Lecture 2: Reward Machines

Natasha Alechina[1,2] and Brian Logan[2,3]

[1]Open University NL, [2]Utrecht University, [3]University of Aberdeen

natasha.alechina@ou.nl, b.s.logan@uu.nl

Outline of this lecture

Specifying reward functions

Recap: specifying reward functions

- agents based on reinforcement learning have been remarkably successful in a variety of domains
- however specifying reward functions can be difficult, particularly when tasks are complex and/or safety critical

Recap: declarative reward specifications

- **key idea:** instead of using a reward function, we can use a declarative specification in a logical language to generate rewards
- makes it easier to specify and explain the behaviour of the agent
- guarantees that the agent's policy conforms to specification
- may also make learning more efficient (fewer episodes needed)

Reward Machines

Structured reward functions

- one way to overcome the difficulties of specifying reward functions is to expose the structure of the task to the learning agent, e.g., in the form of an automaton
- a reward machine (RM) is a Mealy machine (automaton with outputs)
- states represent abstract ‘steps’ or ‘phases’ in a task
- transitions correspond to observations of *high-level events* in the environment indicating that an abstract step/phase in the task has (or has not) been completed

Reward machines

Definition (Simple Reward Machine, Toro Icarte et al. (2022))

A reward machine is a tuple $R = (U, u_I, F, \Sigma, \delta_R, r_R)$ where:

- U is a finite non-empty set of states;
- $u_I \in U$ is the initial state;
- F is a set of terminal states ($U \cap F = \emptyset$);
- Σ is a finite set of environment events;
- $\delta_R : U \times \Sigma \to U \cup F$ is a transition function that, for every state $u \in U$ and environment event $\sigma \in \Sigma$, gives the state resulting from observing event σ in state u; and
- $r_R : U \times \Sigma \to \mathbb{R} \cup \{-\infty\}$ is a reward function that for every state $u \in U$ and event $\sigma \in \Sigma$ gives the reward resulting from observing event σ in state u.

Labelling function

- we assume that the events Σ are generated by a *labelling function*

$$L : S \times Act \times S \to 2^{\overline{\mathcal{PV}}}$$

  where $\mathcal{PV}$ is a set of propositions representing high-level events in the environment, and $\overline{\mathcal{PV}}$ is the set of literals derived from $\mathcal{PV}$

- for example, a labelling could consist of (a relevant subset of) the postconditions of actions

Task completion Reward Machines

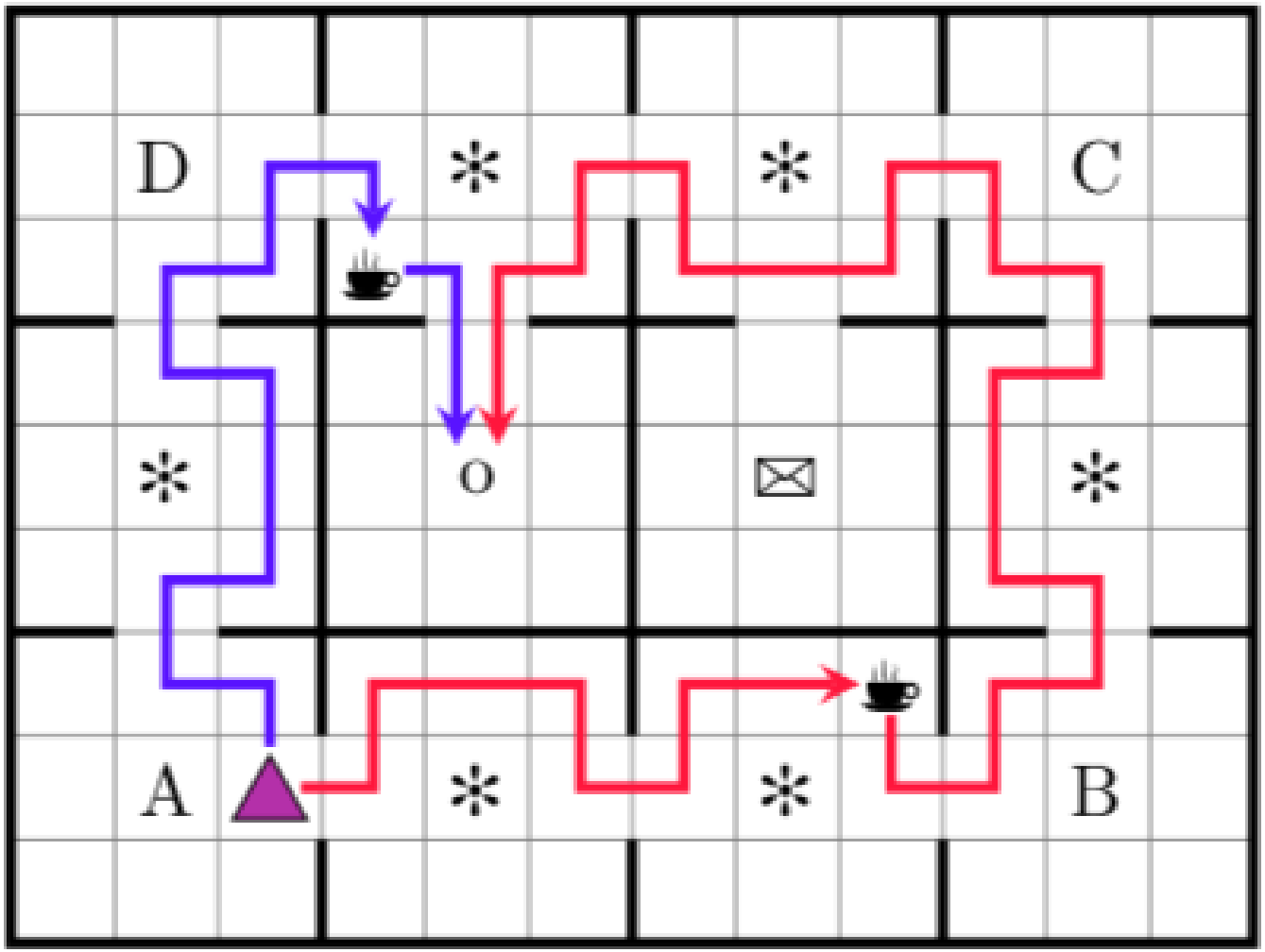
A task completion reward machine is an RM which:

- only produces a non-zero rewards on transition to a terminal state
- if the terminal state is accepting, the reward is positive, e.g., 1
- if the terminal state is not accepting (e.g., indicates the violation of an invariant), the reward is negative, e.g., $-\infty$

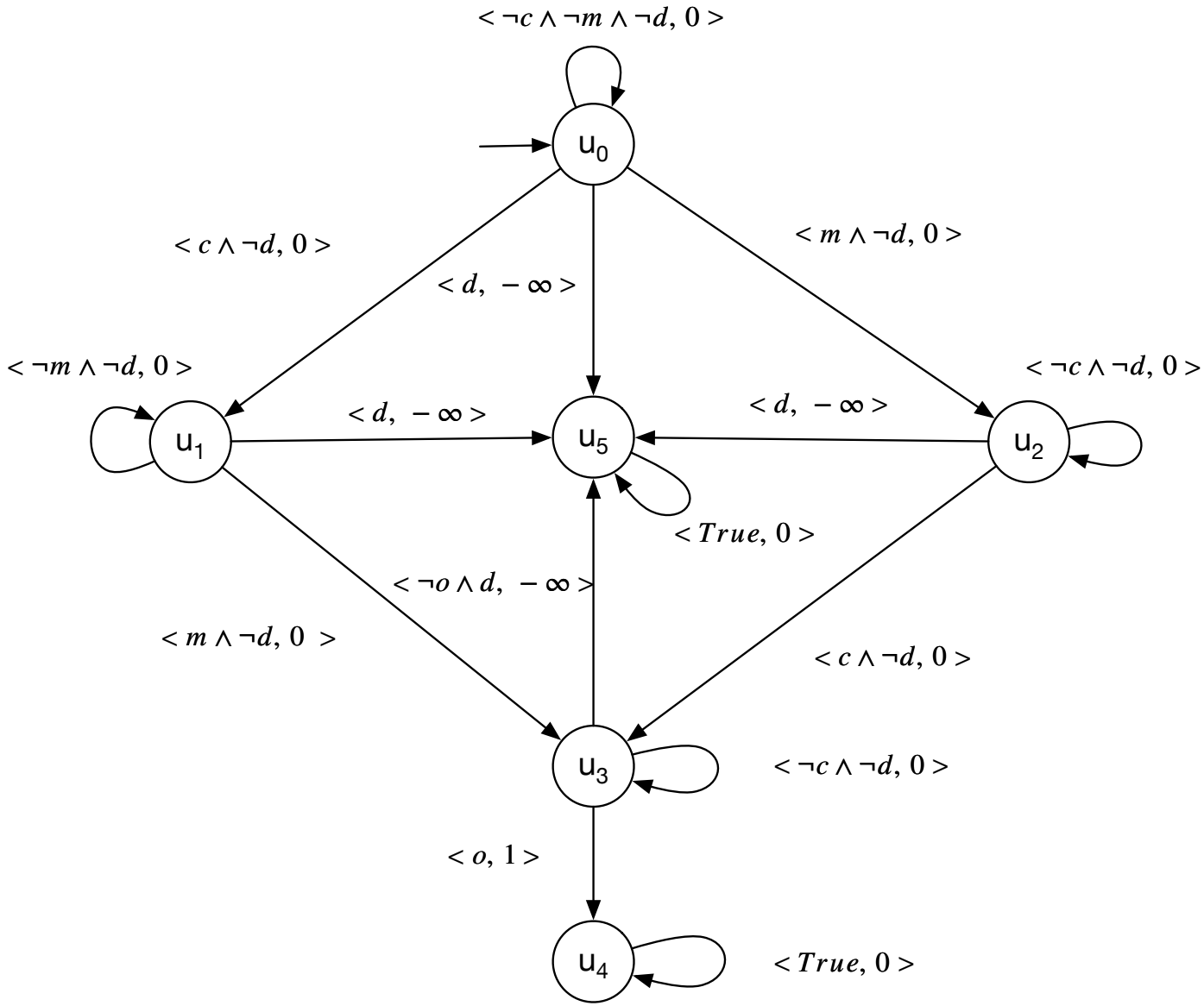
Example: OfficeWorld

- RMs allow the specification of non-Markovian rewards, i.e., rewards based on the history of the agent's actions
- OfficeWorld is a simple grid-world environment that includes coffee stations and a mail room
- not all rooms are connected to adjacent rooms, and some rooms contain fragile "decorations"
- goal of the agent is to deliver coffee and mail (in either order) exactly once while not stepping on the decorations
- difficult to do with Markovian reward functions without task-specific modifications to states of the MDP

Example:OfficeWorld



OfficeWorld: reward machine



From LTL to reward machines

Specifying reward machines

- reward machines can be computed from task specifications expressed in a range of property specification languages, including regular expressions and goal specification languages
- in particular, they can be computed from task specifications expressed in LTL and LTL_f (LTL on finite traces)
- any optimal policy learned using a task completion RM expressed in LTL_f is guaranteed to satisfy the LTL_f specification (if it is possible to satisfy it at all)

OfficeWorld: specification of behaviour in LTL_f

- the agent should reach a state where coffee and mail are true (in any order), and then return to the office and maintain stop forever, without stepping on decorations

$$G\neg \mathrm{d} \wedge (F(\mathrm{c} \wedge F(\mathrm{m} \wedge F(\mathrm{o} \wedge XG\,\mathrm{f}))) \vee F(\mathrm{m} \wedge F(\mathrm{c} \wedge F(\mathrm{o} \wedge XG\,\mathrm{f}))))$$

- where d, c, m, o and f are propositions denoting states in which the agent has: stepped on a decoration, collected coffee, collected mail, is at the office, and is stopped

LTL on finite traces

- reward machines can be specified LTL and many of its fragments, e.g., Safety LTL, CoSafety LTL
- we will focus on LTL on finite traces (LTL_f)
- makes it easy to mix reachability properties with invariants
- straightforward translation to DFAs (and hence to RMs)
- many AI problems have a natural interpretation in terms of finite traces, e.g., episodic tasks

Syntax of LTL$_f$

The syntax of LTL$_f$ formulas is given by:

Definition (Syntax of LTL$_f$)

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid X\varphi \mid \overline{X}\varphi \mid F\varphi \mid G\varphi \mid \varphi_1 \, U \, \varphi_2$$

- note that this is nearly identical to the syntax of LTL
- however LTL$_f$ adds a "weak next", $\overline{X}$, which is defined as

$$\overline{X}\varphi = \neg X \neg \varphi$$

- there are also small differences in the semantics

Semantics of LTL_f

Formulas are evaluated given a path λ, where $\lambda \models \varphi$ means that λ satisfies φ

Definition (Semantics of LTL_f)

$\lambda \models p$	iff p is true at state $\lambda[0]$ (that is, $\lambda[0] \in \mathcal{V}(p)$);
$\lambda \models \neg\varphi$	iff $\lambda \not\models \varphi$;
$\lambda \models \varphi \wedge \psi$	iff $\lambda \models \varphi$ and $\lambda \models \psi$;
$\lambda \models X\varphi$	iff $\lambda[1..\infty] \models \varphi$ and $\lambda \not\models \mathit{last}$;
$\lambda \models F\varphi$	iff $\lambda[i..\infty] \models \varphi$ for some $i \geq 0$;
$\lambda \models G\varphi$	iff $\lambda[i..\infty] \models \varphi$ for all $i \geq 0$;
$\lambda \models \varphi U \psi$	iff $\lambda[i..\infty] \models \psi$ for some $i \geq 0$, and $\lambda[j..\infty] \models \varphi$ for all $0 \leq j < i$.

where $\lambda[i]$ means the ith state of λ, $\lambda[i..\infty]$ means the suffix of λ starting at i, and *last* is a propositional symbol true exactly in the last state on a path.

LTL_f to Reward machines

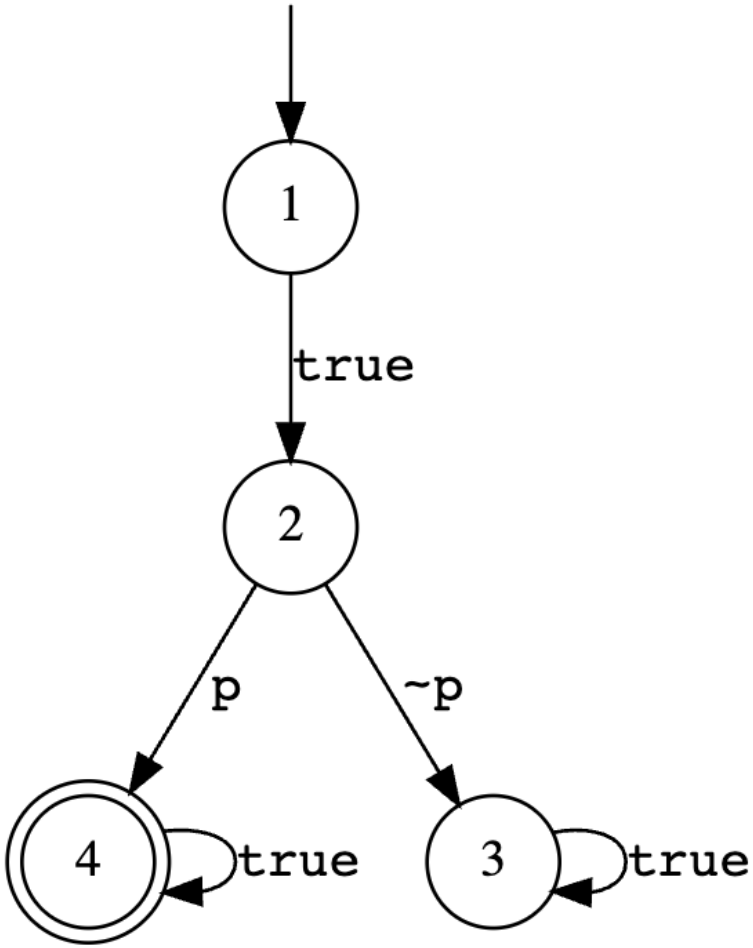
- a reward machine for an LTL_f formula corresponds closely to a DFA which accepts paths where the formula is true
- any LTL_f formula can be converted to a DFA
- usually an LTL_f formula is first converted to an NFA
- then NFA is determinised (see Appendix)
- downside: in the worst case, the reward machine is double exponential in the size of the LTL_f formula

Tools for converting LTL and LTL_f to automata

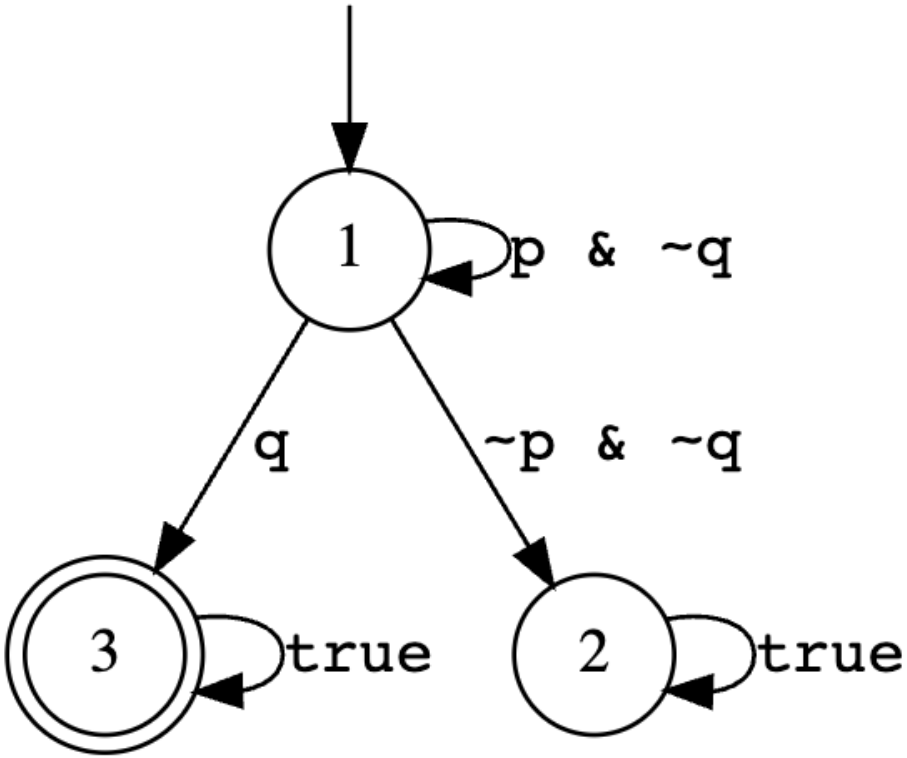
There are several tools for converting LTL specifications to automata:

- Spot: https://spot.lre.epita.fr/
- LTLf2DFA: online interface http://ltlf2dfa.diag.uniroma1.it/ , github https://github.com/whitemech/ltlf2dfa
- Lydia http://videos.icaps-conference.org/demos/demos/393.pdf
- Lisa 2 https://github.com/vardigroup/lisa
- Owl https://www.cs.cit.tum.de/tcs/tools/

Example: LTLf2DFA *Xp*



Example: LTLf2DFA $p\,U\,q$



DFA to RM

- to obtain a (task completion) reward machine given a DFA:
- add a reward function which
  - outputs a reward of 0 on transitions to states which are not terminal
  - outputs, e.g., a reward of 1, on transitions to a terminal state which is accepting
  - outputs, e.g., a reward of $-\infty$ on transitions to a terminal state which is not accepting

Other ways of generating reward machines

- generate an abstract planning domain (e.g., where ‘getMail’ and ‘getCoffee’ are actions), then generate a plan for achieving a task, and convert the plan into an RM (Illanes, et al. RLDM 2019)
- generate an abstract planning domain and all partially ordered plans for the task, then generate a “maximally permissive” RM from the set of partially ordered plans (Varricchione et al. ECAI 2024)
- generate an abstract state transition system and use model-checking to generate a strategy to achieve the task, and convert the strategy to a reward machine (we will see this in lecture 3) (Varricchione et al. JAIR 2026)

Learning with RMs

Learning with reward machines

- key idea is a Markov Decision process with a Reward Machine (MDPRM) — a combination of an MPD and an RM
- an MDPRM can be thought of as defining an MDP where:
  - the set of states is the cross product $S \times U$ of the environment states and the reward machine states
  - the rewards are generated by the RM
- downside: since the reward machine is double exponential in the size of the LTLf formula, the state space of the MDPRM is also double exponential in the size of the LTLf formula

MDPRM

Definition (MDPRM, Toro Icarte et al. (2022))

A Markov decision process with a Reward Machine (MDPRM) is tuple $T = \langle S, s_0, A, p, \gamma, \mathcal{PV}, L, U, u_I, F, \delta_R, r_R \rangle$ where:

- S, s_0, A, p and γ are defined as in an MDP;
- $L : S \times Act \times S \to 2^{\overline{\mathcal{PV}}}$;
- U, u_I, F, δ_R, r_R are defined as in a reward machine.

Learning in MDPRMs

- upside: since an MDPRM is just a particular kind of MDP, any RL algorithm can be used to learn a policy $\pi(a \mid s, u)$ in an MDPRM including tabular RL methods and deep RL methods
- agent cannot "look ahead" in the reward machine, but gains information that, e.g., a step in the task has been completed when it happens to pick up the mail
- if the RL algorithm is guaranteed to converge to optimal policies, then it will also find optimal policies for the MDPRM

Q-learning with an MDPRM

Initialise $\tilde{q}(s, u, a)$ arbitrarily for each $s \in S, u \in U, a \in A$

for 0 to num_episodes **do**

$s \leftarrow$ EnvInitialState(), $u \leftarrow u_0$

while s is not terminal and $u \notin F$ **do**

Sample action a using policy derived from $\tilde{q}$ (e.g., ϵ-greedy)

Take action a and observe the next state s'

$r \leftarrow r_R(u, L(s, a, s')); u' \leftarrow \delta_R(u, L(s, a, s'))$

if s' is terminal or $u' \in F$ **then**

$\tilde{q}(s, u, a) \overset{\alpha}{\leftarrow} r$

else

$\tilde{q}(s, u, a) \overset{\alpha}{\leftarrow} r + \gamma \max_{a' \in A} \tilde{q}(s', u', a')$

Update $s \leftarrow s'$ and $u \leftarrow u'$

Algorithms for learning with RMs

- while any standard RL algorithm can be used with an MDPRM, it is often possible to improve sample efficiency by using algorithms which exploit the structure of the RM
- we will focus on Counterfactual experiences for Reward Machines (CRM)
- Q-learning for Reward Machines (QRM) (Toro Icarte et al. 2018) is similar but learns a separate Q-value function $\tilde{q}_u$ for each RM state $u \in U$
- key idea is to augment the real experience of the agent with a set of synthetic experiences corresponding to the same (s, a, s') transition but experienced in a counterfactual state of the RM

Exploiting counterfactual experiences

- CRM can be easily incorporated into off-policy learning methods such as Q-learning
- for DQN and DDPG, the counterfactual experiences can simply be added to the experience replay buffer and then used for learning in the usual way

Q-learning with CRM

Initialise $\tilde{q}(s, u, a)$ arbitrarily for each $s \in S, u \in U, a \in A$

for 0 to num_episodes **do**

$s \leftarrow$ EnvInitialState(), $u \leftarrow u_0$

while s is not terminal and $u \notin F$ **do**

Sample action a using policy derived from $\tilde{q}$ (e.g., ϵ-greedy)

Take action a and observe the next state s'

$r \leftarrow r_R(u, L(s, a, s')); u' \leftarrow \delta_R(u, L(s, a, s'))$

$E_c \leftarrow \{\langle s, \bar{u}, a, r_R(u, L(s, a, s')), s', \delta_R(\bar{u}, L(s, a, s'))\rangle \mid \forall \bar{u} \in U\}$

for $\langle s, \bar{u}, a, \bar{r}, s', \bar{u}'\rangle \in E_c$ **do**

if s' is terminal or $u' \in F$ **then**

$\tilde{q}(s, \bar{u}, a) \xleftarrow{\alpha} r$

else

$\tilde{q}(s, \bar{u}, a) \xleftarrow{\alpha} r + \gamma \max_{a' \in A} \tilde{q}(s', \bar{u}', a')$

Update $s \leftarrow s'$ and $u \leftarrow u'$

Hierarchical reinforcement learning for RMs

- RMs can also be used with hierarchical reinforcement learning
- the agent will learn a set of options for the cross-product MDP
- each option is a policy which moves from one RM state to another RM state
- a higher-level policy is then learnt to select among those options in order to collect reward

Reward shaping with RMs

- (simple) RMs can also be used with potential-based reward shaping
- value iteration over the RM states is used to compute the potential function
- key idea is to is to approximate the expected discounted reward of being in any RM state by treating the RM itself as an MDP
- a potential is assigned to each RM state which is used to define a shaped reward function that encourages the agent to make progress in performing the task

Sample efficiency

Sample efficiency

- even when an RL agent can learn a policy, it may take a very large number of steps (samples) to do so
- rewards are often *sparse* (very sparse in the case of terminal rewards)
- e.g., learning to play Atari games from pixels requires tens of millions of steps, which is impractical in many applications such as robotics
- in addition to ensuring that the policy learned satisfies the LTL specification, reward machines can also be more sample efficient than classical RL approaches

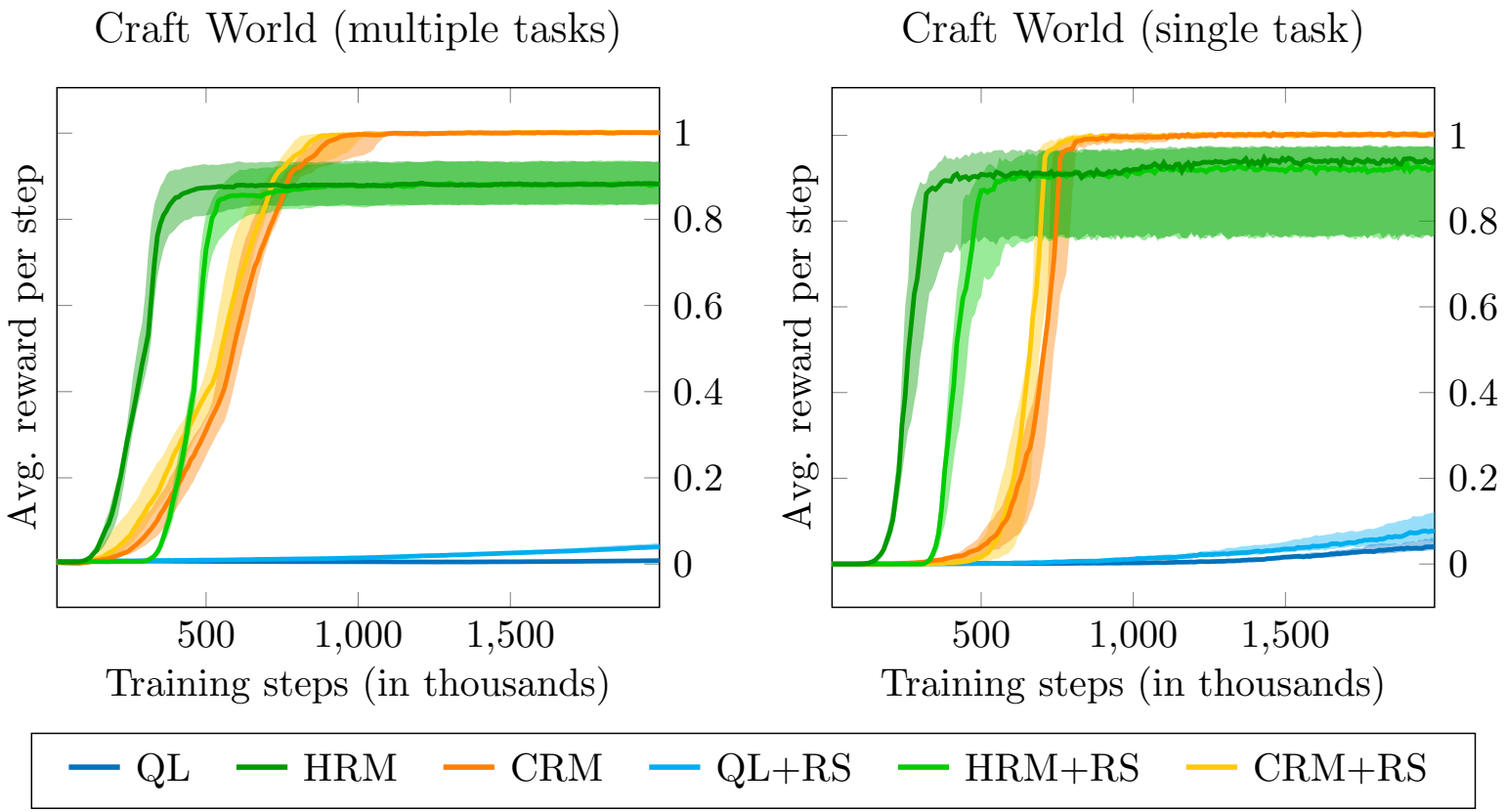
Example: CraftWorld

- CraftWorld is a simplified Minecraft-like grid (discrete) environment
- agent was evaluated on 10 tasks of increasing difficulty in single and multi-task settings
- e.g., the ‘get gem’ task involves ‘get wood, use workbench, get iron, use toolshed, use axe’; some of the steps are ordered, some are not
- in the single-task setting, the agent learns a different policy for each task
- in the multi-task setting, the agent learns a single policy for all 10 tasks, iterating through the tasks at the completion of each episode

Example: CraftWorld algorithms

- Q-learning on the cross-product MDP (QL)
- Q-learning on the cross-product MDP with reward shaping (QL+RS)
- Q-learning with counterfactual experiences (CRM)
- Q-learning with CRM and reward shaping (CRM+RS)
- hierarchical RM (HRM)
- hierarchical RM with reward shaping (HRM+RS)

Example: CraftWorld results



From Toro Icarte et al. Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning JAIR Vol. 73, 2022, 173–208

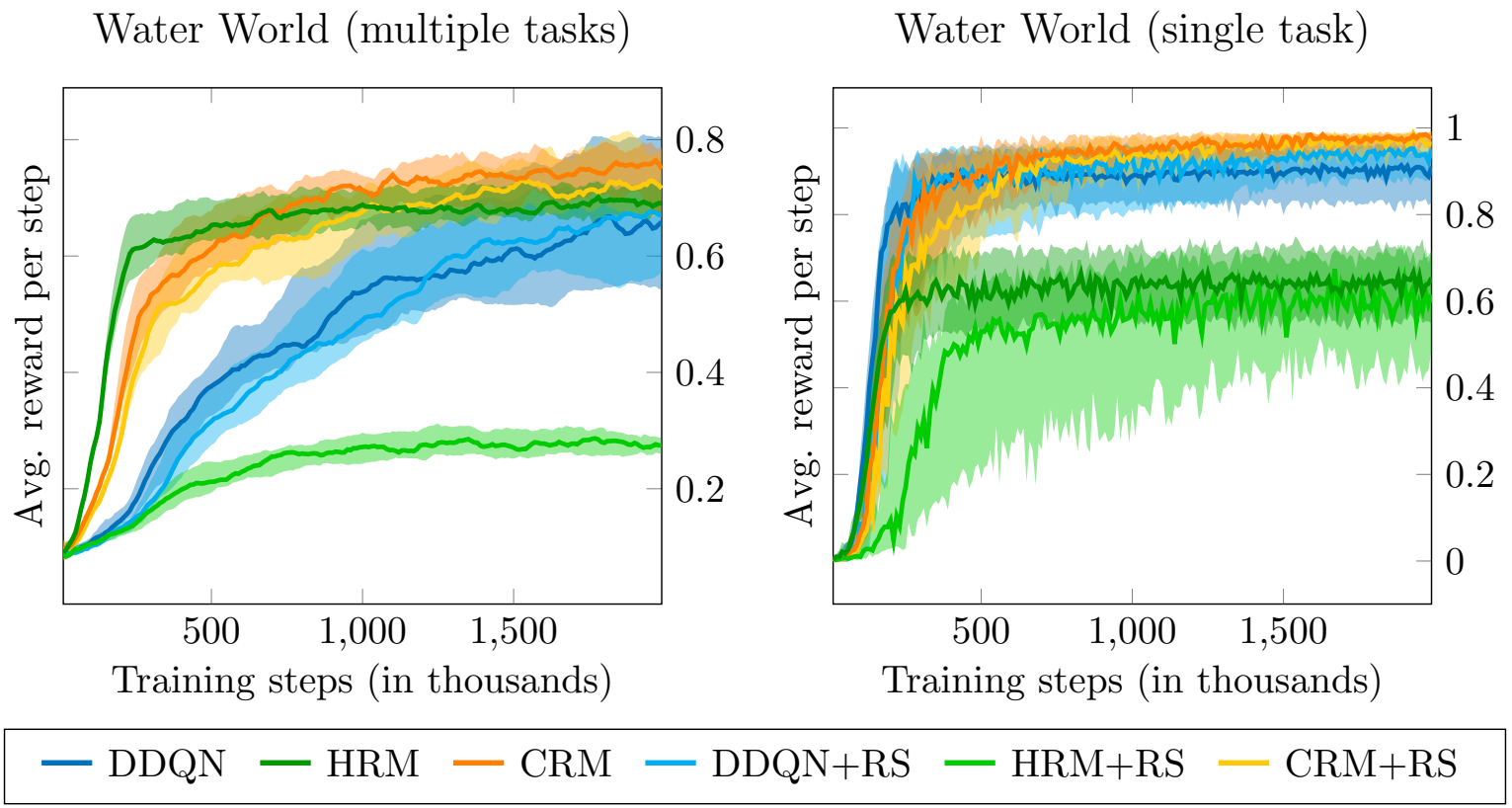
Example: WaterWorld

- WaterWorld is a continuous environment consisting of a two dimensional box containing moving balls of different colours
- each ball moves in one direction at a constant speed and bounces when it collides with an edge of the box
- the agent can increase its velocity in any of the four cardinal directions
- agent was evaluated on 10 tasks of increasing difficulty in single and multi-task settings
- tasks require touching balls of different colours; some in sequence and some in any order (and not touching balls of other colours)

Example: WaterWorld algorithms

- Double DQN on the cross-product MDP (DDQN)
- Double DQN on the cross-product MDP with reward shaping (DDQN+RS)
- DDQN with counterfactual experiences (CRM)
- DDQN with counterfactual experiences and reward shaping (CRM+RS)
- hierarchical RM (HRM)
- hierarchical RM with reward shaping (HRM+RS)

Example: WaterWorld results



From Toro Icarte et al. Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning JAIR Vol. 73, 2022, 173–208

Sample efficiency: summary

- standard RL algorithms are often unable to learn simple tasks if the rewards are non-Markovian
- for relatively simple tasks with non-Markovian rewards, standard RL algorithms on the cross product MDP are often sufficient
- for complex tasks with many sub-tasks, counterfactual experiences (CRM) can significantly reduce the number of samples required to learn the task
- reward shaping can be useful with simple reward machines (where the reward depends only on the state of the reward machine)

Beyond regular specifications

More expressive RMs

- RMs are essentially DFAs, and so can be used to encode reward functions for computational problems which can be expressed in regular languages
- recently, there has been work on extending RMs to encode problems expressible in richer languages
- Pushdown Reward Machines extend the definition of RMs to add a stack and can can encode problems expressible in context free languages
- Counting Reward Automata (CRAs) extend RMs with one or more counters: a CRA with two counters can encode any problem representable by a Turing Machine
- Recursive MDPs (RMDPs) generalise MDPs in that they consist of a set of MDPs where each MDP can “call” other MDPs; by keeping a stack of calls, RMDPs can encode tasks representable as DCFLs

References

- Leike et al. AI Safety Gridworlds. arXiv:1711.09883
- Camacho et al. LTL and Beyond: Formal Languages for Reward Function Specification in Reinforcement Learning. IJCAI 2019: 6065-6073
- Toro Icarte et al. Reward Machines: Exploiting Reward Function Structure in Reinforcement Learning JAIR Vol. 73, 2022, 173–208
- De Giacomo & Vardi. Synthesis for LDL and LTL on Finite Traces. IJCAI 2015: 1558–1564
- Toro Icarte et al. Using Reward Machines for High-Level Task Specification and Decomposition in Reinforcement Learning. ICML 2019
- Illanes, et al. Symbolic planning and model-free reinforcement learning: Training taskable agents. In Proceedings of 4th Multidisciplinary Conference on Reinforcement Learning and Decision Making (RLDM), pages 191–195, 2019
- Varricchione et al. Maximally Permissive Reward Machines. ECAI 2024: 1181–1188
- Varricchione et al. Synthesising Reward Machines for Cooperative Multi-Agent Reinforcement Learning JAIR vol. 85, 2026

Appendix

LTL$_f$ to NFA: auxiliary function δ

δ is a function that takes

- an LTL$_f$ formula ψ in negation normal form, and
- a propositional assignment $\Pi \in \mathcal{PV}$
- returns what should hold of ψ's subformulas (and the formulas in its Fischer-Ladner closure) in the next state

LTL$_f$ to NFA: auxiliary function δ

- $\delta(\text{“}\varphi\text{”}, \Pi) = \mathit{true}$ if $\Pi \models \varphi$ and false otherwise (for propositional φ)
- $\delta(\text{“}\varphi_1 \wedge \varphi_2\text{”}, \Pi) = \delta(\text{“}\varphi_1\text{”}, \Pi) \wedge \delta(\text{“}\varphi_2\text{”}, \Pi)$
- $\delta(\text{“}\varphi_1 \vee \varphi_2\text{”}, \Pi) = \delta(\text{“}\varphi_1\text{”}, \Pi) \vee \delta(\text{“}\varphi_2\text{”}, \Pi)$
- $\delta(\text{“}X\varphi\text{”}, \Pi) = \text{“}\varphi\text{”}$ if $\mathit{last} \notin \Pi$, otherwise false
- $\delta(\text{“}F\varphi\text{”}, \Pi) = \delta(\text{“}\varphi\text{”}, \Pi) \vee \delta(\text{“}XF\varphi\text{”}, \Pi)$
- $\delta(\text{“}G\varphi\text{”}, \Pi) = \delta(\text{“}\varphi\text{”}, \Pi) \wedge \delta(\text{“}\overline{X}G\varphi\text{”}, \Pi)$
- $\delta(\text{“}\varphi_1 \, U \, \varphi_2\text{”}, \Pi) = \delta(\text{“}\varphi_2\text{”}, \Pi) \vee (\delta(\text{“}\varphi_1\text{”}, \Pi) \wedge \delta(\text{“}X(\varphi_1 \, U \, \varphi_2)\text{”}, \Pi))$

Algorithm to convert LTL_f to NFA

input: LTL_f formula φ

output: NFA $\mathcal{A}_\varphi = (Q, 2^{\mathcal{PV}}, q_0, \Delta, q_f)$

$q_0 \leftarrow \{\text{“}\varphi\text{”}\}$

$q_f \leftarrow \emptyset$

$Q \leftarrow \{q_0, q_f\}$

$\Delta \leftarrow \emptyset$

while Q or Δ change **do**

  if $q \in Q$ and $q' \models \bigwedge_{\text{“}\psi\text{”} \in q} \delta(\text{“}\psi\text{”}, \Pi)$ **then**

    $Q \leftarrow Q \cup \{q'\}$

    $\Delta \leftarrow \Delta \cup \{(q, \Pi, q')\}$

De Giacomo & Vardi. Synthesis for LDL and LTL on Finite Traces. IJCAI 2015: 1558–1564

From NFA to DFA

- given an NFA $\mathcal{A}_\varphi = (Q, q_0, \Sigma, \Delta, F)$ we construct a DFA $\mathcal{A}'_\varphi = (2^Q, \{q_0\}, \Sigma, \delta, F')$ that accepts the same language as $\mathcal{A}_\varphi$
- the transition function δ of the DFA maps a state $s \in 2^Q$ and an input symbol σ to the set of all states that can be reached by an σ-transition from a state in s

$$\delta(s, \sigma) = \{q' \mid \exists q\,(q \in s \wedge \Delta(q, \sigma, q'))\}$$

- a state s of the DFA is an accepting state if and only if at least one member of s is an accepting state of the NFA

From DFAs to Reward Machines

- suppose we are given reward specification $(r_1 : \varphi_1), \ldots, (r_n : \varphi_n)$ (where r_i are rewards and φ_i are LTL$_f$ formulas)
- we generate n DFAs $\mathcal{A}^{(i)} = (Q(i), \Sigma, q_0^{(i)}, \delta^{(i)}, F^{(i)})$ which are transformations of each φ_i,
- the reward machine $M_R = (Q, q_0, \Sigma, R, \delta, \rho)$ is given by:
  - $Q = Q^{(1)} \times \cdots \times Q^{(n)}$
  - $q_0 = (q_0^{(1)}, \ldots, q_0^{(n)})$
  - $\delta((q^{(1)}, \ldots, q^{(n)}), \sigma) = (\delta^{(1)}(q^{(1)}, \sigma), \ldots, \delta^{(n)}(q^{(n)}, \sigma))$
  - $\rho((q^{(1)}, \ldots, q^{(n)}), \sigma) = \Sigma_{k=1}^{n} = \rho^{(k)}(q^{(k)}, \sigma)$ where $\rho^{(k)}(q^{(k)}, \sigma) = r_k$ if $\delta^{(k)}(q^{(k)}, \sigma) \in F^{(k)}$ and 0 otherwise (rewards are for reaching accepting states)