AI6125 Multi-Agent Systems

Chen Yongquan

Nanyang Technological University

Group 1:
Arthi Thiyagarajan
Chen Yongquan
Fong Lin Qiang

Tileworld

Tileworld is an agent testbed first introduced in 1990 [1] as a simulated environment for investigating efficacy of intelligent agents. The objective of agents it to fill as many holes with tiles which can be found and picked up in the environment. Agents have a definable fuel capacity and can refuel at a fuel station grid cell. The sensing range of agents is also specifiable. At every time step, agents can perform one of these actions: {WAIT, MOVE_UP, MOVE_DOWN, MOVE_LEFT, MOVE_RIGHT, PICK_UP, DROP, REFUEL}. Moving consumes one fuel per step. The environment is highly parameterized and can be configured to simulate a particular situation, such as the creation rate and lifetime of objects. For our multi-agent environment, agents perform a sense-communicate-plan-act cycle once per time step.

Agent Designs

Our team experimented with 3 agent architectures. The first is a simple reactive agent with no communication. The second is a reactive architecture with limited communication for sharing the fuel location station. The third agent uses a hybrid architecture with advanced communication for information sharing and task sharing.

Hybrid Agent

This agent uses a horizontally layered hybrid architecture with procedural reasoning in its control subsystem, as shown in the Figure 1 below.

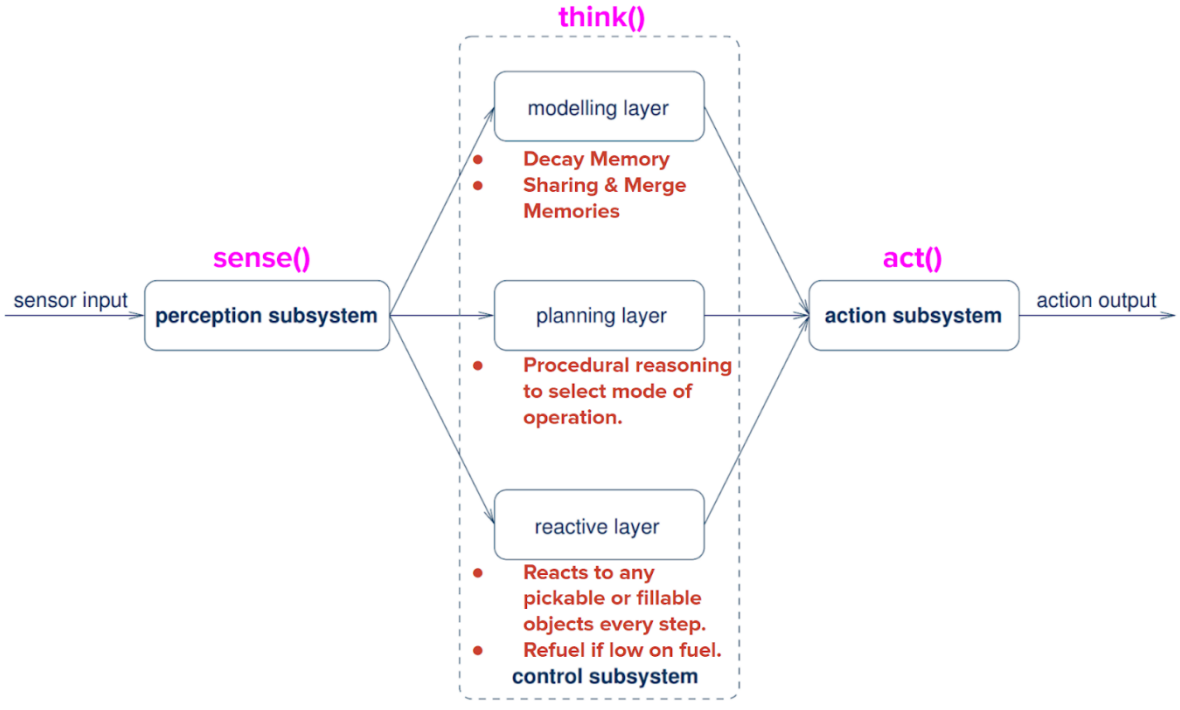

Figure 1

Source: http://www.inf.ed.ac.uk/teaching/courses/abs/slides/abs06-reactivehybrid.pdf

Modelling Layer

The agent uses a decaying memory to store the position of objects. Tiles and holes have their memory percepts decayed every time step and are forgotten once they expire or on sensing that they have disappeared. Expiry is determined by the maximum lifetime defined in the

environment parameters and the estimated remaining life of the objects. The estimated remaining lifetime is given by the following formula:

$$\begin{aligned} &Estimated\ Remaining\ Life(Obj) \\ &= (Max\ Lifetime \times objectLifetimeThreshold) \\ &-\big(Current\ Step - StepFirstSensed(Obj)\big) \end{aligned}$$

The *objectLifetimeThreshold* hyperparameter can be used to augment the maximum lifetime and be more conservative in our estimation. $StepFirstSensed$ is the timestamp the memory percept is created. The original memory class overwrites this timestamp with the current time on updates. Our extended class retains the original timestamp if the percept is already present. Thus, we do not break any rules by retrieving the actual life from the environment instance and is estimated based on the memory percept instead. Agents are able to share their memory and are also able to merge shared memories into their own, without overwriting their own or other agents' sensed regions.

Planning Layer

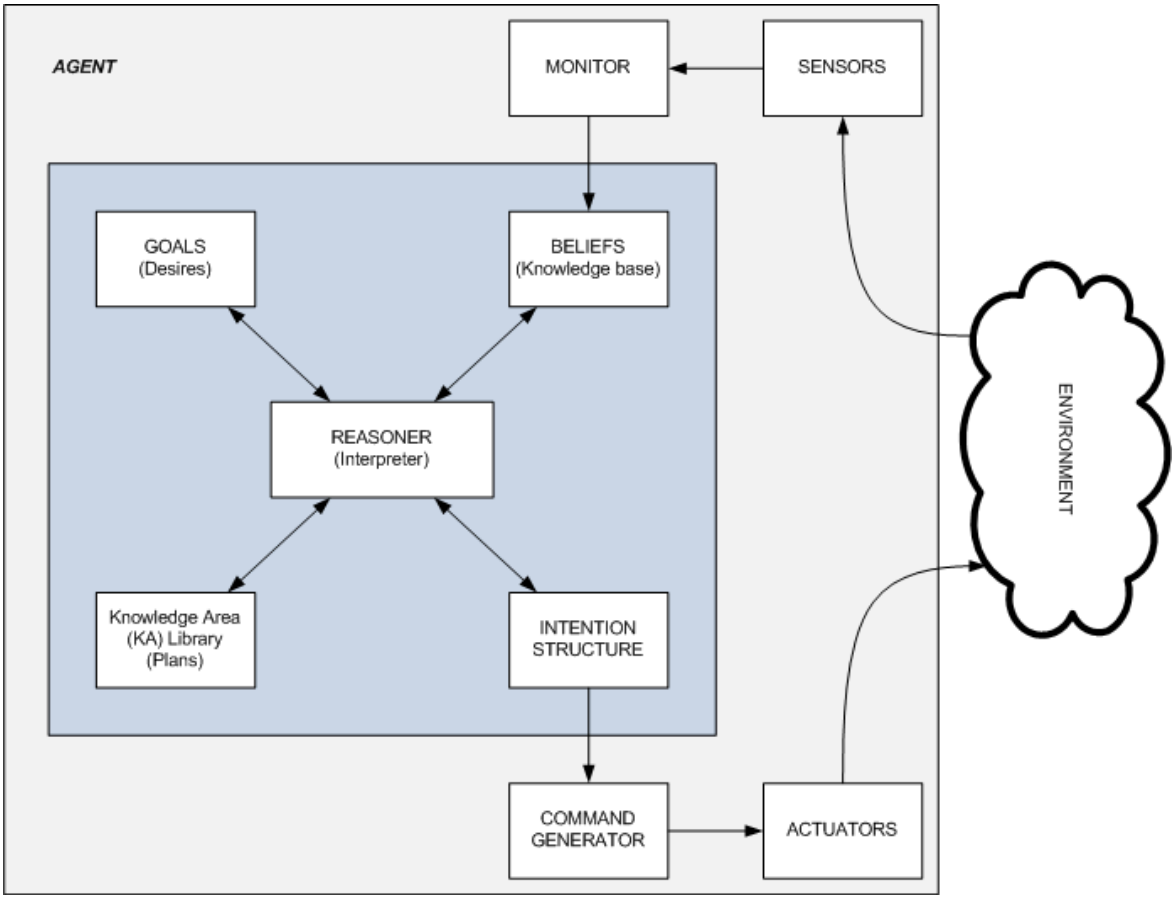

Figure 2
Source: https://en.wikipedia.org/wiki/Procedural_reasoning_system

We employ a procedural reasoning system for deciding which mode of action to take every step. The database which contains beliefs about the world is provided through the modelling layer. The intention structure takes the form of one of the following modes: {EXPLORE, COLLECT, FILL, REFUEL, ASSIST_COLLECT, ASSIST_FILL, REACT_COLLECT, REACT_FILL, WAIT}. The command generator takes in the mode and generates a new **TWThought** instance with a **TWAction** and an additional **TWDirection** if applicable. This is then passed to the *act()* method

for execution. The knowledge area defines sequences of low-level actions towards achieving a goal. Thus, the **DefaultTWPlanner** class generates and stores the optimal steps to take to a specified goal coordinate using the A* search algorithm. Low-level actions like picking up tiles and filling holes are defined in the **TWAgent** class with the *pickUpTile()*, *putTileInHole()*, *move()* and *refuel()* methods. The reasoner is implemented in *think()* following the decision flow chart below:

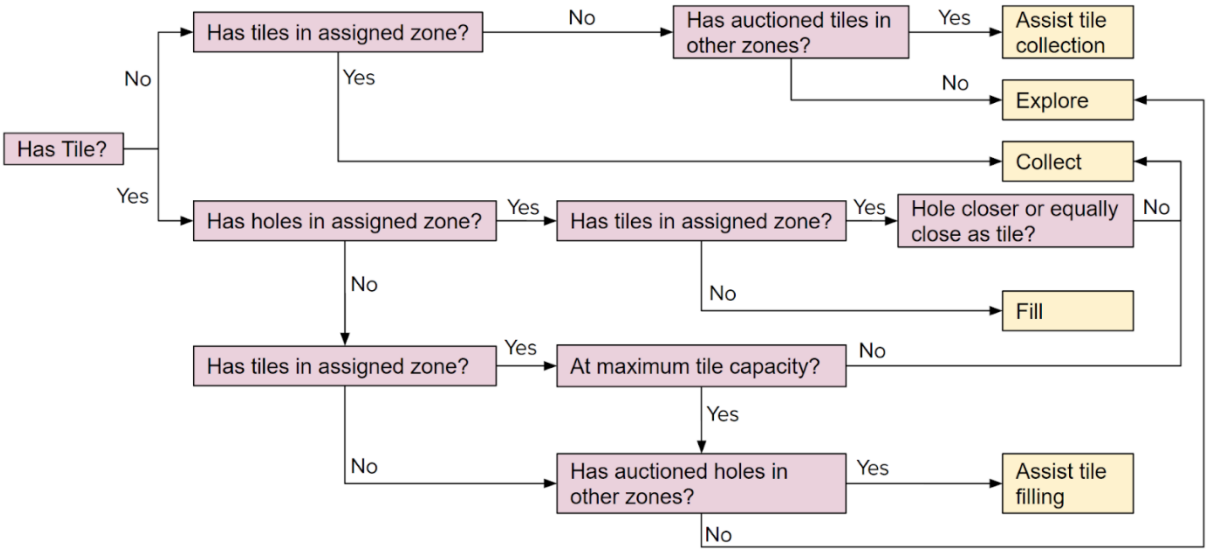

Figure 3

Reactive Layer

The reactive layer allows the agent to react to any tiles or holes it stumbles upon and to manage its fuel level. Fuel management is enforced through two policies. Firstly, *hardFuelLimit* which forces the agent to refuel if fuel level dips below it. This is normally set to the longer dimension of the map, maximizing operation uptime while minimizing downtime due to refueling. Secondly, a dynamic adjustment policy limits the distance of the agent from the fuel station by its fuel level. A tolerance factor, *fuelTolerance*, to allow for more fuel buffer, accounts for the risk of obstacles blocking the direct path to the fuel station. This second policy allows the agent to dynamically adjust its position so that it does not overextend its fuel. We normally set this tolerance to 0.95. Decreasing it further would reduce the chances of fuel failures. By following the two hyperparameter guidelines above to tune them on-the-spot, we were able to achieve a competitive average score of 76.7 in the unknown 100×100 environment, with the best being 93.

Exploration Strategy

Our exploration strategy divides the map into as many zones as agents detected in the environment. Zones are not hard coded and is instead dynamically inferred based on the number of unique agent identifiers broadcasted into the environment. This means the multi-agent system would function flawlessly with any number of agents, without any changes to the codebase (excluding *createAgent()* calls). Map division is decentralized to prevent systemic failure in case of communication breakdowns. For instance, if we were to designate a leader agent to divide the map and message other agents their assigned zones, there would be a failure if the leader agent does not initialize correctly or its communication is somehow

blocked. Decentralization also means there is no single point of failure in the multi-agent system.

The division is deterministic, and all agents generate the same results using the same positions broadcasted. Zones are assigned to their nearest agent, and all agents know everyone's zones without directly communicating this information.

Major zones are further subdivided into anchor zones, determined by the sensor range. Referring to Figure 4, the red lines delineate how the 50×50 map is divided into 3 zones. The first two are 50×16 while the last takes up all remaining estate and is 50×18. Square maps are divided into horizontal zones. However, agents also recognize rectangular environments and divides along the longer edge so that zones are not too elongated. The yellow boxes show example anchor zones. It is usually the case that zones cannot be perfectly divided by the anchor zone. As such, we add additional anchor zones from the end of the opposite edges, as shown by the orange box. The center of these anchor zones are used as goal coordinates.

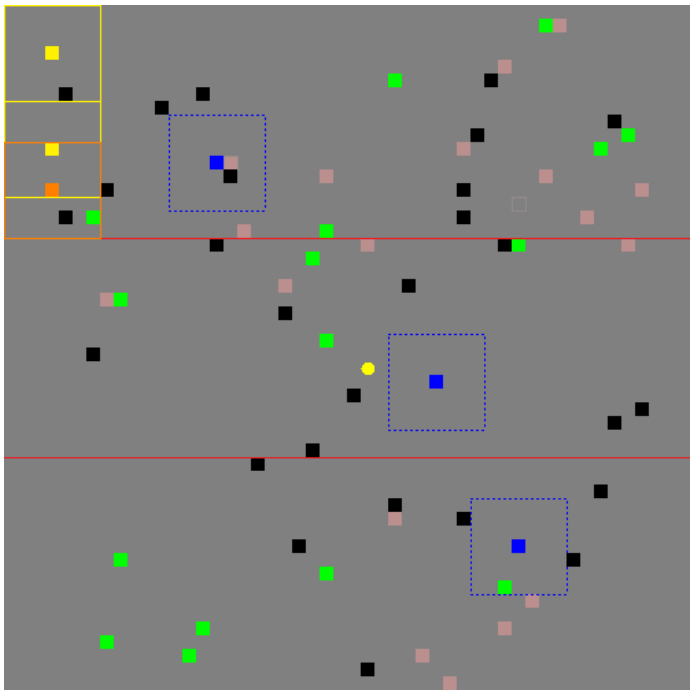

Figure 4

Agents keep an exploration decay map in their working memory and prioritizes the least explored anchor zone. Cells in this decay map are reset to 0 when they are in sensing range and decays otherwise. The exploration score of an anchor zone is the sum of all cells in that zone and the anchor zone with the largest decay is targeted. We initially used an arithmetic decay function, i.e. +1/step. However, this resulted in rubberbanding when the exploration scores of two anchor zones are similar. Because agents do not instantly teleport into the center of the anchor zones, when they approach an anchor zone, its total score will gradually drop to 0. However, because another anchor zone has a similar score to this zone, the gradual drop causes that zone's decay to inch above this's, and the agent targets that zone before it completely explores the current zone. On leaving this zone, the decay of this zone starts increasing and the phenomenon repeats. We thus used geometric decay, i.e *2/step, which solves the rubberbanding yet does not decay as fast as exponential decay, where cells hit the cap of Double.*POSITIVE_INFINITY* too fast, resulting in decay scores becoming indifferentiable

from one another. We initialize all decay cells to Double.*POSITIVE_INFINITY* to force exploration of the entire map at least once until the fuel station is found.

Target Prioritization

Agents often need to prioritize between multiple objectives. We note that the Tileworld task is analogical to the Travelling Salesman Problem albeit with additional time and resource constraints such as object decay and fuel limit. Alternatively, a ride sharing route planner is also applicable, where it plans the optimal route for drivers to ferry passengers, while considering the risk of booking cancellations if drivers are late. Thus, prioritization using Manhattan distance is simple but not optimal. Instead, we use an additional time modifier as shown below. Between two equidistant objects, the one at risk of disappearing is then prioritized.

$$TSPDistance(Obj) = \frac{Estimated\ Remaining\ Life(Obj)}{Max\ Lifetime} \times ManhattanDist(Obj)$$

Communication

Our communication strategy takes a two-pronged approach, with both information sharing and task sharing. For information sharing, agents broadcast their own memory into the environment for other agents to merge into their own losslessly.

For task sharing, agents auction out untenable objectives or surplus objectives that can be expedited with assistance. Before defaulting back to exploration when no objectives are found in their own zone, agents may pursue available contracts broadcasted in the environment. The *maxAssistZoneDistance* parameter limits how many zones agents can cross, so that they do not waste too many steps assisting. To determine whether an objective can be reached in time, agents compare the object's Manhattan distance to its estimated lifetime. Furthermore, agents can reserve a set number of objects using *goalAnnounceCount* and auction out the rest. If there are available tiles or holes but the agent is already at maximum tile capacity or it has no tiles to fill holes with, respectively, those objects are also auctioned out.

Because there is a risk of goal collisions when agents are allowed to interact with objects outside of their zones, such as when stumbling upon objects, or when assisting other agents, agents will be required to broadcast their goals for other agents to resolve goal collisions. Thus, this mechanism also marks contracts as assigned.

Continue on next page

Results

Environment 1:

- Environment Size: 50×50
- Average Object Creation Rate: NormalDistribution$(\mu = 0.2, \sigma = 0.05)$
- Lifetime: 100

Environment 2:

- Environment Size: 80×80
- Average Object Creation Rate: NormalDistribution$(\mu = 2, \sigma = 0.5)$
- Lifetime: 30

Environment 3:

- Environment Size: 100×100
- Average Object Creation Rate: NormalDistribution$(\mu = 0.1, \sigma = 0.025)$
- Lifetime: 150

We tuned the following hyperparameters on-the-spot during the presentation:

- *fuelTolerance* = 0.95
- *hardFuelLimit* = 100
- *TSPHeuristic* = *true*
- *objectLifetimeThreshold* = 1.0
- *allowAssistance* = ***false***

Unfortunately, agents still ran out of fuel on some runs.

Nonetheless, we further tuned the hyperparameters of the agent after the presentation:

- *fuelTolerance* = 0.95
- *hardFuelLimit* = 100
- *TSPHeuristic* = *true*
- *objectLifetimeThreshold* = 1.0
- *allowAssistance* = ***true***

We also present the results using the tuned hyperparameters. The rest of the codebase is entirely the same as the one used for presentation.

Environment	Simple Reactive										Reactive + Limited Communication										Hybrid + Advanced Communication (Tuned)										Hybrid + Advanced Communication (Presentation)									
1	**30**	14	31	25	20	26	35	11	29	29	303	315	**338**	295	298	292	310	295	301	297	382	**419**	411	388	391	384	398	388	393	412	382	**419**	411	388	391	384	398	388	393	412
	25										304.4										396.6										396.6									
2	122	134	117	**151**	133	83	140	122	129	134	**504**	486	463	484	476	486	497	467	490	489	498	513	541	533	537	**553**	505	476	498	498	498	513	541	533	537	**553**	505	476	498	498
	126.5										484.2										515.2										515.2									
3	0	0	1	0	1	0	**2**	0	0	0	1	88	**90**	3	3	32	73	1	78	76	102	88	91	99	97	93	31	93	100	**107**	67	83	50	65	**93**	78	80	81	79	91
	0.4										44.5										90.1										76.7									

Analysis

The hybrid agent performed the best in all environments. Our results show that adding communication to the reactive architecture improved overall scores tremendously. For the hybrid agent, turning on assistance can stabilize scores across different runs in configurations with low object creation rate (Env1). However, the maximum score is lower as time is wasted travelling between zones. This effect can be worse when we set the *maxAssistZoneDistance* to larger than 1. In configurations with high object creation rate (Env2), turning on assistance can increase the maximum score barely, with no disadvantages. Using the TSP heuristic can be beneficial in environments with low object lifetimes (Env2), while there is negligible difference vice versa (Env1), though it is harmless to use it. We found that proper tuning of the *fuelTolerance* and *hardFuelLimit* thresholds is vital towards preventing fuel failures. *goalAnnouceCount*, which determines which objects are surpluses, is best set to 1. This minimizes goal collision while still maximizing available contracts for other agents. We find that *objectLifetimeThreshold* is not a very useful modifier for tuning the agent. Though it can sometimes increase the average rewards across runs, it is a fairly unstable modifier and in other configurations can decrease it instead. Being that it is hard to tune, we just set it to 1.0 for most setups.

References

[1] M. E. Pollack and M. Ringuette, "Introducing the Tileworld: Experimentally Evaluating Agent Architectures," 1990.

Appendix

- When map is too large relative to default fuel level, agents may run out of fuel before map can be completely explored. If the fuel station is in their zone, all agents eventually run out of fuel. Thus, an additional failsafe is needed, where the failing agent broadcasts out its unexplored cells for other agents to assist.
- For large maps, we considered limiting the working area to a set radius around the fuel station after it is found, to maximize agents' operation time and reduce time spent on refueling.
- We mulled over the possibility of using a probability-based model such as a utility-based model. However, we relegated that due to time constraints and reservations about its final performance as it would be hard to assign the utility values. An optimal setup would require us setting arbitrary utilities and slowly approach the optimal values either through genetic algorithms, random forest or gradient boosting. Furthermore, the unknown environment is also a major consideration for us, as we doubt we could tune so many hyperparameters on-the-spot during presentation and still outperform our exploration algorithm.
- An efficient path planner would calculate an optimal path to a goal and use the exact same path generated until a new obstacle appears along the path or an existing one disappears, rendering the original path no longer optimal. Code wise, this would mean either checking for changes within a bounded zone from the agent to the target every timestep and generating a new path on detecting changes. A less optimal solution that does not take into account existing obstacles disappearing would be to only generate a new path when the *act()* method triggers a *CellBlockedException*. This solution would only generate a path when an obstacle appears along a previously generated path. After careful deliberation, we opted for generating an optimal path every time step instead. The second solution required making compromises in optimality while the first solution required additional computation. When comparing these computations with that of calling the A* search algorithm every time step, the benefits in speed are rather minimal if we tune the A* search depth properly. However, we do note that this is the case because the Tileworld environment and its search space are much smaller compared with real-world environments. Nonetheless, using this method, we were still able to run the agents reasonably fast, without catastrophic errors, and still achieve competitive scores for all 3 environments during the presentation.