CS391L Machine Learning HW5: Reinforcement Learning

Zeyuan Hu, iamzeyuanhu@utexas.edu

EID:zh4378 Fall 2017

1 Introduction

In this task, we use the modular reinforcement learning (MRL) to explore a navigation task in a 2D grid world of size 10×10, as shown in Figure 1. Our agent starts randomly on the left side of the grid (denoted as `start` in the figure) and its action space A = {*up*, *down*, *forward*}. There are *objects* (orange square in the figure) that can be picked up, which leads to positive reward. In addition, there are *litters* (red triangle) that cause negative reward when step onto it. Furthermore, the green cross represents *exits*, which give a large positive reward. The goal of the agent is to walk from the left side of the grid to the right side while collecting as many objects as possible and trying his best to avoid litters. MRL divides the task into subtasks (modules) and each module will learn its own policy using reinforcement learning. Then, module policies are combined to construct a global policy. Concretely, we model each module as a Markov decision process (MDP), which is defined as a tuple $< S, A, T, R, \gamma >$ [1]. Suppose we have n modules and n corresponding MDPs $< S_1, A, T_1, R_1, \gamma_1 >, \ldots, < S_n, A, T_n, R_n, \gamma >$. Each module is trained with reinforcement learning and learned module policies $\pi_1, \ldots, \pi_n$. Then, our job is to form a global policy π that maximize global reward based on module policies. In other words, we need to select a global action a^* for global state s based on the optimal actions $a_1, \ldots, a_n$ proposed by each module based on their module states $s_1, \ldots, s_n$.

2 Method

2.1 MRL

MRL can be break down into three parts: decomposition, module training, and navigation and real-time global policy construction. For decomposition part, litters, objects, and exits form modules individually. The goal of agent for each module is defined as: avoiding

litters, picking up objects, and walking towards exits. Each module is modelled as a MDP independently. The state is defined as the (x, y) position in the grid and the action space A is the same across all three modules. For module training part, we train each module with Sarsa, which is defined as following:

Algorithm 1 Sarsa

Let **s,a** be the original state and action, **r** is the reward observed in the following state, and $\boldsymbol{s'}, \boldsymbol{a'}$ are the new state-action pair.
Initialize $\boldsymbol{Q(s, a)}$ arbitrarily
for each episode **do**
  Initialize $\boldsymbol{s}$
  Choose $\boldsymbol{a}$ from $\boldsymbol{s}$ using policy derived from $\boldsymbol{Q}$ (i.e., ϵ-greedy)
  for each step of episode **do**
    Take action $\boldsymbol{a}$, observe $\boldsymbol{r}, \boldsymbol{s'}$
    Choose $\boldsymbol{a'}$ from $\boldsymbol{s'}$ using policy derived from $\boldsymbol{Q}$ (i.e., ϵ-greedy)
    $\boldsymbol{Q(s, a)} \leftarrow \boldsymbol{Q(s, a)} + \mu(\boldsymbol{r} + \gamma \boldsymbol{Q(s, a)} - \boldsymbol{Q(s, a)})$
    $\boldsymbol{s} \leftarrow \boldsymbol{s'}; \boldsymbol{a} \leftarrow \boldsymbol{a'}$
  end for
end for

Intuitively, the agent will explore from state to state until it reaches either the step limit, which we specify in the inner loop of the aglorithm, or the final state (i.e., exit). We call each exploration an episode. The major difference between Sarsa and Q-Learning is

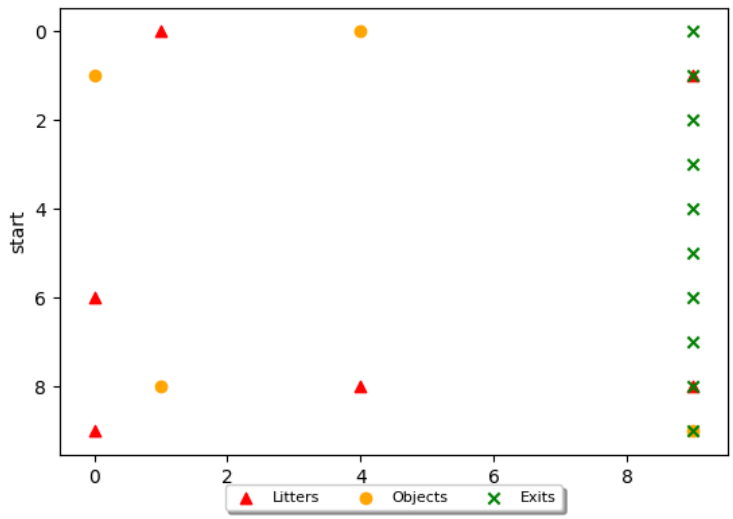

Figure 1: Environment

that the maximum reward for the next state is not necessarily used for updating the Q values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. We keep the trained Q table for each module.

For the navigation and real-time global policy construction part, at each global state s, our agent looks up the Q values in each module for the given state s and then we select a global action a^* based on the global policy construction algorithm and execute the action to move to a new global state s'. We implement three global policy construction algorithms, which will be described in details in the following section.

2.2 Global policy construction

The three algorithms are *Module aggregation algorithm*, *Module selection algorithm*, and *Module voting algorithm* [2].

- **Module aggregation algorithm** selects the action that maximizes the Q value of all modules combined for a given state. Concretelym we choose action $a^* = \text{argmax}_a Q(s, a)$, where $Q(s, a) = \sum_i Q_i(s_i, a)$ with i representing module i.

- **Module selection algorithm** selects the action that has the highest weight across the all modules. Let $W_i(s_i)$ denote the weight of module i at state s_i. We choose $W_i(s_i) = \sigma(Q_i(s_i, a))$, where $\sigma(Q_i(s_i, a))$ is the standard deviation of Q values acorss actions. The intuition behind the scene is that the variance of Q values across all actions indicates how indifferent the module is about its action selection. Large variance indicates that choosing different ations lead to very different expected reward.

- **Module voting algorithm** selects the action that has highest votes. Let $K(s_i, a)$ be vote count for global action a, then $K(s_i, a) = \sum_i W_i(s_i)$ for all module i whose optimal action $a_i = a$. In other words, each module i put $W_i(s_i)$ votes on its optimal action a_i. Global action is selected as the action with highest number of votes $a^* = \text{argmax}_a K(s_i, a)$.

2.3 Implementation

We use `REWARD_EMPTY`, `REWARD_OBJ`, `REWARD_LITTER`, and `REWARD_EXIT` to denote the rewards for empty grid, objects, litters, and exits respectively, which are initialized to 0, 5, -4,

and 100. Functions `avoidLittersReward`, `pickingObjsReward`, and `exitingMapReward` are used to generate the reward matrix for avoiding litters, picking up objects, and walking towards exit modules individually. Each reward matrix is generated independently and thus litters, objects, and exits can appear simultaneously in the same cell. In addition to reward matrix, we also generate the cell state matrix (i.e., `R_state`), which is to keep track of the states of the cells of the grid. There are four possible states: `L`, `O`, `X`, and `E`, which represents that the cell has "Litters", "Objects", "Exits", or nothing. The purpose for the state matrix is that the litters or the objects can be stepped onto once and we need to change the reward once the agent passes those cells.

Sarsa algorithm is implemented as `sarsa` that takes in reward matrix `R`, state matrix `R_state`, a set of hyperparameters including μ, γ, ϵ, numEpisode, and numSteps. The implementation follows closely to the algorithm stated above. The output of the algorithm is the trained Q table. `drawGrid` contains all three global policy construction algorithms, the action execution based on the global policy, and the plotting of the grid and the agent exploration route. `drawGrid` takes in the Q table(s), reward matrices, maximum number of steps that agent can move, mode, policy, and a boolean `checkRepeatActions`. The input Q table and reward matrices have to be packed as a list of Q table matrix and reward matrix resepectively. This allows us to use one single `drawGrid` function to handle all possible cases: avoiding litters module only, picking up objects only, walking towards the exit only, and all three modules combined. Which module we are working with is specified with `mode`. Accepted values ranged from 0 to 3 inclusive to indicate avoiding litters module, picking up objects, walking towards the exit, and all three modules combined. `policy` variable specifies which global policy construction we want to use and `checkRepeatActions` specifies whether we want to detect the repeative actions, which is shown in figure 2. Once the agent moves to the cell $[0, 1]$, the optimal global action keeps instructing the agent to go up, which is not possible. Thus the agent stays at the original place and stay here forever (the path of agent is shown in blue). To handle case like this, we invent a protocol that randomly select the action to execute when we detect there have been repeat actions in the past using `checkCycle` function. The impact of `checkRepeatAction` is shown in figure 3.

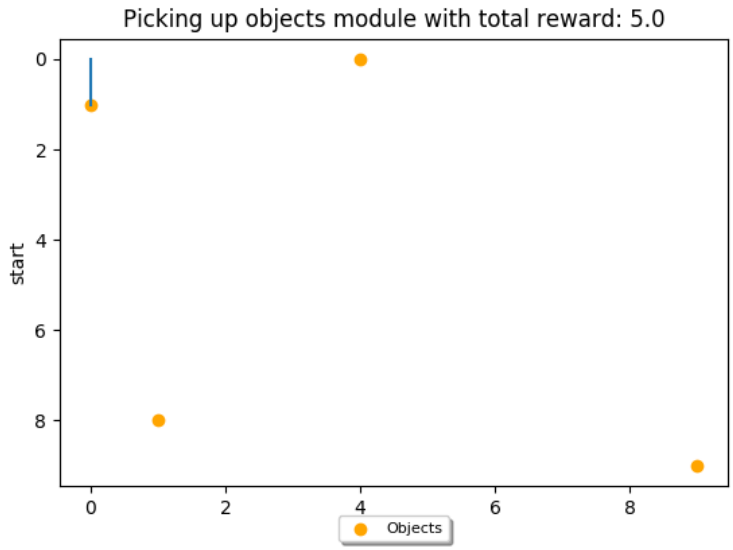

Figure 2: checkRpeatActions=False for picking up objects module

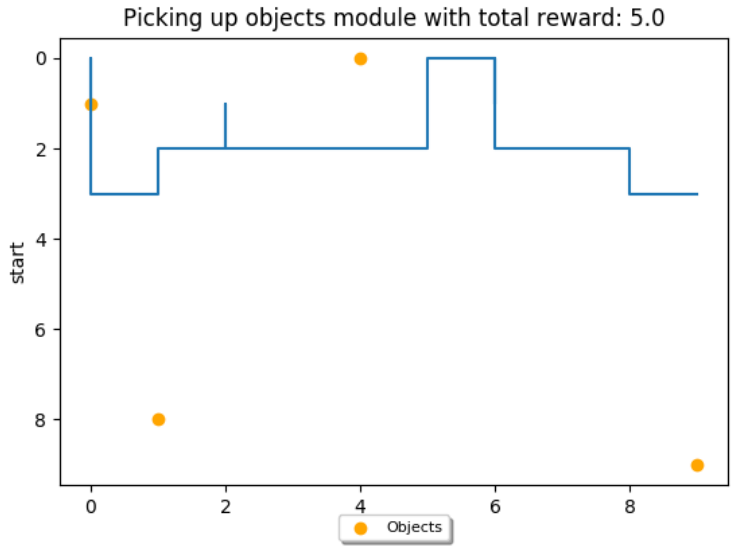

Figure 3: checkRpeatActions=True for picking up objects module

Table 1: Hyperparameter configuration for the navigation task

Description	Values
grid width	10
grid length	10
number of litters	10
number of objects	4
μ	0.7
ϵ	0.5
numEpisode (module training)	200
numSteps (module training)	100
maximum number of steps that agent can move	200
γ for avoiding litters	0.4
γ for picking up objects	0.7
γ for walking towards exit	1

Table 2: Hyperparameter configuration for globlal policy construction algorithms

Description	Values
grid width	100
grid length	100
number of litters	30
number of objects	40
maximum number of steps that agent can move	1000

3 Results

Without specific note, we use the hyperparameters shown in the table 1. The heuristic for setting up the discount factor γ for each module is that we want to prioritize walking towards exits goal. In general γ has a range of 0 to 1. If γ is closer to zero, the agent will tend to consider only immediate rewards. If γ is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

The performance of each individual module given `checkRepeatActions=True` is shown 4, 5, and 6. The performance for picking up objects module is shown in Figure 2 above. Additional plots of all three modules combined can be seen in the appendices.

We also experiment with the efficiency of the global policy construction algorithm by measuring how many steps that the agent has to perform in order to reach the exits and the total reward collected in the end. For this experiment, we change the following hyperparameters shown in table 2. The rest of hyperparameters are kept untouched.

The performance of using *module voting algorithm* is shown in Figure 7 and the performance of using *module aggregation algorithm* is shown in Figure 8.

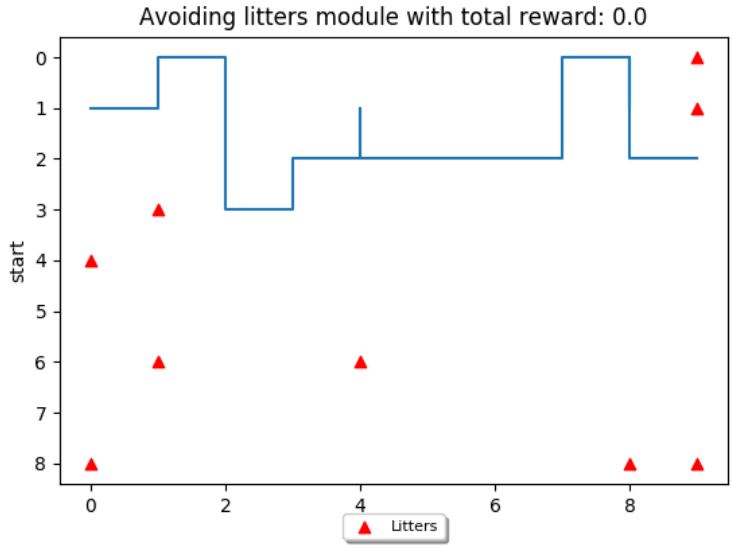

Figure 4: Performance of avoiding litters module

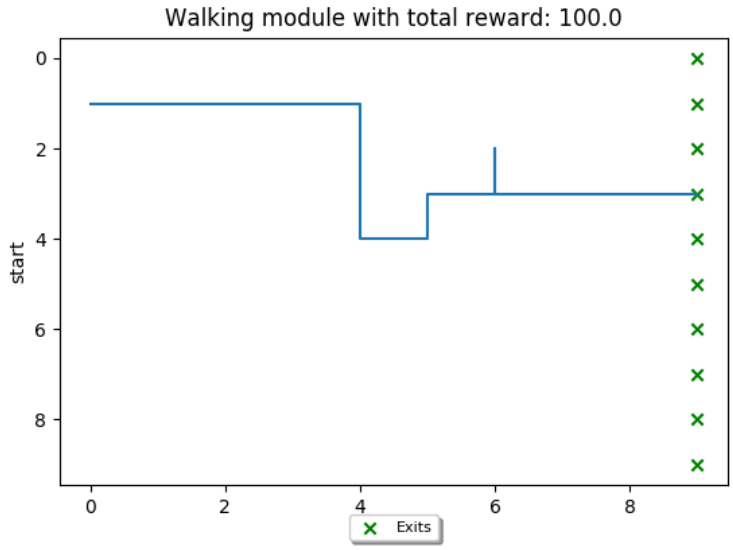

Figure 5: Performance of walking towards exits module

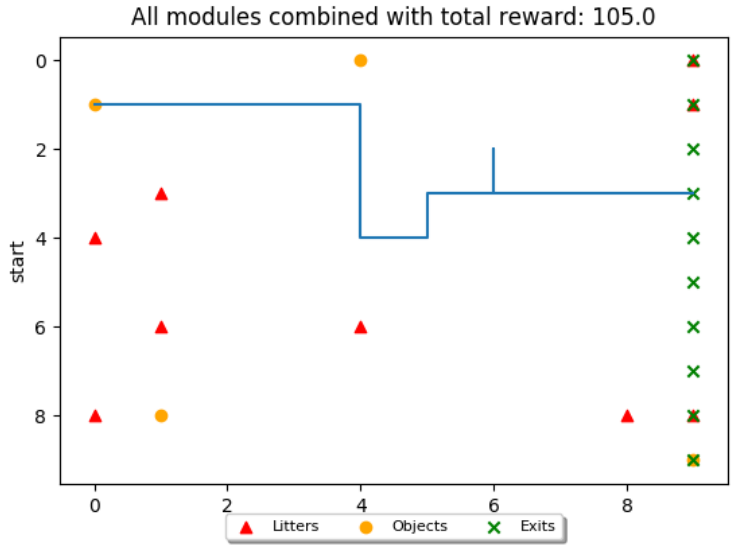

Figure 6: Performance of all three modules combined

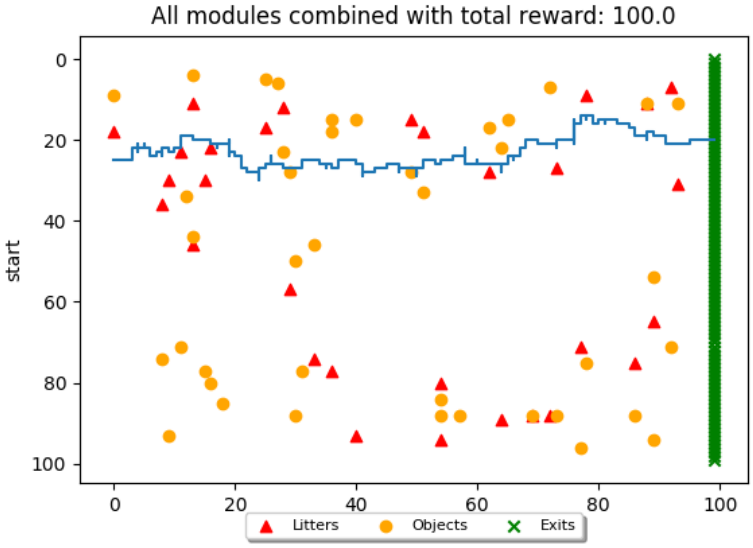

Figure 7: Performance of all three modules combined with module voting algorithm

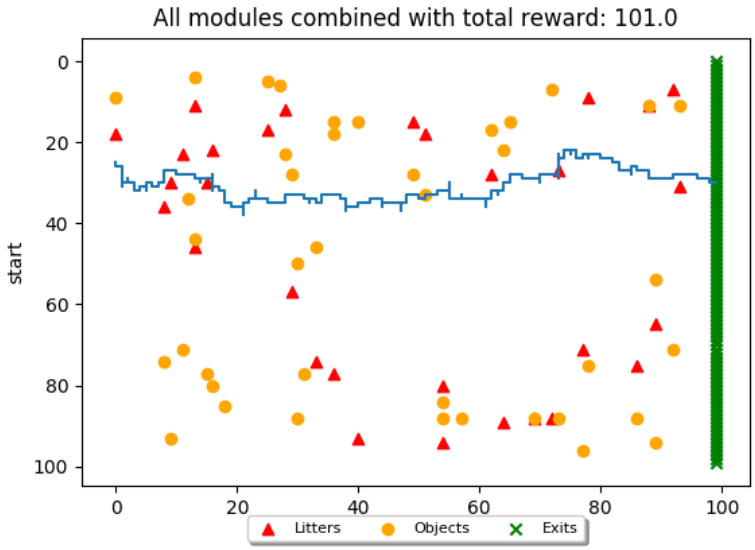

Figure 8: Performance of all three modules combined with module aggregation algorithm

Table 3: Performance of globlal policy construction algorithms

Algorithm	Number of Steps	Total Rewards
module aggregation algorithm	264	**101**
module selection algorithm	264	100
module voting algorithm	264	100

Table 2 summarizes the performance of three global policy construction algorithms measured by the number of steps used and the total reward collected. As shown in the table, all three algorithms do not make fundamentally difference. However, module aggregation algorithm works better than the other two algorithms in terms of total rewards collected. In addition, the peformance of algorithms stay the same no matter whether we turn on `checkRepeatActions` or not during the training phase of each module.

We also experiment with the impact of the discount factor γ onto the agent's performance. We vary the discount factor for walking towards the exits module and measure the number of steps and total rewards collected. The hyperparameters configuration is the same with the previous experiment except that now we vary γ from 0 to 1 with 0.1 intervals (i.e., $0, 0.1, 0.2, \ldots, 1$). We use *module voting algorithm* and turn off `checkRepeatActions` during the training phase for this experiment. The performance of the agent is summarized in table 3.

Table 4: Performance of globlal policy construction algorithms

γ	Number of Steps	Total Rewards
0	260	**105**
0.1	260	**105**
0.2	260	**105**
0.3	264	96
0.4	264	96
0.5	264	96
0.6	264	96
0.7	264	96
0.8	265	**105**
0.9	**262**	100
1.0	264	100

The importance of `checkRepeatActions` protocol becomes apparent when we deal with large grid in this case. Figure 9 shows the result of the agent exploration after 1000 steps. As shown in the figure, the agent cannot fully explore the space and fail to reach the exits given the step constraint.

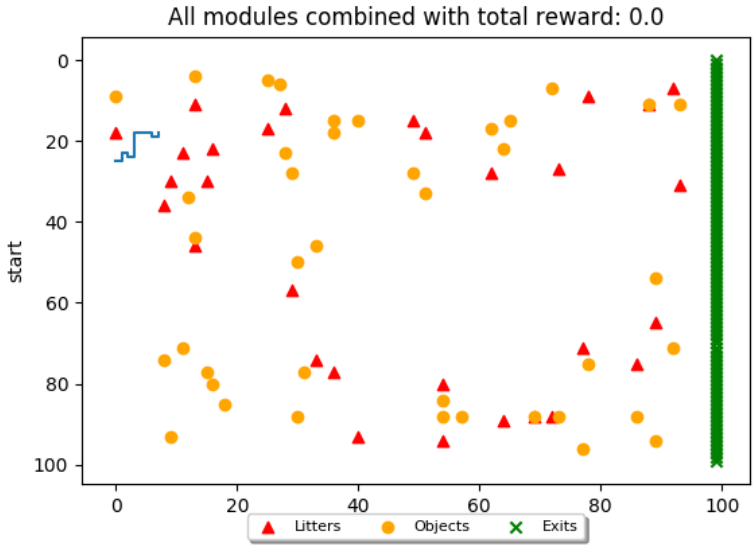

Figure 9: checkRpeatActions=False for three modules combined with 1000 maximum number of steps

4 Conclusion

In this task, we implement MRL and apply it towards the navigation task. We implement three global policy construction algorithms and show the effectiveness of the algorithms for the given task. In addition, we explore the impact of the discount factors on the agent's performance and demonstrate the importance of `checkRepeatActions` protocol on achieving desirable task outcome.

References

[1] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed., 1998.

[2] R. Zhang, Z. Song, and D. H. Ballard, "Global policy construction in modular reinforcement learning," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, pp. 4226–4227, AAAI Press, 2015.

Appendices

A How to run the code

To run my code, unzip the `hw5.zip` and get `rl.py`. Then, you can run the code with the following commands:

- `python rl.py L` Run the MRL on avoiding litters module only
- `python rl.py O` Run the MRL on picking up objects module only
- `python rl.py X` Run the MRL on walking towards exit module only
- `python rl.py A` Run the MRL on all three modules combined

My code is heavily commented and please take a look if there are any type of questions.

B Additional plots of agent explorations

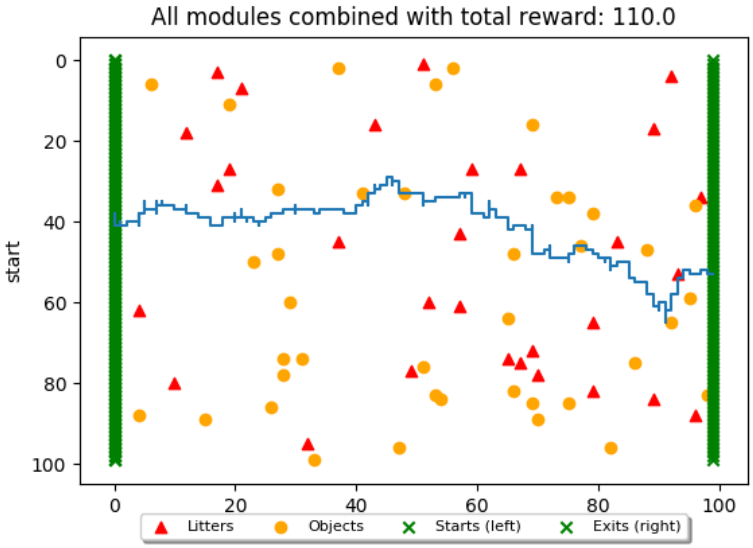

Figure 10: Performance of all three modules combined

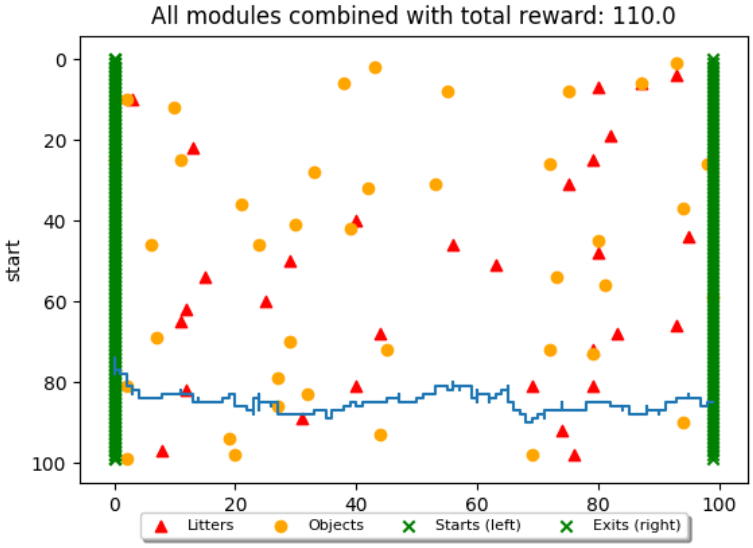

Figure 11: Performance of all three modules combined

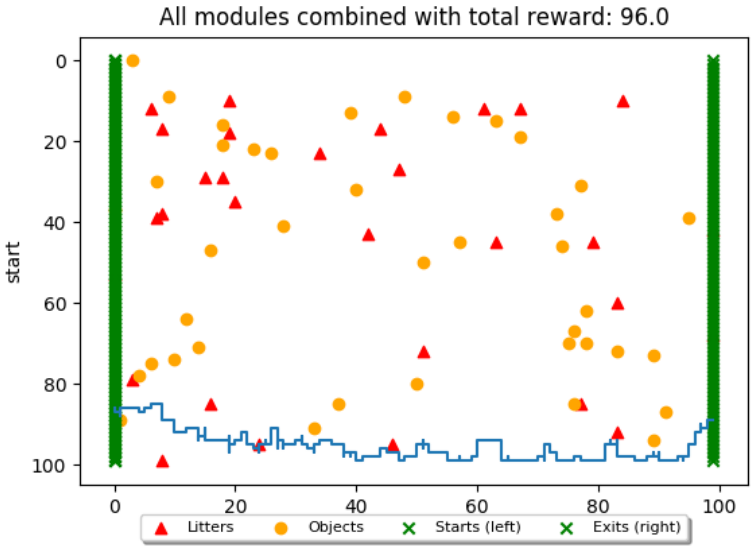

Figure 12: Performance of all three modules combined