Supplementary Material:
Learning Portable Representations for High-Level Planning

Steven James [1] **Benjamin Rosman** [1] **George Konidaris** [2]

1. Proof of Sufficiency

In this section, we show that a combination of agent-space symbols with problem-space partition labels provides a sufficient symbolic vocabulary for planning. We begin by defining the notion of $\mathcal{X}$-space options, whose initiation sets, policies and termination conditions are all defined in state space $\mathcal{X}$.

Definition 1. Let $\mathcal{O}^{\mathcal{X}}$ be the set of all options defined over some state space $\mathcal{X}$. That is, each option $o \in \mathcal{O}^{\mathcal{X}}$ has a policy $\pi_o : \mathcal{X} \to \mathcal{A}$, an initiation set $\mathcal{I}_o \subseteq \mathcal{X}$ and a termination function $\beta_o : \mathcal{X} \to [0, 1]$.

Problem-space options are thus denoted $\mathcal{O}^{\mathcal{S}}$, while $\mathcal{O}^{\mathcal{D}}$ are agent-space options. We now define a partitioned option as follows:

Definition 2. Given an option $o \in \mathcal{O}^{\mathcal{X}}$, define a relation $\sim_o$ on I_o so that $x \sim_o y \iff \Pr(x' \mid x, o) = \Pr(x' \mid y, o)$ for all $x, y, x' \in \mathcal{X}$. Then $\sim_o$ is an equivalence relation which partitions I_o. Label each equivalence class in $I_o / \sim_o$ with a unique integer α. A *partitioned subgoal option* is then the parameterised option $o(\alpha) = \langle [\alpha], \pi_o, \beta_o \rangle$, where $[\alpha] \subseteq I_o$ is the set of states in equivalence class α.

We define a *probabilistic plan* $p_Z = \{o_1, \ldots, o_n\}$ to be the sequence of options to be executed, starting from some state drawn from distribution Z. It is useful to introduce the notion of a *goal option*, which can only be executed when the agent has reached its goal. Appending the goal option to a plan means that the probability of successfully executing a plan is equivalent to the probability of reaching some goal. The act of planning now reduces to a search through the space of all possible plans— ending with a goal option—to find the one most likely to succeed.

Our representation must therefore be able to evaluate the probability that an arbitrary plan, ending in goal option, successfully executes. However, the options in the plan may be either problem- or agent-space options. In order to show that agent-space representations and their associated problem-space partition labels are sufficient for planning with both types of options, we first define a function that maps problem-space partitions to subsequent problem-space partitions:

Definition 3. A *linking function* L is a function that specifies the problem-space partition the agent will enter, given the current problem-space partition and executed option. That is, $L(\alpha, o, \beta) = \Pr(\beta \mid o, \alpha)$, where $o \in \mathcal{O}$, $\alpha, \beta \in \Lambda$ and Λ is the set of problem-space partitions induced by all options.

We next need the following result, which demonstrates that we are able to model the true dynamics using problem-space partitions and agent-space effects:

[1]School of Computer Science and Applied Mathematics, University of the Witwatersrand, Johannesburg, South Africa [2]Department of Computer Science, Brown University, Providence RI 02912, USA. Correspondence to: Steven James <steven.james@wits.ac.za>.

Proceedings of the 37th International Conference on Machine Learning, Online, PMLR 119, 2020. Copyright 2020 by the author(s).

Lemma 1. Let $\omega \in \mathcal{O}^{\mathcal{D}}$ be a partitioned agent-space option, and denote $\omega(\alpha)$ as that same option which has been further partitioned in problem space, with problem-space partition α. Let $s, s' \in \mathcal{S}$ and $x, x' \in \mathcal{D}$ such that $x' \sim \Pr(\cdot \mid x, s, \omega)$ and $s' \sim \Pr(\cdot \mid s, \omega(\alpha))$ with $\mathbb{E}[\phi(s')] = x'$. Finally, assume $s \in [\beta]$, where β is some partition label. Then,

$$\Pr(s' \mid s, x, x', \omega(\alpha), \beta) = \frac{g(x', \omega, \alpha, \beta)}{\int_{[\alpha]} \Pr(t \mid o(\alpha)) dt}, \text{ where}$$
$$g(x', \omega, \alpha, \beta) = \begin{cases} \Pr(x' \mid \omega) & \text{if } \beta = \alpha \\ 0 & \text{otherwise.} \end{cases}$$

Proof. Recall that $\omega(\alpha)$ obeys the subgoal property in both $\mathcal{D}$ and $\mathcal{S}$. Thus the transition probability is simply its image, given that it is executable at the current state. Thus we have:

$$\begin{aligned} \Pr(s' \mid s, x, x', \omega(\alpha), \beta) &= \Pr(x' \mid s \in [\alpha], \omega(\alpha), \beta) \\ &= \frac{\Pr(x', s \in [\alpha] \mid \omega(\alpha), \beta)}{\Pr(s \in [\alpha] \mid \omega(\alpha), \beta)}. \end{aligned}$$

Now $\beta \neq \alpha \implies s \notin [\alpha]$, and so $\Pr(x', s \in [\alpha] \mid \omega(\alpha), \beta) = 0$. Conversely, $\beta = \alpha \implies s \in [\alpha]$ and so $\Pr(x', s \in [\alpha] \mid \omega(\alpha), \beta) = \Pr(x' \mid \omega)$. Furthermore,

$$\Pr(s \in [\alpha] \mid \omega(\alpha), \beta) = \int_{[\alpha]} \Pr(t \mid \omega[\alpha]) dt.$$

Therefore, we have

$$\Pr(s' \mid s, x, x', \omega(\alpha), \beta) = \frac{g(x', \omega, \alpha, \beta)}{\int_{[\alpha]} \Pr(t \mid \omega(\alpha)) dt},$$

where g is defined above.

□

The above states that, if the starting state s is in the problem-space partition of the executed option, then the transition probabilities are exactly those under the subgoal option. However, if s is not in the correct partition, then the probability is 0 because we cannot execute the option. Thus we consider only starting states in $[\alpha]$ and set everything else to 0. Finally, we renormalise over $[\alpha]$ to ensure that the transition remains a proper distribution. This is sufficient to predict the effect in agent space, since we can just apply the observation function ϕ to s' to compute $\Pr(x' \mid s')$. We can now proceed with our main result:

Theorem 1. The ability to represent the preconditions and image of each option in agent space, together with the partitioning in $\mathcal{S}$, is sufficient for determining the probability of being able to execute any probabilistic plan p from starting distribution Z.

Proof. For notational convenience, we denote ω as a partitioned agent-space option, $\omega(\alpha)$ as a partitioned agent-space option with *problem-space* partition α, and $o(\alpha)$ as a problem-space option with $I_o = [\alpha] \subseteq \mathcal{S}$. Because the only difficulty lies in evaluating the precondition of a problem-space option, assume without loss of generality that $p_Z = \{\omega_0, \ldots, \omega_{n-1}, o(\alpha_n)\}$. p_Z is a plan consisting of a number of agent-space options followed by a problem-space option. Finally, we note that Z is a start distribution over $\mathcal{D}$ and $\mathcal{S}$. We denote the initial agent- and problem-space distributions as D_0 and S_0 respectively.

The image of an option in agent space is specified by the image operator

$$Z_{i+1} = \text{Im}(Z_i, \omega_i; \alpha_i), \text{ with } Z_0 = D_0.$$

Note that the agent-space image is conditioned on the problem-space partition, as Lemma 1 showed that we required it to compute the effects in agent space. We can define the problem-space image similarly, although we will not require it to learn a sufficient representation:

$$\hat{Z}_{i+1} = \text{Im}(\hat{Z}_i, \omega_i, \alpha_i), \text{ with } \hat{Z}_0 = S_0.$$

The probability of being able to execute p_Z is given by

$$\Pr(x_0 \in I_{\omega_0}, \ldots, x_{n-1} \in I_{\omega_{n-1}}, s_n \in I_{o(\alpha_n)}),$$

where $x_i \sim Z_i$ and $s_n \sim \hat{Z}_n$. By the Markov property, we can write this as

$$\Pr(s_n \in I_{o(\alpha_n)}) \prod_{i=0}^{n-1} \left[\Pr(x_i \in I_{\omega_i})\right].$$

If we can estimate the starting problem-space partition α_0 and linking function L, then we can evaluate this quantity as follows:

$$\begin{aligned}
\Pr(s_n \in I_{o(\alpha_n)}) &= \Pr(s_n \in [\alpha_n]) \\
&= \Pr(s_0 \in [\alpha_0]) \prod_{i=0}^{n-1} L(\alpha_i, \omega_i(\alpha_i), \alpha_{i+1}) \\
&= \int_{\mathcal{S}} \Pr(s \in [\alpha_0]) S_0(s) ds \times \prod_{i=0}^{n-1} L(\alpha_i, \omega_i(\alpha_i), \alpha_{i+1}),
\end{aligned}$$

and

$$\Pr(x_i \in I_{\omega_i}) = \prod_{j=1}^{i} L(\alpha_{j-1}, \omega_{j-1}, \alpha_j) \times \int_{\mathcal{D}} \Pr(x_i \in I_{\omega_i}) Z_i(x; \alpha_i) dx.$$

Thus by learning the precondition and image operators in $\mathcal{D}$, partitioning the options in problem-space, and learning the links between these partitions, we can evaluate the probability of an arbitrary plan executing. □

2. Learning a Portable Representation in Agent Space

Partitioning We collect data from a task by executing options uniformly at random and scale the state variables to be in the range $[0, 1]$. We record state transition data as well as, for each state, which options could be executed. We then partition options using the DBSCAN clustering algorithm ($\epsilon = 0.03$) to cluster the terminating states of each option into separate effects, which approximately preserves the subgoal property.

Preconditions Next, the agent learns a precondition classifier for each of these approximately partitioned options using an SVM with Platt scaling. We use states initially collected as negative examples, and data from the actual transitions as positive examples. We employ a simple feature selection procedure to determine which state variables are relevant to the option's precondition. We first compute the accuracy of the SVM applied to all variables, performing a grid search to find the best hyperparameters for the SVM using 3-fold cross validation. Then, we check the effect of removing each state variable in turn, recording those that cause the accuracy to decrease by at least 0.02. Finally, we check whether adding each of the state variables back improves the SVM, in which case they are kept too. Having determined the relevant features, we fit a probabilistic SVM to the relevant state variables' data.

Effects A kernel density estimator with Gaussian kernel is used to estimate the effect of each partitioned option. We learn distributions over only the variables affected by the option. We use a grid search with 3-fold cross validation to find the best bandwidth hyperparameter for each estimator. Each of these KDEs is an abstract symbol in our propositional PDDL representation.

Propositional PDDL For each partitioned option, we now have a classifier and set of effect distributions (propositions). However, to generate the PDDL, the precondition must be specified in terms of these propositions. We use the same approach as prior work to generate the PDDL: for all combinations of valid effect distributions, we test whether data sampled from their conjunction is evaluated positively by our classifiers. If they are, then that combination of distributions serves as the precondition of the high-level operator.

3. Learning Linking Functions

We can learn linking functions by simply executing options, and recording for each transition the start and end partition labels. Let $\Gamma^{(o)}$ be the set of problem-space partition labels for option o, and $\Lambda = \bigcup_{o \in \mathcal{O}} \Gamma^{(o)}$ the set of all partition labels over all options. Note that each label $\lambda \in \Lambda$ refers to a set of initiation states $[\lambda] \subseteq \mathcal{S}$. We present a simple count-based approach to learning these functions, but note that any appropriate function-learning scheme would suffice:

1. Given a set of agent-space subgoal options that have subsequently been partitioned in $\mathcal{S}$, gather data from trajectories, recording tuples $\langle s, d, o, s', d' \rangle$ representing initial states in both $\mathcal{S}$ and $\mathcal{D}$, the executed option, and the subsequent states.

2. Determine the start and end partitions of the transition. The start partition is the singleton $c = \{\gamma \mid \gamma \in \Gamma^{(o)}, s \in [\Gamma^{(o)}]\}$, while the end labels are given by the set $\beta = \{\lambda \mid \lambda \in \Lambda, s' \in [\lambda]\}$. In practice, we keep all states belonging to each partition and then calculate the L2-norm to the closest states in each partition. We select those partitions whose distance is less than some threshold.

3. Denote L_o as the linking function for option o which stores the number of times transitions between different partition labels occur. Increment the existing count stored by $L_o(c, \beta)$, and keep count of the number of times the entry (o, c) has been updated.

4. Normalise the linking functions L_o by dividing the frequency counts by the number of times the entry for c was updated. We have now learned the link between the parameters of the precondition and effect for each option.

4. PPDDL Description for the Navigation Task

```
; Automatically generated ToyDomainV0 domain PPDDL file.
(define (domain ToyDomain)
        (:requirements :strips :probabilistic-effects :conditional-effects :rewards :fluents)
        (:predicates
                (notfailed)
                (wall-junction)
                (window-junction)
                (dead-end)
        )

        (:functions (partition))

        ;Action Inward-partition-0
        (:action Inward_0
         :parameters()
         :precondition (and (dead-end) (notfailed))
         :effect (and (when (= (partition) 6) (and (wall-junction) (not (dead-end)
                                              (decrease (reward) 1.00) (assign (partition) 5))))
                             (when (= (partition) 3) (and (wall-junction) (not (dead-end)
                                               (decrease (reward) 1.00) (assign (partition) 4))))
                             (when (= (partition) 1) (and (window-junction) (not (dead-end)
                                               (decrease (reward) 1.00) (assign (partition) 2))))
                             (when (= (partition) 8) (and (window-junction) (not (dead-end)
                                               (decrease (reward) 1.00) (assign (partition) 7))))
                    )
        )


        ;Action Outward-partition-0
        (:action Outward_1
         :parameters()
         :precondition (and (wall-junction) (notfailed))
         :effect (and (when (= (partition) 2) (and (dead-end) (not (wall-junction)
                                              (decrease (reward) 1.00) (assign (partition) 1))))
                             (when (= (partition) 5) (and (dead-end) (not (wall-junction)
                                               (decrease (reward) 1.00) (assign (partition) 6))))
                             (when (= (partition) 4) (and (dead-end) (not (wall-junction)
                                               (decrease (reward) 1.00) (assign (partition) 3))))
                             (when (= (partition) 7) (and (dead-end) (not (wall-junction)
                                               (decrease (reward) 1.00) (assign (partition) 8))))
                    )
        )


        ;Action Outward-partition-0
        (:action Outward_2
         :parameters()
         :precondition (and (window-junction) (notfailed))
         :effect (and (when (= (partition) 2) (and (dead-end) (not (window-junction)
                                              (decrease (reward) 1.00) (assign (partition) 1))))
                             (when (= (partition) 5) (and (dead-end) (not (window-junction)
                                               (decrease (reward) 1.00) (assign (partition) 6))))
                             (when (= (partition) 4) (and (dead-end) (not (window-junction)
                                               (decrease (reward) 1.00) (assign (partition) 3))))
                             (when (= (partition) 7) (and (dead-end) (not (window-junction)
                                               (decrease (reward) 1.00) (assign (partition) 8))))
                    )
        )


        ;Action Clockwise-partition-0
        (:action Clockwise_3
         :parameters()
         :precondition (and (wall-junction) (notfailed))
         :effect (and (when (= (partition) 4) (and (window-junction) (not (wall-junction)
                                              (decrease (reward) 1.00) (assign (partition) 2))))
                             (when (= (partition) 5) (and (window-junction) (not (wall-junction)
                                               (decrease (reward) 1.00) (assign (partition) 7))))
                    )
        )


        ;Action Clockwise-partition-1
        (:action Clockwise_4
         :parameters()
         :precondition (and (window-junction) (notfailed))
         :effect (and (when (= (partition) 7) (and (wall-junction) (not (window-junction)
```

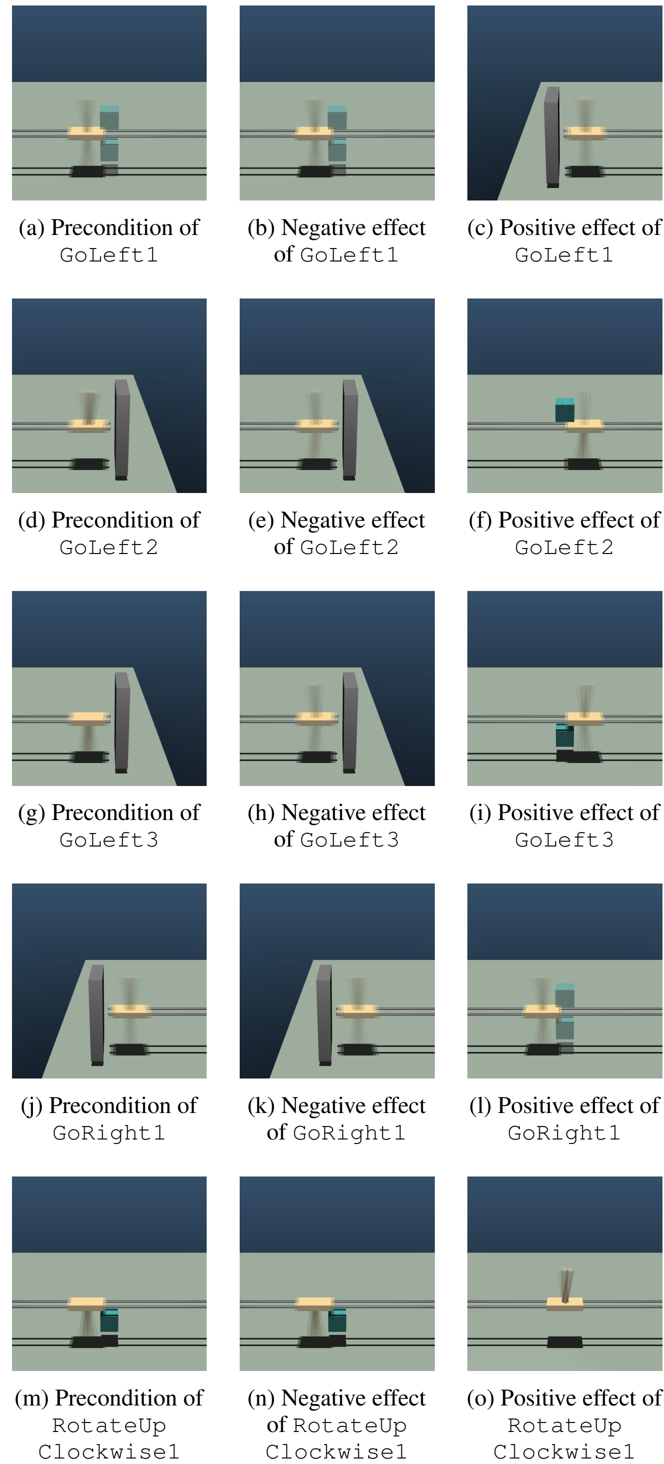
```
                                        (decrease (reward) 1.00) (assign (partition) 4))))
                              (when (= (partition) 2) (and (wall-junction) (not (window-junction)
                                         (decrease (reward) 1.00) (assign (partition) 5))))
                     )
        )


        ;Action Anticlockwise-partition-0
        (:action Anticlockwise_5
         :parameters()
         :precondition (and (window-junction) (notfailed))
         :effect (and (when (= (partition) 7) (and (wall-junction) (not (window-junction)
                                        (decrease (reward) 1.00) (assign (partition) 5))))
                              (when (= (partition) 2) (and (wall-junction) (not (window-junction)
                                         (decrease (reward) 1.00) (assign (partition) 4))))
                     )
        )


        ;Action Anticlockwise-partition-1
        (:action Anticlockwise_6
         :parameters()
         :precondition (and (wall-junction) (notfailed))
         :effect (and (when (= (partition) 4) (and (window-junction) (not (wall-junction)
                                        (decrease (reward) 1.00) (assign (partition) 7))))
                              (when (= (partition) 5) (and (window-junction) (not (wall-junction)
                                         (decrease (reward) 1.00) (assign (partition) 2))))
                     )
        )

)
```

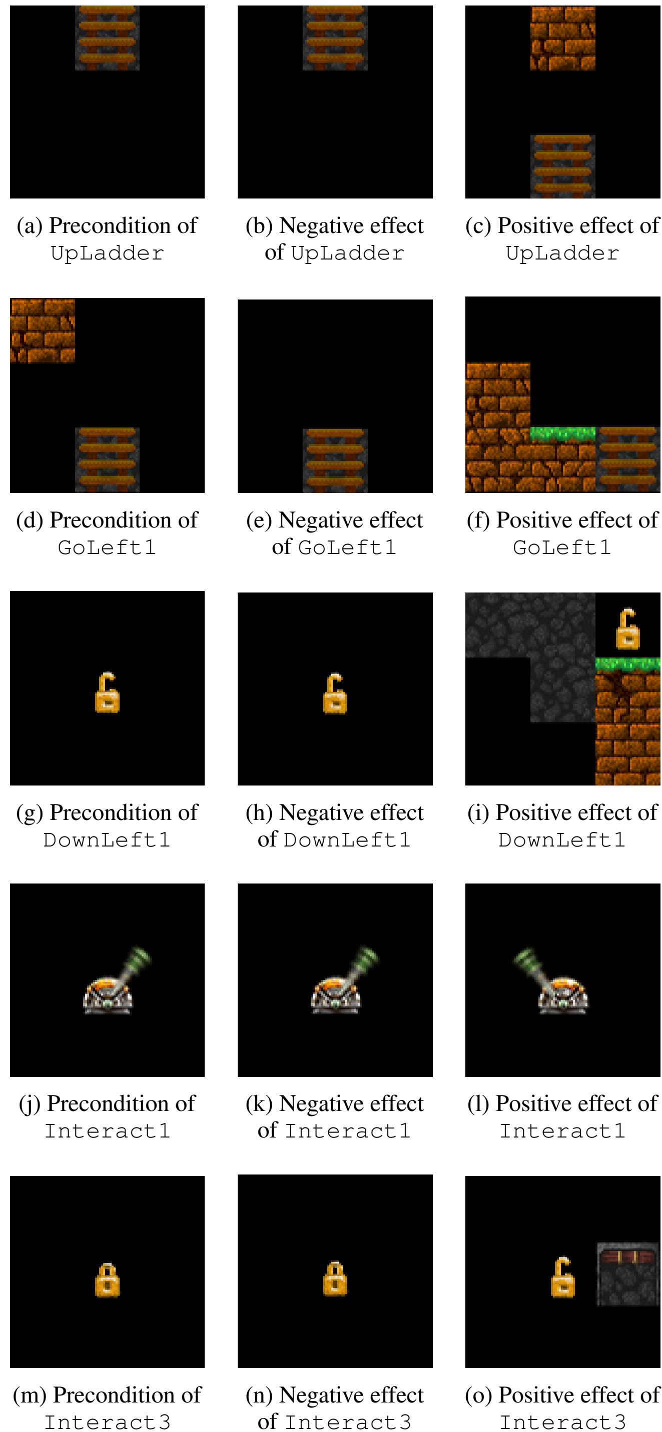
5. Examples of Portable *Rod-and-Block* Rules



(a) Precondition of GoLeft1
(b) Negative effect of GoLeft1
(c) Positive effect of GoLeft1
(d) Precondition of GoLeft2
(e) Negative effect of GoLeft2
(f) Positive effect of GoLeft2
(g) Precondition of GoLeft3
(h) Negative effect of GoLeft3
(i) Positive effect of GoLeft3
(j) Precondition of GoRight1
(k) Negative effect of GoRight1
(l) Positive effect of GoRight1
(m) Precondition of RotateUp Clockwise1
(n) Negative effect of RotateUp Clockwise1
(o) Positive effect of RotateUp Clockwise1

Figure 1: A subset of symbolic rules learned for the task in Figure 8.

6. Examples of Portable *Treasure Game* Rules



(a) Precondition of `UpLadder`
(b) Negative effect of `UpLadder`
(c) Positive effect of `UpLadder`
(d) Precondition of `GoLeft1`
(e) Negative effect of `GoLeft1`
(f) Positive effect of `GoLeft1`
(g) Precondition of `DownLeft1`
(h) Negative effect of `DownLeft1`
(i) Positive effect of `DownLeft1`
(j) Precondition of `Interact1`
(k) Negative effect of `Interact1`
(l) Positive effect of `Interact1`
(m) Precondition of `Interact3`
(n) Negative effect of `Interact3`
(o) Positive effect of `Interact3`

Figure 2: A subset of symbolic rules learned in Level 1

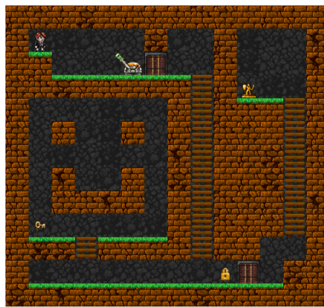
7. *Treasure Game* Level Layouts

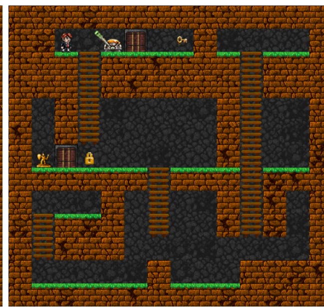
(a) Level 1

(b) Level 2

(c) Level 3

(d) Level 4


(e) Level 5


(f) Level 6

(g) Level 7

(h) Level 8

(i) Level 9

(j) Level 10