COSC343: Artificial Intelligence

Lecture 2: Starting from scratch: robotics and embodied AI

Alistair Knott

Dept. of Computer Science, University of Otago

Embodied AI: starting at the beginning

How hard is it to build something like us?

One way of measuring the difficulty of a task is to look at how long evolution took to discover a solution.

Evolving plants from single-cell organisms	1 billion years
Evolving fish/vertebrates from plants	1.5 billion years
Evolving mammals	300 million years
Evolving primates	130 million years
Evolving the ancestors of great apes	100 million years
Evolving homo sapiens	15.5 million years

- Most of evolutionary time was spent designing robust systems for physical survival.
- In this light, what's 'distinctively human' seems relatively unimportant!

An evolutionary approach to AI

Proponents of embodied AI believe that in order to reproduce human intelligence, we need to retrace an evolutionary process:

- We should begin by building robust systems that perform very simple tasks—but *in the real world*.
- When we've solved this problem, we can progressively add new functionality to our systems, to allow them to perform more complex tasks.
- At every stage, we need to have a robust, working real-world system.

Maybe systems of this kind will provide a good framework on which to implement distinctively human capabilities.

This line of thought is associated with a researcher called Rodney Brooks.

Beginning with some simple agents

Some AI researchers are interested in modelling simple biological organisms like cockroaches, woodlice, etc.
These creatures are basically reflex agents (c.f. Lecture 1):

- They sense the world, and their actions are linked directly to what they sense.
- They don't have any internal representations of the world.
- They don't do any 'reasoning'.
- They don't do any 'planning'.

Simple reflex agents

Recall: in Lecture 1, you heard about the agent function, which maps from percept histories to actions:

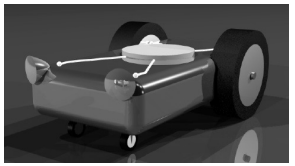
$$f : \mathcal{P}^* \to \mathcal{A}$$

For a simple reflex agent, this function is much simpler: it just maps the *current percept* to actions.

$$f : \mathcal{P} \to \mathcal{A}$$

Braitenberg Vehicles

One very simple simulated organism is a Braitenberg vehicle.



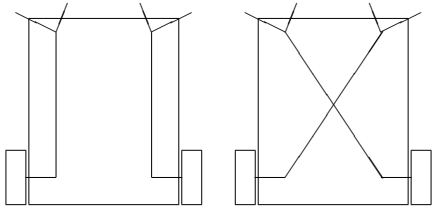
The discs on the front are light sensors.

- Lots of light → strong signal.
- Less light → weaker signal.

The sensors are connected to motors driving the wheels.

Braitenberg Vehicles

Two initial configurations:



A : Same side connections
B : Cross connection

From these simple connections can you work out the behaviour?

Perception to Action

These simple agents blur the lines between perception and action.

Perception processes that are used to interpret the environment of an agent. These are processes that turn stimulation into meaningful features.

Action any process that changes the environment. (Including the position of the agent *in* the environment!)

Actually, the lines are blurred in human agents too.

Simple Agents—Complex behaviour

Often you can get complex behaviour emerging from a simple action function being executed in a complex *environment*.

If an agent is working in the real world, its behaviour is a result of *interactions* between its agent function and the environment it is in.

We can distinguish between

- Algorithmic Complexity
- Behavioural Complexity

Emergent Behaviour

Behaviours which result from the interaction between the agent function and the environment can be termed emergent behaviours.

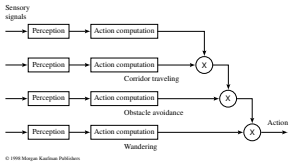
Some particularly interesting emergent behaviours occur when several agents are placed in the same environment.

It's very hard to experiment with emergent behaviours except by building simulations and seeing how they work.

Often even simulations are not enough - you need to build real robots.

The subsumption architecture

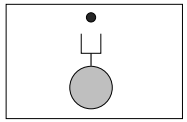
Rodney Brooks developed an influential model of the architecture of a reflex agent function. The basic idea is that there are lots of *separate* reflex functions, built on top of one another.



- Each agent function is called a behaviour.
- Behaviours all run concurrently. E.g. 'wander around', 'avoid obstacle'.
- One behaviour's output can *override* that of less urgent behaviours.

An example of the subsumption architecture

Say you're building a robot whose task is to grab any drink can it sees.



- Behaviour 1: move around the environment at random.
- Behaviour 2: if you bump into an obstacle, inhibit Behaviour 1, and avoid the obstacle.
- Behaviour 3: if you see a drink can directly ahead of you, inhibit Behaviour 1, and extend your arm.
- Behaviour 4: if something appears between your fingers, inhibit Behaviour 3 and execute a grasp action.

Embodied AI: the importance of practical work

Roboticists stress the importance of working in the physical world.

- It's very hard to *simulate* all the relevant aspects of a robot's physical environment.
- The physical world is far harder to work with than we might think—there are always unexpected problems.
- There are also often unexpected benefits from working with real physical systems.

So we're going to do some practical work.

The LEGO Mindstorms project

Researchers at the MIT Media Lab have been using LEGO for prototyping robotic systems for a long time.

- One of these projects led to a collaboration with LEGO, which resulted in the 'Mindstorms' LEGO kit.

Mindstorms is now a very popular product, which is used by many schools and universities, and comes with a range of different operating systems and programming languages.

We're using the third generation of Mindstorms, called EV3.

Mindstorms EV3 components

As well as a lot of different ordinary LEGO pieces, the EV3 kit comes with some special ones.

1. The EV3 brick: a microcontroller, with four inputs and three outputs.
   - The heart of the EV3 is a 32-bit ARM microcontroller chip. This has a CPU, ROM, RAM and I/O routines.
   - To run, the chip needs an *operating system*. This is known as its firmware.
   - The chip also needs a *program* for the O/S to run. The program needs to be given in bytecode (one step up from assembler).
   - Both the firmware and the bytecode are downloaded onto the NXT from an ordinary computer, via a USB link.

Mindstorms components

2. A number of different sensors.
   - Two touch sensors: basically on-off buttons.
     These are often connected to whisker sensors.
   - A colour sensor detects light intensity in 3 wavelengths (RGB).
     - This can be used to pick up either ambient light, or to detect the reflectance of a (close) object.
     - The sensor shines a beam of light outwards. If there's a close object, it picks up the light reflected off the object in question.
   - A sonar sensor, which calculates the distance of surfaces in front of it.
   - A microphone, which records sound intensity.

Mindstorms components

3. Two servomotors (actuators).
   - The motors all deliver *rotational* force (i.e. torque).
   - Motors can be programmed to turn at particular speeds, either forwards or backwards.
   - They can also be programmed to lock, or to freewheel.
   - Servomotors come with rotation sensors. So they can be programmed to rotate a precise angle and then stop.

A simple mobile robot

The robots you will be working with all have the same design.

- They're navigational robots - i.e. they move around in their environment.
- They are turtles: they have a chassis with two separately controllable wheels at the front, and a pivot wheel at the back.
- They have a light sensor underneath, for sensing the 'colour' of the ground at this point.
- They have two whisker sensors on the front, for sensing contact with objects 'to the left' and 'to the right'.
- They also have a microphone and a sonar device (but there's only space to plug in one of these at a time).

What commands would you need to give to a turtle to make it turn?

The ROBOTC programming language

We'll be programming the robots using a language based on C, called ROBOTC. Here's a simple example program.

```
task main()
{
 motor[motorC] = 100;  // Start Motor C running at power 100
 motor[motorB] = 100;  // Start Motor B running at power 100
 wait1Msec(4000);      // Wait 4000 milliseconds

 motor[motorC] = -100; // Start Motor C running backwards
 motor[motorB] = -100; // Start Motor B running backwards
 wait1Msec(4000);      // Wait 4000 milliseconds
}
```


NXC program development and execution

Program development cycle:

- Boot up on Windows.
- Turn the robot on, and connect it to your machine's USB port.
- Start the ROBOTC app.
- Write a program (in the top panel).
- Then hit 'Compile Program'. (Errors appear in the bottom panel.)
- When it compilies cleanly, hit 'Download to Robot'.

See the 343 web page for much more info.

The EV3 brick control panel

The EV3 brick has an LCD display, and a few buttons to navigate a hierarchical menu.

- Hit the right button, and then select 'rc' to get a listing of all ROBOTC programs on the robot.
- When you select a program, it will run. *Make sure the robot's not on the edge of a table when you do this!*
- To abort the program, hit the top-left button.

Synched motors and servomotors

```
task main()
{

  nSyncedMotors = synchBC; // Motor B is 'master', C is 'slave'
  nSyncedTurnRatio = 100;  // motors move with 100% alignment

  nMotorEncoder[motorB] = 0; // set the position of Motor B to 0

  while(nMotorEncoder[motorB] < 720) // run Motor B at 30% power,
  {                                  // until it reaches posn 720.
    motor[motorB] = 30;              // (Motor C is synched)
  }

  motor[motorB] = 0;        // turn both motors off

  wait1Msec(3000);      // Wait for the above routine to finish!

}
```


Threads in ROBOTC

The NXT supports multiple threads (called tasks).
Every program has to contain a task called `main`.
In this example, the `main` starts `TOne` and `TTwo`. `TTwo` kills `TOne` 1 if the bump sensor is pressed, and restarts `TOne` when the bump sensor is released.

```
task TOne();
task TTwo();
task main()
{
  startTask(TOne);
  startTask(TTwo);
  while(true)
  {
    wait1Msec(300); [then do something else, maybe]
  }
  return;
}
```


Threads in ROBOTC

```
task TOne()
{
  while(true)
  { wait1Msec(300); [then do something] }
  return;
}
task TTwo()
{
  while(true)
  {
    wait1Msec(300);
    while(SensorValue(touchSensor) == 1)
    {
      stopTask(TOne);
    }
    startTask(TOne);
  }
  return;
}
```

For more information about ROBOTC. . .

There are lots of sample programs in the programming environment. (‘file’ → ‘open sample program’.)

The 343 web page (LEGO resources) has lots more info, including:

- A user manual
- A useful API guide.

The challenge

The physical capabilities of a LEGO robot are very limited.

The challenge is: to program this simple robot to produce *complex behaviours*.

Summary

- Embodied AI: models the aspects of intelligence which relate to how an agent operates in the real world.
- In complex environments, simple systems can generate complex behaviours.
- The subsumption architecture is a useful architecture for embodied agents.
- We can study embodied agents using LEGO Mindstorms robots.

Reading

For this lecture: take a look at the info about LEGO robots on the COSC343 webpage.

For next lecture: AIMA Sections 18.1, 18.2