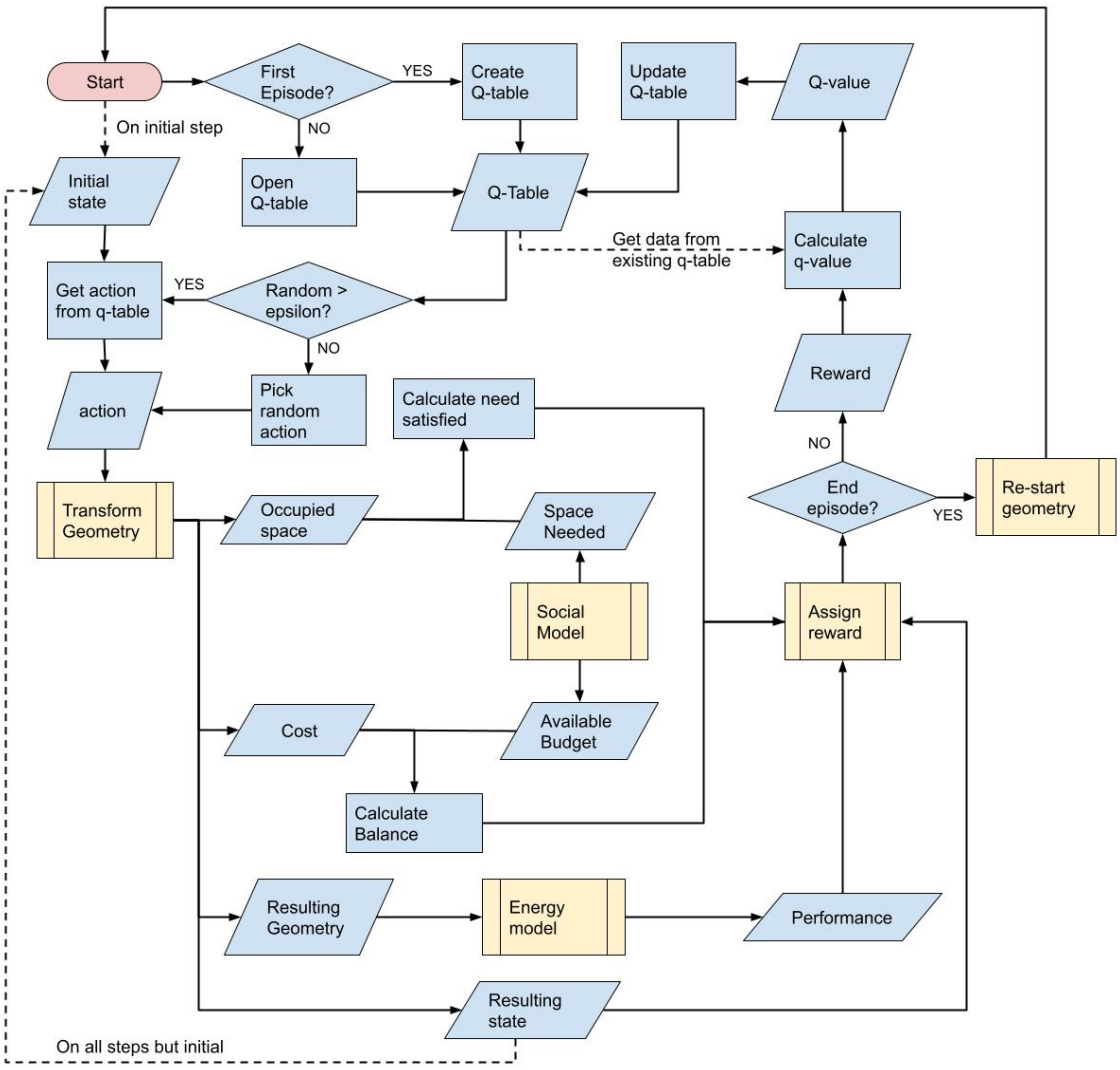

Figure 3: Process followed by the Q-learning algorithm at each time-step.

The *Q*-table is at the centre of the *Q-learning* process. This is a double-entry table where actions are columns and states are rows. The internal cells (action-state values) are the so-called "*Q*-values", which result from applying a *Bellman equation* (1). This considers the previous *Q*-value ($Q_{(s,\ a)}$), the reward obtained from performing an action on the given state (R_{t+1}), the maximum future *Q*-value for the resulting state (max $Q_{(s',\ a')}$) and the learning rate (α) and discount factor (γ). Both α and γ are a number between 0 and 1, where α determines the weighting of previous *Q*-values relative to the new ones, and γ controls how important are the long-term rewards in comparison to the immediate. As such a higher α implies a "fast learning" process, in which previous values are rapidly "forgotten", while a lower γ implies a "myopic" agent, one that weights more the reward to be received in the immediate next time-step.

$$Q_{new(s,\ a)} = (1 - \alpha)\ Q_{(s,\ a)} + \alpha\ (R_{t+1} + \gamma \max Q_{(s',\ a')}) \qquad (1)$$

As time passes and the table is updated, an optimal policy (series of state-actions that accumulate the highest *Q*-values) evolves. Nevertheless, to maximize the likelihood of finding the truly highest-performing individual, a degree of randomness must be added to the process. An epsilon (ε) value is used for this, determining the exploration (high ε) and exploitation (low ε) stages.

In a multi-agent scenario, the stationary feature of the environment that allows a single agent's actions to converge to an optimal value is violated. To overcome this, a slightly more sophisticated *Q* function, which includes knowledge of the actions of the other agents, is needed. As this scenario might include a large number of agents, computing the actions of all the others becomes inefficient and unlikely. Selecting only those local neighbours, whose behaviour does affect a particular agent's performance, makes *Q* MARL possible. The *Q*-table meanwhile takes the joint action of all the other agents along the state of the current agent to select individual actions and update *Q*-values.

Geometry generation model

Our geometry generation model creates a '*HHagent*' object (Figure 4) with the inputs of locations and dimensions of both the lot and the original incremental housing unit. A *module* input is also needed. This is to both facilitate the generation of unique *state* identifiers and to represent the spatial and structural configuration of a residential unit. As such, this input represents a side of one of the equally sized cubes that conform to the maximum buildable space within a lot. All dimension inputs are in terms of a number of modules.

For the *Q-learning* algorithm to work, the geometric solution has to communicate its own *state*, along with the current cost of extending, and its solar performance to the *Q-learning* algorithm. The performance is sorted via the '*OutBrep*' property of the agent class, which outputs the geometric envelope of the building in the form of a boundary representation (*B-rep*). Once this is created, it can be passed to the energy model, which assesses it and outputs its performance. The state self-identification is sorted by using a list of points at the centre of each of the modular cubes defining the buildable space. The agent identifies its own state with a string formed by '*T*'s and '*F*'s which come from evaluating each point as being inside (*true*) or not (*false*) of the current envelope. By counting how many points lay inside, it informs its size (in *modules*) to the *Q-learning* component, so the latter can determine if it meets the space requirements given by the social model. Finally, by calculating the area of new wall and roof surfaces and receiving inputs on the cost of building per sq. meter of wall and roof, the agent calculates the cost of building beyond its initial state.

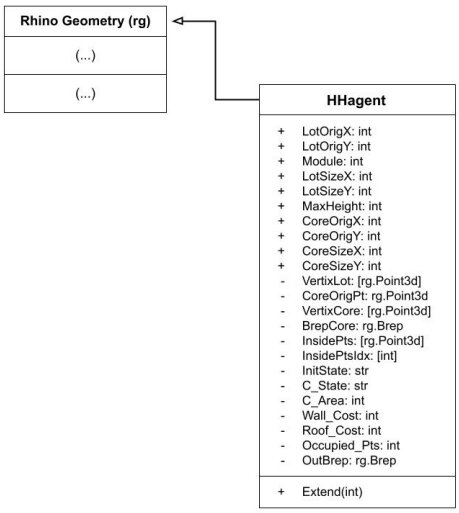

Figure 4: UML class diagram for the HHagent class.

By calling the function *Extend()*, which takes a positive integer representing the *action* to be performed as an argument, the geometry extends and occupies one additional module. The input integer represents an index in the list of faces available to grow, which is a filtered list of all the faces of the *B-rep*. This list is also re-ordered according to the closest centre point of the cubic modules, so the jump from one initial state (s_n) to another (s_{n+1}) is ensured every time a given action (a) is taken, disregarding the sequence of actions that lead to s_n on the first instance. Exceptionally, the *Extend()* function also takes the *actions 0* and *-1*. The earlier determines that during this time-step the geometry should remain the same, and the latter indicates that the episode is finished and we must reset the geometry to its initial state to proceed to the next episode.

The *Extend()* function works differently depending on whether the face to extend is a wall or a roof. When faced with a wall (it only takes the ones on the ground floor) it first recreates the footprint of the building, then calculates the central point of that polygon, projects it to the polygon side where the face lays and creates a vector between the two points. It re-sets the length of the vector to the same of a module and then uses it to extrude the face in question (Figure 5a). It finally deletes the original face and detects if any of the new faces have the same vertices of an existing face. If so, it deletes them. For a roof, it does the same, except that the vector guiding the extrusion is defined as the absolute difference between one of the ground vertices of the *B-rep* and its copy with *Z* equal to the *module* (Figure 5b).

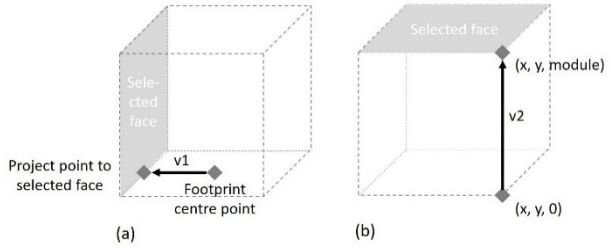

Figure 5: Generation of vectors for extrusion as part of the Extend() function. To get its dimension, v1 is unitized and multiplied by the "module" input. V2 already shows its correct direction and dimension.

Due to the iterative extension process, an initially solid *B-rep* (closed and non-manifold) might eventually stop being so. Solid *B-reps* have the capacity of automatically detecting the orientation of their faces. Keeping all faces outward-looking ensures the accuracy of the energy modelling. When the *B-rep* opens, the geometry generation algorithm converts it to a *Mesh*, heals its naked Edges and reconverts it to an output *B-rep*. When we get a *non-manifold B-rep*, as there is not a geometrical fix, it is sent as it is to the *Q-learning* algorithm, which detects its state and gives it a maximum penalization so we avoid falling into it again.

Interaction between the models

The typical process of the workflow is the following: The social model iterates over the probabilities of expansion of each year for a determined number of years. These probabilities are determined from a number of factors such as available savings, the number of household members, etc. and are unique to each household. If the model determines that in a given year a particular dwelling should expand, it sends a signal to the *Q-learning* algorithm (*iii*), with the available budget and need for space, to start the expansion process. By taking these inputs and the initial geometry, the agent occupies one additional *module* per step until it has satisfied the need for space (in *modules*) or depleted the available resources. It repeats this process for a given number of episodes until it reaches a stable solution, indicated by a flat line at the high end of the performance (resulting from the energy modelling) during a number of episodes. This solution is saved as the outcome for that year and replaces the initial expandable dwelling the next time the *Q-learning* process is activated.

In a multi-agent learning scenario, the joint performance of the dwellings forming part of an incremental housing neighbourhood is shared among the individual agents. The goal is to coordinate individual actions to minimize the environmental impact of the operational and embodied energy use at the neighbourhood scale.

Given the complexity of our task, our workflow will demand high computational power. The reinforcement repetitions imply various sub-tasks, like calculating the geometry, evaluating its performance and calculating the next state-action to be taken. This, multiplied by the number of agents in the neighbourhood implies long waiting times for a personal computer. Because of this, our workflow had to be mindful of a possible integration with local or cloud-based *High-Performance Computing* (*HPC*). While the market currently offers several solutions to implement this, we found that using *Rhinoceros 3D* (*Rhino*), a commercial 3D computer graphics and *CAD* application software, is well suited to this task. While at earlier stages of development we can operate locally making use of the less demanding environmental analysis plugins available for *Grasshopper* (Rutten, 2010), eventually, we can take our geometry generation tasks to a cloud production environment using *Rhino.Compute*, a web server that provides an API for geometry calculations using *Rhino*'s geometry library.

With this in mind, our workflow is written using the popular object-oriented programming language *Python*, which both *Grasshopper* and *Rhino.Compute* are able to understand. At this trial stage, we use the *Ladybug Tools* (Roudsari et Al., 2013), a *Grasshopper* component to analyse weather data and to perform solar irradiation computations, as our energy model.

Case Study

To test the interaction between these components, we designed two experiments. In the first one, we tested single-agent reinforcement learning while in the second we employed MARL.

The single agent started with a single-module extendable housing unit. This unit was located in a lot surrounded on three of its sides by fixed static buildings with a height of two *modules* (Figure 6). The size of the lot was 2*3 *modules* and the agent could build up to three *modules* in height. Each *module* was fixed at 3 meters per side. As previously noted, the agent's objective is to maximise its solar accessibility without depleting its financial resources. A particular challenge of the setting came from the fact that the open side of the lot was oriented southwards. As we simulate a southern hemisphere location (-16.32 latitude, -71.55 longitude, 2520 m.a.s.l.) in wintertime (21/03 until 23/09), the agent had to find a policy that allowed it to surpass its shadowing neighbours. For this experiment, the agent was given a fixed budget of 15k monetary units (MU). Building one sq. meter of additional walls costs 100 MU, while every additional sq. meter of roof costs 150 MU. There was a pre-defined need for space of 6 occupied *modules* in total (including the original building and its expansions).

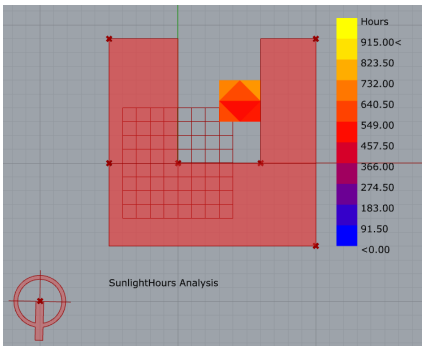

Figure 6: Initial state of the housing core for experiment one. Surrounding buildings on red, core coloured with the analysis palette.

The pseudocode for the reward function used in this experiment can be seen in Box 1. The simulation was configured for 500 episodes and the maximum number of steps per episode was 10, although the agent finishes the episode before completing the ten steps, as per the reward function (space need and money availability). Finally, the inputs for the bellman equation were: The *learning rate* (α) started at 0.99 with a decay of 0.99976 per step, while the *discount rate* (γ) was fixed at 0.9 (a non-myopic agent). *Epsilon* (ε) started at 0.99 and had a decay of 0.9982 per step. These parameters are consistent with an epsilon-greedy configuration, where we start with an exploration phase in which new record performances are discovered and a high α is needed to "forget" rapidly the outgrown solutions. The agent needs to value future rewards more than the immediate, as the goal will only be reached at the last time-step.

In our second experiment, we repeat the settings from experiment one but replace the static neighbouring building on the west of the initial agent with another agent. In this scenario, both agents contribute to maximising the performance of only the initial one. Therefore, the second agent, despite having its own *Q*-table, just acts to benefit its neighbour. The reward function is repeated from experiment one but adds the evaluation of the *non-manifold* state for agent two. As per the reward function, agent two does not need to know its own need for space or keep an eye on its budget. The only condition that emanates from agent two that ends an episode and gives a penalization is when it falls in a *non-manifold* state.

In this second experiment, to reduce the processing time and computational demands, the agents were asked to perform a simpler task: the space needed by agent one was reduced to four modules in total and the budget was consequently diminished to 10.5k MU. *Epsilon* (ε) was fixed at 0.05 to contribute to this effort, while the *learning rate* (α) was initialized at 0.99 with a decay of 0.99984 per step. The *discount rate* (γ) was kept exactly as in experiment one.

Box 1: Reward function pseudocode for experiment one.

```
#### First we create a dictionary to keep record performances per number of occupied modules
Records = {} # This creates an empty dictionary
For point in occupied_points: # Fills the dictionary with possibly occupied modular cubes at each step
        PerfomancesDictionary[voxel] = 0 # Initial record is 0. This is updated iteratively.

##### Then we get the Boolean conditions #####
If Agent.Performance >= Records[Agent.OccupiedModules]: # If we break the record
        Records[Agent.OccupiedModules] = Agent.Performance # Update record dictionary
        record_broken = True # Boolean condition 1
If Agent.NewlyOccupiedVoxels >= SpaceNeeded: # If we build as much or more than we need
        filled_space = True # Boolean condition 2
If Agent.State is NonManifold: # The geometry is non-manifold
        Non_mani = True # Boolean condition 3
agent_balance = Agent.Budget – Current_Expenses # Get Current balance

##### Finally, we test the conditions and assign the reward #####
If record_broken and filled_space: # We occupy as much as we should and break record performance
        reward = GOAL_REWARD # Receives max reward
        Episode.EndEpisode() # End Current episode
elif not record_broken and filled_space: # We occupy as we should but don't break the perf. record
        reward = -GOAL_REWARD # Receives max penalization
        Episode.EndEpisode() # End Current episode
elif agent_balance < 0 or Non_mani: # We get out of funds or the state is non-manifold
        reward = -GOAL_REWARD # Receives max penalization
        Episode.EndEpisode() # End Current episode
else:
    reward = -TIME_PENALTY # Receive time penalization to promote fastest solution
    Episode.NextStep() # Proceed to next step
```

Results

Our first experiment showed promising results. Figure 7 shows how as ε diminishes, a high-reward policy stabilises. Only a few remains of random solutions in the last episodes interrupt the flat line on the highest end. A completely flat line will be only possible if ε was an absolute zero, as just one incorrect (random) action taken in a previous time-step is enough to block the way to the optimal solution. Figure 8 shows how the policy leading to the highest recorded performance is found in the exploration phase (higher ε), and it is later recovered in the exploitation phase.

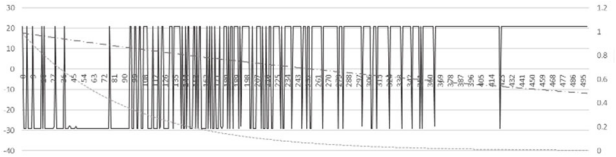

Figure 7: Reward accumulation per episode for experiment one. Upper dotted line represents average γ lower does average ε.

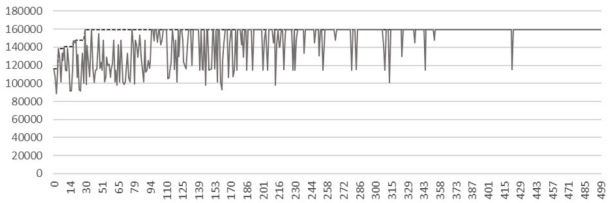

Figure 8: Performance per episode (in sunlight hours per year) on largest possible development stage for experiment one. The upper dotted line is the highest performance on record.

Figure 9 shows reward accumulation per episode for experiment 2. We can notice that it took fewer episodes for the agents to find an optimal joint policy to maximize the performance of agent one. This is probably because their task is less demanding than previously. We can also notice how in such a small search space, the agents find a maximum performance quite early (Figure 10). In a single-agent scenario, with such a low ε, we would expect the agent to follow the highest rewarding policy immediately after finding a record performance. Nevertheless, in this scenario, we observe that after reaching the record, the performance wanders for several episodes before recovering the highest performance on record. This could be the result of the search it takes to recover the combination (own state-others agent's joint action-own action) leading to maximum reward. Only when both agents find the right combination for a second time, the high *Q*-values can start back-propagating. In this search space, this is a straightforward process; nevertheless, it might take considerably more time for them to find it when the goal is several steps ahead of the initial state and there are more agents to coordinate. A possible solution could be to distribute partial rewards on the intermediate steps, so the *Q*-values backpropagate earlier.

Other adaptations would need to be considered when the workflow tackles situations that are more complex. Nevertheless, these initial experiments indicate that this initial implementation of the multi-agent *Q-learning* algorithm is a promising start.

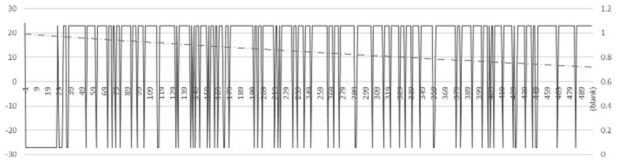

Figure 9: Reward accumulation per episode for experiment two. Dotted line represents average γ.

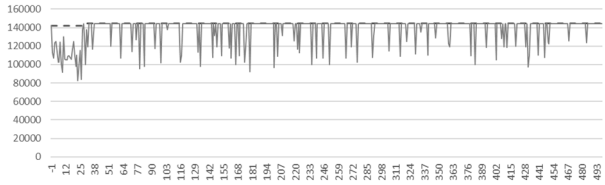

Figure 10: Performance per episode (in sunlight hours per year) on largest possible development stage for experiment two. The upper dotted line is the highest performance on record.

Conclusions

We have presented here an initial version of our simulation-optimization-generation loop for energy-driven incremental housing development, which relies on a MARL algorithm.

We aimed to develop a tool that was able to comply with three basic requirements. It should be able to: account for households' changing socioeconomic circumstances and how these trigger a decision to enlarge their home (R1); minimize the joint objectives of financial cost and carbon emissions whilst satisfying the demand for additional space (R2); account for the impacts of changes in neighbours' homes on the target home's performance (R3).

Although the presented workflow aims at tackling these three requirements, the present paper only presents progress with respect to R2 and R3. The social model (*i*), which will tackle R1, is still under development.

For the time being, we have shown that by coding a multi-agent *Q-learning* algorithm, taking space needs and budget availability as inputs, we are able to generate optimal building geometries as per the evaluation of a simple energy model. As such, we feel that we are on track to meet our requisites two and three.

Nevertheless, there is still a long way to go. First, the social model must be implemented and tested in conjunction with the already developed sub-processes. Second, a larger number of agents must be included to test the resilience of the workflow. And third, the building performance must be evaluated using a more sophisticated model, which would require further input data and more complex geometrical modelling of the dwellings.

References

Fernández-Maldonado, A. M. (2014). Incremental Housing in Peru and the role of the Social Housing Sector. In Bredenoord, J., Van Lindert, P., & Smets P. *Affordable Housing in the Urban Global South: Seeking Sustainable Solutions.* Routledge. New York (USA).

Luna Victoria Vereau, A. (23th of July, 2016). TMS: Caída de 7% en autoconstrucción se debe a baja de ingresos. *El Comercio*. https://elcomercio.pe/economia/peru/tms-caida-7-autoconstruccion-debe-baja-ingresos-221378-noticia/

Reinhart, C. F., & Davila, C. C. (2016). Urban building energy modeling–A review of a nascent field. *Building and Environment(97)*, 196-202.

Roudsari, M. S., Pak, M., & Smith, A. (2013). Ladybug: a parametric environmental plugin for grasshopper to help designers create an environmentally-conscious design. *Proceedings from the 13th international IBPSA conference*. Chambéry (France), August 25-28

Rutten, D., & McNeel, R. (2010). *Grasshopper*. Robert McNeel and associates.

Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine learning, 8(3)*, 279-292.