G54DIA: Designing Intelligent Agents

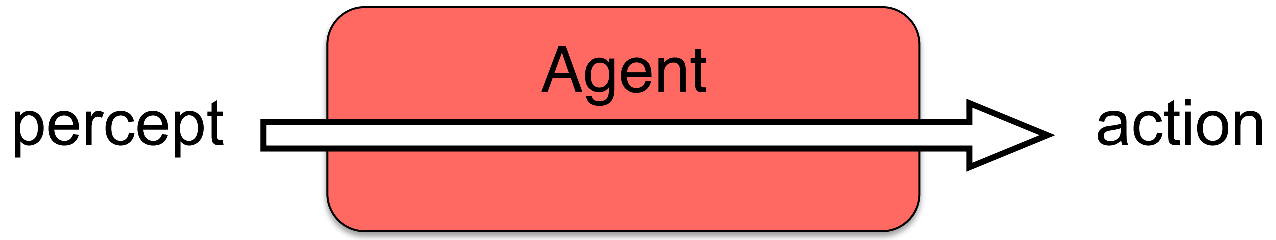
Lecture 4: Reactive Architectures II

Natasha Alechina
School of Computer Science
nza@cs.nott.ac.uk

Outline of this lecture

- limitations of simple reactive systems
- more complex reactive systems
  - with state
  - with multiple components
- example: '*Avoid Past*' behaviour
- example: *Subsumption* architecture

Simple reactive architectures

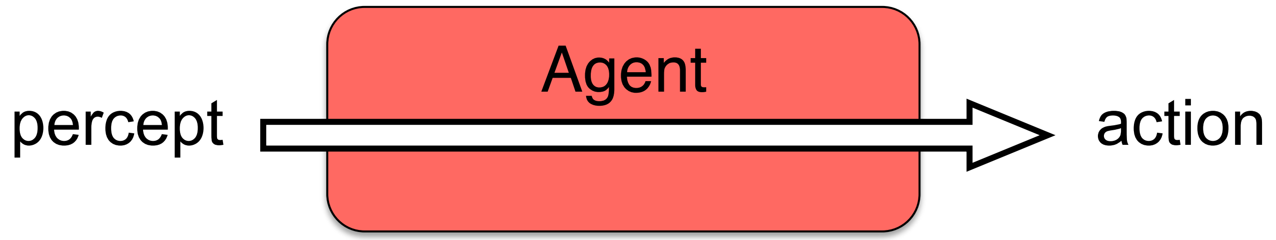


- *actions* are directly triggered by *percepts*
  - no representations of the environment
  - predefined, fixed response to a situation
  - fast response to changes in the environment

Limitations of a simple reactive agent

- its knowledge of the world is limited by the range of its sensors
- it’s unable to count
- it’s unable to recover from actions which fail silently
- and many others ...

Modelling reactive behaviours

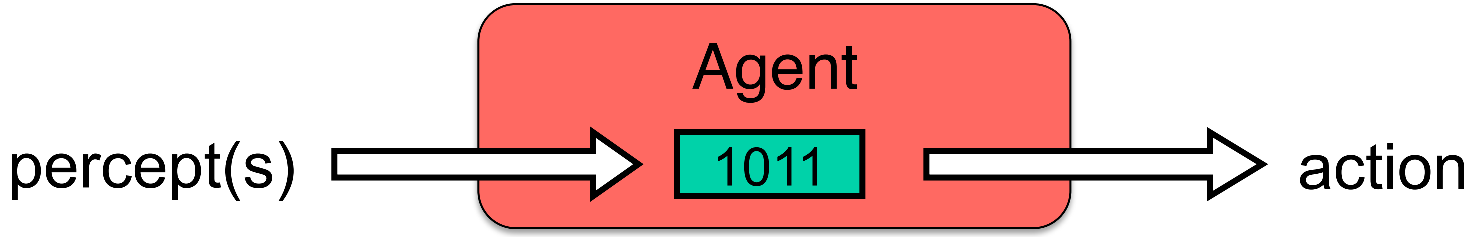


- we can model reactive behaviours as *condition-action rules*
- if the *condition* matches the agent's precepts, it triggers an *action*

```
if percept then action
```

- a simple reactive agent maintains no internal representation of the state of the world, whether the rule has been fired before etc.

Reactive architectures with state



- some rules match against an *internal representation* of aspects of the environment

- representations can be built using simple percept-driven rules (internal actions) which record simple '*beliefs*' about the state of the world

Reactive architectures with state

- from the point of view of expressive power, this adds nothing
- we have simply taken a condition-action rule which matched against a percept and generated an action and split it in two, with a mediating internal representation
- needs extra machinery to store the state
- requires at least two computation steps to choose an action rather than one

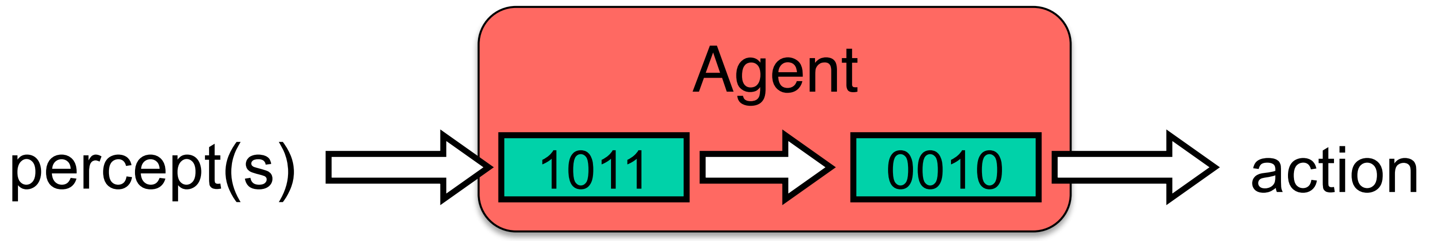
Action selection function

- the action selection function for a simple reactive agent looks like

$$\mathit{selectAction} : \mathit{Event} \times \mathit{State} \rightarrow \mathit{Action} \times \mathit{State}$$

- a reactive agent with state (finite-state machine) can respond to regular sequences of events
- we can add more complex state data structures to increase the capabilities of the agent, e.g.,
  - add a stack (pushdown automata) to respond to context-free sequences
  - add a random-access array to get a Turing machine, etc.

Actions which modify the internal state

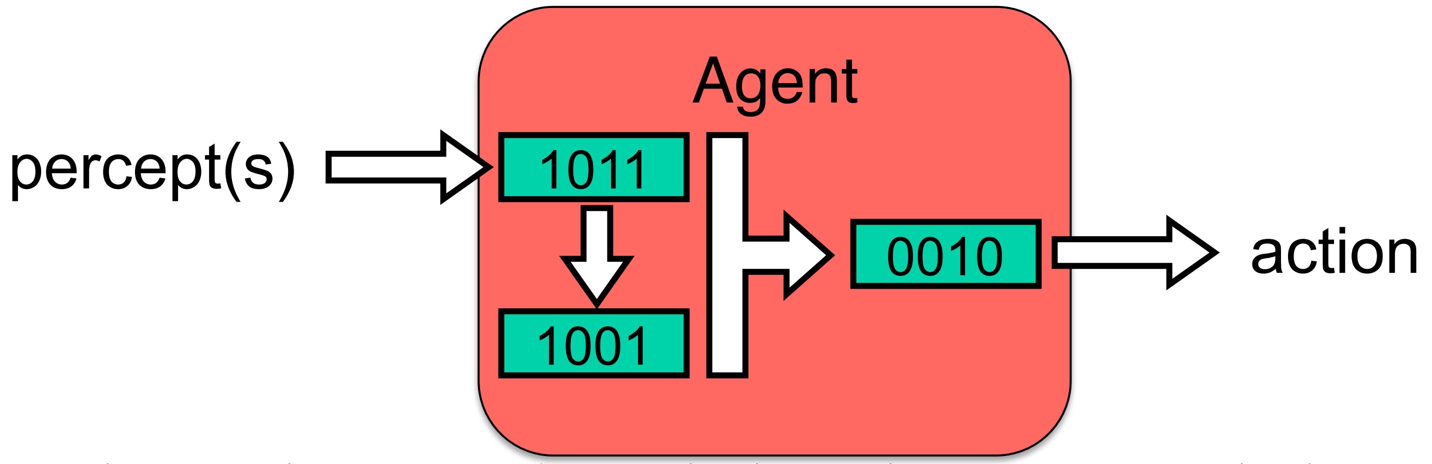


- we can extend this to rules
  - whose conditions match against the agent's internal state; and
  - whose actions modify the agent's internal state
- again, this appears to make things worse—requires even more space and more steps to choose an action

Importance of representations

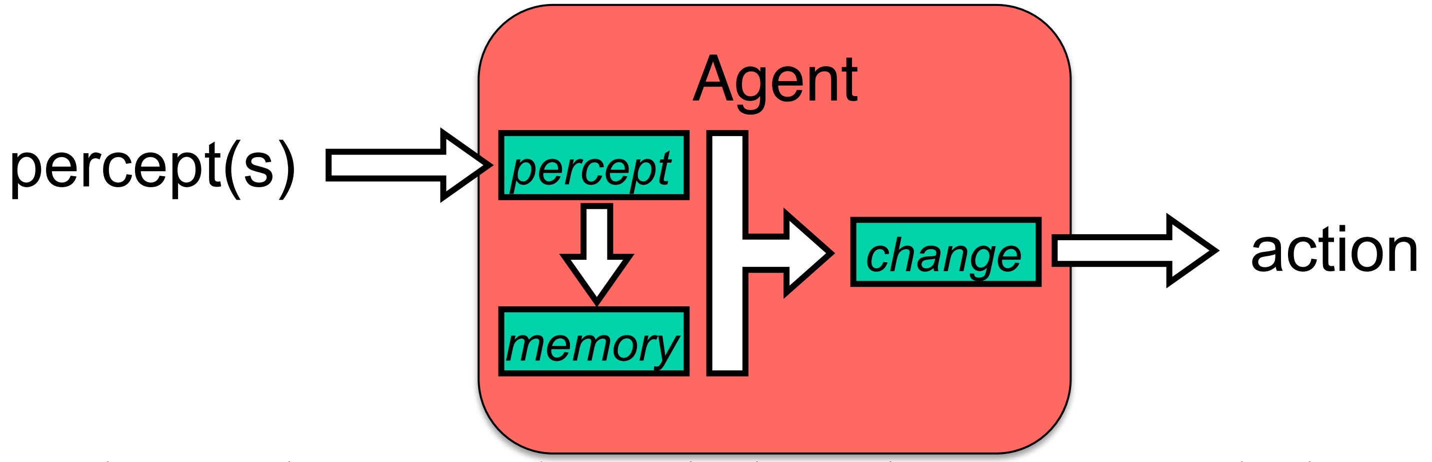
- notion of a rule which only responds to and generates *internal* changes in the agent is a key step
- forms the basis of all derived representations, and of representations which refer to other aspects of the agent's internal state
- e.g., allows the agent to respond only to *changes* in the environment, ignoring features that are constant
- without some representation of the previous state, we can't say what is novel in the current state

Detecting change



- to detect and represent *changes* in the environment we need rules
  - whose conditions match against representations of the current and previous precepts
  - whose action is to *remember* (copy) the state representing the current percept for use at the next cycle

Detecting change



- to detect and represent *changes* in the environment we need rules
  – whose conditions match against representations of the current and previous precepts
  – whose action is to *remember* (copy) the state representing the current percept for use at the next cycle

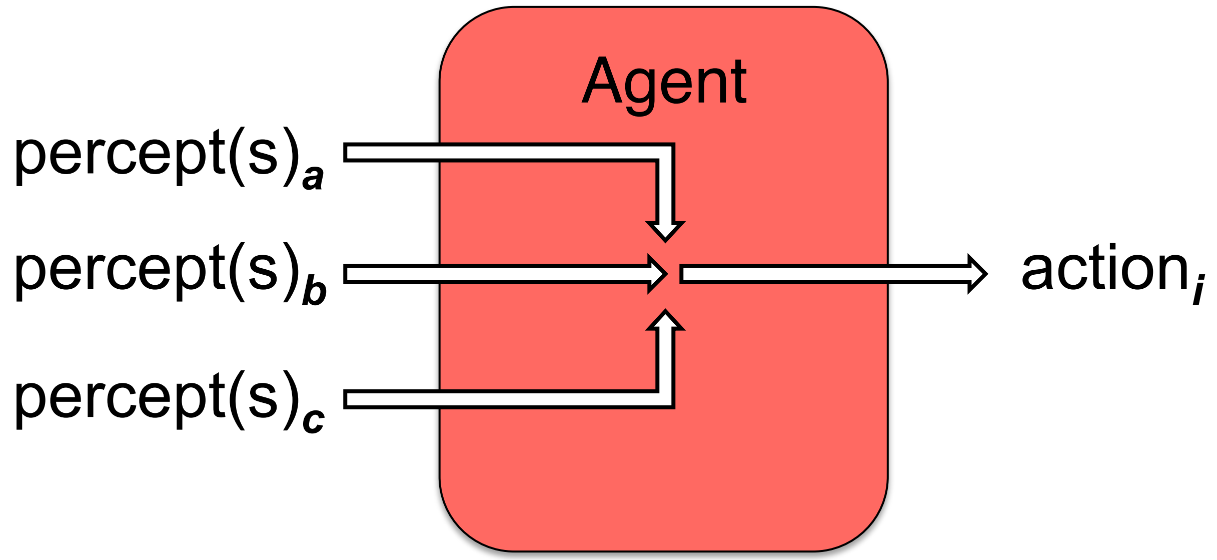
Internal representations

- require more space and incur the cost of maintaining the representation
- allow the choice of actions based on *sequences of states*, e.g.:
  - to react to change
  - to react to *lack of* change
- given such internal behaviours, much more complex external behaviours are possible

Advantages of state

- agent's knowledge of the world is no longer limited by the range of its sensors–it can remember parts of the environment it can't currently sense
- agent is able to count–allowing it to execute behaviours that require some action to be iterated a given number of times
- agent is able to recover from actions which fail silently–it can remember which actions it has tried before or how many times an action has been tried, and try something else if the action doesn't have the desired effect

Combined actions



- distinct actions triggered by different percepts are *combined* into a single composite action

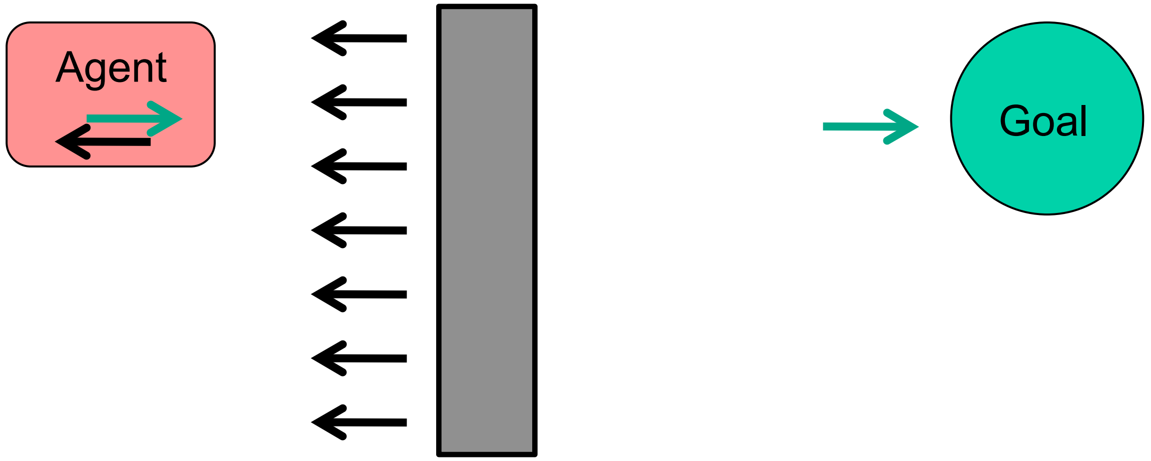
Problems with combining actions

- action selection based on combining actions is prone to a number of problems:
  - local minima: e.g., Braitenberg vehicles get suck in a corner of a box, unable to turn around; and
  - cyclic behaviour: e.g., Braitenberg vehicles get trapped orbiting an obstacle
- one way to solve these problems is simply to inject *noise* or *randomness* into the agent's behaviours

Local minima

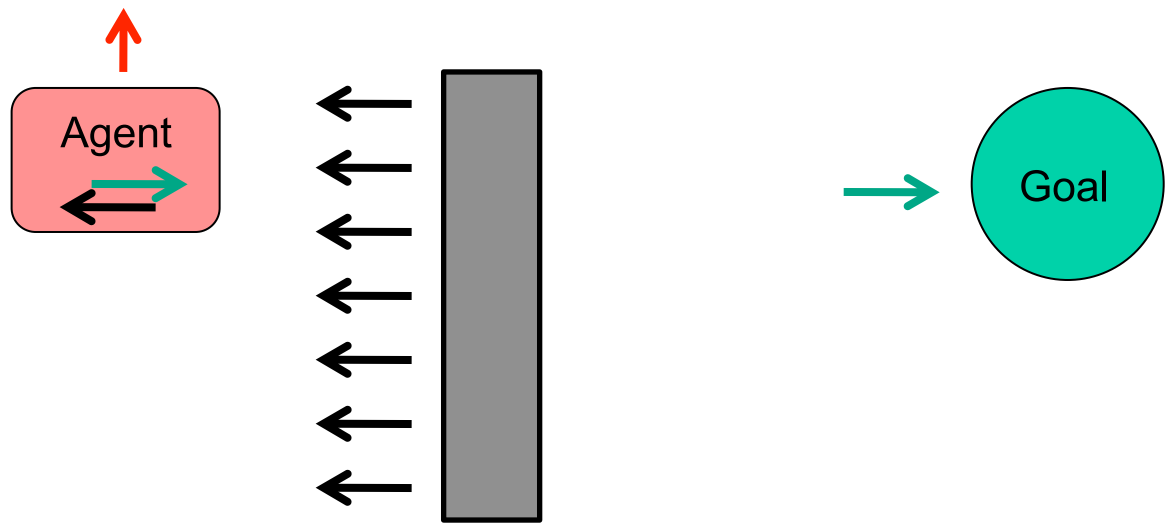
- consider a simple ‘boids-like’ agent which chooses an action by combining
  - a vector towards the goal
  - a vector away from an obstacle (if any)
- if there is an obstacle on the way to the goal, the agent can get stuck

Example: local minima



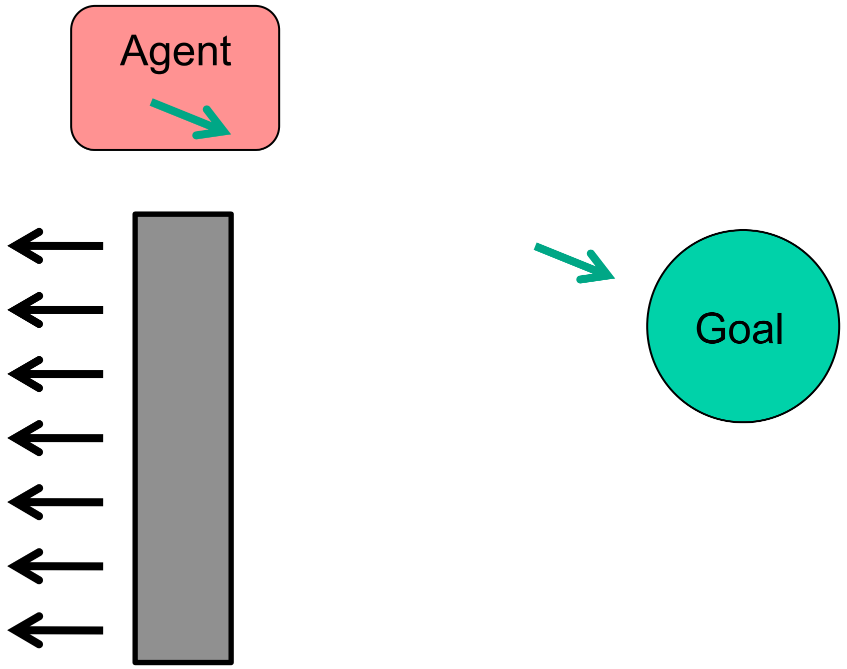
- vectors sum to zero and the agent's combined action is nil

Example: local minima



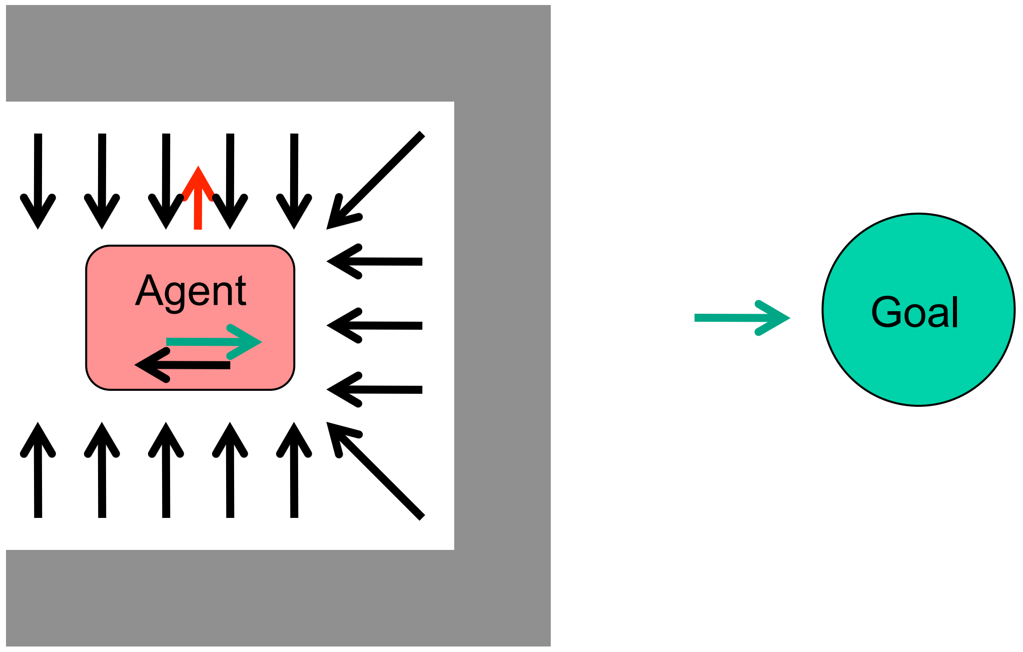
- adding a (small) random vector can take the agent past the obstacle, “unsticking” it

Example: local minima



- adding a (small) random vector can take the agent past the obstacle, “unsticking” it

Local minima example

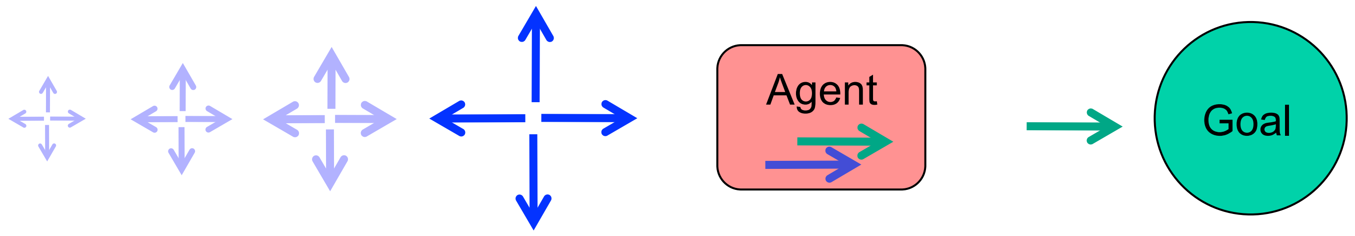


- however for some obstacles, this doesn’t work

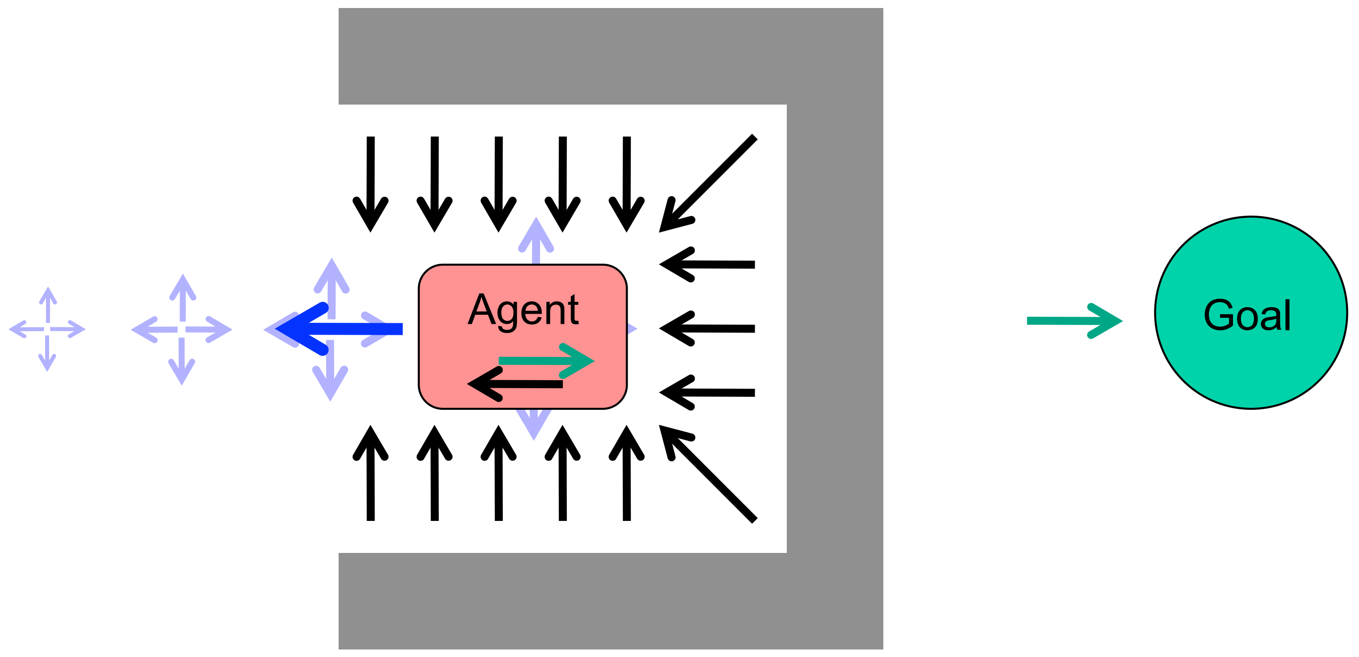
Example: the ‘Avoid Past’ behaviour

- another solution involves the use of memory
- *Avoid Past* behaviour uses a short term representation of where the agent has been recently
- *repulsive* forces are generated from recently visited areas
- the output of the Avoid Past behaviour is a vector of the same form as the vectors produced by the other behaviours (e.g., move to goal, avoid obstacles etc.) and is combined in the same way
- memory used by Avoid Past is short term—the agent can return to previously visited locations when the memory decays

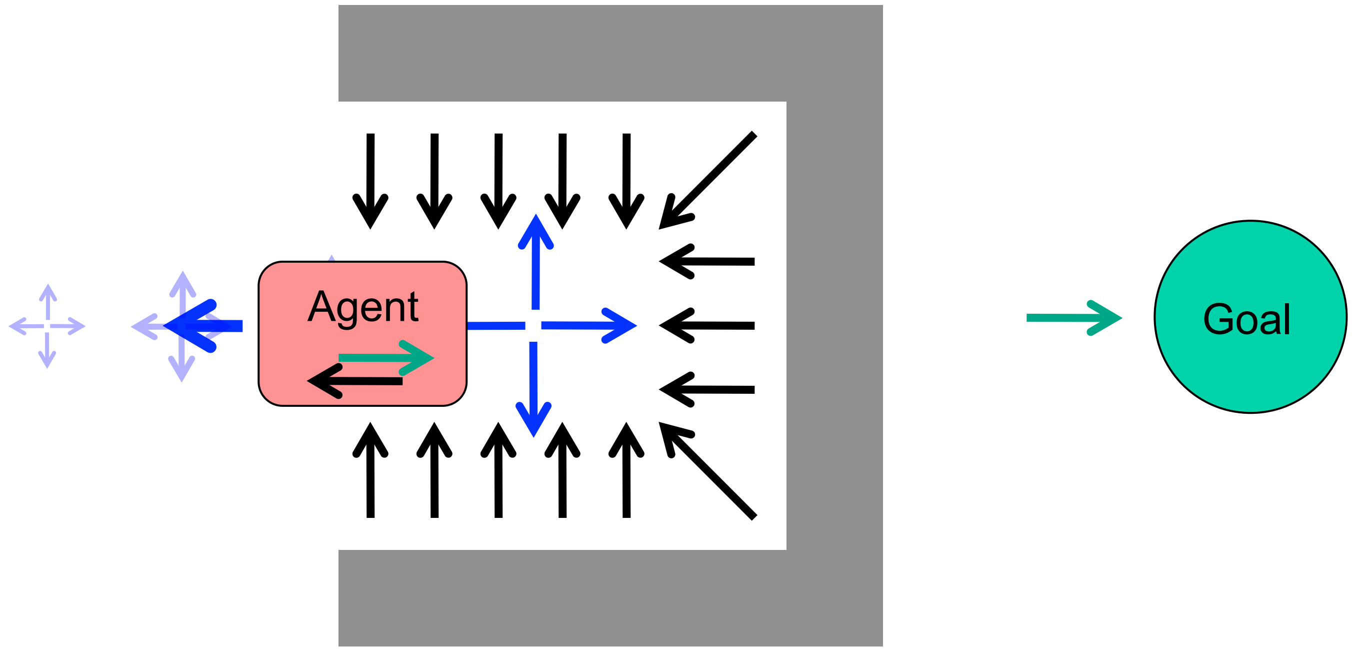
Example: the Avoid Past behaviour



Example: the Avoid Past behaviour

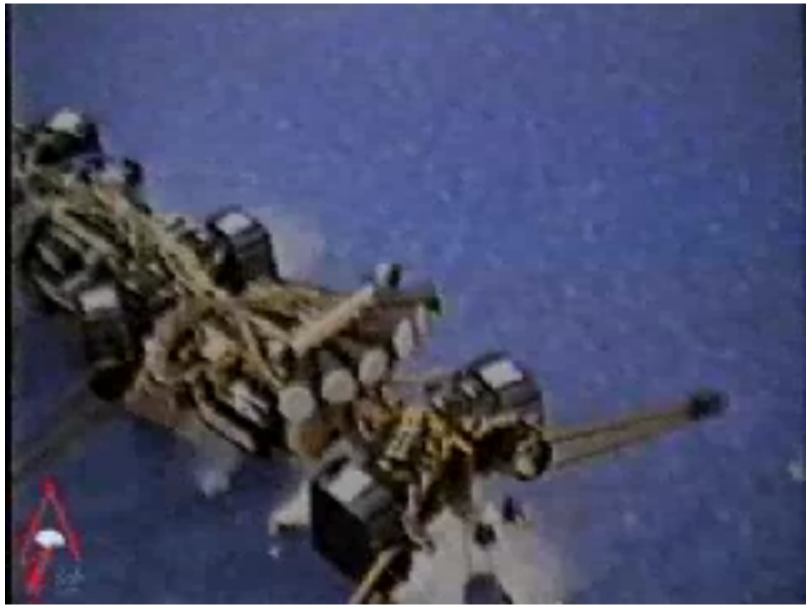


Example: the Avoid Past behaviour

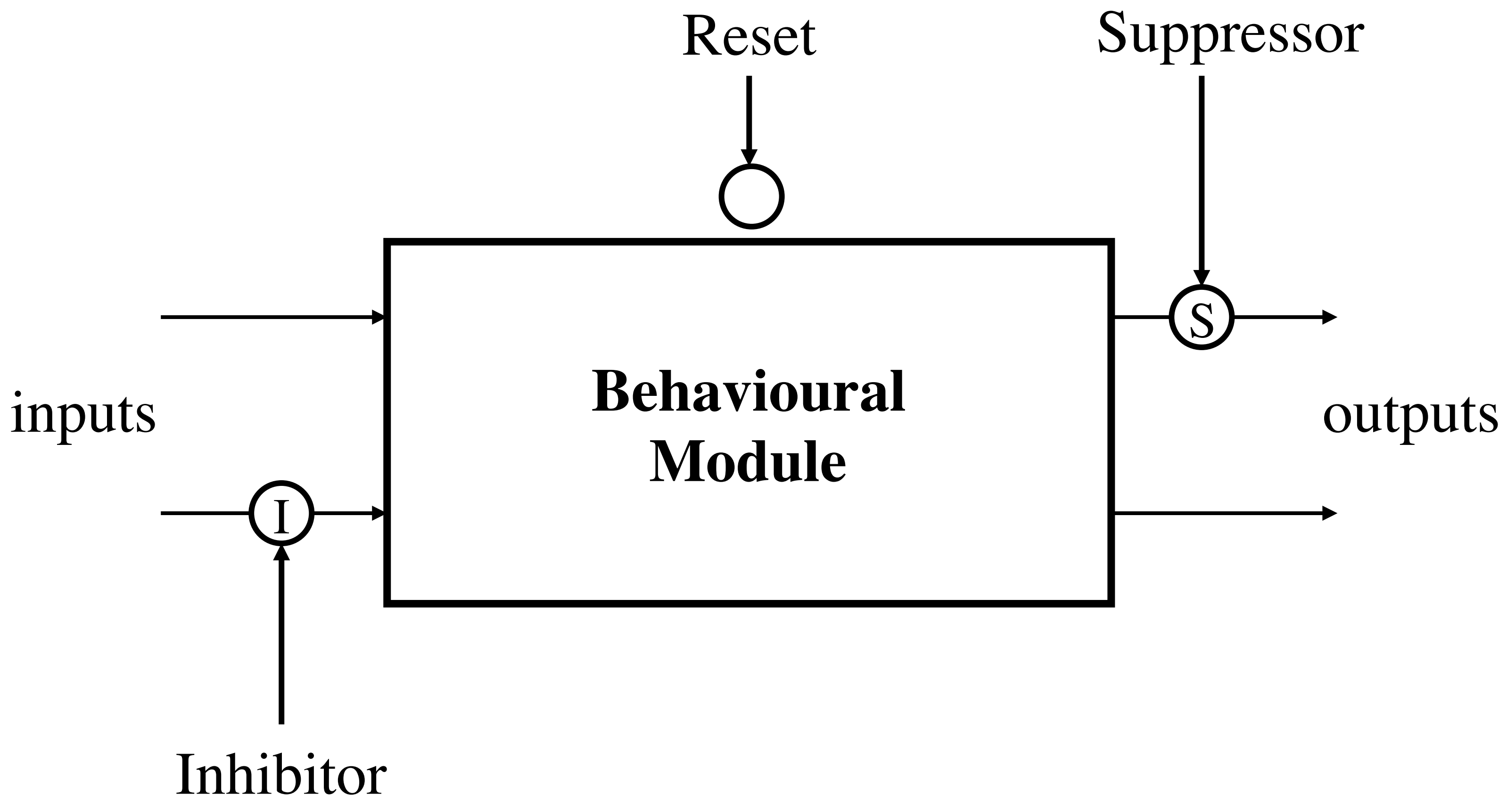


Example: Subsumption architecture

- collection of *behaviours* specified as a set of rules in Behavior Language
- behaviours compile to an Augmented Finite State Machine (AFSM)
- each AFSM performs an action and is responsible for its own perception of the world
- the output of one behaviour can form the input to another



Behavioural module



Subsumption architecture layers

- behaviours are organised into *layers*, each of which is responsible for independently achieving a goal
- complex actions *subsume* simpler behaviours lower in the hierarchy
- output of lower layers can be read by higher layers
- lower layers have no knowledge of higher layers
- layers operate concurrently and asynchronously

Subsumption architecture layers

- higher layers control lower layers using:
  - **inhibition**: prevents transmission
  - **suppression**: replaces a message with a suppressing message
  - **reset**: restores behaviour to its original state

Subsumption example: Foraging 1

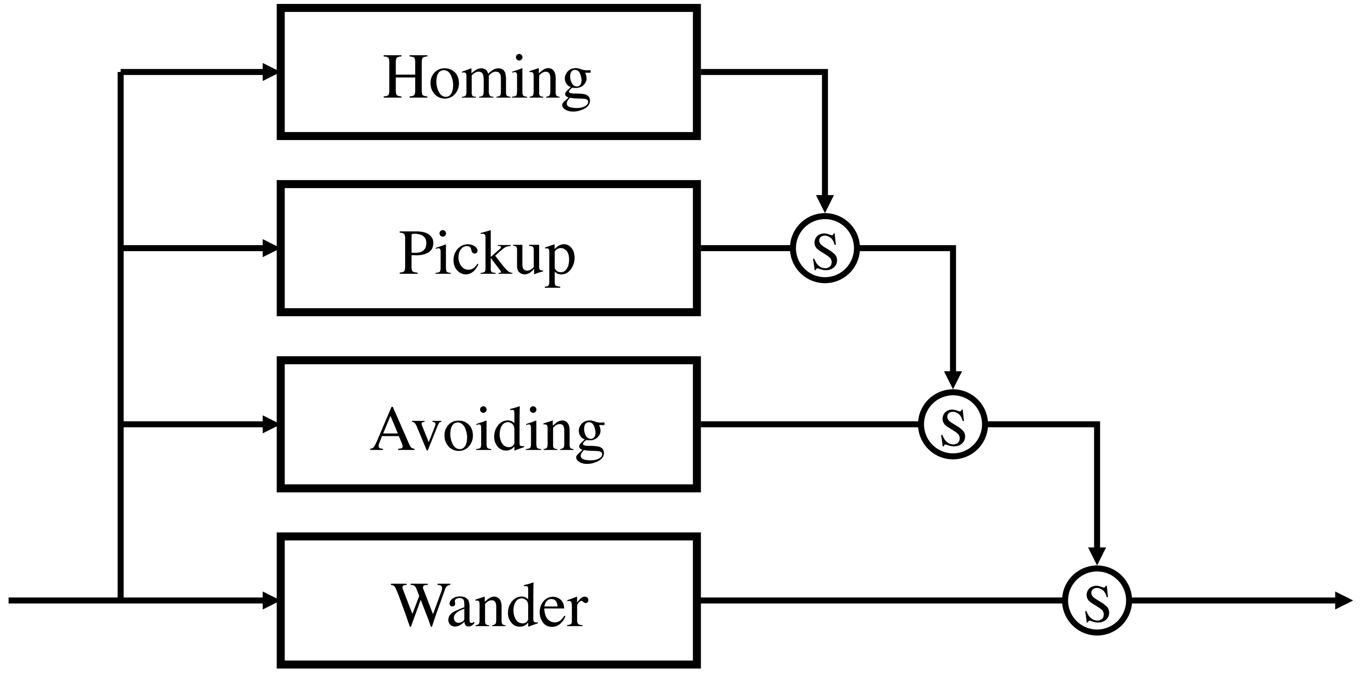
- *foraging task* consists of a robot moving away from a home base looking for attractor objects
- when it detects an object, the robot moves towards it, picks it up and returns to home base
- this sequence of actions is repeated until it has returned all the objects in the environment

Subsumption example: Foraging 2

Mataric (1993) implemented a simple robotic foraging system using 4 behaviours:

- **wandering:** move in a random direction for some time
- **avoiding:**
  - turn to the [right | left] if the obstacle is on the [left | right], then go
  - after *three attempts*, back up and turn
  - if there is an obstacle on both sides, randomly turn and back up
- **pickup:** turn towards the object and go forward; if at the object, close gripper
- **homing**: turn towards home and go forward, otherwise, if at home, stop

Subsumption example: Foraging 3



Subsumption example: Foraging 4

- behaviours are *prioritised* and the robot is executing only *one* behaviour at any one time
- when the robot senses an obstacle, *wandering* is suppressed, so that the *avoidance* behaviour can get the robot away from the obstacle
- when the robot senses the target object, collision *avoidance* is suppressed—otherwise the *pickup* behaviour couldn't get the robot close enough to the object to pick it up
- when the object is grabbed, homing then suppresses *pickup* (allowing the robot to ignore the potential distraction of of other objects it might encounter on its way back to base)

Advantages of reactive architectures

- a reactive architecture with state can produce *any* kind of behaviour
- if the state can be partitioned (e.g., subsumption architecture)
  - development is easier
  - can still use dedicated, parallel hardware
  - fast (real-time) response to changes in the environment

Disadvantages of reactive architectures

- no complex problem solving
- even with the addition of state and multiple components, reactive architectures can't consider alternative plans/solutions to a problem (states are veridical)
- every solution to every problem must be coded in advance, either by the designer of the system, or by evolution
- can't cope with novel situations for which they don't have a predefined behaviour
- agent programs for complex problems can be *very* large

The next lecture

Deliberative Architectures I

Suggested reading:

- Russell & Norvig (2003), chapter 11
- Wooldridge (2002), chapter 4