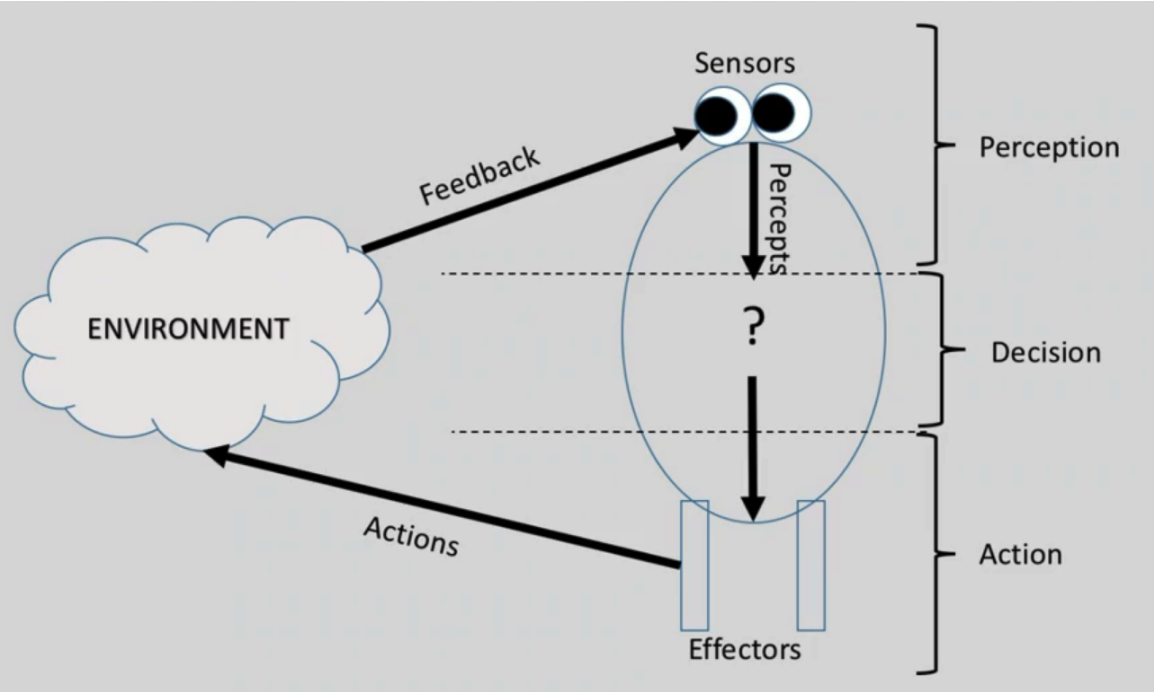
Intelligent Agents

▼ Refined definition of agent

An agent is a computer system that is situated in some environment, and that is capable of autonomous action in that environment in order to meet it's delegated objectives. It must decide both **what** actions to perform and **when** to perform an action.

▼ Agent within an environment

Here is a depiction of an agent.

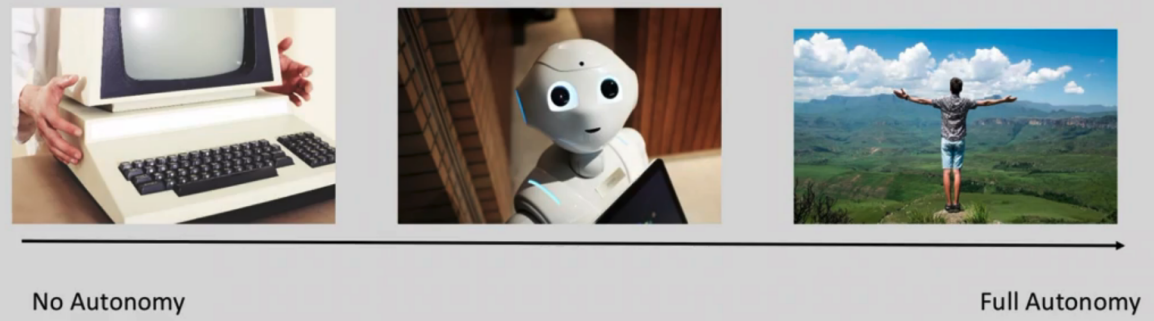


It's effectors help an action to be carried out. The environment is constantly giving feedback to the agent through it's sensors, and this generates a percept. For example, a thermostat might sense the temperature in the room. In the question mark, the agent is making some kind of decision, regarding how to take advantage of the situation in order to bring about it's goal. It will decide the best action to take, take the action and that will affect the environment in some way.

This is all part of the Agent Control Loop.

▼ Autonomy

Autonomy is the freedom of an agent to do what they desire. It is on a spectrum, with agents sitting somewhere in the middle of it



Agents have a restricted amount of autonomy, being able of deciding what to do within some confinements.

Also, autonomy is adjustable, so if an agent feels like it can't make a decision it can come back to us to get some feedback. An agent that either never comes back for feedback and does their own thing, or comes for it too often is not really desirable.

▼ Examples of simple agents

- Thermostat
  - Environment: physical room
  - Delegated goal: maintain room temperature
  - Actions: heat on / heat off
- Software demon (Biff email alert program)
  - Environment: Unix OS
  - Delegated goal: check for email, flag if new email arrives
  - Actions: GUI interactions

They are "simple" because the decisions made are trivial, but we can argue they are agents nonetheless.

▼ Comparing agents to other things

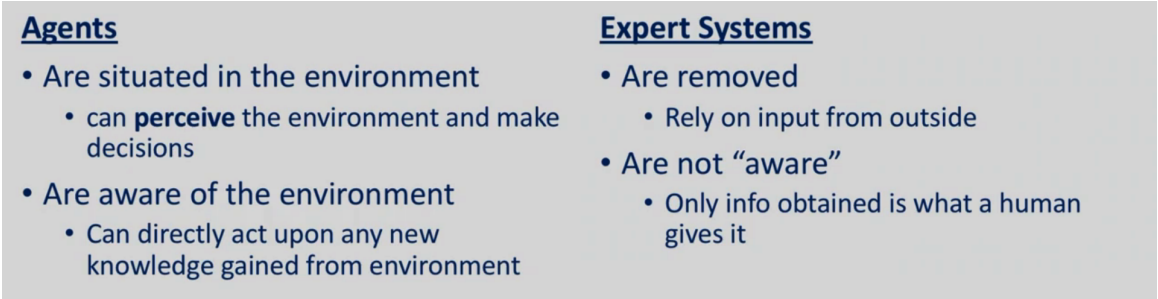
It could be said agents are objects "with attitude", a special type of object. Objects perform an action because they have to, while agents have a decision-making ability, so they choose to perform an action. Objects are not smart nor flexible, doing exactly what is programmed, while agents are smart and have a flexible behaviour, doing something only if it is worth it for them, being quite "selfish".

Agents	Objects

• Are autonomous
  • can **choose** to perform an action
	• Are not autonomous
	  • Perform an action because they **must**

• Are Smart
  • Capable of flexible behaviour
	• Are not “smart”
	  • OO says nothing about flexible behaviour per se

Agents could also be compared to expert systems. ES's are essentially separate computer programs, relying on human input to gather information (of course also being able to infer some of it), but they are not situated within an environment. They are "disembodied", while agents are in an environment and able to perceive things. Everything an ES has is what a human provides them, while agents are aware of their environment.



▼ Environments

An environment is not just a physical world, it could also be a software world. It's characteristics can have a massive effect in agents and how we build them. It's main characteristics are:

- Fully observable or partially observable: in the first one we can have perfect information, allowing us to make very good decisions. If it is partially observable decisions made can be good, but something may come to light later on indicating we didn't make the best decision. Partially observable environments are harder to work with.
- Deterministic or non-deterministic: if an environment is deterministic, you take an action and it always has the same action. That gives you some certainty about what is going to happen. If an environment is not deterministic, the same action could have different effects, introducing certain uncertainty into the environment, having to figure that out when building the agent.
- Static vs dynamic environment: a dynamic environment is constantly changing, while a static environment is the same every time unless we change it ourselves. This could be exemplified by looking at a road (where cars change every time you look) vs looking at a painting, which is always the same.
- Discrete vs continuous: in a discrete environment, the amount of states in the environment or actions that can be taken are countable. In a continuous environment, these are not countable, being a lot harder to deal with while also being much closer to reality.
- Episodic vs non-episodic environment: the same way as in tv shows, the actions taken in an episodic environment will only affect the current episode (like The Simpsons episodes) and be reset at the end of the episode, while in a non-episodic environment, every action taken will affect it until the end of time. For example,

driving a car is sequential or non-episodic, since a bad turn in the beginning of the journey will affect the entirety of the journey.

- Real Time environment: in a real time environment, time does not stop and keeps on going.

▼ "Intelligent" agents

We want an agent to choose the correct action and perform it at the correct time. This does not need to solve all the AI problems in the world, but requires 3 properties:

- Reactive: an agent needs to be aware of their environment and has to react to it. This is a stimuli-response model. For example, if the agent has a goal and the environment changes in such a way it is impossible to achieve such goal, the agent should stop trying to achieve it. The agent should decide if it is worth it to carry on with their activity, or if it maybe needs to change something to achieve their goal. This is important in an environment that could be constantly changed by other agents.
- Proactive: an agent needs to seek opportunities and exploit them to bring about their goal states.
- Social: this is not just sending and receiving data all the time, but to take the data they're receiving and being able to negociate, coordinate and cooperate.

▼ Intentional systems

An intentional system is focused around human attributes and mental states, like beliefs, feels, wants, needs... and how we attribute them to machines (agents).

For example, I **believe** it will rain and I **want** to stay dry, so I'm going to bring an umbrella with me.

We can see we have a **belief** of the world and a **desire**.

This means we can **predict** the way people will act based on their beliefs and desires. That they will act a particular way.

This is the whole thing about intentional systems. Agents' behaviour can be predicted by attributing beliefs, desires and a rational argument to them.

▼ Logic orders

Dennit described that Intentional Systems can be placed in several different levels, increasing in complexity each time.

For example, the statement "Craig thinks that spaghetti bolognese is acceptable as a breakfast food" would be a **first order** statement.

The statement "I think my friend believes in magic" would be second order logic, since it contains **a belief about a belief**.

These orders are not covered in depth in the module, but play an important role in epistemic logic.

John McCarthy asked if it is legitimate to give human-like mental states to machines. It is, when it expresses the same information about a machine as it would a person. We said beliefs and desires can be used to predict how a person would act (bringing an

umbrella when wishing to stay dry), so if a robot has the same intention, we can expect it would take the same action.

So, it is useful to do that when the ascription helps us understand:

- The structure of the machine
- The machine's past or future behaviour
- How to repair or improve the machine

This is not ever logically required, since there are many other descriptions we can use. However, it may come in handy depending on the situation, like trying to teach your grandma how to use a computer, instead of explaining her how it works at a bit-level. This abstraction is more useful when we want to understand the behaviour of the machine when we don't know it's entire structure.

Intentional systems give us a familiar, non-technical way of explaining how agents work.

▼ What can be explained using intentional stance

Let's use a toaster as an example.

A toaster is a very cooperative agent. It will always toast bread when it thinks we want toast. We communicate this desire for toast to the toaster by inserting bread, and then it gives us toast. There is nothing stopping us from believing there is a dragon inside toasting the bread for us.

The more we know about a system, the less intentional stance is needed, (like thinking weather depends on the gods' mood in ancient times).

At the same time, the more the complexity of a system increases, the more useful these abstractions become.

In conclusion, taking agents as intentional systems:

- Is good for characterising agents (they want, believe...)
- Allows for nested representation (second order logic)
- Allows for an agent to work as a post-declarative system, generating an agent's actions based on their beliefs and desires

▼ Abstract architecture for agents and environments

An abstract architecture helps us to

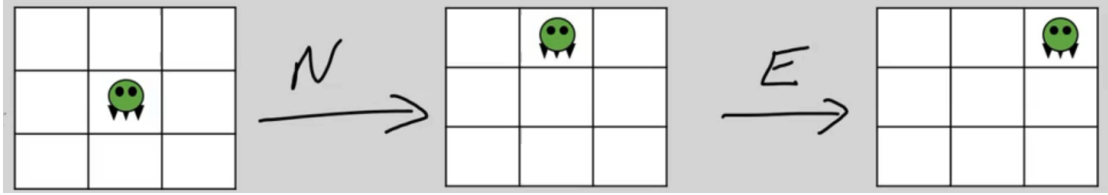
- Model states of the environment
- Model possible actions
- Model decision making
- Model actions taken

▼ Example: tileworld

We have a finite set of states (0,0) to (2,2) and a finite set of actions "North", "East", "South", "West".

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

An agent can move through the environment, giving us a set of actions taken and states it has been in. An action directly links it's adjacent states.



This set of interleaved environmental states and actions is called a run.

▼ Notation

The environment can be in any of a finite set of states

$$E = \{e, e', ...\}$$

There is a finite set of actions available to the agent

$$Ac = \{\alpha, \alpha', ...\}$$

A run is a set of interleaved world states and actions

$$r : e_0 \rightarrow^{\alpha_0} e_1 \rightarrow^{\alpha_1} e_2 ...$$

We have to make sure that all "e"'s are members of the set of states and all "alpha"'s are members of the set of actions. All of them have to be valid and not random states and actions.

$$\forall i \geq 0 : (e_i \in E \ \wedge \alpha_i \in Ac)$$

▼ Applying notation to Tileworld

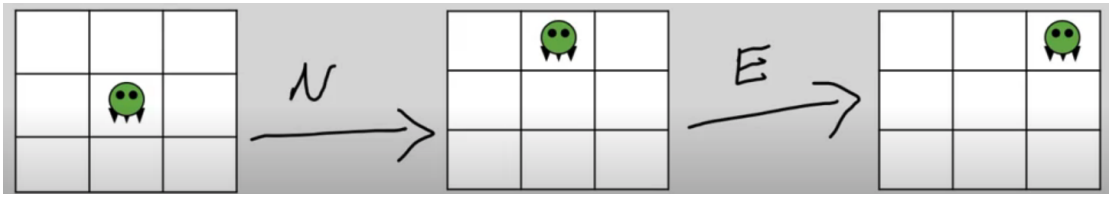
We have a finite set of states E

$$E = \{(0,0),(0,1),(0,2),(1,0),...,(2,2)\}$$

And a finite set of actions Ac

$$Ac = \{N, S, E, W\}$$

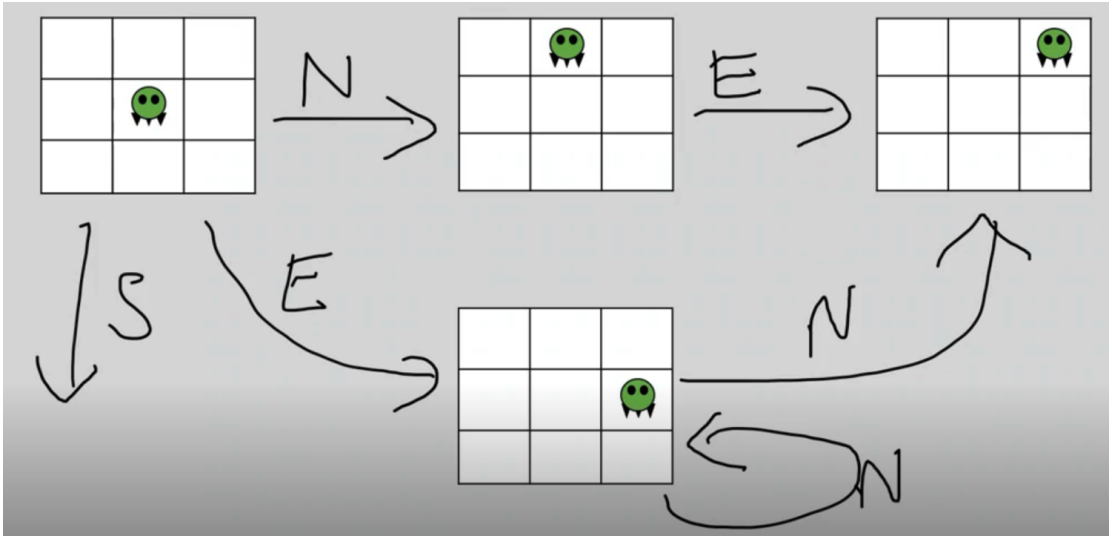
We can write the run



As

$$r : (1,1) \rightarrow^{N} (0,1) \rightarrow^{E} (0,2)$$

In the case we have multiple runs, as in



We also have to be able to consider **all possible runs** from **all possible starting states**.

▼ Runs

The set of all possible runs is expressed as

$$\mathcal{R} = \{r, r', ...\}$$

The set of all runs that end with an action is expressed as

$$\mathcal{R}^{Ac} \subseteq R$$

The set of all runs that end with a state is expressed as

$$\mathcal{R}^{E} \subseteq R$$

We give different notations to these specific subsets because agents and environments will react to these runs in a different ways.

▼ Behaviour of environment

A state transformer function represents the behaviour of the environment

$$\tau : \mathcal{R}^{Ac} \rightarrow 2^{E}$$

It transforms a run ending in an action into a new environmental state. It maps an action to a powerset of the environmental set. That is the set of **all possible** subsets. It's a way of saying "this action will lead to some available states".

▼ Example

$$\tau(e_0 \rightarrow^{\alpha_0} e_1 \rightarrow^{\alpha_1} e_2 \rightarrow^{\alpha_2}) = e_3$$

$$\tau(e_0 \rightarrow^{\alpha_0} e_1 \rightarrow^{\alpha_1} e_2 \rightarrow^{\alpha_2'}) = \{e_3, e_4, ...\}$$

The second run is non-deterministic, since the transformer action gives us a set of possible states.

If for a particular run, the transformer function returns an empty set, the agent is stuck and the run over

$$\tau(r) = \emptyset$$

If, for every single run that ends in an action, the transformer function will only ever return one state, it is deterministic.

$$\forall r \in \mathcal{R}^{Ac} \, |\tau(r)| = 1$$

If there is at least one run that ends in an action where the transformer function returns more than one state, it is non-deterministic, since we can't guarantee what the effect of each and every action is going to be.

$$\exists r \in \mathcal{R}^{Ac} \, |\tau(r)| > 1$$

▼ Environment

An environment is a triple of a set of states E, an initial state e0, and a transformer function tau. The transformer function indicates how does the environment behave.

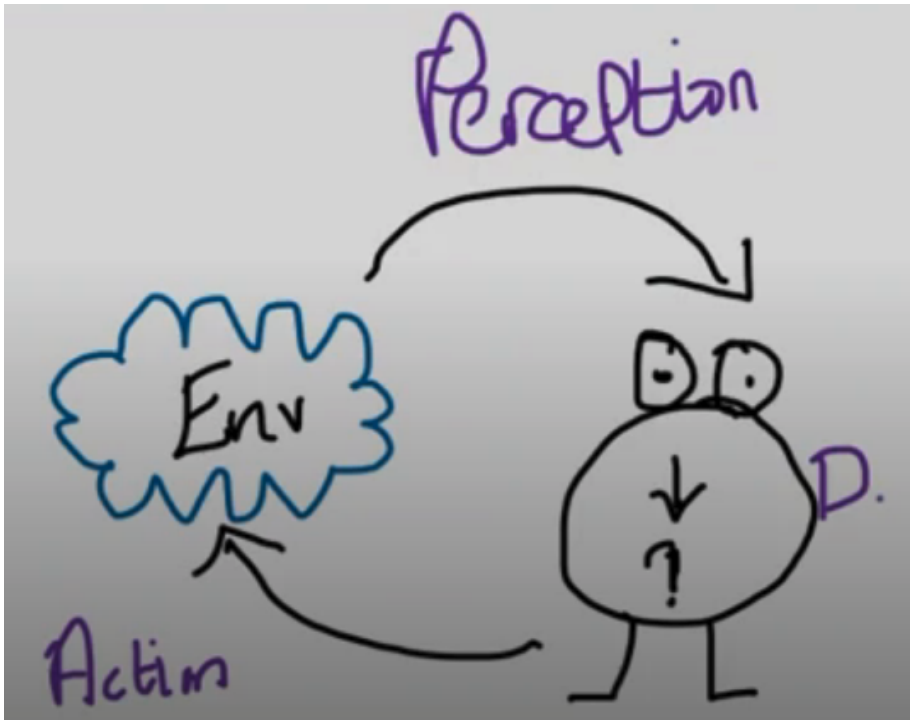
$$Env = \langle E, e_0, \tau \rangle$$

▼ Agent

An agent behaves like a function that maps runs ending in states to an action.

$$Ag : \mathcal{R}^E \to Ac$$

The run ending in a state is perceived by the agent, which then makes a decision and performs an action. This forms a loop



▼ Purely reactive agent

Some agents only care about the present, ignoring how they got to where they are, merely "reacting without any thought or consideration". This means they only map a state to an action

$$Ag : E \to Ac$$

They will always do the same thing when they get to a state.

▼ System

A system is a pair of an agent and an environment.

Formally, we define a sequence as

$$e_0 \alpha_0 e_1 \alpha_1 e_2 \alpha_2$$

In order to see if it is a run of an agent in an environment

$$Env = \langle E, e_0, \tau \rangle$$

we need to check some conditions.

First, we need to check if e0 is the initial state of the environment.

Second, we need to check for the very first action. So, if the agent is in e0, is the action they choose alpha0?

$$Ag(e_0) = \alpha_0$$

Third, we need to check if, for each of the environmental states in the run, does the agent perform the same action as the sequence says?

$$For\ u > 0 : e_u \in \tau((e_0, \alpha_0, ...e_{n-1}, \alpha_{n-1}))\ and \alpha_u \in Ag(e_0, \alpha_0, ...e_n)$$

Then we can put together all possible runs of agents in the environment

$$\mathcal{R}(Ag, Env)$$

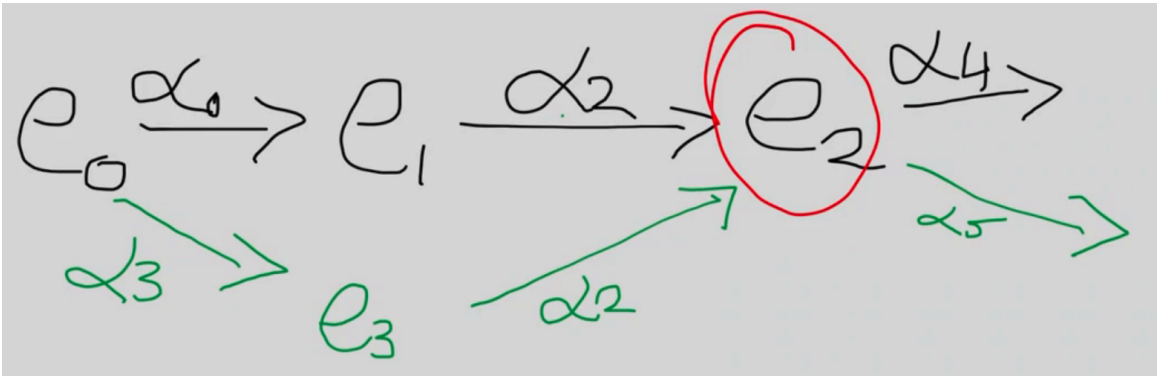
If we have two sets of runs of in an environment, each of them with a different agent, we can not tell the difference if those sets are identical

$$\mathcal{R}(Ag, Env) = \mathcal{R}(Ag', Env)$$

▼ Agent Control Loop

As we previously saw, we can see an agent as a function, of which there are two types:

- Purely reactive agent, which does the same actions every time it gets to a certain state
- Deliberative agent, which maps a particular run to an action, being able to take different actions from a given state depending on all of it's previous states.



The main element behind all this is the Agent Control Loop

It has the following components:

- State: internal state of the agent, data structure that has information on environmental state and history, like the run taken or what the agent believes about other agents or the environment. The set of all possible internal states is "I".

$$I = \{i_0, i_1, ...\}$$

- "See" function: the agent perceives the environmental state. It takes in an environmental state and outputs a percept (an internal representation that the agent has seen and can make sense of).

$$See : E \rightarrow Per$$

- "Next" function: it takes in the percept outputted by the "See" function and the current internal state and uses it to update it's internal state (it's beliefs about the world)

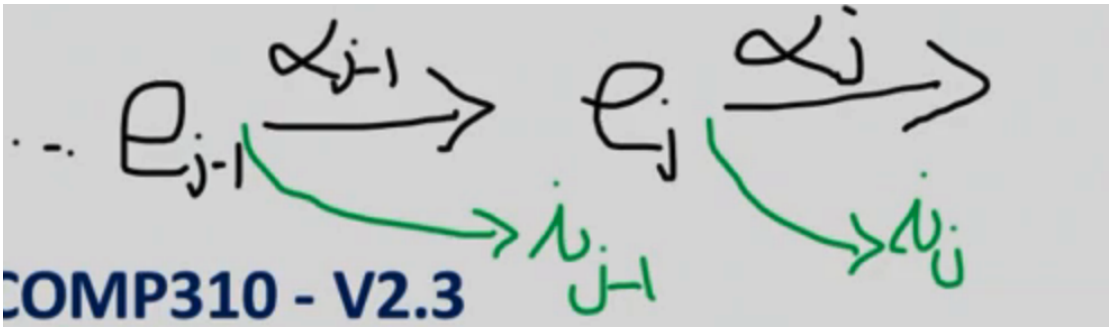
$$next : I \times Per \rightarrow I$$

- Action: the agent maps it's internal state to an action, sort of asking "what do I do now?". It takes in the current state and outputs an action

$$action : I \rightarrow Ac$$

▼ Example

We have this run



As the environment gets to state e_{j-1}, we do the See and Next functions, and generate the state i_{j-1} and perform the action alpha_{j-1}. Then the environment state changes to e_j. We begin by perceiving the environment with See(e_j), which returns a percept and we pass that along with the internal state i_{j-1} to the Next function, which generates that such as

$$Next(See(e_j), i_{j-1})$$

The Next function generates a new state i_j.

Finally, we move to Action, which maps our internal state i_j to an action, telling us we have to perform alpha_j. Everything combined looks like

$$\alpha_j = Action(Next(See(e_j), i_{j-1}))$$

▼ Utility

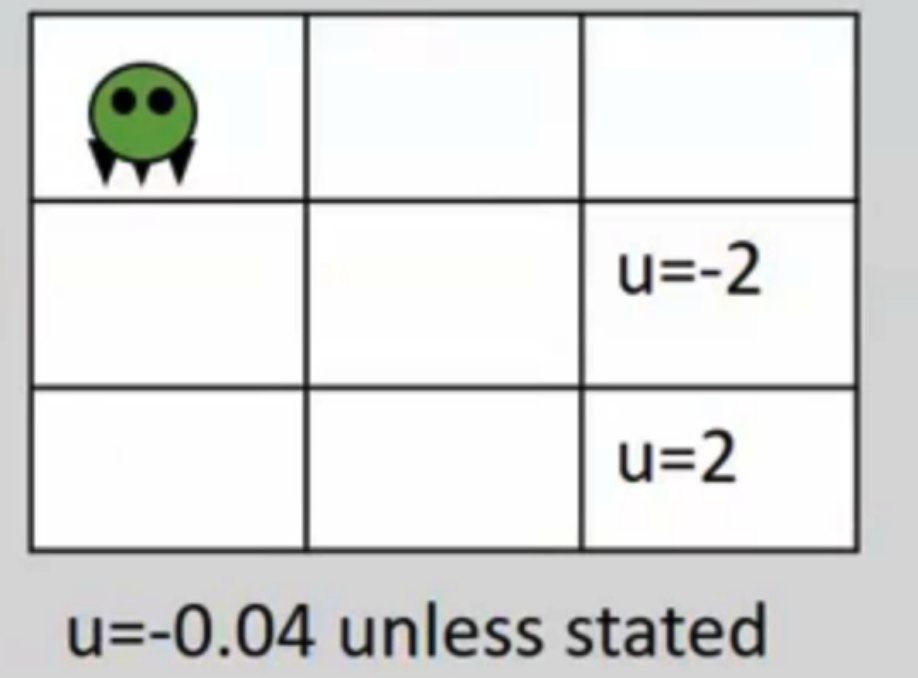
We want agents to carry out tasks for us autonomously, and utility can help them nudge in the correct direction. It is a payoff for the agent.

Utility can be assigned to individual states via the Utility function, taking in a state of the environment and mapping it to a real number.

$$u : E \to \mathbb{R}$$

The problem is this cannot take a long view, like a case when something gets worse before it gets better, making agents take a greedy approach.

For example, let's try to maximise utility in this system



The lower utilities in most of the map are used to force the agent towards the desired states.

In order to calculate the utility of a run we must add up all the utilities in it. For example, for a run that's south-south-east-east, it would be

$$(3 * -0.04) + 2 = 1.88$$

For a run that's west-west-south-south, it would be

$$(2 * -0.04) - 2 + 2 = -0.08$$

Since the first option gives us a better utility, we would prefer to take that route.

As we can see, taking the correct route to get somewhere is more important than the place we want to get to.

▼ Utility on runs

We can also assign utility functions to entire runs, allowing for a long term view.

$$u : \mathcal{R} \to \mathbb{R}$$

▼ For example

There are fruits that can appear and disappear randomly in Tileworld. An agent moves towards the fruit and tries to get it before it disappears. The utility of the run is

$$u(r) = \frac{num\ fruits\ collected\ in\ r}{num\ fruits\ appeared\ in\ r}$$

So utility will be higher the more fruits collected, with 1 if all fruits are collected and 0 if no fruits are collected.

▼ Probability of run occurring

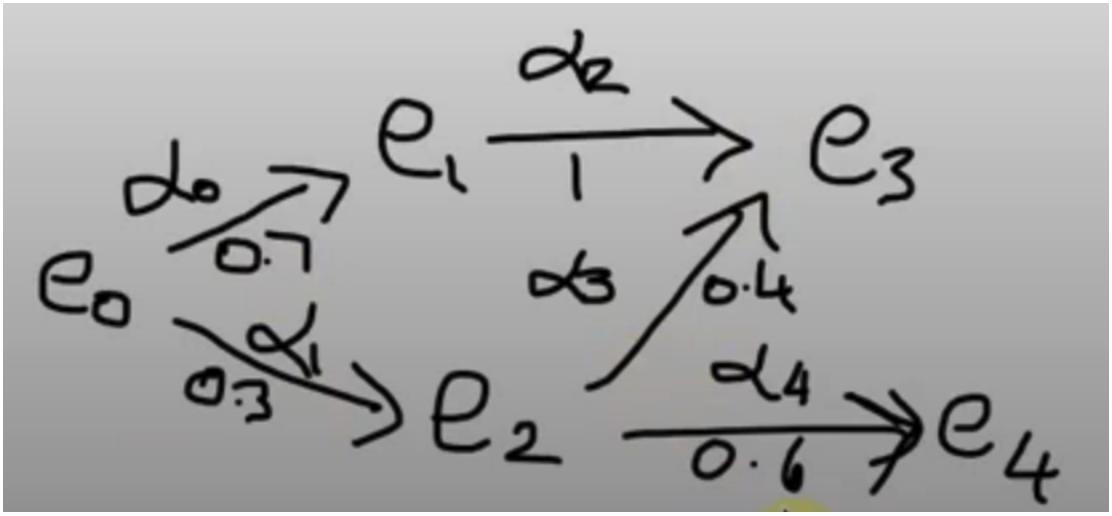
The probability of a run occurring in an environment is expressed as

$$P(r|Ag, Env)$$

To calculate it for a non-deterministic environment, we multiply the probability of each action occurring.

$$For\ run\ r = (e_0, \alpha_0, e_1, \alpha_1, e_2\alpha_2, ...) : P(r|Ag, Env) = P(e_1|e_0, \alpha_0)P(e_2|e_1, \alpha_1)...$$

For example, in the system



the probability of run

$$e_0 \rightarrow^{\alpha_1}_{0.3} e_2 \rightarrow^{\alpha_3}_{0.4} e_3$$

is calculated as

$$P((e_0, \alpha_1, e_2, \alpha_3, e_3)|Ag, Env) = 0.3 * 0.4 = 0.12$$

▼ Expected utility for an agent

To calculate the utility we expect an agent to have (EU), we multiply the utility of each run against the probability of each run occurring in the environment, and sum them all.

$$EU = \sum_{\forall r \in \mathcal{R}} u(r)P(r|Ag, Env)$$

▼ For example

$$U(e_0 \rightarrow e_1 \rightarrow e_3) = 2(0.7 * 1) = 1.4$$

$$U(e_0 \rightarrow e_2 \rightarrow e_3) = 1(0.3 * 0.4) = 0.12$$

$$U(e_0 \rightarrow e_2 \rightarrow e_4) = 3(0.3 * 0.6) = 0.54$$

$$EU = 1.4 + 0.12 + 0.54$$

This is only how much we expect an agent is going to get on average, but we can compare it against other agents' EU to get an idea of which one is better.

▼ Optimal Agent

An optimal agent is the agent within the environment that gets the biggest expected utility

$$Ag_{opt} = ag \max_{Ag \in \mathcal{AG}} EU(Ag, Env)$$

▼ Bounded Agent

Not all agents can be implemented on a machine "m", since machines have finite resources. A bounded agent is an agent that can be implemented on machine "m".

$$\mathcal{AG}_m = \{Ag | Ag \in \mathcal{AG} \wedge Ag \; can \; be \; implemented \; on \; m\}$$

▼ Tasks

▼ Specifying tasks

Utility is one way of telling an agent what it should do (specifying the task) by maximising it, but it is not the only way:

▼ Predicate Task Specification

It's a special type of utility which acts as a predicate over runs, in which runs either success or fail. It is represented by

$$\Psi : \mathcal{R} \rightarrow \{0, 1\}$$

A Task Environment is the pairing of a task and a predicate task specification

$$\langle Env, \Psi \rangle$$

With the environment we have the states, initial states and behaviour of the states, but Psi adds a criteria for judgement of wether or not an agent has done a good job or not.

If we have multiple Task Environments, the set of all task environments is represented as

$$\mathcal{TE}$$

We check if a run is successful with

$$\mathcal{R}_\Psi(Ag, Env) = \{r | r \in \mathcal{R}(Ag, Env) \wedge \Psi(r)\}$$

This is a possible run that exists and satisfies the predicate task specification.

There are two possible ways of populating this:

- Every possible run has to satisfy the PTS

$$\mathcal{R}_\Psi(Ag, Env) = \mathcal{R}(Ag, Env)$$

$$\forall r \in \mathcal{R}(Ag, Env) : \Psi(r)$$

- An agent belongs to "cal PSI" f there exists just one run that satisfies the PTS

$$\exists r \in \mathcal{R}(Ag, Env) : \Psi(r)$$

▼ Probability of success

To know how likely an agent is to succeed in an environment we sum the probability of occurring of every successful run

$$P(\Psi | Ag, Env) = \sum_{\forall r \in \mathcal{R}_\Psi(Ag, Env)} P(r | Ag, Env)$$

▼ Achievement Goals

We want to achieve a particular state of affairs, we don't care how.

We want to force the environment to go into some of the Goal States

$$\mathcal{G} \subseteq E$$

If the run brings about something in G, the PTA judges it to be a good run, or a bad run if it fails.

We say an agent is successful in an achievement task if it is guaranteed to force the environment into a Goal State.

We can write an achievement task like

$$\langle Env, \mathcal{G} \rangle$$

▼ Maintenance Goals

We want to maintain a state of affairs.

We have a set of Bad States which are to be avoided

$$B \subseteq E$$

For example, "we don't want to let the dog out".

If a run is ever in B, the PTA will consider it a bad run.

$$\Psi(r) = 0$$

If it is never in B, the PTA will say it is a good run.

$$\Psi(r) = 1$$

Like before, an agent is going to succeed if they are absolutely guaranteed to avoid every bad state.

A Maintenance Task can be written as

$$\langle Env, B \rangle$$