TÉCNICO LISBOA

AUTONOMOUS AGENTS AND MULTIAGENT SYSTEMS 2020

EXERCISE v0.1 (MARCH 2ND CHANGES IN RED)

Consider the problem of automatization, where agents can perform tasks on behalf of humans.

Agents can possess unique skills and resources that determine their ability to successfully accomplish specific tasks.

The perceived utility from performing a given task might vary with time. Consider, for instance, the following cases:

- a mine working robot can dig at different locations (tasks), yet the utility of mining at a specific location continuously varies in accordance with the characteristics of the collected materials;
- a processor can run different scheduling modes (tasks), yet the utility of scheduling under a given mode changes in accordance with the properties of the waiting processes.

In this context, agents need to be prepared to learn from experience in order to select (decide) the most promising tasks (actions) at a given time.

In the presence of a community of agents (for instance, a group of miners or processors), agents can further pursue different forms of coordination depending on whether they trust or not the other agents.

1. EXERCISE

Consider that our agents have a working autonomy of *n* steps.

During a cycle of *n* steps, agents perform tasks and perceive the utilities resulting from their work in order to form beliefs regarding the expected utility of each task.

Once a cycle ends, agents are recharged and erase their memory in order to renew beliefs for the next cycle.

1.1 SINGLE-AGENT SYSTEMS

1.1.1 Rationale Agents [5v]

Consider that throughout a cycle, the owner of the agent can suggest a task `Ti` for the agent to perform.

A task is suggested on a given step and is accompanied with a speculative utility per step, e.g. "`T1 u=2`".

The agent can execute the suggested task in the next step or at a later step in the cycle and perceive its actual utility every time it performs the task. Once the agent collects ground truth on the utility of a task, it can neglect the speculative utility given by its owner.

The agent sees all historical observations as equally important for its decision.

After a cycle is finished, the agent returns the expected utility for every task and the cumulative gains.

Only then, the agent is allowed to rest and recharge.

Quest: program a rationale agent, where the agent's goal is to optimize the cumulative utility along a cycle.

Let us give an *example* considering a cycle with 5 time points:

```
console>cycle=5 decision=rationale
console>T1 u=3
console>T2 u=4
console>TIK //step 1
//agent executes T2 and perceives utility
console>A u=1
//agent replaces speculative utility of T2 by the observed utility
console>TIK //step 2
//agent executes T1
console>T3 u=3
console>A u=2
//agent updates T1 utility
console>TIK //step 3
//agent executes T3
console>T4 u=2
console>A u=1
//agent updates T3 utility
console>TIK //step 4
//agent executes T1
console>A u=4
//agent updates T1 utility
console>TIK //step 5
//agent executes T1
console>A u=3
//agent updates T1 utility
console>end
//agent offers its owner the achievements of this cycle
agent>state={T1=3,T2=1,T3=1,T4=NA} gain=11
```

1.1.2 Task preparation and exchange costs [2v]

Consider that our agent may need to take some time to start performing each task.

Miners need to move to dig in a new site. Processors need to load a new scheduling mode.

The number of steps required to start a task (the *restart* cost) can thus be given upfront.

Quest: extend the rationale agent considering the preparation time when changing tasks.

```
console>cycle=5 decision=rationale restart=1
console>T1 u=3
console>T2 u=4
console>TIK //step 1
//agent prepares T2
console>TIK //step 2
//agent reassesses, executes T2
console>A u=1
//agent perceives utility
console>TIK //step 3
```

```
//agent reassesses, prepares T1
console>TIK //step 4
//agent reassesses, keeps up with T1, executes T1
console>T3 u=3
console>A u=1
console>TIK //step 5
//agent reassesses, keeps up with T1, executes T1
console>A u=1
console>end
//agent offers its owner the achievements of this cycle
agent>state={T1=1,T2=1,T3=NA} gain=3
```

1.1.3 Selective memory [2v]

In addition, the agent may place a memory factor to be able to differentially weight the probability associated with more recent observations. Given a task `Ti`, the probability of an observed utility can be adjusted by

$$\frac{{t_j}^k}{\sum_{i=1}^{m}({t_i}^k)}$$

where *m* is the number of observations for task `Ti`, t_j is the number of the step of the *j*th observation, and *k* is the memory factor. Illustrating, when the memory factor equals 0, we are in the scenario where the memory of the agent is not selective.

Quest: extend the rationale agent to consider the aforementioned memory factor on the agent's memory.

```
console>cycle=2 decision=rationale memory-factor=0.3
console>T1 u=2
console>TIK //step 1
//agent executes T1
console>A u=1
console>TIK //step 2
//agent executes T1
console>A u=3
console>end
agent>state={T1=2.1} gain=4
```

1.1.4 Flexible and conservative traits [2v]

Note: harder exercise, please only attempt to accomplish it if you have time.

Processors might run more than one scheduling mode.

When there are no task exchanging costs (`restart=0`), the agent can attempt to divide a single step across multiple tasks. For instance, an agent might allocate 60% and 40% of its efforts to T1 and T2 tasks.

In this context, the gain of the step corresponds to the utility affected to the allocated effort. Still, the observed utility is a fair one. Considering an agent with no selective memory, allocating T1 with 50% of efforts on step 1, observing an utility of 1 for T1, selecting T1 with 100% on step 2, and observing an utility of 3 for T1. The gain from T1 is 3.5 and the perceived utility of T1 is 2.

Quest: under the described circumstances, program a rationale agent with a conservative trait: attempting to avoid decisions that can produce a negative utility according to its beliefs.

```
console>cycle=3 decision=flexible
console>T1 u=1
console>T2 u=4
console>TIK //step 1
//agent prepares T2, executes T2
console>A u=11
console>TIK //step 2
//agent executes T2
console>A u=-1
console>TIK //step 3
//agent executes 50% of T1 and 50% of T2
agent>{T1=0.50,T2=0.50}
console>A u={T1=3,T2=2}
console>end
//agent offers its owner the achievements of this cycle
agent>state={T1=3,T2=4} gain=12.5
```

1.2 Multi-Agent Systems

Consider the presence of multiple agents.

The tasks suggested by the system's owner are visible to all the agents.

```
console>cycle=5 agents={A1,A2} decision=society restart=0
console>T1 u=3
console>TIK
//agents prepare T1, execute T1 and perceive utilities
console>A1 u=4
console>A2 u=2
console>TIK
...
console>end
agent>state={A1={...},A2={...}} gain=...
```

1.2.1 Homogeneous and heterogeneous societies [6v]

Quest: program a cooperative society of agents in the following situations:

1. homogeneous society where agents can communicate and know they are alike (`homogeneous-society`)
2. heterogeneous society where agents can communicate moved by a utilitarian social welfare, yet know they are different (`heterogeneous-society`)

```
console>cycle=2 agents={A1,A2} decision=homogeneous-society restart=0
console>T1 u=3
console>T2 u=2
console>TIK //step 1
//agents prepare T1, execute T1 and perceive utilities
console>A1 u=3
console>A2 u=0
console>TIK //step 2
//agents prepare T2, execute T2 and perceive utilities
console>A1 u=2
console>A2 u=3
console>end
agent>state={A1={T1=1.5,T2=2.5}},A2={T1=1.5,T2=2.5}} gain=8

console>cycle=2 agents={A1,A2} decision=heterogeneous-society restart=0
console>T1 u=3
console>T2 u=2
console>TIK //step 1
//agents execute T1 and perceive utilities
console>A1 u=3
console>A2 u=0
console>TIK //step 2
//A1 executes T1, A2 executes T2, and perceive utilities
console>A1 u=1
console>A2 u=3
console>end
agent>state={A1={T1=2,T2=NA},A2={T1=0,T2=3}} gain=7
```

1.2.2 Concurrency penalties [3v]

Note: medium-to-hard difficulty, please attempt to accomplish once previous tasks are implemented.

Consider the fact that tasks are optimally accomplished when undertaken by a single agent at a given time.

For instance, miners can get in the way of the workings of each other when digging the same site.

In this context, a penalty should be considered when two or more agents are concurrently performing the same task. This penalty is deducted to the gained utility for each agent.

If agents in the society are in a conflicting decision, the agent with lower identifier has priority in its choice and will further prefer tasks with lower identifiers.

Quest: extend the previous societies under this knowledge.

```
console>cycle=2 agents={A1,A2,A3} decision=heterogeneous-society restart=0 concurrency-penalty=2
console>T1 u=3
console>T2 u=2
console>TIK //step 1
//agent A1 and A2 executes T1, A3 executes T2
console>A1 u=5
```

```
console>A2 u=3
console>A3 u=0
console>TIK //step 2
//agent A1 and A3 executes T1, A2 executes T2
console>A1 u=3
console>A2 u=2
console>A3 u=2
console>end
agent>state={A1={T1=4,T2=NA},A2={T1=3,T2=2},A3={T1=2,T2=0}} gain=15
```

2. Final considerations

1. Utilities should be always displayed with two decimal places.
2. In case of tied decisions in single-agent systems, the eligible task with lowest index should be pursued.

   In case of tied decisions in multi-agent systems, decisions should be placed by selecting the tasks with lower indexes, starting from lower index agents to higher index agents. For instance, considering options {(A1=T1,A2=T2), (A1=T2,A2=T1),(A1=T2,A2=T2)}, then (A1=T1,A2=T2) option should be pursued.
3. The implementation of architectural principles – tolerance to input errors, decentralized decisions, asynchronicity – will not be evaluated. Architectural principles will be primarily assessed in the context of the course's project.
4. Please use the provided code examples in the course's webpage and do not change the way the agent reads from the standard input and writes on the standard output in your final submission.
5. The essential concepts on decision theory to solve the exercise were already covered in February. No advanced coordination aspects are required to solve the multi-agent tasks, you can carefully analyze the provided input output cases to understand the quest and solve multi-agent tasks right away.

Note: consult FAQ on the course's webpage before posting questions to faculty hosts.

3. Submission Instructions

Deadline: Friday, April 17th

Mooshak platform will be used for code submission. There is no other alternative for submitting the code.

Information on possible programming languages is available on the course webpage and Mooshak system.

The source code will be automatically evaluated by the Mooshak system.

Only the last submission will be considered for evaluation purposes. All previous submissions will be ignored.

Students should submit solutions to Mooshak as early as possible. Late deliveries may not be accepted.

The exercises' code will be subjected to strict copy checkers: if copy is detected after manual clearance, the registration in AASMA is nullified and IST guidelines will apply. Do not share your code. The application of plagiarism guidelines is also valid for students sharing the code, independently of the underlying intent.