ISEMO-SW

1 Description

The execution starts from *main.py*. This file contains three functions: **makeWorlds**, **mainISEMO**, and **mainCoHRL**. **makeWorlds()** is called once to prepare the environment objects and save them into PL files (.pl). The number of objects is equal to the number of training runs or trials (given by the argument *args.nruns*). This is the number of times the training is performed by resetting the agents (i.e. their function parameters) in each run.

makeWorlds() creates objects of the **World** class. This class is defined in the file *World.py*. A World() object contains the true grid-map of the *indoor search & rescue* environment consisting of victims, debris, path blockage, walls, and obstacles. The location of the victims are randomly set during the World() object creation (in main.py - makeWorlds()). The World() object also contains data structures representing the health values of the victims, the lists of victim locations and status, and the lists of other elements. The numerical values used to represent these various elements of the search & resource world are defined in the **Cell** class in *Utils.py*.

When the World() objects are ready, other functions in *main.py* can be called. The **mainCoHRL()** function runs the baseline CoHRL method used for comparison against our proposed method ISEMO. The **mainISEMO()** functions runs ISEMO. The execution can be in training mode or testing mode. The instructions to set the mode are provided under section 2. Following discussion is in the context of training mode, but the structure mostly remains same in testing mode as well.

The argument of mainISEMO(), named *noISER*, determines which type of agents are blocked from receiving the ISER rewards. If *noISER* is None, all types of agents can observe ISER. mainISEMO() further calls the **runISEMO()** function which takes the execution into ***ISEMO.py***. In ISEMO.py, the **OptionControl** class contains following members: (i) the *estimator* object which provides methods to predict the Q-values of the options/subtasks and to update the Q-function (linear function approximators) parameters, (ii) the *policy* object which provides method to sample an option/subtask given a state input, and the *critic* object which performs the Temporal Different Q-function updates. These OptionControl members are initialized in the **runISEMO()** function in ISEMO.py. runISEMO() also calls the **registerAgents()** function defined in *Agents.py*. In this function, the total number of agents is defined as *numagents*. Then, *numagents* number of **Agent** objects are created. The **Agent()** class is also defined in Agents.py. An Agent() object takes two inputs: an **Agent_Environment** object and the agent ID (*agid*). The *Agent_Environment* is an extension of the *World* object discussed above. While the *World* object is a single instance shared by all *Agent* instances, *Agent_Environment* instances are unique to each *Agent* instance. An *Agent_Environment* instance contains the **spatial representation** of the *World* attributes for the corresponding agent using a list of feature values created by the **situupdate()** method in *Agent_Environment* class. This class also contains the **update()** method which calls **situupdate()** and also checks the changes in the *option preconditions* to generate ISER for the eligible agents.

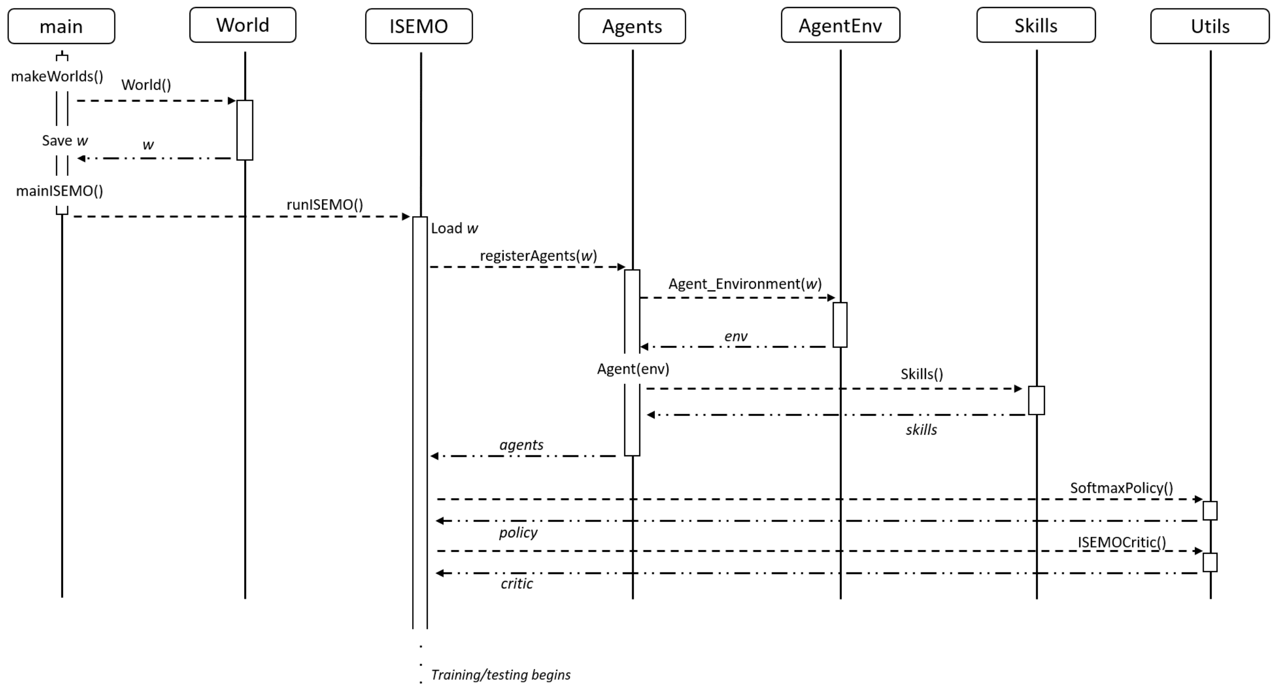

Figure 1: The timing diagram during initiation, before training episodes (or testing run) begin.

Once the **Agent** objects/instances are created, each containing its own *Agent_Environment* instance, each Agent is assigned the eligible *options/subtasks* it can perform by initializing the **oset** list in an *Agent* object. An *Agent* object provides a method called *stepLevel1()* which calls lower level primitive-step functions to execute a given *option.* These lower level functions are defined under the **Skills** class in *Skills.py.* A *Skill* instance is created along with an *Agent* instance. The call diagram for this initiation stage is shown in Figure 1.

After the creation of the *Agent* objects/instances, the training/testing proceeds within **runISEMO()**. At the beginning of an episode, an *option* for each agent is sampled using the *sample()* function of the *policy* object contained within *OptionControl* class. A *policy* object is an instance of the **SoftmaxPolicy** class defined in *Utils.py.* After initial *options* are sampled, the training/testing step loop runs until the termination of an episode. At each step, the health values of the victims is decayed by calling **decayHealth()** function of the *World* object. Then, each agent takes one step according to its own sampled *option* using the **stepLevel1(*option*)** function of the corresponding *Agent* object instance.

The **stepLevel1()** function of an *Agent* object (in *Agents.py*) calls the lower level functions defined in the *Skills* class in *Skills.py* in repsonse to the input *option.* These functions are described below:

- **scan()** function performs scanning of local area around an agent's location. This function also calls the **on_scan()** function of the *World* class which modifies the world attributes in response.

- **save()** function calls the **on_aid()** function of the *World* class which modifies the status of *critical* victims in the proximity of the agent to *stable.*

- **fetch()** function takes either *Cell.station* or *Cell.victim_stable* as argument. If it is the former, then the count of medicine (*med* variable in the *Agent_Environment* class) is increased. Otherwise, this function also calls the **on_carry()** function of the *World* class which modifies status of the victims in the proximity of the agent and adds them to the carried victims list.

- **relocate()** function performs relocation of carried victims if the base station is in proximity of the agent. This function also calls the **on_relocation()** function of the *World* class which modifies the World attributes in response.

- **clear_debris()** function calls the **on_clear_debris()** function of the *World* class which modifies the World attributes to reflect the removal a debris element.

- **clear_blockage()** function calls the **on_clear_blockage()** function of the *World* class which modifies the World attributes to reflect the removal a blockage element.

- **move_to()** function computes A^* path to a given target and moves the agent.

Following the stepLevel1(*option*) calls for all agent, the environment state of each agent is updated. This is done by calling the **update()** function of the *Agent_Environment* class (in AgentEnv.py). The *update()* function calls another function within the *Agent_Environment* class named **situupdate** which computes the state features using the latest attributes of the *World* object. The *update()* function also checks if any precondition factors are satisfied and prepares a list of agents to which ISER should be given. The function returns both the latest environment state (*next_state*) and the ISER list *ISER_to*. *runISEMO()* then calls the **reward_blender()** function defined in *Utils.py*. The **reward_blender()** function gets the global shared rewards from the **shared_global_reward()** function of the *World* defined in the *World* class. The global rewards (task rewards) are blended with ISER by weighted summation and the final reward is return by the *reward_blender()* to *runISEMO()*.

If the execution is in training mode, the parameterized Q-function models defined under the *Estimator* class in *Utils.py* are updated by calling the **update()** function of the **ISEMOCritic** class in *Utils.py*. The *termination* functions (*betamodels* in the *Estimator* class) are also updated by calling the **update()** function of the **TerminationGradient** class in *Utils.py*. The termination of an option/subtask is checked by calling the *option* termination probability sampling function **sample()** of the **SigmoidTermination** class in *Utils.py*. If an *option* terminates, a new *option* is sampled by calling the *sample()* function of the **SoftmaxPolicy** class object.

At the end of each step, the **finish()** function of the *World* class is called. This function checks if an episode should terminate. If the function returns *true*, the episode loop is terminated and a new episode begins. Before termination of the episode, **the trained models are saved** *pickle (.pkl)* format files by calling the **save_models()** function of the *Estimator* class in Utils.py. The models are saved only in the training mode. The timing diagram during training iterations in depicted in Figure 2.

2 Instructions to run training/testing

2.1 Dependencies

- Python $\geq$ 3.5.0

- scikit-learn==0.19.1

- scipy==1.0.0

- opencv-python==4.1.1.26

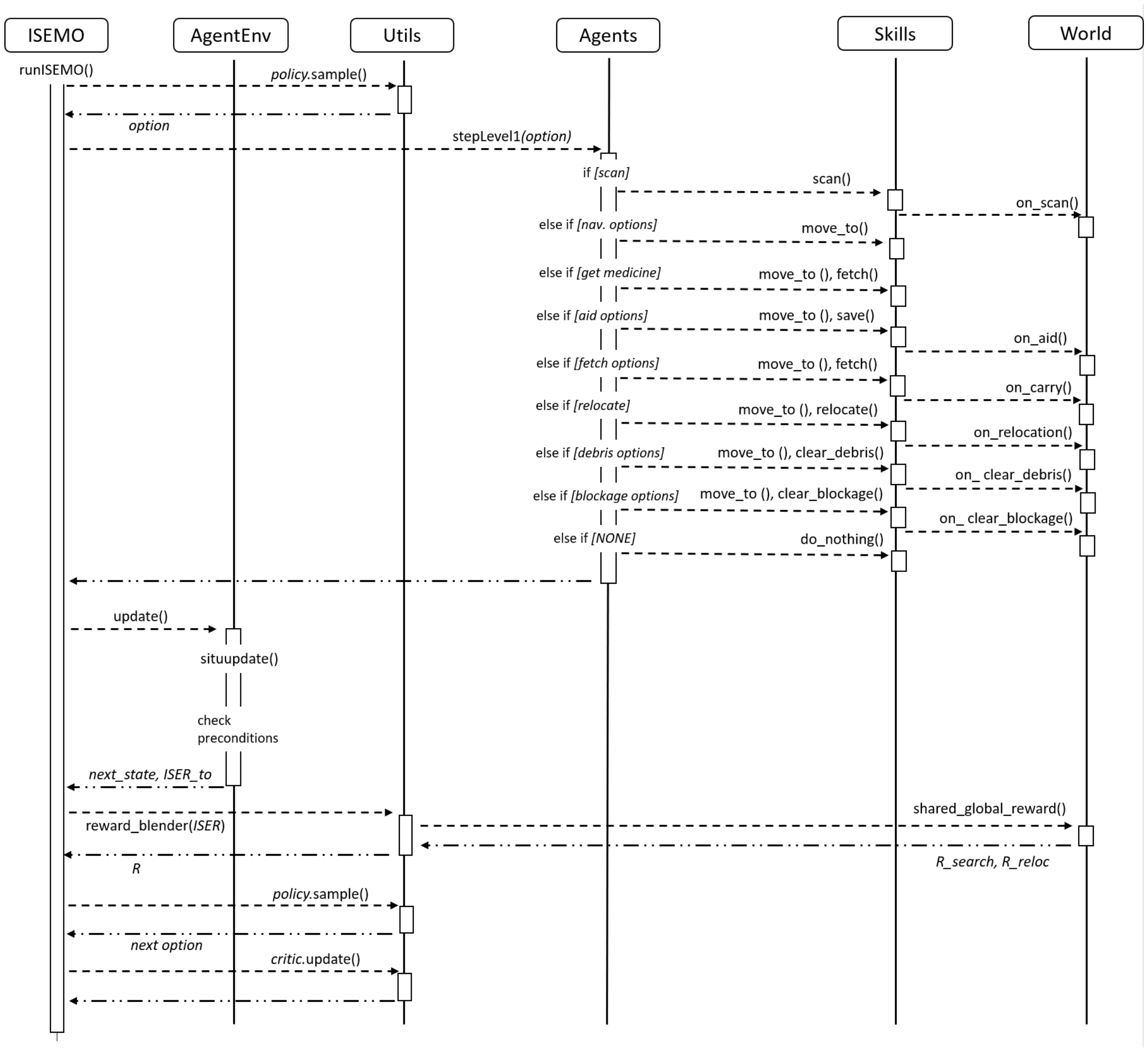

Figure 2: The timing diagram during a step of training.

2.2 Running the code

Before training, it is required to make the *World* objects. To make the world objects, give the following command: *python main.py −−make*. The result will be saved files named as "**MA-World-{i}.pl**, where i ranges from 0 to $nruns - 1$. $nruns$ is defined in the *args* class in main.py.

To run the software in the *training mode*, give the following command: *python main.py*. By default, this runs ISEMO. To run CoHRL instead, give the following command: *python main.py −−runCoHRL*.

During training, data is saved in files with the name as: "historyISEMO_testingFalse_.npy". The list of data items saved can be checked in *ISEMO.py* (refer to the multi-dimensional array *history*). In case of CoHRL, the name "ISEMO" is replaced with "CoHRL". Moreover, the learned models for the Q-functions and the termination functions (*betamodels*) are saved in the "models" folder.

To run the software in the *testing mode*, give the following command: *python main.py −−testing −−testID* {i}. Here, *testID* is the index of the saved World object (MA-World-{i}.pl) to be used for testing.