Introduction: definition of AI and agents

Lubica Benuskova

Reading: AIMA 2nd ed., Chapters 1 and 2

Textbook

- http://aima.cs.berkeley.edu/ has many useful materials (like pdf of some chapters, program codes, etc.).
- It is the leading textbook in AI, used in over 1200 universities in over 100 countries.

Podmienky absolvovania predmetu (UUI)

   1. Mgr. Peter Gergeľ, email: peter.gergeľ@fmph.uniba.sk
   2. Mgr. Juraj Holas, email: holas3@uniba.sk
3. Hodnotenie: cvičenia 30 %, projekty 20 %, skúška 50 %
5. Termíny: budeme sa držať striktne zákonom, skúšame len cez skúšobné, ako opravný termín máte právo využiť vypísané termíny.
6. Termín označený ako posledný bude naozaj posledný, žiadne dodatočné termíny po ňom už nebude.
7. Známky zapisujeme do indexu len v rámci na to určených zapisovacích termínoch.

What is Artificial Intelligence (AI)?

- Here is a fairly uncontroversial definition:
  - AI is the study and creation of machines that perform tasks normally associated with intelligence.
- People are interested in AI for several different reasons:
  - Cognitive scientists and psychologists are interested in finding out how people (i.e. our brains) work. Machine simulations can help with this task.
  - Engineers are interested in building machines which can do useful things. These things (often) require intelligence.
- The tasks to be performed could involve thinking, or acting, or some combination of these.

What is Intelligence?

- Intelligence has been defined in many different ways including logic, abstract thought, understanding, self-awareness, communication, learning, having emotional knowledge, planning, and problem solving (Wikipedia, 2014).
- Intelligence derives from the Latin verb intelligere, meaning to comprehend or perceive. A form of this verb, intellectus became the medieval technical term for understanding.
- Intelligence is a very general (mental) capability, through which organisms possess the abilities to learn, form concepts, understand, and reason, including the capacities to recognize and categorize patterns, comprehend ideas, make plans, solve problems, and use language to communicate.
An embodied 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 (Australian, living in USA).

Embodied AI: starting from 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 in nature took to discover a “solution”, i.e. us.

Disciplines contributing to AI

• Artificial Intelligence (AI) has origins in several disciplines some old, some recent:

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer Science</td>
<td>Alan Turing, John von Neumann,…</td>
</tr>
<tr>
<td>Mathematics</td>
<td>Notions of proof, algorithms, probability…</td>
</tr>
<tr>
<td>Robotics (cybernetics)</td>
<td>Construction of embodied agents</td>
</tr>
<tr>
<td>Economics</td>
<td>Formal theory of rational decisions</td>
</tr>
<tr>
<td>Neuroscience</td>
<td>Study of the brain structure and function</td>
</tr>
<tr>
<td>Experimental psychology</td>
<td>Models of human information processing</td>
</tr>
<tr>
<td>Linguistics</td>
<td>Noam Chomsky, language, concepts,…</td>
</tr>
<tr>
<td>Philosophy</td>
<td>‘Reasoning’: Aristotle, Boole,… ‘The mind’: Descartes, Locke, Hume,…</td>
</tr>
</tbody>
</table>

Agents and environments

• Recall our general definition of AI: **AI is the study and creation of machines that perform tasks normally associated with intelligence.**
• In this course we will use the term **agent** to refer to these machines.
  – The term agent is very general: it covers humans, robots, softbots, computer programs, vacuum cleaners, thermostats, etc.
  – An agent has a set of sensors, a set of actuators, and operates in an environment.

The agent function

• If we want, we can define an agent and its environment formally.
  – We can define a set of actions **A** which the agent can perform.
  – We can define a set of percepts **P** which the agent can receive.
  (Assume that there’s one percept per time point.)
• A simple agent function could simply map from percepts to actions:
  \[ f : P \rightarrow A \]
• A more complex (and general) agent function maps from percept histories to actions, thus allowing modelling an agent with a memory for previous experience:
  \[ f : P' \rightarrow A \]
• The agent program runs on the physical architecture to produce \( f \).

An example: the vacuum-cleaner world

• Here’s a simple example: a formal definition of a robot vacuum cleaner, operating in a two-room environment (room A, room B).
  – Its sensors tell it which room it’s in, and whether it’s clean or dirty.
  – It can do four actions: move left, move right, suck, do-nothing.

<table>
<thead>
<tr>
<th>Room A</th>
<th>Room B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Preliminaries for defining the agent function

- To define the agent function, we need a syntax for percepts and actions.
  - Assume percepts have the form [location; state]: e.g. [A; Dirty].
  - Assume the following four actions: Left, Right, Suck, NoOp.

- We also need a way of specifying the function itself.
  - Assume a simple lookup table, which lists percept histories in the first column, and actions in the second column.

<table>
<thead>
<tr>
<th>Percept history</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

A vacuum-cleaner agent

- Here is an example agent function for the vacuum cleaner agent:

<table>
<thead>
<tr>
<th>Percept history</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, Clean]</td>
<td>Right</td>
</tr>
<tr>
<td>[A, Dirty]</td>
<td>Suck</td>
</tr>
<tr>
<td>[B, Clean]</td>
<td>Left</td>
</tr>
<tr>
<td>[B, Dirty]</td>
<td>Suck</td>
</tr>
<tr>
<td>[A, Clean], [A, Clean]</td>
<td>Right</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

And here is an agent’s pseudo-code which implements this function:

```plaintext
function REFLEX-VACUUM-AGENT([location, status]) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

Evaluating the agent function

- It is useful to evaluate the agent function, to see how well it performs.

- To do this, we need to specify a performance measure, which is defined as a function of the agent’s environment over time.

- Some example performance measures:
  - one point per square cleaned up in time T?
  - one point per clean square per time step, minus one per move?
  - penalize for > k dirty squares
  - Etc.

Formalising the agent’s environment

- As well as a formal description of the agent, we can give a formal description of its environment.

- Environments vary along several different dimensions:
  - Fully observable vs partially observable
  - Deterministic versus stochastic (i.e. involving chance)
  - Episodic vs sequential
  - Offline vs online
  - Discrete vs continuous
  - Single-agent vs multi-agent

- The environment type largely determines the agent design.

Interim summary

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

- **Action** is any process that changes the environment – including the position of the agent in the environment!

- In order to specify a scenario in which an agent performs a certain task, we need to define the so-called PEAS description of the agent/task, i.e.:
  - a Performance measure;
  - an Environment;
  - a set of Actuators;
  - a set of Sensors;

Some different types of agents

- Just as we can identify different types of environment, we can identify different types of agents.

- In order of increasing complexity/adaptability we have:
  - Simple reflex agents
  - Model-based agents (or reflex agents with state)
  - Goal-based agents
  - Utility-based agents

- All these can be turned into learning agents (with which we will deal in the second half of the semester).
Simple reflex agents

- A simple reflex agent is one whose behaviour at each moment is a function of its sensory inputs at that moment
  - It has no memory for previous sensory inputs
  - It doesn’t maintain any state

- For example, here’s a simple reflex vacuum cleaner agent:

```plaintext
function REFLEX-VACUUM-AGENT(location, status) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

Simple reflex agents

- Some AI researchers are interested in modelling simple biological organisms like cockroaches, ants, etc.
- These creatures are basically simple 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’.
- Recall, the agent function for a simple reflex agent is the mapping from the current set of percepts to an action, i.e.
  \[ f : P \rightarrow A \]

Braitenberg vehicle

- 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 that can be in one of two modes: A – same side connections, B – cross connections.
- From these simple conditions can you work out the behaviour?

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
- It seems plausible to require that ‘intelligent’ behaviour be complex. However, this complexity could relate to the complexity of the environment and not the agent itself.

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.
The subsumption architecture

- Each agent function $f: P \rightarrow A$ is called a behaviour.
  - e.g., 'wander around', 'avoid obstacle' are all concrete behaviours.
- The subsumption architecture can be nicely motivated from biology.
  - New more sophisticated behaviours model new evolutionary developments of the organisms.
  - They sit on top of existing more primitive control systems.
- One behaviour's output can override that of less urgent behaviours.
  - For instance, people have got simple reflexes which override more complicated tasks they perform.

An example of the subsumption architecture

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

  ![Subsumption Architecture Example](image)

  - **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.

Interactions between behaviours and the world

- Notice how the 'reach' and 'grasp' actions interact with one another:
  - There's no notion of a 'planned sequence' of actions, where the robot first reaches its arm and then grasps: the 'reach' and 'grasp' behaviours are entirely separate.
  - However, the physical consequences of executing the 'reach' behaviour happen to trigger the 'grasp' behaviour.
    - The physical consequences of the 'reach' behaviour are due to:
      - the way the world is;
      - the way the robot's hardware is set up.
- Notice that if you just offered the robot a drink can, it would also behave appropriately (by grasping it).

Emergent behaviours

- 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.
  - The actions of each individual agent changes the environment which the other agents perceive;
  - So they potentially influence the behaviour of all the other agents.
  - Swarming and flocking are good examples.
- It's very hard to experiment with emergent behaviours except by building simulations and seeing how they work. Often even simulations are not enough – we need to build real robots.

Embodied AI: importance of practical work

- Robotics stress the importance of working in the physical world. It's not enough to develop a theory of how robots work, or to build simulations of robot in an environment; you actually have to build them.
  - 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 during labs with robots very similar to LEGO robots.

The LEGO Mindstorms project

- Researchers at the MIT Media Lab have been using LEGO for prototyping robotic systems for quite a long time.
  - One of these projects, headed by Fred Martin, led to a collaboration with LEGO, which resulted in the Mindstorms product.
  - The active collaboration between MIT researchers and LEGO continued for some years, and was quite productive.
- 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 now onto the second generation of Mindstorms, called NXT.
Mindstorms components 1

- As well as a lot of different ordinary LEGO pieces, the Mindstorms kit comes with some special ones.
- The NXT brick: a microcontroller, with four inputs and three outputs.
  - The heart of the NXT 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 light sensor detects light intensity.
  - 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

- Three 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. (As contrasted with e.g. with manipulatory robots.)
  - 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 NCX programming language

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

```c
int main() {
  SetSensorTouch(TL_3); //Define input 3 as a touch sensor
  OnFWD(OUT_L, 75); //Drive Motors B&C forward at 75% while (true) {
    if (SENSOR_3 == 1) { //If a bump is detected.
      OnRev(OUT_L, 75); Wait(300); //Full reverse for 300ms
      OnFWD(OUT_L, 75); Wait(1000); //Turn reverse for 300ms
      OnRev(OUT_L, 75); //Resume forward movement
    }
  }
}
```

The NCX program development and execution

- Program development cycle:
  - Turn the robot on, and connect it to your machine’s USB port.
  - Start the NXTCC app. (It’s in the Applications folder.)
  - Open the editor, write some NCX source code, save it.
  - Then compile the code. (‘Compile’ menu button in the editor)
  - When this compiles cleanly, it creates a bytecode file called [filename].sym.
  - When you’ve got it compiling cleanly, download the bytecode file to your robot. (‘Download’ menu button in the editor)
- See the http://www.otago.ac.nz/cosc343/lego.php for more info.
Threads in NCX

- Threads are called tasks.
- Every program has to contain a task called `main`. This is started when the program is executed. Threads can then be started and halted by program commands.
- Mutex variables are used to synchronise tasks.
- In the example, the main thread initiates two tasks, by placing them in a ready queue, and then exits.

NXC threads and the subsumption architecture

- Given that NXC is multithreaded, it’s possible to run several different behaviours (tasks) simultaneously.
- Each behaviour can be looking for input, and computing output.
- Behaviours can also trigger other behaviours, suppress their output, and so on.
- So it’s possible to implement an architecture very much like Brooks’ subsumption architecture on the LEGO robots.

Subroutines in NXC

- NXC supports subroutines, with a pretty standard syntax:

```c
sub turn_around(int pwr) {
  OnRev(OUT_C, pwr);
  Wait(900);
  OnPwr(OUT_AC, pwr);
}

task main() {
  OnPwr(OUT_AC, 75);
  Wait(1000);
  turn_around(75);
  Wait(2000);
  turn_around(75);
  Wait(1000);
  Off(OUT_AC);
}
```

NXC documentation

- Other standard things:
  - Variables: declaration, initialisation
  - Data structures: arrays
  - Control structures: if, do, while, for, etc
  - Arithmetic expressions
  - The preprocessor: #define, #include
- On the 343 webpage: Links to the NXT and NXC home pages and detailed information about how to develop and download NXC code.
- Other two useful guides:
  - Programming LEGO NXT robots using NXC, by Daniele Benedetti
  - NXC programmer’s guide, by John Hansen (the creator of NXC).

Challenge

- The physical capabilities of a LEGO robot are very limited.
  - The information about the world, which robot receives via its sensors is minimal.
  - It has very simple motor capabilities.
  - It doesn’t have much memory. (256K FLASH, 64K RAM).
- So, the challenge is this: to program this simple robot to produce complex behaviours.
- Thus, we are going to do some practical exercise.

Model-based / Reflex agents with state

- We can modify a reflex agent to maintain the state.
  - ‘Important’ sensory inputs can cause changes in the agent’s state, while other ‘irrelevant’ inputs can leave its state unchanged.
  - A state can be a complex representation (model) of the world.
  - The agent’s actions can now be based on a combination of its current state and its current sensory inputs:

```c
def function reflex_vacuum_agent(location, status) returns an action
static last_A, last_B, initially set to Clean
if location = A and status = Dirty and last_A = Dirty then return Suck
if location = A and status = Dirty and last_A = Dirty then return Suck
...
```
A reflex agent may behave as if it has goals, but it doesn’t represent them explicitly. This means it can’t reason about how to achieve its goals.

A goal-based agent is a huge step up. In this kind of agent,
- One or more goal states are defined.
- The agent’s state includes a representation of the state of the world.
- The agent has the ability to reason about the consequences of its actions, along the following lines:
  - ‘In state S1, if I do action A1, the next state will be S2’

Now the agent can search amongst all the actions in its repertoire for an action which achieves one of its goals.

Utility-based agents

- Utility function: it maps a new state, which is a consequence of action A onto a single number – measure of agent’s happiness.
- Agent’s happiness is inversely related to the cost of the actions, and influences the choice of subsequent actions.

In this lecture (reading AIMA 2nd ed., chapters 1 and 2, chap. 25.7):
- We looked at definitions of intelligence and AI
- We introduced a formal definition of an agent
- We introduced a taxonomy of different environment types
- We looked at some different agent architectures

Next lecture, we will introduce the logical agents, i.e. model-based agents based on propositional logic.

Next lecture & lab reading: AIMA 2nd ed. Chap. 7.1–7.5.