Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning

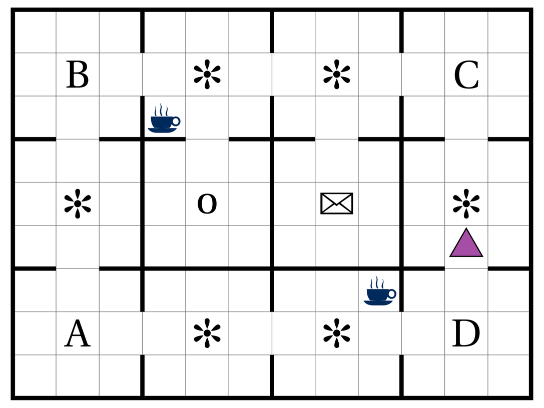
Rodrigo Toro Icarte[†‡] Toryn Q. Klassen[†] Richard Valenzano[§] Sheila A. McIlraith[†‡]

[†]Department of Computer Science, University of Toronto [‡]Vector Institute [§]Element AI

[†]{rntoro, toryn, sheila}@cs.toronto.edu, [§]rick.valenzano@elementai.com

Abstract. In this paper we propose Reward Machines – a type of finite state machine that supports the specification of reward functions while exposing reward function structure to the learner and supporting decomposition. We then present Q-Learning for Reward Machines (QRM), an algorithm which appropriately decomposes the reward machine and uses off-policy q-learning to simultaneously learn subpolicies for the different components. QRM is guaranteed to converge to an optimal policy in the tabular case, in contrast to Hierarchical Reinforcement Learning methods which might converge to suboptimal policies. We demonstrate this behavior experimentally in two discrete domains. We also show how function approximation methods like neural networks can be incorporated into QRM, and that doing so can find better policies more quickly than hierarchical methods in a domain with a continuous state space.

Running Example



Symbol	Meaning
▲	Agent
✻	Furniture
☕	Coffee machine
✉	Mail room
o	Office
A, B, C, D	Marked locations

Motivation

Task: Patrol A, B, C, and D.

Steps to solve the task using RL:

- Someone programs a reward function.
- The learning agent gets the reward function as a black box.

```
m = 0 # global variable
def get_reward(s):
  if m == 0 and s.at("A"):
    m = 1
  if m == 1 and s.at("B"):
    m = 2
  if m == 2 and s.at("C"):
    m = 3
  if m == 3 and s.at("D"):
    m = 0
    return 1
  return 0
```

→ **Reward Function**

What if we give the agent access to the reward function's code?

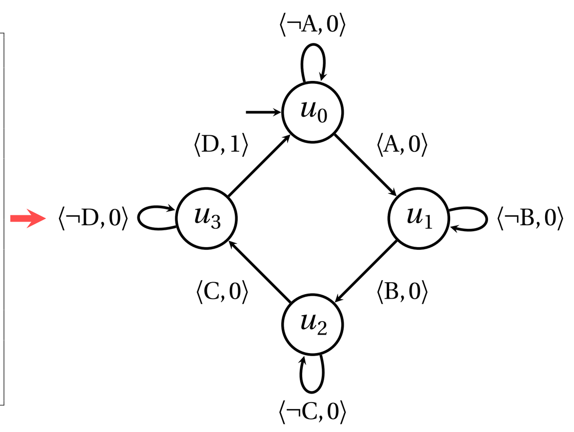
Advantage: The agent can exploit the reward structure! How?

What is a Reward Machine (RM)?

Idea: We encode reward functions using a finite state machine.


```
m = 0 # global variable
def get_reward(s):
  if m == 0 and s.at("A"):
    m = 1
  if m == 1 and s.at("B"):
    m = 2
  if m == 2 and s.at("C"):
    m = 3
  if m == 3 and s.at("D"):
    m = 0
    return 1
  return 0
```


→



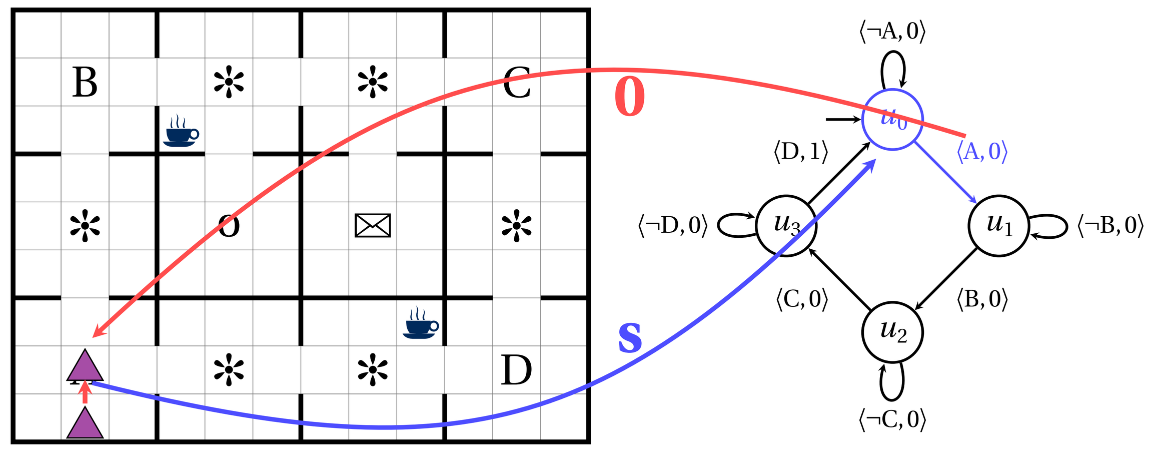
A **reward machine** consists of the following elements:

- A finite set of states U.
- An initial state $u_0 \in U$.
- A set of transitions, each labelled by:
  - a logical condition over properties of the state (e.g. at A, B, ☕, ✉, ...)
  - and a reward function.

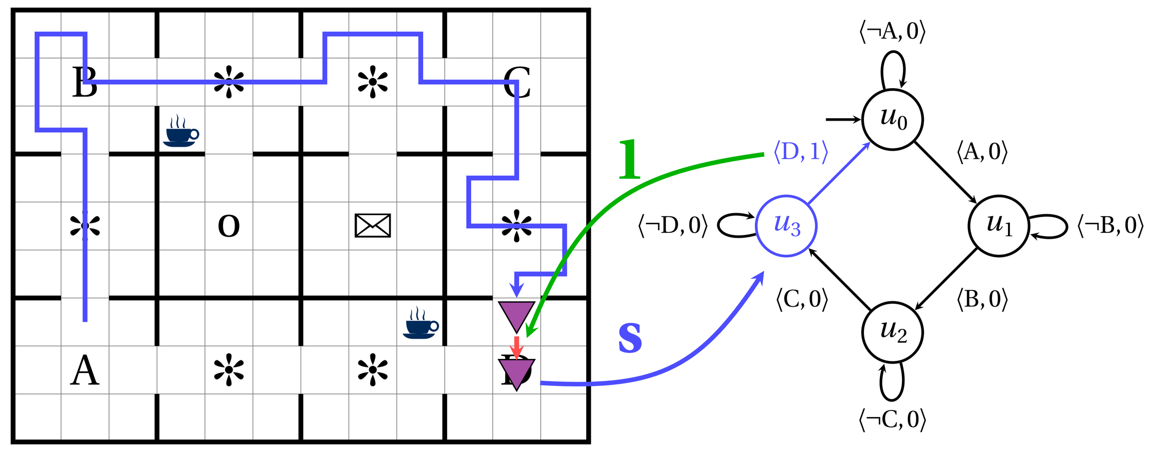
After each move in the environment, an RM makes the transition whose logical condition is satisfied by the environment state, and rewards the agent according to that transition's reward function.

Reward Machines in Action

This RM starts in u_0 and transitions to u_1 when A is reached. The agent gets a reward of 0 from that transition's reward function.

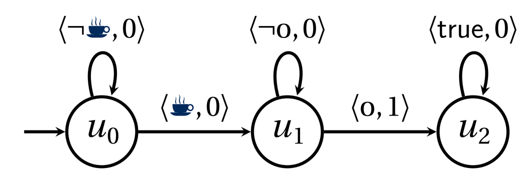


Positive reward is given only when the agent completes a cycle.

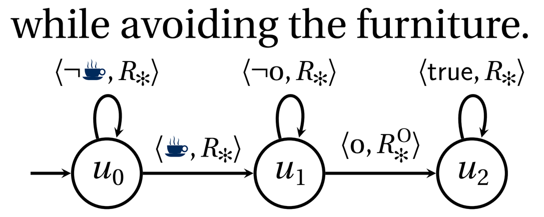


Other Examples of Reward Machines

Deliver coffee to the office.

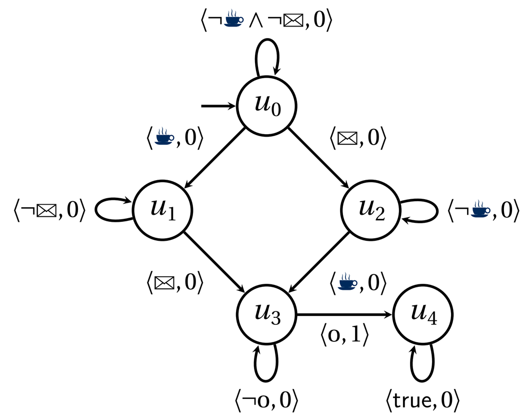


Deliver coffee to the office while avoiding the furniture.

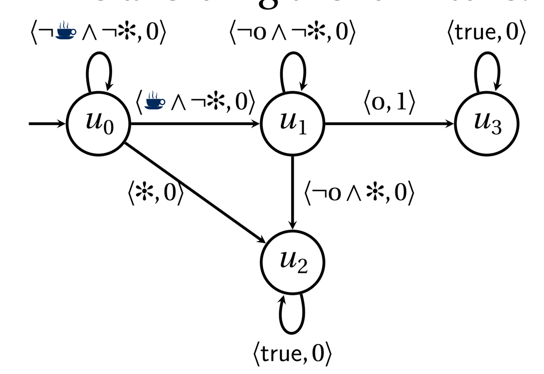


where $R_{*} = -1$ iff the agent is at ✻ (zero otherwise) and R_{*}^{O} is like R_{*} but also gives a reward of 1 when the office is reached.

Deliver coffee and the mail to the office.



Deliver coffee to the office while avoiding the furniture.



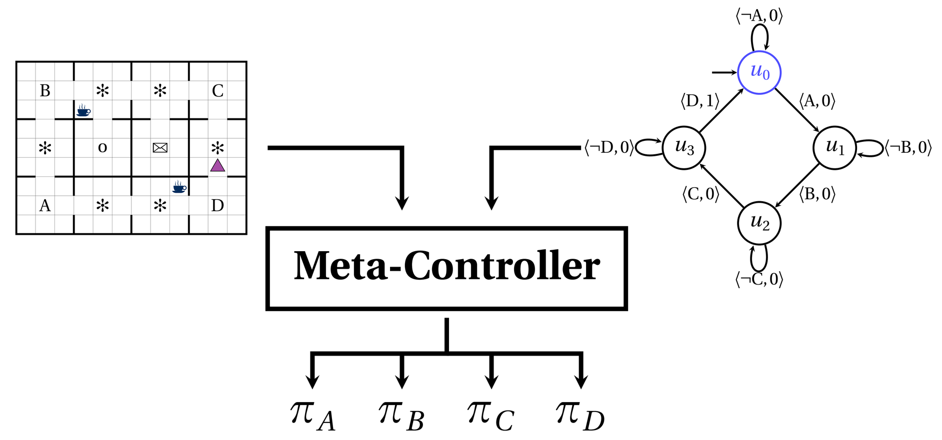
How to exploit an RM's structure

First idea (q-learning baseline)

Reward machines might produce non-Markovian rewards w.r.t. the environment states. To solve this, we add the RM state to the agent's state representation and use q-learning.

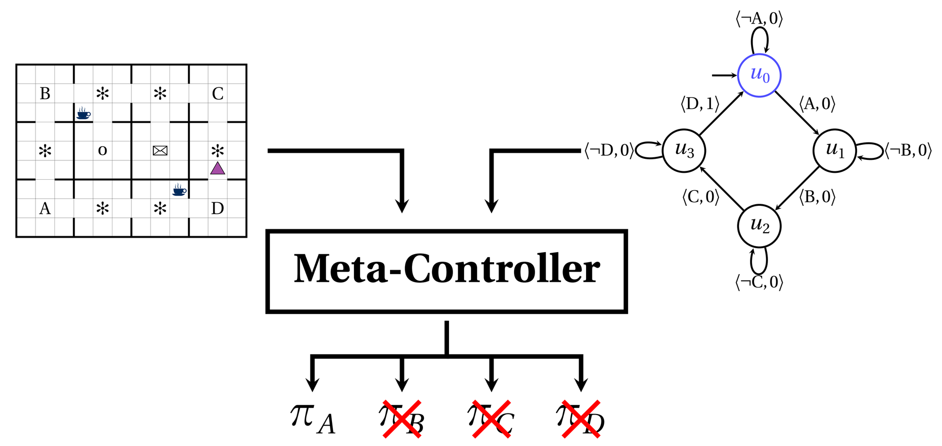
Second idea (HRL baseline)

Use the RM to extract *options* and learn using Hierarchical RL.

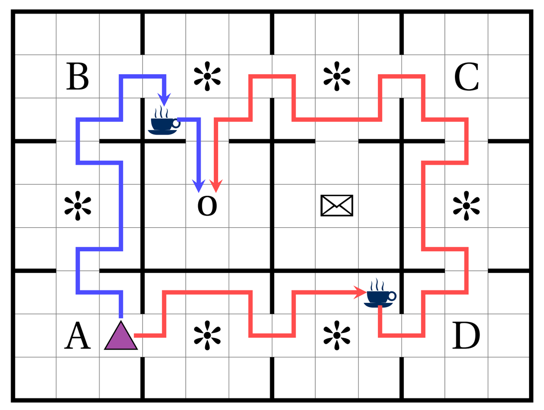


Third idea (HRL-RM baseline)

Use the RM to also prune *useless* options.

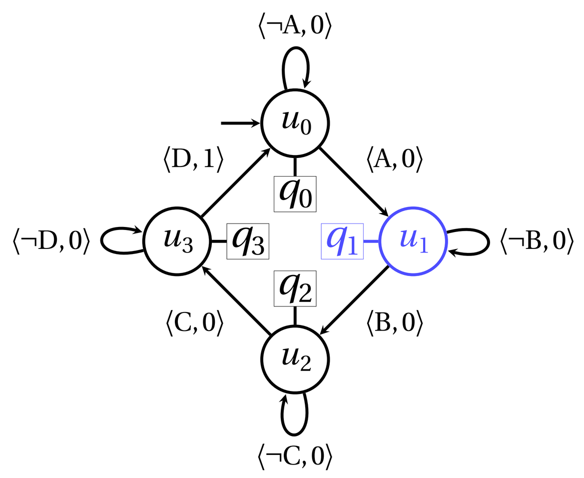


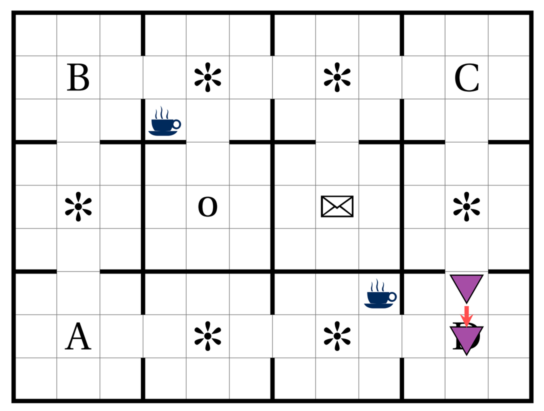
Problem: Hierarchical RL might converge to suboptimal policies!



Our final method (QRM)

- Learn one policy (q-function) per state in the RM.
- Select actions using the policy of the current RM state.
- Reuse experience to update all the q-values at the same time.





Q-updates

$$q_0(s,a) \xleftarrow{\alpha} 0+\gamma \max_{a'} q_0(s',a') \qquad q_2(s,a) \xleftarrow{\alpha} 0+\gamma \max_{a'} q_2(s',a')$$

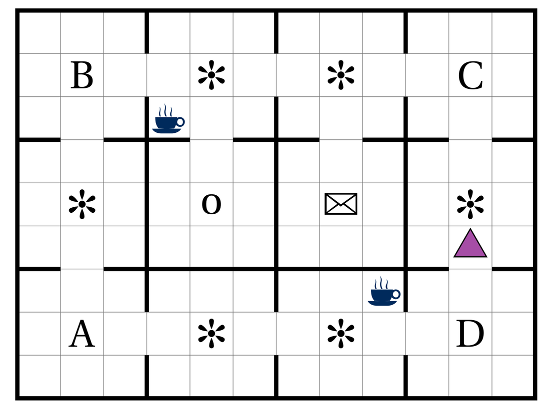
$$q_1(s,a) \xleftarrow{\alpha} 0+\gamma \max_{a'} q_1(s',a') \qquad q_3(s,a) \xleftarrow{\alpha} 1+\gamma \max_{a'} q_0(s',a')$$

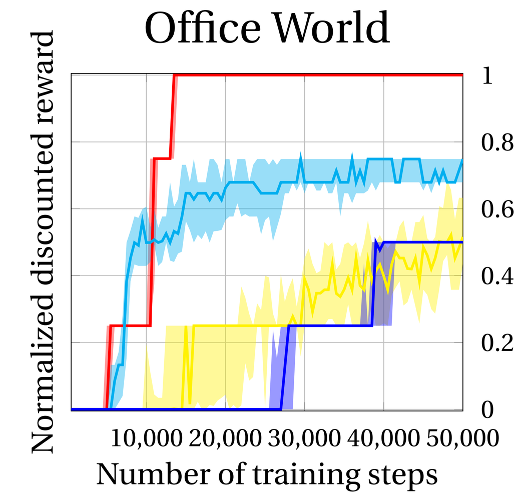
Theorem: QRM converges to an optimal policy in the limit.

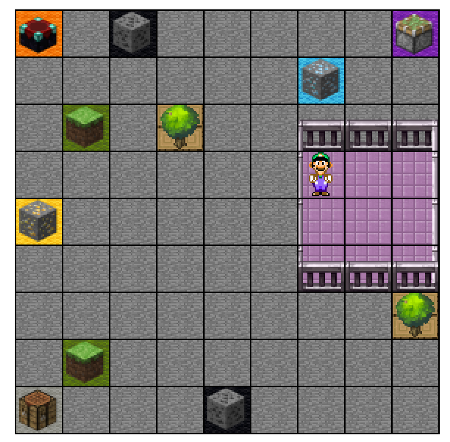
Code (coming soon at): `bitbucket.org/RToroIcarte/qrm`

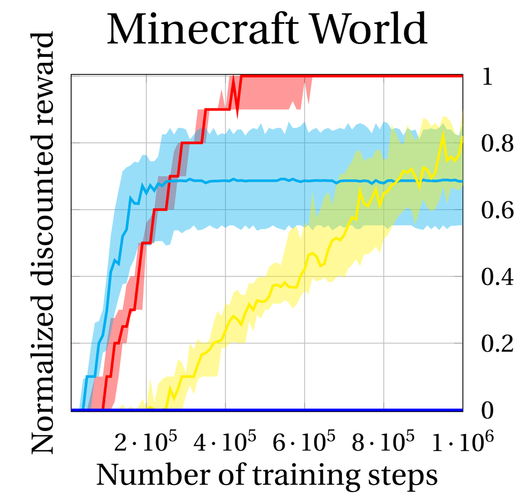
Results

Discrete Domains





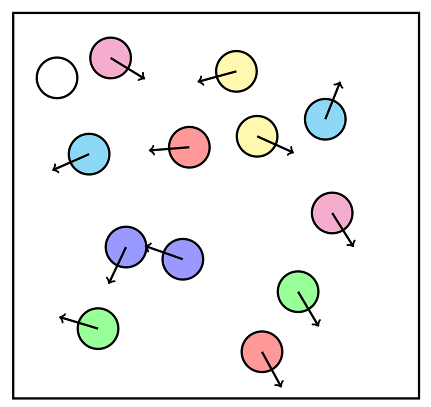


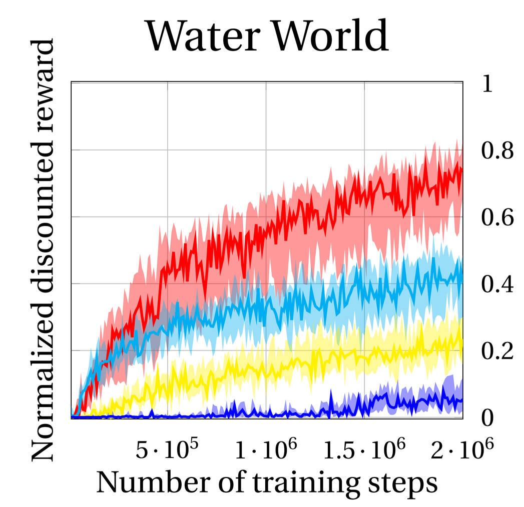


Legend: — Q-Learning — HRL — HRL-RM — QRM

Continuous Domains

Deep QRM uses DDQN with prioritized experience replay.





Legend: — DDQN — DHRL — DHRL-RM — DQRM

Conclusion

"To summarize, a nice simple idea exposing more of the structure of an RL problem and the benefits thereof."

— Third reviewer