Constraint-Procedural Logic Generated Environments for Deep Q-learning Agent training and benchmarking

Stefania Costantini[1], Giovanni De Gasperis[1] and Patrizio Migliarini[1]

[1]*Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica, Università degli Studi dell'Aquila*

Abstract

While training and benchmarking neural networks, a large and precise set of data and an efficient test environment are parts of a successful process. A good data set is usually produced with high effort in terms of cost and human work to satisfy the constraints imposed by the expected results. In the first part of this paper we focus on the specification of the properties of the solutions needed to build a data set rather than using common primitives of imperative programming, exploring the possibility to procedurally generate data-sets using constraint programming in Prolog. In this phase geometric predicates describe a virtual environment according to inter-space requirements. The second part is focused to test the generated data set in a machine learning context by means of an AI gym and space search techniques. We developed a deep Q-learning model based neural network agent in Python able to address the NP search problem in the virtual space; the agent has the goal to explore the generated virtual environment to seek for a target, improving its performance through a reinforced learning process.

Keywords

procedural generation, constraint programming, deep Q-learning, intelligent agent, unknown ambient exploration, neural network training, neural network benchmark, neural network gym, AI gym

1. Introduction

The ability to efficiently train and benchmark a deep learning neural network to solve hard tasks such as, speech processing, image recognition, image classification etc. has grown significantly. While those tasks require large - but available - data sets and powerful computing resources, tasks like environment exploration are more difficult to train and benchmark due to lack of available data sets and the long time needed to compile one by hand. To overcome these limitations, we propose a procedural generator of virtual environments based on a logic constraint solver. In the second part of this work, a deep learning explorer agent will be located in the resulting simulated environments to be trained to search for a specific target.

1.1. Related work

Procedural generation is a method of algorithmic data creation widely associated with the world of computer graphics and video games. This context is generally described as Procedural

CILC 2022: 37th Italian Conference on Computational Logic, June 29 – July 1, 2022, Bologna, Italy

stefania.costantini@univaq.it (S. Costantini); giovanni.degasperis@univaq.it (G. D. Gasperis); patrizio.migliarini@graduate.univaq.it (P. Migliarini)

0000-0002-5686-6124 (S. Costantini); 0000-0001-9521-4711 (G. D. Gasperis); 0000-0002-7824-529X (P. Migliarini)

Generated Content (PGC) *"PCG is the algorithmic creation of game content with limited or indirect user input"* [1]; in this work we focus on the generation of 2D geometric spaces representing a simplified house plan, as shown in Section 2. The procedural generation phase gives us a diverse set of environments which cannot be easily obtained manually. This set can be used to train and benchmark an explorer agent without any given a-priori data set. The PCG is based on search and/or optimization algorithms - i.e. evolutionary algorithms - to find the solution that best satisfies an evaluation metric, which consists of a function that associates each individual component of the generated content with a metric that contributes to the quality of the solution.

1.1.1. Procedural generated spaces

Since we intend to simulate the exploration of an autonomous agent immersed in a virtual space, we focus on procedural generated spaces, i.e., geometrical description of 3D volumes, or 2D partitions that can be assimilated to navigable indoor spaces such as dungeon maps, rooms or house plans. We can find examples of procedural generated spaces in architecture [2]. The dynamic generation video-games dungeons as described in [3] inspired our approach: generate environments that consist of a given number of rooms, connected by a given number of corridors, all enclosed by walls that delimit the possible space.

1.1.2. Constraint generation

Generating content using a constraint problem can be done either using an imperative method [2] where house plans are generated hierarchically over a discrete grid, or using a declarative approach [4] by means of an answer set programming constrained program.

1.1.3. Deep Q-learning for spaces exploration

Deep Q-learning (DQL) is an implementation that substitutes the state-action look-up-table of the classical Q-learning algorithm [5] with a deep learning neural network [6]. It has been recently applied to ambient exploration [7] showing that it is possible to explore unknown environments by an agent that received in input a low resolution image from an on-board simulated camera in the 3D space, while exploring the environment.

2. Geometrical description of 2D environments

Differently from conventional dungeon generation, we opted to avoid internal corridors to reduce the complexity of the generated house plans. So, we introduce a simplification about the inter-rooms connection by having rectangular rooms connected with doors all connected to a single central shared room, as shown in Fig. 1. Also, in order to render a more realistic environment we generated a selection of typical home furnishings and their position inside the rooms, taking into account rules like: a bed shall not stand in the middle of the room, a closet shall not impede doors and windows.

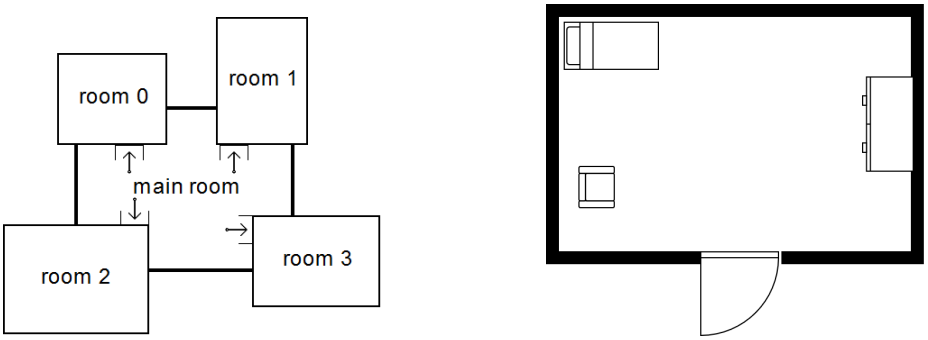

Figure 1: Reference model of the generated rooms set. Left: the room interconnection via the central main room. Right: furniture distribution inside a room of type "bedroom" with a bed, a sofa and a closet.

3. Geometrical Constraints Generation

We defined a set of geometrical constraints that the final virtual house plan has to comply with: room size, room type (bedroom, dining room, bathroom, kitchen), furniture position depending on the room type, allocation of all needed room types to have a complete house.

The set of constraints used to generate the 2D virtual house plans can be summarized as follows:

- the house is made of a central room that connects all the secondary rooms by doors.
- each secondary rooms can have only 1 door.
- all secondary bounding boxes shall not overlap, i.e. have an empty space intersection
- the area of secondary rooms shall never be larger than the main room
- a house shall contain at least 2 rooms of basic types (bedroom, bathroom, kitchen); a shared central room is always generated
- furniture can be positioned only in the secondary rooms
- furniture items have to be compatible with the room type they are in

The rectangles describing the rooms are defined by top-left and bottom-right vertex coordinates in a 2D continuous space.

The problem has been coded in CLP(R) Prolog, *Constraint Logic Programming with Real numbers* [8]; the code is available at the GitHub repository [9], were a full example of the generated Prolog code can be seen at https://github.com/AAAI-DISIM-UnivAQ/bd-procedural-env-deep-learning/blob/master/environments/example1.pl. The constraint generator Python program is called `Main.py` which handles interaction with the user to collect her preferences in terms of number of rooms, type, number and types of furnishings. It then generates the Prolog knowledge base in function of user preferences adding constraints rules; via the PySWIP library it submits the query to the SWI-Prolog interpreter. An example of the generated query is available in the source code repository.

```
generateEnvironment(EnvWidth, EnvHeight, R0X, R0Y, R0W, R0H) :-
   repeat,
```

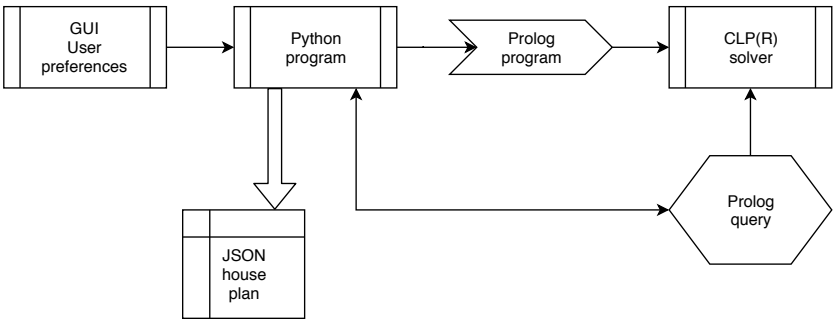

Figure 2: Data flow to obtain the constrained generated virtual house plan: 1: the user fixes the number and type of rooms, with furniture preferences, 2: the Python program generates Prolog rules with CLP(R) constraints expressions, 3: the SWI-Prolog interpreter is invoked and queried, 4: grounded variables from the query result are used to generate the final JSON geometrical house plan description.

```
random(100.0, 145.0, R0W),
random(100.0, 145.0, R0H),
WSUB0 is EnvWidth - R0W,
random(0.0, WSUB0, R0X),
HSUB0 is EnvHeight - R0H,
random(0.0, HSUB0, R0Y),
!.
```

The generated Prolog code above shows the random generation of a room by its origin coordinates (R0X, R0Y) and size (ROW, ROH), respecting the overall space limits (EnvWidth, EnvHeight). It keeps generating random rectangles until an acceptable solution is found according to the following CLP(R) constraints definitions:

```
{(30.0 =< B0X ; B0X + B0W =< 9.5) ;
  (200.0 =< B0Y ; B0Y + B0H =< 175.0)},
{(30.0 =< BS0X ; BS0X + BS0W =< 9.5) ;
  (200.0 =< BS0Y ; BS0Y + BS0H =< 175.0)},
{(30.0 =< W0X ; W0X + W0W =< 9.5) ;
  (200.0 =< W0Y ; W0Y + W0H =< 175.0)},
{(W0X + W0W =< B0X ; B0X + B0W =< W0X) ;
  (W0Y + W0H =< B0Y ; B0Y + B0H =< W0Y)},
{(BS0X + BS0W =< W0X ; W0X + W0W =< BS0X) ;
  (BS0Y + BS0H =< W0Y ; W0Y + W0H =< BS0Y)},
```

This code portion is about the generation of just one bedroom, with bed coordinates starting with B, commode starting with B, closet starting with W. All furniture constraints are in the form described by the formula:

$$\begin{cases} X_{door} + W_{door} \leq X_{obj}; \\ X_{obj} + W_{obj} \leq X_{door}; \\ Y_{door} + H_{door} \leq Y_{obj}; \\ Y_{obj} + H_{obj} \leq Y_{door}; \end{cases} \tag{1}$$

where X and Y are coordinates and W and H are the width and height of the *doors* and *objects* (*objs*) respectively.

The first case represents a situation in which the door of the room is positioned horizontally while, in the second, the door is positioned on a vertical wall. The sub-cases instead represent, in order, the situations in which the object is completely to the right or to the left of the horizontal door or completely above or below the vertical door. It can easily be verified that the simple disjunction of disjunctions ensures the non-overlapping of the elements of a room with its door. A typical graphical result, with much more rooms and constraints, can be seen in Fig. 3 . The colored rendering is the result of a Python/Pygame [1] program that reads the JSON described solution and visualizes it on the computer screen. The temporal complexity of the algorithm is clearly exponential, given that it explores an infinite space of solutions by evaluating the correctness of each one individually. This exploration is based on a random seed, so although the algorithm converges to a solution in a short time for most runs, we cannot currently rule out rare cases where a conforming solution is never found. Furthermore, Prolog queries for the various types of rooms look for a solution that simultaneously satisfies the criteria of each room of that type: this means that, by increasing the number of rooms, the time complexity of the algorithm also increases considerably. In particular, using simple probabilistic terms, called the p probability of finding the solution for a single room, the probability of finding a solution valid for two rooms of the same type is, p^2. More generally this probability is p^{n_i}, with n_i is the numbers rooms for each room type. It is clear that a non-deterministic Turing machine could return any of the valid results in polynomial time as it can attempt every possibility simultaneously, therefore the algorithm can fall into the class of NP-hard problems.

```
{ "roomNumber" : int,
              "floor" : { "x": float, "y": float,
              "width": float, "height": float },
    "RX" : { "x": float, "y": float,
                "width": float, "height": float,
                "type": "bedroom | bathroom | kitchen | hall",
            "children" : [{ "x": float, "y": float,
                "width": float, "height": float,
                "type": "bed | bedside | wardrobe…",
                "orientation": "W | N | E | S" }]
    }
}
```

[1] http://www.pygame.org is Python library optimized in fast screen rendering widely used to implement 2D video games.

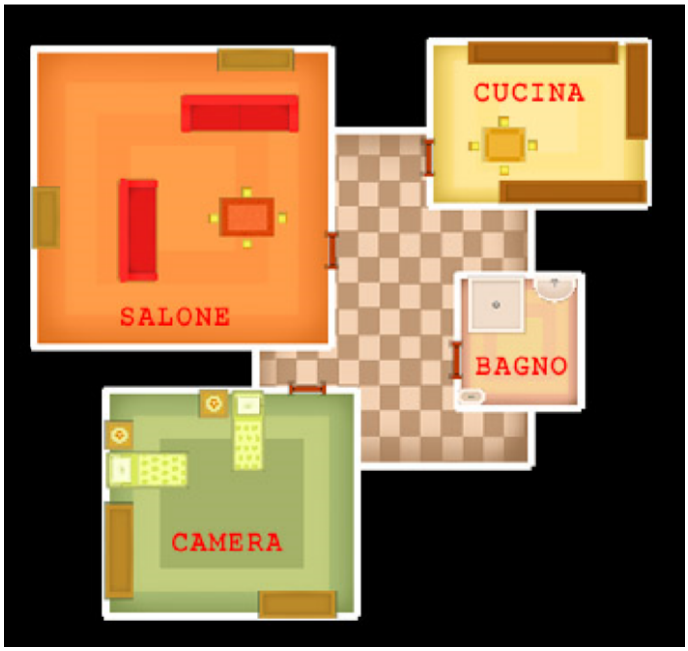

Figure 3: An example of a house plan generated by CLP(R) constraint program. Colored in green is the bedroom, pink is the bathroom, orange is the dining room, yellow is the kitchen.

The above code is the typical JSON definition of a room that is generated by the CLP(R) Prolog system that we called *CoPLEnG* (Costraing-Procedural Logic Environments Generator). Such code is then fed to the simulator were the DQL agent lives in for it to explore.

4. Deep Q-learning explorer agent

The agent starts off in his exploration phase being instantiated by the Python/Pygame program in gym-simulator which takes care of collisions with walls and obstacles and generating sensors data at the agent input layer. The agent has an input array of 40 simulated optical sensors deployed in the front part of its body, in a 120 degree view window; we call them rays, each one measuring the distance to the obstacle/object were is pointed at. Given the 3 possible actions the agent can perform in the environment (rotate left, go straight, rotate right), a totally random selection would have resulted in a 66% chance of rotating against 33% of moving. The observed pattern with these odds was that the agent spent more time wandering around his spawn point rather than exhibiting exploration-oriented behaviours right from the start. So we favored the moving primitive by shifting its chance to as much as 93%, with the rest split among the other 2 actions. This simple change made a huge difference as this modified random agent was now able to move long distances while turning left or right every once in a while, effectively granting a good realism and variability in the forthcoming inputs. In an attempt to raise the neural network from learning unnecessary patterns and thus simplify the model, we have wondered if there were any simpler problems - in fulfilling the main objective - that we could solve in a more mechanical way. Avoiding obstacles and walls was a task which met such description. So,

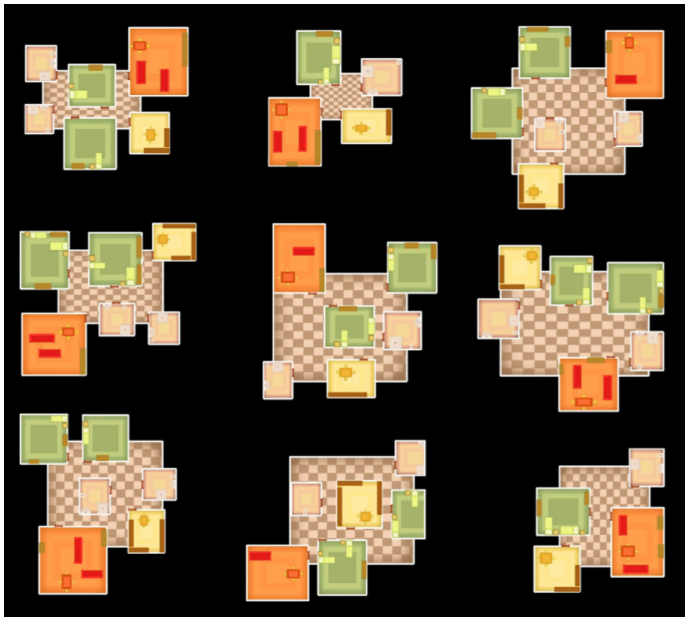

Figure 4: A set of generated home-like spaces derived from the constraints.

drawing inspiration on the Subsumption Architecture introduced by Rodney Brooks [10], we implemented a scripted behaviour - called “Avoidance” - which triggers when the minimum distance perceived in the rays gets lower than an empirically calculated threshold. When it does, it forces the agent to rotate to avoid the incoming collision, purposefully ignoring the prediction from the neural net. With these two mechanics combined, the agent is free to roam with no danger of collision from the beginning and can solidify this initially random behaviour in a predicted one.

4.1. Neural network model

The structure is composed of 7 layers, which follow a so-called “diverging-converging” pattern: the neurons per layer increase in quantity up to half the network and then shrink down to the output layer, whose population is defined by the number of primitive actions, as shown in [11] to have a good compromise between the total number of internal weights, the generalization capability of the neural network and learning and testing performance. The first layer is connected to the input through a pre-processing module; it consists of a vector of 40 tuples bearing the contribution of each ray projected by the agent in the environment. Following is a utility layer (without neurons) called “Flatten” which is particularly useful where the input to the network should consist of a multidimensional vector as it is capable of “spreading” data along a single one-dimensional array to avoid sending through the various layers of “heavy” data to read. The following hidden layers are of the type dense standard and are all activated by the Rectifier Linear Unit (RLU). The function of activation of the output layer is the softmax since the agent shall deliberate over a single motion action . The loss function is the MSE

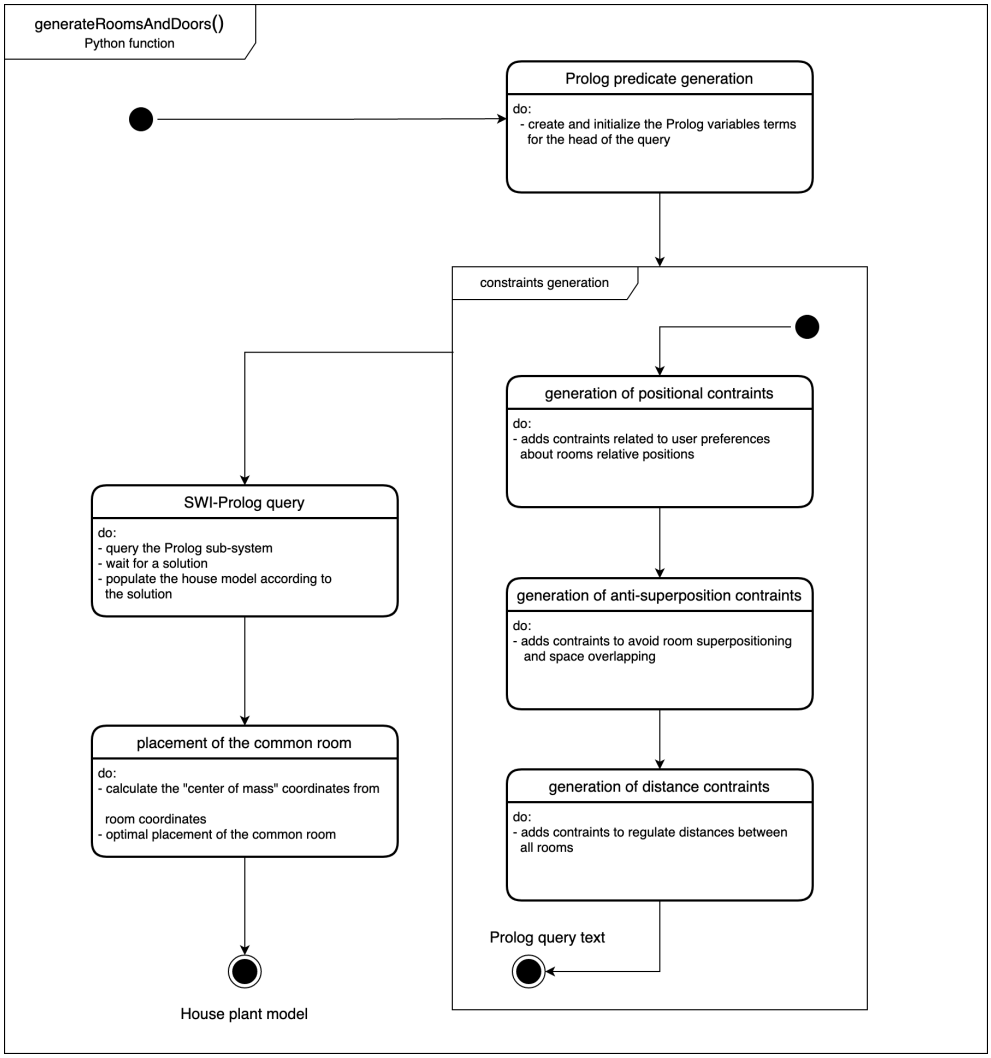

Figure 5: The overall flowchart of the generation process performed by the Python function `generateRoomsAndDoors` which, starting from the user preferences describing the expected house plant model, it generates the appropriate query to submit to the SWI-Prolog/CLP(R) interpreter and solver.

(Mean Square Error), in accordance with the formula derived from the Q-Function on which the network optimizes. The deep learning algorithm is driven by the Adam stochastic gradient optimizer [12].

4.2. Performance metric definition

A simple performance metric could be defined by counting the number of targets reached in a finite time interval from its starting position, even if the agent receives commands by a human. We found that, in this kind of generated environments, the performance the human can achieve

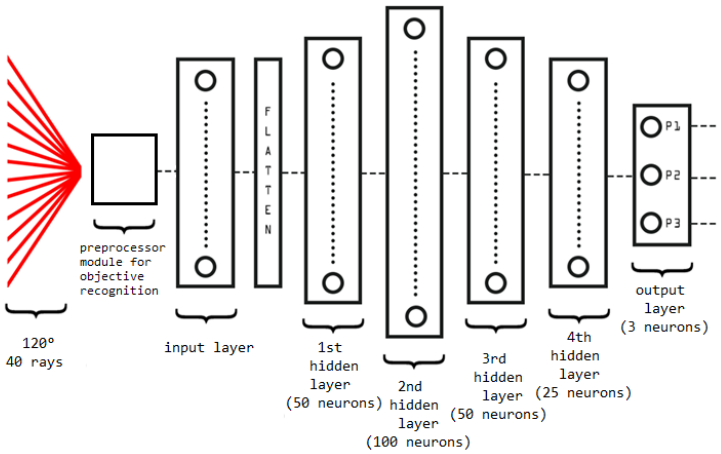

Figure 6: Deep learning neural network model adopted the controller of the explorer agent. Inputs are proximity distance sensors readings, output is direction of motion or rotation.

is - in average - 27 targets in 15 seconds while searching in the main room, and about 10 targets while searching for them in secondary rooms. In this way we introduced a global behavioural metric to judge the agent performance, not just looking at each single action move.

4.3. Training method

The explorer agent is trained by a reinforcement learning algorithm in function of its performance while exploring the generated environment, as shown in Fig. 7. To further help the model converge, we make use of the well known Remember & Replay method [13]. Experiences are first stored in the agent's memory in the form of tuples containing the information pertaining to a single transition from one observation to the next. The tuple structure is $(state, action, reward, next\_state, done)$ where done is a flag indicating whether the episode has ended or not; this is useful to check if there was any more reward achievable in that time step or not. After an episode ends, a random batch of experiences are sampled from the memory and fed to the neural net for learning. Reward is 1 for each transition in which the agent managed to gather an objective.

5. Results: trained agent performance

The trained agent showed excellent capability to reach the target in the main room, always performing better than the human pilot. On the other, hand statistically it fails to reach the target that is positioned inside the secondary rooms. When the target is visible from the main room it can achieve to reach 1 target in 15 seconds. A trained agent behavior example has been recorded in the animated GIF image inside the software repository [9] .

```
env = loadEnvironment(file_name)
agent = SLAMRobot()
while i = 0 < episodes
   env.reset()
   state = env.projectSegments()
   done = false
   step = 1
   while not done
      action = agent.act(state)
      env.update()
      reward = env.assignReward()
      next_state = env.projectSegments()
      agent.remember(state, action, reward, next_state, done)
      state = next_state
      if step++ == time_steps or env.checkCollision()
         done = true
   agent.replay(batch_size)
```

Figure 7: The pseudo-code for the reinforcement learning of the explorer agent.

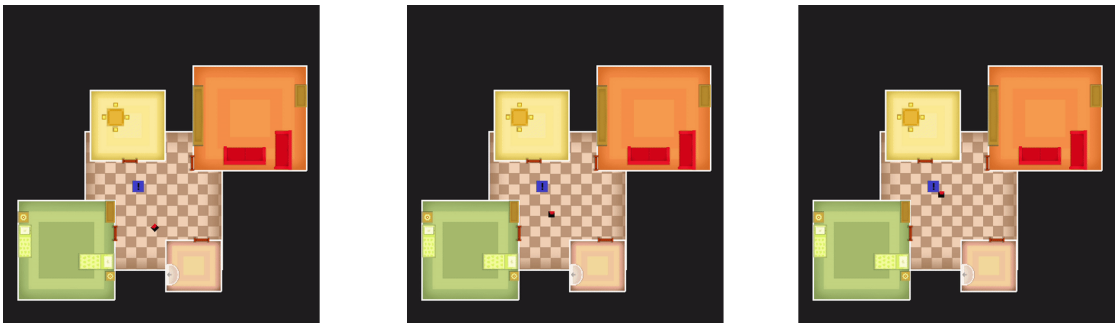

Figure 8: Agent progressing to the objective.

6. Conclusions

In this work we explored the possibility to build a constraint-procedural logic generator for environments to be used as training and benchmarking tool for deep Q-learning agents. The resulting products are a working generator of environments that resembles house floors and an exploring deep Q-learning agent. The generated rooms are connected by a common space and filled with coherent furniture, variably distributed on the room continuous space. The exploring agent is partially capable of solving the given task of finding an object in the generated environment and learning through reinforcement of a reward. It actually outperforms a human competitor when the target is inside the central room. With this work we verified the feasibility of such tool and implemented an instance of both generator and exploring agent. Further evolution of the generator could include support for corridors, multi-story houses and stairs, dynamic elements such as obstacles, simulation for humans, animals and other agents, door states management (such as open, closed, locked etc.), light management, ambience management (such as smoke, fog, humidity etc.), temperature, friction and other challenging elements to

train and benchmark the agent. From the agent side, it could be improved with the ability to explore the rooms and interact with the environment (ie. open doors), be able to consider data from other sensors and act accordingly.

7. Acknowledgments

This work started as a joint Bachelor's Degree Thesis in Computer Science @ University of L'Aquila A.Y. 2017-2018 by Leonardo Formichetti and Samuele Petrucci under the supervision of Giovanni De Gasperis. Afterwards, the work has been extended and rationalized by the authors.

References

[1] N. Shaker, J. Togelius, M. J. Nelson, Procedural content generation in games, Springer, 2016.

[2] R. Lopes, T. Tutenel, R. M. Smelik, K. J. De Kraker, R. Bidarra, A constrained growth method for procedural floor plan generation, in: Proc. 11th Int. Conf. Intell. Games Simul, 2010, pp. 13–20.

[3] R. Van Der Linden, R. Lopes, R. Bidarra, Procedural generation of dungeons, IEEE Transactions on Computational Intelligence and AI in Games 6 (2014) 78–89.

[4] A. M. Smith, M. Mateas, Answer set programming for procedural content generation: A design space approach, IEEE Transactions on Computational Intelligence and AI in Games 3 (2011) 187–200.

[5] C. J. Watkins, P. Dayan, Q-learning, Machine learning 8 (1992) 279–292.

[6] H. Van Hasselt, A. Guez, D. Silver, Deep reinforcement learning with double q-learning, in: Thirtieth AAAI Conference on Artificial Intelligence, 2016.

[7] N. Savinov, A. Raichuk, R. Marinier, D. Vincent, M. Pollefeys, T. Lillicrap, S. Gelly, Episodic curiosity through reachability, arXiv preprint arXiv:1810.02274 (2018).

[8] J. Jaffar, S. Michaylov, P. J. Stuckey, R. H. Yap, The clp (r) language and system, ACM Transactions on Programming Languages and Systems (TOPLAS) 14 (1992) 339–395.

[9] G. De Gasperis, P. Migliarini, Environment generator, simulator and neural agent, http://github.com/AAAI-DISIM-UnivAQ/bd-procedural-env-deep-learning, 2019.

[10] R. A. Brooks, Cambrian intelligence: The early history of the new AI, MIT press, 1999.

[11] I. Letteri, G. Della Penna, G. De Gasperis, Botnet detection in software defined networks by deep learning techniques, in: International Symposium on Cyberspace Safety and Security, Springer, 2018, pp. 49–62.

[12] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, arXiv preprint arXiv:1412.6980 (2014).

[13] L.-J. Lin, Reinforcement learning for robots using neural networks, Technical Report, Carnegie-Mellon University, Pittsburgh, PA, School of Computer Science, 1993.