Molecule Builder: Environment for Testing Reinforcement Learning Agents

Petr Hyner[1,2], Jan Hůla[2,3] and Mikoláš Janota[3]
[1]*Department of Informatics and Computers, Faculty of Science, University of Ostrava, Ostrava, Czech Republic*
[2]*Institute for Research and Applications of Fuzzy Modeling, University of Ostrava, Ostrava, Czech Republic*
[3]*Czech Technical University in Prague, Prague, Czech Republic*

Keywords: Reinforcement Learning, Subgoals, Environment, Agent.

Abstract: We present a reinforcement learning environment designed to test agents' ability to solve problems that can be naturally decomposed using subgoals. This environment is built on top of the PyVGDL game engine and enables to generate problem instances by specifying the dependency structure of subgoals. Its purpose is to enable faster development of Reinforcement Learning algorithms that solve problems by proposing subgoals and then reaching these subgoals.

1 INTRODUCTION

This contribution describes a minimalistic environment called *Molecule Builder* whose purpose is to test the abilities of *Reinforcement Learning* (RL) agents. Concretely, this environment tests the agents' ability to solve problems that require the completion of many subgoals before reaching the final state. Reinforcement learning methods can solve problems of surprising difficulty (Silver et al., 2017; Mirhoseini et al., 2021). Nevertheless, they are not very effective in terms of sample complexity. They require an extensive computational budget for trial and error exploration to discover an effective behaviour for a given problem. If the goal state (or a state with a reward) first requires the completion of several subgoals that depend on each other, it is possible that the agent will never reach such a state and, therefore, will not obtain any learning signal.

This attribute is present in many real-life scenarios, which should justify the search for novel approaches that relate to sequential decision-making problems. It is a mark of human cognition that we create hierarchical plans when we try to solve a complex and novel task. Additionally, factorization of complex structures into more simple structures is an essential tool in problem solving.

We believe that RL agents should also be equipped with this ability. To develop such agents/algorithms, it is necessary to have a benchmark environment that naturally contains the concept of subgoals first. Our motivation for the work presented here was to develop such a benchmark that would provide challenging problems for the current generation of RL algorithms. Such a benchmark should be as minimalistic as possible, while the environment should be easily configurable by anyone who considers implementing an agent to solve it.

The environment presented consists of a random generator of problem instances in which the goal of the agent is to build a structure from simpler building blocks. The structure could contain substructures, and the construction of these substructures represents a natural subgoal. Moreover, the building blocks may be blocked by various obstacles that may be destroyed only by the corresponding structures. Therefore, each problem instance could be created from a dependency graph that reflects the order in which the individual structures could be built.

The text is structured as follows. Section 2 describes fundamental concepts in reinforcement learning and provides motivation for developing agents that explore the *state-space* by proposing subgoals to themselves. Section 3 provides a high-level overview of the Molecule Builder environment. Section 4 describes the generative model for the problem instances, and the last two Sections 5 and 6 are devoted to related work and conclusion.

Hyner, P., Hůla, J. and Janota, M.
Molecule Builder: Environment for Testing Reinforcement Learning Agents.
DOI: 10.5220/0012257900003595
In *Proceedings of the 15th International Joint Conference on Computational Intelligence (IJCCI 2023)*, pages 450-458
ISBN: 978-989-758-674-3; ISSN: 2184-3236

2 SUBGOALS IN REINFORCEMENT LEARNING

The field of Reinforcement Learning is focused on methods for *sequential decision-making problems*. These problems are formalized by a *Markov decision process* (MDP), which is a 4-tuple (S, A, P_a, R_a).

- S is a set of states called the state space
- A is a set of actions called the action-space
- $P_a(s,s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t+1$
- $R_a(s,s')$ is the immediate reward (or expected immediate reward) received after transitioning from state s to state s', due to action a

Many problems can be formulated as MDPs. For example, many examples can be found in combinatorial optimization, with the canonical example being the *Travelling Salesman Problem* (TSP) (Gavish and Graves, 1978). In the TSP, we have a directed (weighted) graph, and the goal is to find the shortest cycle, which visits all the vertices. Here the state $s \in S$ can be represented as a set of already visited vertices, a set of still non-visited vertices, and the vertex the agent is currently in. The set of actions A available in a given state corresponds to the choice over the set of non-visited vertices. The transitions are deterministic and, therefore, $P_a(s,s') = 1$ only for one (consistent) triple (a,s,s') and 0 otherwise. The rewards could be, for example, set in such a way that the agent receives a non-zero reward only for the action which closes the cycle. The value of the reward would be equal to the length of the cycle.

To solve a given MDP means to find a policy function $\pi : S \to A$ (which can be potentially probabilistic) that maximizes the expected cumulative reward:

$$E\left[\sum_{t=0}^{\infty} \gamma^t R_{a_t}(s_t, s_{t+1})\right]. \tag{1}$$

In this expression, E is the expected value, a_t is the action sampled according to the policy π and s_{t+1} is the state sampled according to $\Pr(s_{t+1} \mid s_t, a_t)$. The expected value E is over the randomness in these two variables.

In practice, we do not necessarily need to find the optimal policy π^*, which maximizes this expression. It is often enough to find any policy that is ”good enough.”

Various methods exist to find such policy (Mnih et al., 2013; Sutton et al., 1999a) but the general idea is that the agent usually starts to explore the state space by choosing actions according to a random policy, and this policy is repeatedly updated according to rewards that the agent observes. If the agent does not observe any reward during its exploration, it cannot learn anything. This problem of *sparse rewards* is often approached by *reward shaping*, where additional (intermediate) rewards are given to the agent to guide it towards a more desirable behaviour. Creating these additional rewards often requires domain expertise.

The policy function can take various forms. In the simplest case, when the state space is small, it can be a lookup table. It can also be determined by another function called *value function*, which, in simple words, measures how valuable it is for the agent to be in a given state. When the agent has access to this value function and the (probabilistic) transition function, it can choose the action which leads to a state with the highest (expected) value.

Very often, the policy/value function is represented by a neural network which is learned during the exploration process. Sometimes, the agent also needs to learn the transition function, sometimes called a *world model*, because it may not be known (i.e. in robotics.)

When the agent makes decisions according to the learned value function, which may be imprecise, the quality of each decision is often considerably improved if the agent first simulates many possible future trajectories (sequences of actions and states) and observes which states they lead to before it decides how to act. This is known under the term *model predictive control* (MPC) and it vaguely resembles certain cognitive processes of humans and other animals, which can run mental simulations before they decide which actions to take.

Nevertheless, the crucial difference between MPC and human planning abilities is that our mental simulations do not necessarily follow the low-level dynamics of the environment. For example, when we want to safely travel to a different country, we do not imagine what our body will be doing at every moment of the journey. Instead, we plan over high-level actions and states such as ”buy a ticket” or ”get to the airport”.

We believe that this ability to freely move in space and time in our minds is crucial for our effectiveness in problem-solving and that figuring out how to enrich RL agents with this ability will allow us to solve much more complicated problems than the ones we are currently able to solve. This motivated us to create a minimalistic environment for testing RL agents, which contain a clear notion of subgoals and where the dependency graph of subgoals is easily controllable.

3 THE MOLECULE BUILDER ENVIRONMENT

We have built the Molecule builder environment on top of the *Video Game Description Language* (concretely, the Python implementation PyVGDL (Schaul, 2013; Vereecken, 2018), which was designed to allow for quick prototyping of test environments for AI agents.

As the name suggests, the basic principle of this game/environment is to build *"molecules"* from *"atoms."* The agent is placed in a simple grid world that contains rooms and corridors, and the atoms are scattered in the rooms. In each game, the goal is to assemble a particular molecule that may consist of sub-molecules. Moreover, some corridors may be blocked by obstacles that may prevent the agent from accessing the required atoms needed to construct the target molecule. The given obstacle can be destroyed by a different molecule, and the construction of this molecule naturally represents a subgoal in this game. There may be several such subgoals that may depend on each other and the *dependency graph* of these molecules constitutes a high-level structure of each game instance. The action space $\mathcal{A}$ consists of four actions: move up, move down, move left and move right.

Our framework enables the generation of games of desired complexity by specifying this dependency graph. This control over the complexity of the problem allows us to create curricula of problems in which the agent can first learn to assemble simple molecules and then proceed to more complex games in which the assembly of these molecules represents a subgoal. Each game consists of a layout that represents the initial state of the game (the position of walls, atoms, obstacles, and the agent). It also consists of a definition file that describes the default behaviour of individual entities in the game, their interactions, and their appearance (each entity is represented by an ASCII symbol). The main part in the definition file is the interaction set, which dictates how the molecules could be built from atoms and sub-molecules.

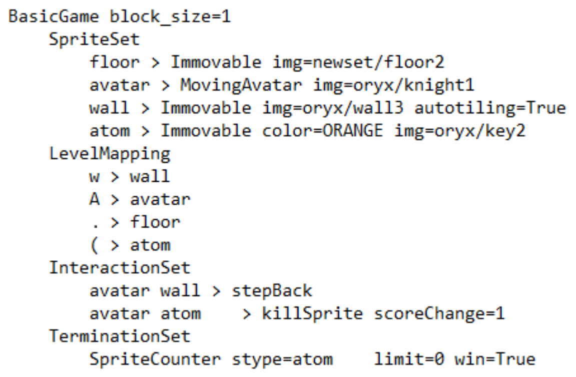
3.1 The Layout of the Game Instance

Figure 1 shows a randomly sampled layout of a game with many atoms scattered throughout the rooms. Each symbol represents a given entity defined inside the *SpriteSet* block and mapped in the *LevelMapping* block in the definition file shown in Figure 2. In this layout, the dots represent a floor where the agent can move freely. The letter *w* represents a wall through which the agent cannot move. Other symbols, such as *3, 5, /,)* etc., are the atom symbols. Several other types of symbols *(C, o)* correspond to obstacles. These symbols are placed inside corridors that connect two rooms and the agent cannot pass through them. Finally, the symbol *A* corresponds to the agent.

```
wwwwwwwwwwwwwwwwwwwwwwwwwwwwww
w......wwwwwwww.......w......w
w......wwwwwwww.......w.../5.w
w......C.wwwwww.......w).....w
w/.....w..............wwwwww.w
wwwwww.wwwwwwww...1...wwwwww.w
wwwwww.wwwwwwww.......wwwwww.w
wwwwww.wwwwwwww.......wwwwww.w
wwwwww.wwwwwwwwwwwww.wwwwwww.w
wwwwww......).wwwwww.wwwwwww.w
wwwwwwww......wwwwww.wwwwwww.w
wwwwwwww......wwwwww.wwwwwww.w
wwwwwwww...(..wwwwww.wwwwwww.w
wwwwwwww......wwwwww.wwwwwww.w
wwwwwwww......wwwwww....A....w
wwwwwwww......wwwwwwwwwwwwww.w
wwwwwwwwwwwwwww.......wwwwww.w
wwwwwwwwwwwwwww..3....wwwwww.w
wwwwwwwwwwwwwww.......wwwwww.w
wwwwwwww..............wwwwww.w
w......w.wwwwwwwwwwww.wwwwww.w
w........wwwwwwwwwwww.wwwwww.w
w......wwwwwwwwwwwwww.o.)....w
w......wwwwwwwwwwwwwwww....!.w
w......wwwwwwwwwwwwwwww.(....w
w......wwwwwwwwwwwwwwww......w
w......wwwwwwwwwwwwwwww...5..w
w......wwwwwwwwwwwwwwww......w
w......wwwwwwwwwwwwwwww......w
wwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

Figure 1: An example of a simplified layout.

Here, we describe how such a game instance can be solved. The goal is to build a target molecule consisting of atoms *), (, /, 5, 3*. As can be seen in Figure 1, there are obstacles that prevent the agent from reaching some parts of the environment. In detail, there are three separate parts of the environment that are inaccessible without the removal of obstacles. Part 1, is the part where the agent starts and contains atoms *), (, /, 5, !*. Part 2 contains symbols *), (, /* and could be accessed from part 1 after removing the obstacle *C*. Finally, part 3 contains atom *3* and can be accessed from part 1 after removing the obstacle *o*.



```
BasicGame block_size=1
    SpriteSet
        floor > Immovable img=newset/floor2
        avatar > MovingAvatar img=oryx/knight1
        wall > Immovable img=oryx/wall3 autotiling=True
        atom > Immovable color=ORANGE img=oryx/key2
    LevelMapping
        w > wall
        A > avatar
        . > floor
        ( > atom
    InteractionSet
        avatar wall > stepBack
        avatar atom    > killSprite scoreChange=1
    TerminationSet
        SpriteCounter stype=atom    limit=0 win=True
```

Figure 2: An example of a simplified definition file.

The agent needs to access part 3 to collect the symbol *3* needed to build the target molecule. The obstacle *o* can be destroyed by a molecule that can be built from atoms *), (, /*. If the agent uses the atoms *), (, /* present in part 1 to destroy the obstacle *o*, then it will not be able to build the target molecule for which these atoms are required. Therefore, it also needs to reach part 2. To destroy the obstacle *C*, it is necessary to build a molecule from atoms *1,5,!* (the agent must learn this knowledge by playing different instances of the game).

Here are the high-level steps by which the agent might solve the game.

1. The agent starts in part 1 and collects atoms *1, 5, !*.
2. The agent destroys the obstacle *C* with the created molecule and gathers atoms */,), (* in part 2.
3. The agent destroys the obstacle *o* with the created molecule */,), (* and gathers the atom *3*.
4. The agent returns to part 1 to collect the atoms */,), (, 5*, and this finishes the game.

Figure 3 shows the molecules that the agent builds in this game instance. Note that the order of collecting atoms for a given molecule is given by its corresponding tree.

3.2 The Definition File of the Game Instance

Figure 2 shows a (simplified) definition file with only one atom. The syntax is simple to understand, as there are only four different types of definitions available (*SpriteSet*, *LevelMapping*, *InteractionSet*, *TerminationSet*). The *SpriteSet* code block defines the entities of the game, their names, properties, and graphics. *LevelMapping* then assigns a unique symbol to each entity defined above. The *InteractionSet* specifies what happens when two defined entities meet. Multiple types of interactions are possible, and these correspond to the event that occurs when the two entities meet. The most commonly used events are the *killSprite* and *stepBack* events. A large number of (simpler) games can be built using these two events only. Furthermore, the interaction *transformTo* is also very helpful in cases where we need to change the affected entity to a different entity. In our case, we use this event to transform multiple atoms into molecules or sub-molecules. Finally, the *TerminationSet* defines condition(s) of what must happen in the environment for the game to end.

For a detailed description of the syntax of the definition files, see the original PyVGDL publication (Schaul, 2013).

Our framework enables us to generate a practically infinite number of such games that run very fast and can be used to develop RL agents that can solve problems by proposing subgoals to themselves. An example of a specific environment's definition file and layout generated by our framework can be found in Appendix A, where both the definition and layout correspond to one environment instance.

4 GENERATIVE MODEL FOR GAME INSTANCES

In the previous section, we have established how the Molecule Builder environment works and what is the goal of the agent. In this section, we describe how each instance of the game is generated. More specifically, how the final layout and definition files mentioned in Section 3 are produced.

4.1 Generation of the Layout

The generation of layouts occurs in several steps. The first step includes the generation of molecule graphs, which describe how each molecule can be built. Figure 3 shows graphs for three different molecules that are used in the instance depicted in Figure 1.

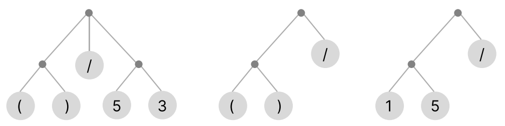


Figure 3: The structure of molecules that are build during the game shown in Figure 1. The subtrees correspond to sub-molecules which need to be built first, before merging them with other atoms.

As shown, each molecule corresponds to a tree, where the leaf nodes contain symbols (these are the atoms that the agent can collect). The sub-trees correspond to sub-molecules. For example, for the leftmost graph, the agent first needs to create a sub-molecule from atoms *),(* before it can compose it with atom */* or the second sub-molecule which is built from atoms *5,3*. In total, we instantiate 20 unique atoms from which 20 unique sub-molecules are built. From these atoms and sub-molecules, 50 unique molecules are built. These are generated only once and used in each game instance. One particular molecule is always used as the target molecule for each game.

To generate the concrete game instance, we first sample the number of molecules *n* that will be required to finish the game. Then we sample *n* random molecules from the 50 available molecules and create a *dependency graph* determining the order in which these molecules could be built.

Once we have this graph, we generate the layout by creating *n* parts with a variable number of rooms in each part and connect these rooms by corridors where each corridor connecting two different parts is blocked by an obstacle. Finally, we position the atoms and the agent in the rooms in such a way that the molecules can be built in the order dictated by the dependency graph.

4.2 Generation of the Definition File

The generation of the definition file is dependent only on the dependency graph, which contains a description of the molecules used at its vertices. We first instantiate the *SpriteSet* and *LevelMapping* by creating an entity represented by a unique symbol for each atom, sub-molecule and molecule. Then we instantiate the rules of how these atoms interact with each other. For certain atoms/sub-molecules/molecules, nothing happens when they touch each other, and these interactions are handled by the *stepBack* event. For each pair of atoms/sub-molecules that can be combined, we create two rules. One rule uses the *killSprite* event to remove one of these entities, and the second rule uses the *transformTo* event to transform the second entity into the entity corresponding to its composition. Additionally, we instantiate rules determining what happens when these atoms/sub-molecules/molecules interact with obstacles, walls, or the agent.

The definition of the game instance is complete by adding the *TerminationSet* block, which is the same for each game and checks whether the target molecule has been built.

5 RELATED WORK

5.1 Reinforcement Learning with Subgoals

Originally, subgoals have been investigated in (Sutton et al., 1999b). Many articles have been published on the topic of subgoals or, more generally, on goal-oriented reinforcement learning, such as (Czechowski et al., 2021; Chane-Sane et al., 2021; Nasiriany et al., 2019; Eysenbach et al., 2019; Zawalski et al., 2023).. All of these methods have been studied on problems that are either too simplistic (the subgoal can correspond to moving the agent from one room to the other) or do not contain a clear notion of a subgoal (i.e., solving a Rubik's cube). A closely related topic, called Hierarchical Reinforcement Learning, studies RL algorithms that use the so-called *options* (Vezhnevets et al., 2017; Barto and Mahadevan, 2003; Sutton et al., 1999b; Precup and Sutton, 2000; Aubret et al., 2019). These are high-level actions that consist of many low-level ones. Most option-based algorithms were developed and tested on very basic environments, and therefore our environment can be used as a challenging benchmark for these algorithms.

5.2 Minimalistic Reinforcement Learning Environments

During the last decade, several frameworks have been developed for designing RL environments. They range from very complex ones, which are based on 3D rendering engines (Beattie et al., 2016; Juliani et al., 2019), through simulators of Atari games (Bellemare et al., 2013), to ones with minimalistic ASCII graphics such as MiniHack (Samvelyan et al., 2021) or PyVGDL (Schaul, 2013). We decided to build our environment on top of the minimalistic PyVGDL because we are interested only in testing the agent's ability to solve problems that can be decomposed into subgoals and we view the ability to deal with complex visual patterns as a separate issue.

6 CONCLUSION

We presented a minimalistic RL environment which is designed to test agents' ability to solve problems that can be naturally decomposed into subgoals. We believe that the ability to solve problems by first setting a subgoal and then achieving this subgoal is crucial for humans and will be crucial for RL agents in the future. We believe that the environment presented will provide a shared testing ground for researchers interested in this topic. In future work, we plan to release algorithms developed using this environment that will explore the state space by proposing subgoals to themselves.

ACKNOWLEDGEMENT

This scientific article is part of the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306. The results were supported by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902.

REFERENCES

Aubret, A., Matignon, L., and Hassas, S. (2019). A survey on intrinsic motivation in reinforcement learning. *CoRR*, abs/1908.06976.

Barto, A. G. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems*, 13(4):341–379.

Beattie, C., Leibo, J. Z., Teplyashin, D., Ward, T., Wainwright, M., Küttler, H., Lefrancq, A., Green, S., Valdés, V., Sadik, A., Schrittwieser, J., Anderson, K., York, S., Cant, M., Cain, A., Bolton, A., Gaffney, S., King, H., Hassabis, D., Legg, S., and Petersen, S. (2016). DeepMind lab.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. 47:253–279.

Chane-Sane, E., Schmid, C., and Laptev, I. (2021). Goal-conditioned reinforcement learning with imagined subgoals. *CoRR*, abs/2107.00541.

Czechowski, K., Odrzygózdz, T., Zbysinski, M., Zawalski, M., Olejnik, K., Wu, Y., Kucinski, L., and Milos, P. (2021). Subgoal search for complex reasoning tasks. *CoRR*, abs/2108.11204.

Eysenbach, B., Salakhutdinov, R., and Levine, S. (2019). Search on the replay buffer: Bridging planning and reinforcement learning.

Gavish, B. and Graves, S. C. (1978). The travelling salesman problem and related problems. Publisher: Massachusetts Institute of Technology, Operations Research Center.

Juliani, A., Khalifa, A., Berges, V.-P., Harper, J., Teng, E., Henry, H., Crespi, A., Togelius, J., and Lange, D. (2019). Obstacle tower: A generalization challenge in vision, control, and planning.

Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J. W., Songhori, E., Wang, S., Lee, Y.-J., Johnson, E., Pathak, O., Nazi, A., et al. (2021). A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning.

Nasiriany, S., Pong, V. H., Lin, S., and Levine, S. (2019). Planning with goal-conditioned policies.

Precup, D. and Sutton, R. S. (2000). *Temporal Abstraction in Reinforcement Learning*. phdthesis. ISBN: 0599844884.

Samvelyan, M., Kirk, R., Kurin, V., Parker-Holder, J., Jiang, M., Hambro, E., Petroni, F., Küttler, H., Grefenstette, E., and Rocktäschel, T. (2021). MiniHack the Planet: A Sandbox for Open-Ended Reinforcement Learning Research.

Schaul, T. (2013). A video game description language for model-based or interactive learning. In *Proceedings of the IEEE Conference on Computational Intelligence in Games*, Niagara Falls. IEEE Press.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm.

Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. (1999a). Policy gradient methods for reinforcement learning with function approximation. In Solla, S., Leen, T., and Müller, K., editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press.

Sutton, R. S., Precup, D., and Singh, S. (1999b). Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1):181–211.

Vereecken, R. (2018). PyVGDL 2.0. original-date: 2018-07-10T16:54:41Z.

Vezhnevets, A. S., Osindero, S., Schaul, T., Heess, N., Jaderberg, M., Silver, D., and Kavukcuoglu, K. (2017). Feudal networks for hierarchical reinforcement learning. *CoRR*, abs/1703.01161.

Zawalski, M., Tyrolski, M., Czechowski, K., Stachura, D., Piekos, P., Odrzygóźdź, T., Wu, Y., Kuciński, L., and Miłoś, P. (2023). Fast and Precise: Adjusting Planning Horizon with Adaptive Subgoal Search.

APPENDIX A

Sampled Environment

```
SpriteSet
        background > Immovable randomtiling=0.9 img=oryx/floor3 hidden=True
        avatar > MovingAvatar img=oryx/knight1
        wall > Immovable autotiling=true img=oryx/wall3
 movable >
          1  > Passive img=atom_sprites/alienShotgun_0.png
          3  > Passive img=atom_sprites/belt1.png
          4  > Passive img=atom_sprites/book1.png
          5  > Passive img=atom_sprites/bookDown.png
          6  > Passive img=atom_sprites/bookUp.png
          7  > Passive img=atom_sprites/boots1.png
          8  > Passive img=atom_sprites/bow1.png
          10  > Passive img=atom_sprites/bullet1.png
          16  > Passive img=atom_sprites/butterfly1.png
          100113  > Passive img=atom_sprites/candle1.png
          100115  > Passive img=atom_sprites/cape1.png
          20  > Passive img=atom_sprites/chair1.png
          100117  > Passive img=atom_sprites/chicken.png
          100118  > Passive img=atom_sprites/city1.png
          10040  > Passive img=atom_sprites/cloak1.png
          10059  > Passive img=atom_sprites/diamond1.png
          10062  > Passive img=atom_sprites/eggcracked.png
          10067  > Passive img=atom_sprites/eye1.png
          10070  > Passive img=atom_sprites/flag1.png
          10072  > Passive img=atom_sprites/goldsack.png
          10074  > Passive img=atom_sprites/heart1.png
          10075  > Passive img=atom_sprites/helmet1.png
 obstacle >
          101118  > Immovable img=obstacle_sprites/barrel1.png
          11067  > Immovable img=obstacle_sprites/barrel2.png
          11075  > Immovable img=obstacle_sprites/block1.png

LevelMapping
        A > background avatar
        w > wall
        . > background
        β > background 1
        δ > background 3
        ε > background 4
        ζ > background 5
        η > background 6
        θ > background 7
        ι > background 8
        λ > background 10
        ρ > background 16
        ѥ > background 100113
        Ѧ > background 100115
        υ > background 20
        ѧ > background 100117
        Ѩ > background 100118
        Þ > background 10040
```

Figure 4: Part 1: Sampled environment definition file.

```
SpriteSet
        background > Immovable randomtiling=0.9 img=oryx/floor3 hidden=True
        avatar > MovingAvatar img=oryx/knight1
        wall > Immovable autotiling=true img=oryx/wall3
 movable >
         1  > Passive img=atom_sprites/alienShotgun_0.png
         3  > Passive img=atom_sprites/belt1.png
         4  > Passive img=atom_sprites/book1.png
         5  > Passive img=atom_sprites/bookDown.png
         6  > Passive img=atom_sprites/bookUp.png
         7  > Passive img=atom_sprites/boots1.png
         8  > Passive img=atom_sprites/bow1.png
         10  > Passive img=atom_sprites/bullet1.png
         16  > Passive img=atom_sprites/butterfly1.png
         100113  > Passive img=atom_sprites/candle1.png
         100115  > Passive img=atom_sprites/cape1.png
         20  > Passive img=atom_sprites/chair1.png
         100117  > Passive img=atom_sprites/chicken.png
         100118  > Passive img=atom_sprites/city1.png
         10040  > Passive img=atom_sprites/cloak1.png
         10059  > Passive img=atom_sprites/diamond1.png
         10062  > Passive img=atom_sprites/eggcracked.png
         10067  > Passive img=atom_sprites/eye1.png
         10070  > Passive img=atom_sprites/flag1.png
         10072  > Passive img=atom_sprites/goldsack.png
         10074  > Passive img=atom_sprites/heart1.png
         10075  > Passive img=atom_sprites/helmet1.png
 obstacle >
         101118  > Immovable img=obstacle_sprites/barrel1.png
         11067  > Immovable img=obstacle_sprites/barrel2.png
         11075  > Immovable img=obstacle_sprites/block1.png

LevelMapping
        A > background avatar
        w > wall
        . > background
        β > background 1
        δ > background 3
        ε > background 4
        ζ > background 5
        η > background 6
        θ > background 7
        ι > background 8
        λ > background 10
        ρ > background 16
        ѥ > background 100113
        Ѧ > background 100115
        υ > background 20
        ѧ > background 100117
        Ѩ > background 100118
        Þ > background 10040
```

Figure 5: Part 2: Sampled environment definition file.

```
wwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwww....w
wwwwwwwwwwwwwwwwwwwwwwwww....w
wwwwwwwwwwwwwwwwwwwwwwwww....w
wwwwwwwwwwwwwwwwwwwwww.......w
wwwwwwwwwwwwwwwwwwwwww.ww....w
wwwwwwwwwwwwww..........wwwwww
wwwwwwwwwwwwww....ρ.....wwwwww
wwwwwwwwwwwwww..........wwwwww
wwwwwwwwwwwwww..........wwwwww
wwwwwwwwwwwwww..........wwwwww
wwwwwwww...δ.w..........wwwwww
wwwwwwww.....wwwwwwwЮwwww....w
wwwwwwww.....w........δ.w....w
wwwwwwww....ηw..........w..ι.w
wwwwwwww.....w..........w.υ..w
w......w.....w..........w....w
w......wwwww.wε.......λ.wθ...w
w......wwwww.w..........w....w
w......wwwww............w....w
w......wwwwwww..........wwww.w
w.ζ....Я................www..w
w...Aβ.wwwwwww..........www.ww
w......wwwwwwwwwwwwwwwwwww..ww
wwwwww.wwwwwwwwwwwwwwwwww..www
wwwwww........wwwwwwwwww..wwww
wwwwwwwwwwwww............wwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwww
wwwwwwwwwwwwwwwwwwwwwwwwwwwwww
```

Figure 6: Sampled environment layout.