Graphs for hierarchical RL macro planning: A Sokoban testbed

Mathieu Orhan
Master MVA, ENS Cachan
mathieu.orhan@eleves.enpc.fr

Bastien Dechamps
Master MVA, ENS Cachan
bastien.dechamps@eleves.enpc.fr

Abstract

In this work, we address the problem of learning general policies with Graph Neural Networks to solve the Sokoban environment, a difficult planning puzzle. We build a graph representation for Sokoban levels, learn policies with a Q-learning approach and then investigate the transfer and generalization capacities enabled by GNNs of the learned policies.

1 Introduction

In the game Sokoban, the player has to push boxes on targets by evolving in a board of squares, where each of those squares are either a wall or a floor. The puzzle is solved when all the boxes are brought on the targets. The problem of learning a policy to solve the Sokoban is difficult because it has to simultaneously learn the physics of the game (moving the player, pushing boxes, ...) and the macro planning of the level as it has been proven to be NP-Hard [4]. Furthermore, many moves are irreversible and can make the level unsolvable, forcing the agent to plan moves ahead of time.

In very recent works, Graph Neural Networks have been used to learn heuristics for combinatorial problems [9, 3] as well as for the task of learning a policy with reinforcement learning algorithms [10, 13]. Compared to other reinforcement learning approaches in which the environment is fixed and the agent interact with it, GNNs would bring the possibility of doing a kind of *curriculum learning*[1] [2]. Indeed, using such networks allows to transfer the learned strategy to any Sokoban level, by varying the number of boxes, the size and shape of the level, etc.

In the following sections, we first define the graph representation for any Sokoban levels and introduce the reinforcement learning algorithm as well as the GNN models we choose to learn a policy. Then, we test the learned strategies and their capacities to generalize to other Sokoban levels.

2 Method

2.1 Graph embedding

There are many ways to embed a state as a graph, where the notion "graph" is defined in [1] and might include node, edge and global features, be directed or not, and may be a multi-graph. Here, a state is defined as a particular configuration of a Sokoban puzzle. In figure 1, an example of such a state is represented along with its possible graph embeddings.

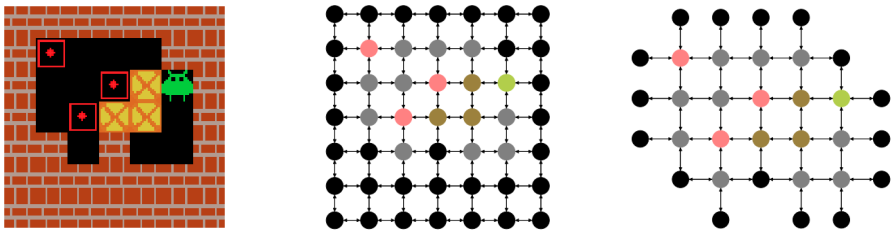
Nodes. Sokoban is organized in a grid composed of small elementary units which are wall, box, floor, player or target. We identify these units as the nodes of the graph (see for example figure 1b). Some units are unnecessary and we can remove them, such as walls surrounded only by walls as represented on figure 1c. Basic node features are binary variables indicating if the units has box, is a

[1] Progressively increase the difficulty of the training levels.

wall, has a target, has a player, and is the floor. Note that some of these variables are not mutually exclusive. Additionally, we add normalized positions in Cartesian coordinates. They break rotational invariance as they discriminate the x and y axis, but solving the Sokoban requires a precision notion of direction and position.

Edges. We mostly considered the case where each node is connected to its neighbors on the grid, but we could also consider the fully connected case which might be interesting with attention mechanism to propagate further information. On the edges, we encode direction as 4-dimensional binary vectors. Currently, successful experiences do not directly use these features but rather differences of relative position, which provide similar information.

Global features. We didn't embed any global feature, though additional information could be encoded such as the number of boxes, their positions, the position of the player, etc.



(a) (b) (c)

Figure 1: Minimal embedding (b) and our embedding (c) for a Sokoban level generated with `gym-sokoban` (a).

We developed our own environment to interact directly with it following Sokoban rules and used the level generator of [8] along with our own level generators to create custom levels. We support both [8] and [12] formats to build datasets. In opposition to [8], we restrict the number of actions to 4, as a distinction between box pushing and box moving seems artificial, and *not moving* cannot help to reach any objective in the game. We also identify actions as neighboring nodes of the player, and further restrict actions to moves that make sense, unlike e.g. pushing against a wall. In some of our experiments, actions are also equivalently interpreted as directions in a global binary vector.

2.2 Graph Neural Networks

We then introduce the GNN models we used to learn a suitable policy. We considered two approaches: a *node-centered*, were Q-values are regressed on each node of the graph, and a *graph-centered* where direction are predicted. As we did not obtain any relevant results with the node-centered approach, we only focus on the graph-centered in this work.

2.2.1 Message Passing scheme

Our graph-centered model is based on a message passing algorithm introduced in [11] called *Edge-Conv*. Let $G = (V, E)$ the graph of a Sokoban level state, with $V = \{1, ..., N\}$. Each node i has a feature vector $x_i \in \mathbb{R}^F$. The EdgeConv operation is defined at each node i by:

$$x'_i = \square_{j \in \mathcal{N}(i)} h_\Theta \left([x_i, x_j - x_i] \right), \tag{1}$$

where $\mathcal{N}(i)$ are the neighbors of i in $G = (V, E)$ and $[\cdot, \cdot]$ is the concatenation operator. The term $h_\Theta : \mathbb{R}^{2F} \rightarrow \mathbb{R}^{F'}$ is a nonlinear function with a set of learnable parameters Θ. In our model, it is defined [2] as a MLP with a ReLU activation given to a linear layer:

$$h_\Theta(x) = \theta_3 \cdot \text{ReLU} \left(\theta_1 \cdot x_1 + \theta_2 \cdot (x_2 - x_1) \right), \tag{2}$$

where $x = [x_1, x_2] \in \mathbb{R}^{2F}$ and $\Theta = [\theta_1, \theta_2, \theta_3]$. The aggregation function $\square$ can be the sum, the max, or any differentiable permutation invariant function.

[2] Biases are omitted for clarity.

We chose this model over simple graph convolutions (eg. GCN in [5]) as the difference between the feature vectors and in particular node positions of two neighboring nodes $x_j - x_i$ contains information about the direction in the graph. If it is not provided, all the neighboring nodes with same nature (floor for example) will contribute the same during a node feature update, regardless of the direction.

2.2.2 Graph-centered model

The whole graph-centered model is formed by stacking D EdgeConv units with MLP having H hidden units each. Multiple message passing layers enable all the nodes of the graph to communicate and allow the model to have a global understanding of the current state. Then, if $\left(x_1^D, ..., x_N^D\right) \in \mathbb{R}^H$ are the nodes features vectors from the last EdgeConv layer, we apply a global max pool operation:

$$\forall j \in \{1, ..., H\}, \quad r_j = \max_{i \in \{1,...,N\}} \left(x_i^D\right)_j . \tag{3}$$

This operation returns a H-dimensional vector that we feed to a 3-layers MLP with ReLU activations[3], corresponding to the Q-values $(Q(G(s), a))_{a \in \{1,...,4\}}$ where $G(s)$ is the input state graph. The model is represented on figure 2.

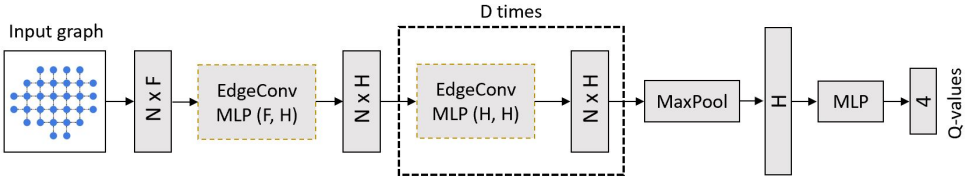


Figure 2: Model used for the graph-centered approach.

2.3 Reinforcement learning

We had multiple options for the choice of a reinforcement learning algorithm. We chose to adopt deep Q-learning [6] with replay memory and target network, a simple yet powerful baseline — it was able to play multiple Atari games at superhuman level. However, we argue that Sokoban is (much) harder than these Atari environments, and more sophisticated reinforcement learning algorithms may be more suited. For instance, [12] introduces Imagination-Augmented Agents to partially solve a variant of the Sokoban environment.

Algorithm 1 Deep Q-Learning algorithm with Target Network and Experience Replay

Require: policy net $\mathcal{M}_P$, target net $\mathcal{M}_T$, replay buffer $\mathcal{B}$, dataset $\mathcal{D}$
  for $s_0 \in \mathcal{D}$ **do**
    while episode is not terminated **do**
      Sample action a_t using ϵ-greedy exploration
      Take action a_t, observe (s_{t+1}, r_t)
      Push a transition (s_t, a_t, s_{t+1}, r_t) into $\mathcal{B}$
      if $\mathcal{B}$ is large enough **then**
        Sample a batch (S_t, A_t, S_{t+1}, R_t) of transitions from $\mathcal{B}$
        Compute predicted Q-values $Q_p = \mathcal{M}_P(S_t, A_t)$
        Compute expected Q-values $Q_e = R_t + \gamma \max_A \mathcal{M}_T(S_{t+1}, A)$
        Compute the loss $\mathcal{L}(Q_p, Q_e)$ and backpropagate the gradient on $\mathcal{M}_P$
      end if
    end while
    Update $\mathcal{M}_T$ at regular intervals using $\mathcal{M}_P$ weights
  end for

A key difficulty in this environment is that many moves are irreversible — they make the game unsolvable. In most cases, it is hard to know if it is unsolvable. To explore, following [6], we use an ϵ-greedy strategy with linear annealing. We improve the strategy by considering only sensible

[3] except for the last linear layer which is kept without activation.

moves — that is, we remove some legal moves such as moving towards a wall that cannot be part of the optimal solution.

Replaying experiences is a key ingredient of deep Q-learning. It enables temporal decorrelation of samples which stabilizes training. In our case, solving a given Sokoban level randomly is impossible or at least very rare for the simplest levels, so storing these rare experiences should be a good idea — even if a basic Replay Memory samples *uniformly* experiences. We used two additional variations of the replay buffer. The first one, that we call *Threshold Replay Buffer*, samples transitions with reward above a threshold more often than other transitions. The second one is an implementation of Prioritized Experience Replay [7]. It is promising, because the Threshold Replay Buffer suffers from over-fitting issues, yet it was one the key to get our first results. However, the Prioritized Replay Buffer did not improve the results and did not manage to solve harder levels presented in section 3.

We used the Huber loss to compute the temporal differences between the expected and predicted Q-values, which has the form:

$$\mathcal{L}(x, y) = \frac{1}{n} \sum_i z_i \quad \text{where} \quad z_i = \begin{cases} 0.5(x_i - y_i)^2 & \text{if } |x_i - y_i| < 1 \\ |x_i - y_i| - 0.5 & \text{otherwise} \end{cases} . \tag{4}$$

Reward shaping is quite an art — in some environments, it can be a decisive ingredient to a working algorithm. In most of our experiments, we kept the reward shaping of [8], that is 1 if a box is pushed on a target, -1 if a box is pushed off a target, 10 if all boxes are pushed on all targets, and -0.1 otherwise. The base negative reward is supposed to penalize sub-optimal solutions. A simpler alternative would be binary reward if the level is solved, but we found the reward signal to be already very sparse.

Algorithm 1 describes how the agent is trained for a single epoch. To evaluate, at state s we simply pick the action maximizing $\mathcal{M}_P(s, .)$. Of course, it is necessary to bound the number of steps both for training and evaluating to avoid an infinite loop.

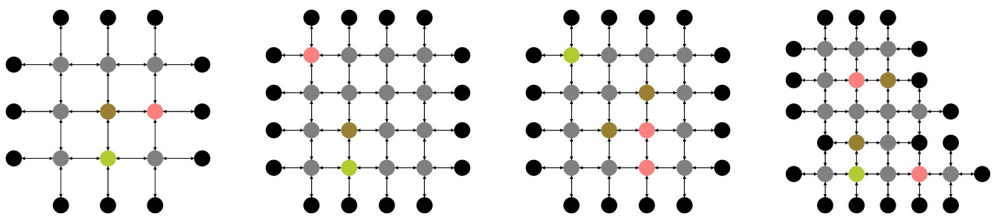
3 Experiments

3.1 Settings

We searched for the best hyperparameters manually. Tuning these parameters is crucial to obtain results. We use RMSProp as our optimizer, following [6], and gradient clipping between -1 and 1. We also use a batch size of 32, and a learning rate of 5×10^{-3}. We used a Thresholded Replay Buffer, and an annealing epsilon scheduling between 1.0 and 0.1 reaching its final value at the quarter of the training steps. We always used $\gamma = 1$. In these experiments, the aggregation function $\square$ is the maximum. Each episode stops when the level is solved or after a maximum of 25 steps.

3.2 Simple levels

We started our experiments with very simple levels which consisted on a $n \times n$ grid with one or two boxes for $n = 5, 6$. Examples of such levels are represented on figures 3a, 3b and 3c. In each case, we generated 100 different levels as our train dataset and 50 others to evaluate our models.



(a) 5×5 + 1 box (b) 6×6 + 1 box (c) 6×6 + 2 boxes (d) easy level

Figure 3: Increasing difficulty Sokoban levels.

We first consider 5×5 levels with one box. We use a simple model with $D = 3, H = 32$. As represented on figure 4a and reported in table 1, the model manages to solve all the levels in the training set and have good performances in the test set.

level type	train	test
$5 \times 5(1)$	1.0	0.94
$6 \times 6(1)$	0.99	0.64
$6 \times 6(2)$	0.62	0.02

Table 1: Fraction of levels solved for level type.

Once 5×5 levels were solved, we moved to a similar task using 6×6 levels and 1 or 2 boxes. By using a buffer size of 30000, $D = 5, H = 256$, we obtain the results represented on table 1 as well as the training curves on figures 4b and 4c. As we can observe, the performances are still good for the 6×6 with 1 box, but with 2 boxes the model only managed to solve one level of the test set. Some of those levels solution found by the models are represented in appendix A.

The next step is to solve real levels of Sokoban on a 7x7 grid generated by [8], as the one represented in figure 3d. Currently, none of our experiments succeeded, even after two entire days of training (see figure 5). These levels are much harder, requiring complex patterns compared with the previous levels. With more walls, the game can be unsolvable after a wrong first move. Exploration is a huge issue, as most levels (more than 90%) are never randomly solved after millions of transitions and the reward signal tends to be very sparse.

3.3 Generalization capacities

Finally, we tested the generalization capacities by applying trained models on other levels that they were trained on. The results, represented on table 2, are unfortunately not very convincing.

		Tested on			
		5×5 (1)	6×6 (1)	6×6 (2)	easy
Trained on	5×5 (1)	–	0.02	0	0
	6×6 (1)	0.26	–	0.01	0.01
	6×6 (2)	0.14	0.04	–	0.01

Table 2: Fraction of level solved for generalization experiences.

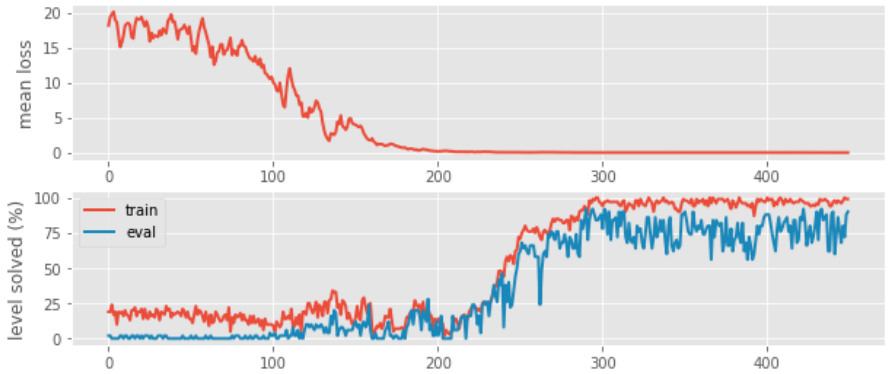
Only the policies learned on 6×6 levels manage to solve a small fraction of simpler levels (for example 26% of 5×5 levels solved by the model trained on 6×6 levels with one box. Furthermore, the model trained on 6×6 levels with 2 boxes should easily manage to solve 6×6 levels with one box, but this is actually not the case. However, in view of the relatively bad training performances represented on figure 4c, it is not surprising either.

We also tried to train a model on a mixed dataset with different types of levels (5×5, 6×6, 7×7, and easy levels) to hope for more generalization, but we didn't get any relevant results, even after 2 days of training.

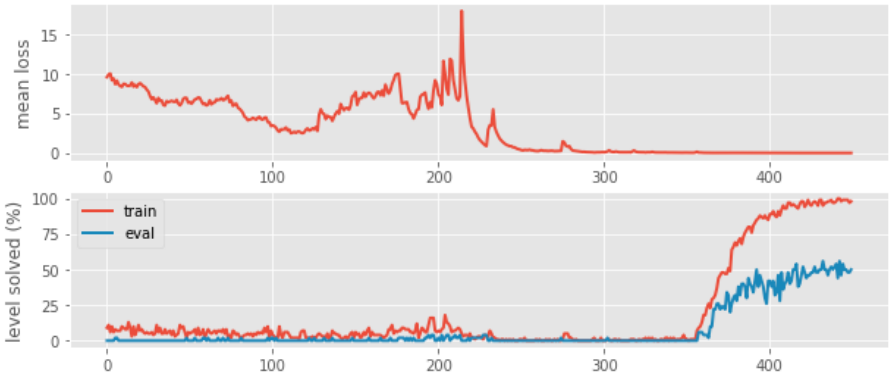
4 Conclusion

4.1 Conclusion

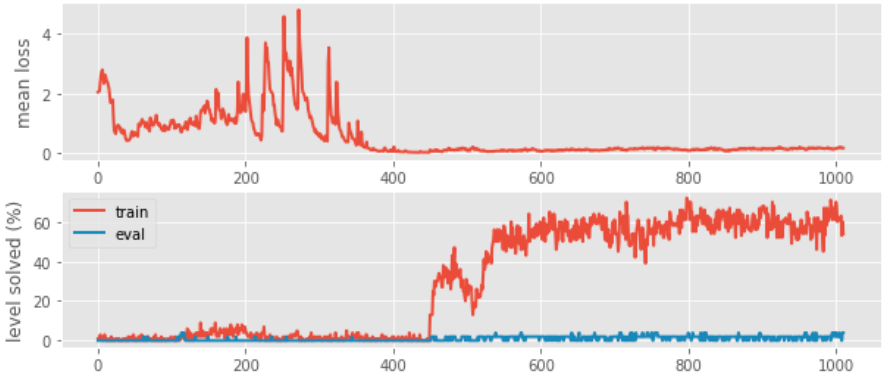
Our work has many possible developments, as it combines reinforcement learning and graph neural networks which are both very active research topics, with a very difficult environment. Experiments are time-costly, taking up to days on high-end computers. With little prior research work on this environment, and approaches using graphs neural networks to the best of our knowledge, we had many interesting design choices to make. Having results at all, even on the easier levels, was very challenging and did not happen early. We may be close to obtain results on harder datasets, but our approach seems computationally inefficient, and more powerful models may be required.



(a) 5×5, 1 box, total training time: 6 hours.



(b) 6×6, 1 box, total training time: 11 hours.



(c) 6×6, 2 box, total training time: 36 hours.

Figure 4: Training curves for different levels size / number of boxes.

4.2 Work directions

We would like to present some of research tracks that could improve our approach.

Graph modeling A different way to approach the problem is to use a fully connected graph instead of a grid-like graph. A combination with an attention mechanism could be interesting. The idea is to let communicate arbitrarily distant nodes using only one message passing. In our approach, receptive field is an important issue, and directly depends of the number of message passing. As more complex levels require longer paths, this is an obstacle to generalization.

Graph Neural Networks Graph neural networks allow to incorporate priors into models. For instance, as we use positional features, we include node features differences in the message passing. This gives the model a much better sense of directionality than simply using positions with normal

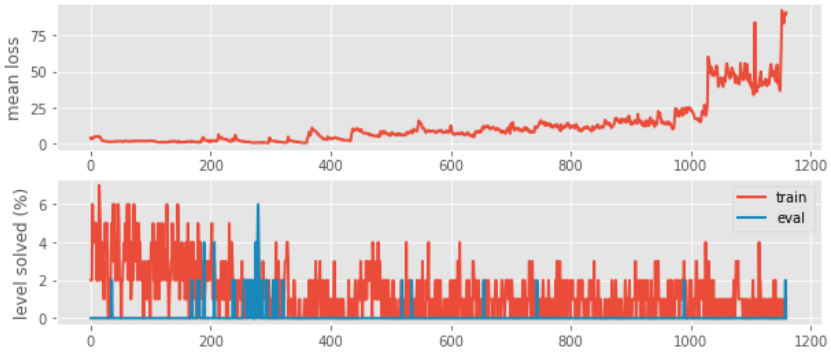


Figure 5: Training for real easy levels (after 2 days of training).

convolutions. Models that incorporate more directionality priors and favor generalization across different levels should be investigated.

Another issue with stacking message passing is the very high number of parameters related to the MLPs. We think sharing parameters across core message passing layers could alleviate this issue.

Reinforcement learning We stuck to deep Q-learning [6] with minor improvements and ϵ-greedy exploration, but this approach seems rather not sample efficient. We could use a form of supervision using Sokoban solvers to bootstrap the training. If the model generalizes enough, it may be able to solve some levels of higher size and difficulty than the supervised ones.

Curriculum learning This is one of the advantages of using graph neural networks. Models can be naturally extended to various levels size while keeping the same structure. The models we trained did not generalize enough to enable some kind of curriculum. However, if one succeed to solve real Sokoban levels, this research track is very exciting. It could enable to solve much harder levels that could never be solved by the existing solvers.

References

[1] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andrew Ballard, Justin Gilmer, George Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks, 2018.

[2] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[3] Hanjun Dai, Elias B. Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs, 2017.

[4] Dorit Dor and Uri Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[5] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2016.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

[7] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[8] Max-Philipp B. Schrader. gym-sokoban. `https://github.com/mpSchrader/gym-sokoban`, 2018.

[9] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a sat solver from single-bit supervision, 2018.

[10] Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. Nervenet: Learning structured policy with graph neural networks. In *International Conference on Learning Representations*, 2018.

[11] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics*, 38(5):1–12, Oct 2019.

[12] Théophane Weber, Sébastien Racanière, David P. Reichert, Lars Buesing, Arthur Guez, Danilo Jimenez Rezende, Adria Puigdomènech Badia, Oriol Vinyals, Nicolas Heess, Yujia Li, Razvan Pascanu, Peter Battaglia, Demis Hassabis, David Silver, and Daan Wierstra. Imagination-augmented agents for deep reinforcement learning, 2017.

[13] Vinicius Zambaldi, David Raposo, Adam Santoro, Victor Bapst, Yujia Li, Igor Babuschkin, Karl Tuyls, David Reichert, Timothy Lillicrap, Edward Lockhart, Murray Shanahan, Victoria Langston, Razvan Pascanu, Matthew Botvinick, Oriol Vinyals, and Peter Battaglia. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*, 2019.

A Example of puzzles solved

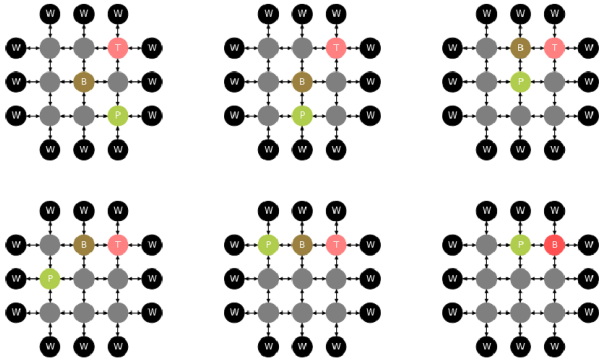


Figure 6: Example of 5×5 levels solved.

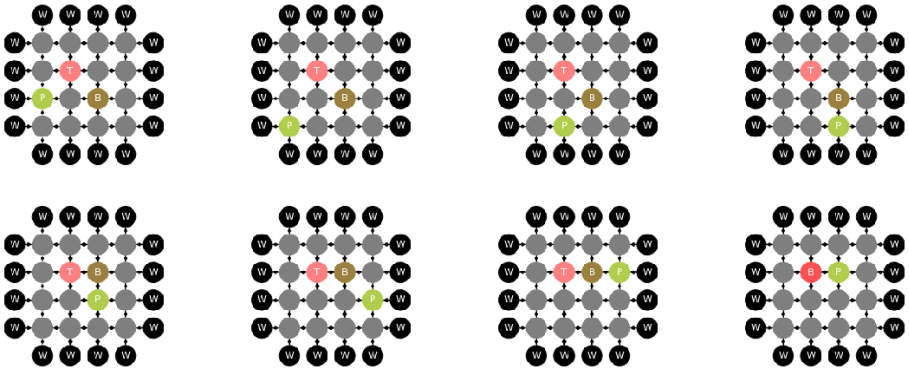


Figure 7: Example of 6×6 levels with 1 box solved.

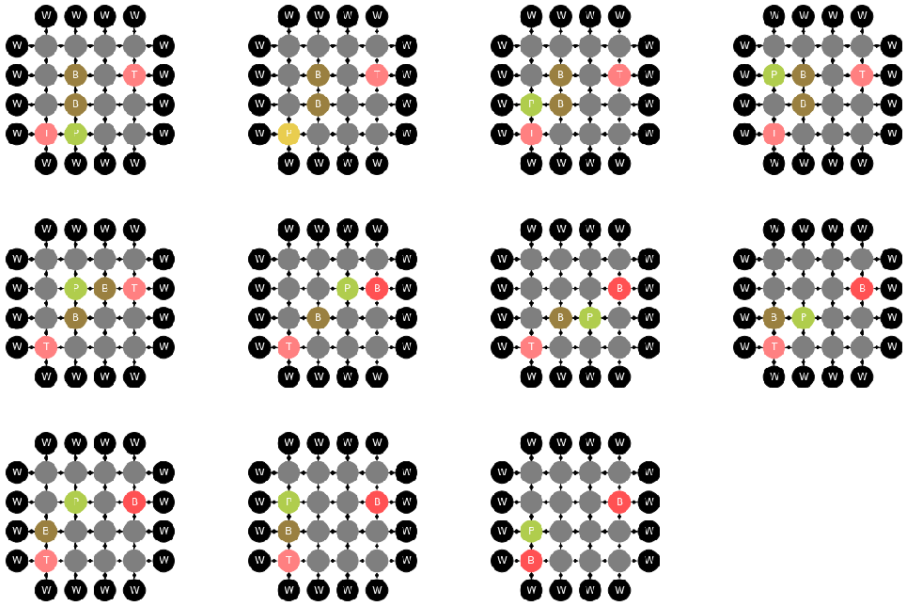


Figure 8: Example of 6×6 levels with 2 boxes solved.