Intelligent Agents, AI & Interactive Entertainment (CS7032)

Lecturer: S Luz
School of Computer Science and Statistics
Trinity College Dublin
luzs@cs.tcd.ie

December 11, 2014

Contents

1 Course Overview

What

- Artificial Intelligence, Agents & Games:
  - AI
    - (+ *Dialogue Systems* and/or *Natural Language* Generation, TBD)
  - *Distributed* AI,
  - Machine *learning*, specially reinforcement learning
- A *unifying framework*:
  - "Intelligent" *agents*

  An *agent* is a computer system that is *situated* in some *environment* and that is capable of *autonomous action* in its environment in order to meet its design objectives.

  (Wooldridge, 2002)

Why

- Theory: *Agents* provide a resonably well-defined framework for investigating a number of questions in *practical* AI
  - (unlike approaches that attempt to emulate *human behaviour* or discover universal "laws of thought")
- Practice:
  - Applications in the area of *computer games*,
  - Computer games as a *dropsophila* for AI (borrowing from (McCarthy, 1998))

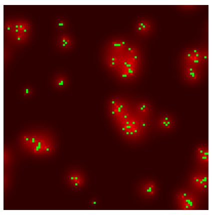
Course structure

- Main topics:
  - Agent *architectures and platforms*
  - Reasoning (*action* selection)
  - *Learning*
- Practicals:
  - project-based *coursework* involving, for example, implementation of a multi-agent system, a machine learning and reasoning module, a natural language generation system, ...
- Knowledge of Java (and perhaps a little Prolog) is desirable.

Agent architectures and multi-agent platforms

- Abstract agent architecture: formal *foundations*
- Abstract architectures instantiated: *case studies*
  - theory,
  - standards, and
  - agent development platforms (e.g. Repast)
  - and libraries (e.g. mason[a], swarm[b])

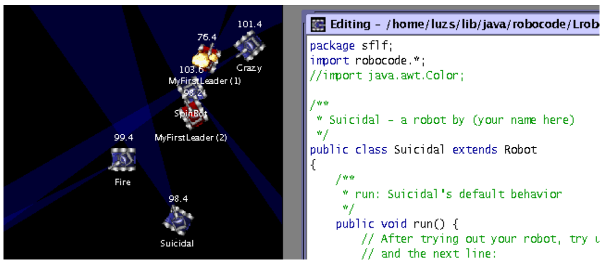
[a]`http://cs.gmu.edu/~eclab/projects/mason/`
[b]`https://savannah.nongnu.org/projects/swarm`



- Agents and simulation:

Reasoning

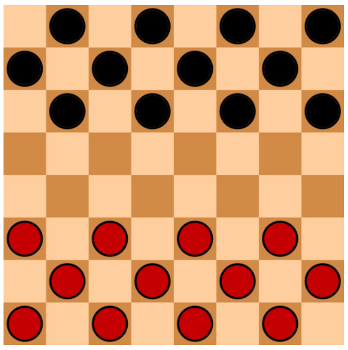
- Probabilistic Reasoning:
  - theoretical *background*
  - knowledge *representation* and *inference*
  - Survey of *alternative approaches* to reasoning under uncertainty



```
package sflf;
import robocode.*;
//import java.awt.Color;

/**
 * Suicidal - a robot by (your name here)
 */
public class Suicidal extends Robot
{
    /**
     * run: Suicidal's default behavior
     */
    public void run() {
        // After trying out your robot, try u
        // and the next line:
```

Learning

- *Machine learning*: theoretical background
- Probabilistic learning: *theory* and *applications*
- *Reinforcement* learning 

Coursework and Project

- Coursework-based *assessment*:
- Weekly labs and mini-assignments (30%)
- Longer project, to be delivered in January (70%):
  - Entering an AI Game competition
  - Reinforcement Learning for Off-The-Shelf games: use of a game simulator/server such as "Unreal Tournament 4" (UT2004) to create an agent that learns.
  - A Natural Language Generation system to enter the Give Challenge
  - A project on communicating cooperative agents,
  - ...

Multiagent systems course plan

- An overview: (Wooldridge, 2002) (Weiss, 1999, prologue):
  - What are agents?
  - What properties characterise them?
- Examples and issues
- Human vs. artificial agents
- Agents vs. objects
- Agents vs. expert systems
- Agents and User Interfaces

A taxonomy of agent architectures

- Formal definition of an agent architecture (Weiss, 1999, chapter 1) and (Wooldridge, 2002)
- "Reactive" agents:
  - Brooks' subsumption architecture (Brooks, 1986)
  - Artificial life
  - Individual-based simulation
- "Reasoning" agents:
  - Logic-based and BDI architectures
  - Layered architectures

The term "reactive agents" is generally associated in the literature with the work of Brooks and the use of the subsumption architecture in distributed problem-solving. In this course, we will use the term to denote a larger class of agents not necessarily circumscribed to problem-solving. These include domains where one is primarily interested in observing, or systematically studying patterns emerging from interactions among many agents.

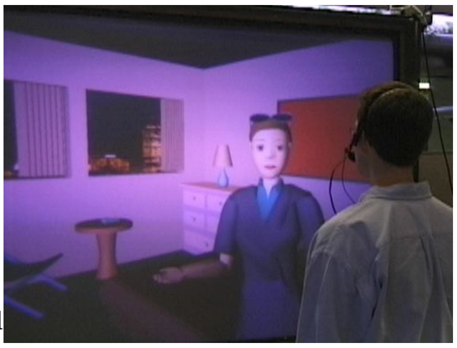
Reactive agents in detail

- Reactive agents in problem solving
- Individual-based simulation and modelling and computational modelling of complex phenomena:
  - Discussion of different approaches
- Simulations as multi-agent interactions
- Case studies:
  - Agent-based modelling of social interaction
  - Agent-based modelling in biology
  - Ant-colony optimisation

Logic-based agents (Not covered this year.)

- Coordination issues
- Communication protocols
- Reasoning
- Agent Communication Languages (ACL):
  - KQML
  - FIPA-ACL
- Ontology and semantics:
  - KIF
  - FIPA-SL

"Talking" agents



- Embodied conversational Agents
- Natural Language processing and generation by machines
- Some case studies

Learning

- Learning and agent architectures (Mitchell, 1997)
- Learning and uncertainty
- Reinforcement learning:
  - Foundations
  - Problem definition: examples, evaluative feedback, formalisation,
  - Elementary solution methods, temporal-difference learning,
  - Case studies: various game-playing learners

Course organisation and resources

- Course Web page:

  `http://www.cs.tcd.ie/~luzs/t/cs7032/`

- The page will contain:
  - Slides and notes
  - links to suggested reading and software to be used in the course.
- It will also contain the latest announcements ("course news"), so please check it regularly

Software resources

- A game simulator (no AI methods incorporated):
  - Robocode: `http://robocode.sf.net/`
- Agent-based simulation software:
  - MASON: `http://http://cs.gmu.edu/~eclab/projects/mason/` or
  - SWARM: `http://www.swarm.org/` or

  - REPAST `http://repast.sourceforge.net/`
- The Give Challenge website: `http://www.give-challenge.org/research/`
- See also:
  - `http://www.cs.tcd.ie/~luzs/t/cs7032/sw/`

Reading for next week

- "Partial Formalizations and the Lemmings Game" (McCarthy, 1998).
  - available on-line[1];
  - Prepare a short presentation (3 mins; work in groups of 2) for next Tuesday

Texts

- Useful references:
  - Agents, probabilistic reasoning (Russell and Norvig, 2003)
  - Agents, multiagent systems (Wooldridge, 2002)
  - Reinforcement learning (Bertsekas and Tsitsiklis, 1996), (Sutton and Barto, 1998)
- and also:
  - (some) probabilistic reasoning, machine learning (Mitchell, 1997)
  - Artificial Intelligence for Games (Millington, 2006)
  - Research papers TBA...

[1] `http://www.scss.tcd.ie/~luzs/t/cs7032/bib/McCarthyLemmings.pdf`

2 Agents: definition and formal architectures

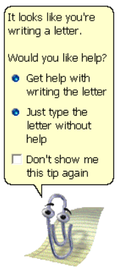
Outline

- What is an agent?
- Properties one would like to model
- Mathematical notation
- A generic model
- A purely reactive model
- A state-transition model
- Modelling concrete architectures
- References: (Wooldridge, 2009, ch 1 and ch 2 up to section2.6), (Russell and Norvig, 2003, ch 2)

Abstract architectures will (hopefully) help us describe a taxonomy (as it were) of agent-based systems, and the way designers and modellers of multi-agent systems see these artificial creatures. Although one might feel tempted to see them as design tools, abstract architectures are really descriptive devices. Arguably their main purpose is to provide a framework in which a broad variety of systems which one way or another have been identified as "agent systems" can be classified in terms of certain formal properties. In order to be able to describe such a variety of systems, the abstract architecture has to be as general as possible (i.e. it should not commit the modeller to a particular way of, say, describing how an agent chooses an action from its action repertoire). One could argue that such level of generality limits the usefulness of an abstract architecture to the creation of taxonomies. This might explain why this way of looking at multi-agent systems hasn't been widely adopted by agent developers. In any case, it has the merit of attempting to provide precise (if contentious) definitions of the main concepts involved thereby moving the discussion into a more formal domain.

What is an "agent"?

- "A person who acts on behalf of another" (Oxford American Dictionaries)
- Action, action delegation: state agents, travel agents, Agent Smith
- Action, simply (autonomy)
- "Artificial agents": understanding machines by ascribing human qualities to them (McCarthy, 1979) (humans do that all the time)
- Interface agents: that irritating paper-clip



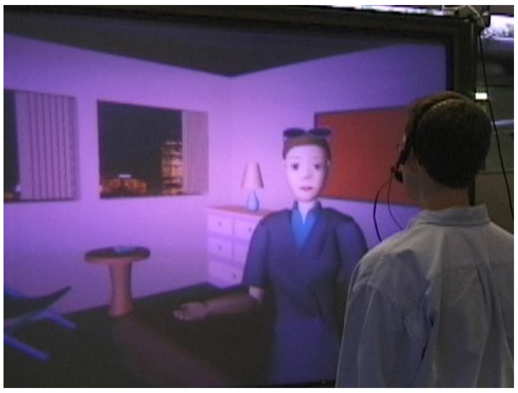
It looks like you're writing a letter.

Would you like help?

- Get help with writing the letter
- Just type the letter without help
- ☐ Don't show me this tip again

An example: Interface agents

- Embodied life-like characters REA: real estate agent (Cassell, 2000), speech recognition and synthesis, multi-modal interaction (gestures, gaze, life-size graphics etc)



- Not all UI designers like the idea, though (Shneiderman and Maes, 1997)

AI's Nobel prize winner

- Scientific and philosophical background:
  - Natural Sciences ☒
    - Conformance to "natural laws"
    - An air of necessity about them
  - Sciences of the artificial (Simon, 1981): ☑
    - Prescription vs. description
    - A science of the contingent
    - A science of "design"

From a prescriptive (design) perspective...

- Agent properties:
  - Autonomy: acting independently (of user/human intervention)
  - Situatedness: sensing and modifying the environment
  - Flexibility:
    - *re-activity* to changes
    - *pro-activity* in bringing about environmental conditions under which the agent's goals can be achieved, and
    - *sociability*: communicating with other agents, collaborating, and sometimes competing

In contrast with the notion of abstract architecture we develop below, these properties of multi-agent systems are useful from the point of view of conceptualising individual systems (or prompting the formation of a similar conceptual model by the user, in the case of interface agents). Abstract architectures, as descriptive devices, will be mostly interested in the property of "situatedness". The starting point is the assumption that an agent is essentially defined by its actions, and that an agent's actions might be reactions to and bring about changes in the environment. Obviously, there is more to agents and environments than this. Agents typically exhibit goal-oriented behaviour, assess future actions in terms of the measured performance of past actions, etc. The architectures described below will have little to say about these more complex notions.

However, before presenting the abstract architecture, let's take a look at a taxonomy that does try to place goals in context:

PAGE Descriptions

- P.A.G.E.: Percepts, Actions, Goals and Environment

Agent Type	*Percepts*	*Actions*	*Goals*	*Environment*
Medical diagnosis system	Symptoms, findings, patient's answers	Questions, tests, treatments	Healthy patient, minimize costs	Patient, hospital
Satellite image analysis system	Pixels of varying intensity, color	Print a categorization of scene	Correct categorization	Images from orbiting satellite
Part-picking robot	Pixels of varying intensity	Pick up parts and sort into bins	Place parts in correct bins	Conveyor belt with parts
Interactive English tutor	Typed words	Print exercises, suggestions, corrections	Maximize student's score on test	Set of students

As in (Russell and Norvig, 1995)

...alternatively, P.E.A.S. descriptions

Agent Type	*Performance measure*	*Environment*	*Actuators*	*Sensors*
Medical diagnosis system	Healthy patient, minimize costs	Patient, hospital	Questions, tests, treatments	Symptoms, findings, patient's answers
Satellite image analysis system	Correct categorization	Satellite link	Print a categorization of scene	Pixels of varying intensity, color
Part-picking robot	% parts in correct bins	Conveyor belt with parts	Pick up parts and sort into bins	Pixels of varying intensity
Interactive English tutor	Student's score on test	Set of students; testing agency	Print exercises, suggestions, corrections	Keyboard input

From (Russell and Norvig, 2003)

Environment properties

- Fully vs. partially observable: whether agent's can obtain complete and accurate information about the environment
- deterministic vs. stochastic: whether the next state of the environment is fully determined by the current state and action performed by the agent
- episodic vs. sequential: whether agent's next action depends only on the current state of the environment (episodic), or on assessment of past environment states (sequential)
- static vs. dynamic: whether the environment changes independently of the agent's actions
- discrete vs. continuous: whether the possible actions and percepts on an environment are finite (discrete environment) or not (continuous environment)
- single vs. multiple agents

Types of environments

Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	fully	yes	sequential	static	yes	single
Chess w/ clock	fully	strategic	sequential	semi	yes	multi
Poker	partially	strategic	sequential	static	discrete	multi
Backgammon	fully	stochastic	sequential	static	discrete	multi
Car driving	partially	stochastic	sequential	dynamic	continuous	multi
Medical diagnosis	partially	stochastic	sequential	dynamic	continuous	single
Image analysis	fully	deterministic	episodic	semi	continuous	single
Robot arm	partially	stochastic	episodic	dynamic	continuous	single
English tutor	partially	stochastic	sequential	dynamic	discrete	multi
Plant controller	partially	stochastic	sequential	dynamic	continuous	single

From (Russell and Norvig, 2003)

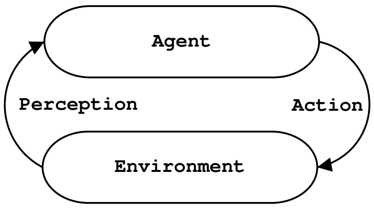
Chess with clock is semi-static (semi-dynamic) because even though the environment changes (time passes, players are penalised if they fail to play by a certain deadline) independently of the agent's actions, it does so in a regular and predictable way. The environment a taxi driver operates in, on the other hand, may change in unpredictable ways, being therefore a *dynamic environment*. Medical diagnosis typically depends on patient history, so its environment is *sequential*. In contrast, a robot which examines various parts which pass on a conveyor belt in front of the robot's camera, one at the time, and moves defective parts off the conveyor belt does not have to consider its past decisions in order to decide whether the part it is currently examining is defective. The robot is therefore said to operate in an *episodic* environment.

Abstract agent architectures

- Why?
- What we would like to describe:

– agent

– environment



– their interactions

Notation

- A few tools from discrete maths and logic:
  - $A, S, ...$: sets
  - $\wp(S)$: the powerset of S
  - S^* : all sequences of elements (i.e. ordered subsets) of S
  - $\Rightarrow, \wedge, \vee, \neg$: material implication, conjunction, disjunction and negation
  - Quantifiers: $\forall$ and $\exists$

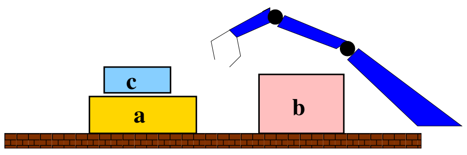
Refreshing your memory: The powerset of a set S is a set containing all of S's subsets. So, if $S = \{s_0, s_1\}$, its powerset $\wp(S) = \{\emptyset, \{s_0\}, \{s_1\}, \{s_0, s_1\}\}$. Sequence sets S^* are similar to power sets except that the order of their elements matters. So, if $S = \{s_0, s_1\}$, we have $S^* = \{<>, < s_0 >, < s_1 >, < s_0, s_1 >, < s_1, s_0 >\}$.

The logical connectives behave in the familiar way. $X \wedge Y$ is true iff both X and Y are true; $X \vee Y$ is true if X is true, Y is true, or both; $X \Rightarrow Y$ is false if X is true and Y is false, and true otherwise. Since these connectives are used here simply to illustrate particular examples of agent architectures, we will deliberately leave their semantics underspecified. Feel free to interpret them as they are interpreted in, say, PROLOG. One could also add existential and universal quantifiers as needed.

Defining an architecture

- A *standard* architecture is a 4-tuple: $Arch_s =< S, A, action, env >$ where
  - $S = \{s_1, s_2, ..., s_n\}$ is a finite set describing all (possible) environment states, and
  - $A = \{a_1, a_2, ..., a_n\}$ is the set of all actions an agent is capable of performing
  - *action* is a function describing the agents behaviour, and
  - *env* is a function describing the "behaviour" of the environment

An example: consider the "blocks world" below:



Assume the robotic arm is capable of performing a set of actions $A = \{move(a, b), move(c, floor), \dots\}$. Environment states could be represented by an $S = \{s_1, s_2, ..., s_n\}$, where we could have

$$s_1 = \{left(a, b), on(floor, a), on(a, c), \dots\} \tag{2.1}$$

and (after a $move(c, floor)$):

$$s_2 = \{left(a, b), on(floor, a), on(floor, c), \dots\} \tag{2.2}$$

One could also, for instance, encode general *properties* such as

$$\forall x, y, z. left(x, y) \land left(y, z) \Rightarrow left(x, z) \tag{2.3}$$

etc, to remain constant in all environment states.

"Agenthood"

- An agent's behaviour will be characterised by the following function:

$$action : S^* \to A$$

- Does having S^* as the domain of *action* make the function most naturally suited to modelling episodic or sequential environments? Why?
- Requirement captured: the current action may depend on the interaction history (i.e. the sequence of environment states)

Actions consider an essentially *sequential* environment, since their arguments are sequences. In our blocks world example, one could have $action(\langle s_1, s_2 \rangle) = move(a, b)$, assuming that the *purpose* of moving block c off block a was to be able to place a on b. Note, however, that the formalism does not explicitly represent *purpose*, planning, or any such notion.

Environment dynamics

- Changes in the environments will be characterised by the following function:

$$env : S \times A \to \wp(S)$$

- Intuition: $env(s_j, a_k) = S'$ performing an action a_k on an environment whose state is s_j results in a number of scenarios (S')
- In the general case, what type of environment does *env* model? Answer: *env* models a non-deterministic environment.
- If $|S'| = 1$, then the environment is deterministic.

In the example above, if nothing else is said, action $move(c, floor)$ would affect the environment as follows:

$$env(s_1, move(c, floor)) = \{\{left(c, a), \dots\}, \{left(a, c), \dots\}\} \tag{2.4}$$

If the relative positions of the blocks were irrelevant, $env(s_1, move(c, floor))$ would be a singleton.

Interaction history

- The agent-environment interaction will be characterized as follows:

$$h : s_0 \overset{a_0}{\rightarrow} s_1 \overset{a_1}{\rightarrow} ... \overset{a_{u-1}}{\rightarrow} s_u \overset{a_u}{\rightarrow} ...$$

- h is a *possible history* of the agent in the environment *iff*:

$$\forall u \in \mathbb{N}, a_u = action(< s_0, ..., s_u >) \tag{2.5}$$

  and

$$\forall u > 0 \in \mathbb{N}, s_u \in env(s_{u-1}, a_{u-1}) \tag{2.6}$$

Condition (2.5) guarantees that all actions in a history apply to environment states in that history.

Condition (2.6) guarantees that all environment states in a history (except the initial state) "result" from actions performed on environment states.

Characteristic behaviour

- Characteristic behaviour is defined as a set of interaction histories

$$hist = \{h_0, h_1, ..., h_n\}$$

  where

- each h_i is a possible history for the agent in the environment

Invariant properties

- We say that ϕ is an invariant property of an agent architecture *iff*

> For all histories $h \in hist$ and states $s \in S$ ϕ is a property of s (written $s \models \phi$)

- We will leave the relation $\models$ underspecified for the time being.
  - In architectures where agents have minimal symbolic reasoning abilities (e.g. some concrete reactive architectures), $\models$ could be translated simply as set membership, that is $s \models \phi \Leftrightarrow \phi \in s$, in architectures where reasoning isn't performed

Behavioural equivalence

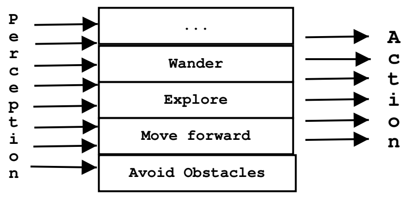
- Equivalence of behaviours is defined as follows (in an abstract architecture)

> An agent $ag1$ is regarded as *equivalent* to agent $ag2$ *with respect to environment* env_i iff $hist(ag1, env_i) = hist(ag2, env_i)$

- When the condition above holds for all environments env_i, then we simply say that $ag1$ and $ag2$ have **equivalent behaviour**

Modelling reactive agents

- Reactive architectures
  - production rules
  - a scheme for defining priorities in the application of rules (e.g. subsumption (Brooks, 1986))



  - no reasoning (theorem proving or planning) involved

Abstract reactive agents

- *Purely reactive agents* can be modelled by assuming

$$action : S \rightarrow A$$

- Everything else remains as in the general (standard abstract architecture) case:

$$Arch_r =< S, A, action_r, env >$$

- Abstract reactive agents operate essentially on an episodic view of environments (i.e. they are memoryless agents). See slide 16

Example: a thermostat.
$temperature(cold) \Rightarrow do(heater(on))$
$\neg temperature(cold) \Rightarrow do(heater(off))$
$S = \{\{temperature(cold)\}, \{\neg temperature(cold)\}\}$
$A = \{heater(on), heater(off)\}$

Purely reactive vs. standard agents

- Proposition 1 (exercise):

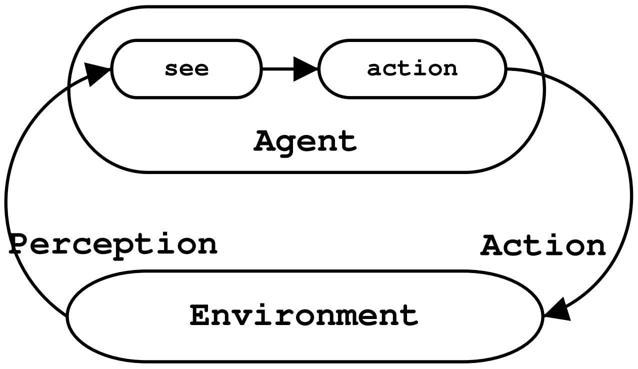
Purely reactive agents form a proper subclass of standard agents. That is, for any given environment description S, and action repertoire A:

(i) every purely reactive agent is behaviourally equivalent to a standard agent, and

(ii) the reverse does not hold.

Modelling Perception

- Refining the agent's decision function



- Types and sources of perception

Percepts and actions

- Facts perceived by an agent will be represented as set

$$P = \{p_0, p_1, ..., p_n\}$$

- The decision function becomes

$$\mathit{action} : P^* \rightarrow A$$

- which is then linked to environment states via the perception fumction

$$\mathit{see} : S \rightarrow P$$

Properties of perception

- If $see(s_i) = see(s_j)$, we say that s_i and s_j are *indistinguishable*, even if $s_i \neq s_j$.
- Define $\equiv$, an equivalence relation over S by saying $s \equiv s'$ iff $see(s) = see(s')$
- If $| \equiv | = |S|$, then we say that the agent is *perceptually ominiscient*
- On the other hand, if $| \equiv | = 1$, then the agents perceptual ability is nil

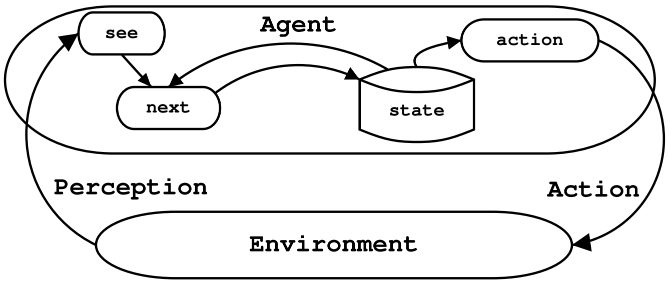
In the following example, four environment states are (rightfully, assuming that a butterfly doesn't cause changes in the weather) as two:
$x = weather(cold)$
$y = moving\_wings(butterfly)$
$S = \{\{\neg x, \neg y\}, \{\neg x, y\}, \{x, \neg y\}, \{x, y\}\}$
$see(\{x, \neg y\}) = see(\{x, y\})$
$see(\{\neg x, \neg y\}) = see(\{\neg x, y\})$

The equivalence relation $\equiv$ is then $\{\{\{x, y\}, \{x, \neg y\}, \}\{\{\neg x, y\}, \{\neg x, \neg y\}\}\}$

For perceptually omniscient agents, $see(s_i) = s_i$, for all $s_i \in S$. In an agent with no perceptual abaility, $see(s_i) = see(s_j)$ for all $s_i, s_j \in S$.

Refining the interaction history

- Representation used so far: history as a sequence of environment states



- The next step: represent history (in the agent architecture) as environment changes as perceived by the agent

State-based decision function

- The state-based architecture will be represented as

$$Arch =< S, A, P, I, action, env, see, next >$$

  where

- I is the set of all internal states of an agent,
- $see : S \to P$,
- $action : I \to A$, and
- $next : I \times P \to I$

Properties of State-based architectures

- Proposition 2:

> State-based architectures are equivalent to standard architectures with respect to the behaviours they are able to represent.

Exercise

- Prove Proposition 2.

Further information

- (Weiss, 1999): overview; this presentation mostly followed the material found there;
- (Russell and Norvig, 2003): $\boxed{\text{agent = architecture + program}}$
- (Genesereth and Nilsson, 1987): foundations of a theory of rational agency

3 Practical: Creating a simple robocode agent

3.1 Goals

In this exercise we will aim at (1) understanding how the Robocode[1] simulator works, (2) to building a simple “robot”, and (3) describing it in terms an the abstract agent architecture introduced in the last lecture[2].

In this lab we use Robocode version **1.0.7** which can be downloaded from sourceforge[3]. Alternatively, if you prefer experimenting with PROLOG, you can specify robots declaratively using a modified (and not very well tested, I'm afraid) version of the simulator.

3.2 Robocode: Getting started

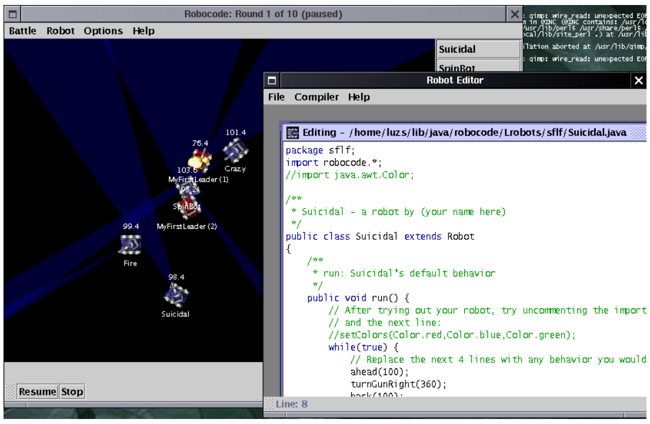
The Robocode simulator should be installed in the IET Lab.

- `http://robocode.sf.net/`

In order to run the simulator, change directory to `ROBOCODE_HOME` and run `robocode.sh` (on Unix) or `robocode.bat` (on Windows). This will bring up a window displaying the “battlefield” (Fig. 3.1). Once the battlefield is visible, you may start a new battle by selecting `Battle->New`. You can create your first robot by starting the editor `Robot->Editor` and `File->New->Robot`. Inspect the robot template (that's already a fully functional, if not very clever, robot). Create a sub-directory of `robots` named by your initials (in lowercase) so that your robots will have a unique identifier, and select it as your code repository through `File->Save as...`, ignoring Robocode's suggestions.

Now, start a new battle and put your newly created robot to fight against some of the robots included in the Robocode distribution.

[1]`http://robocode.sf.net/`
[2]`http://www.cs.tcd.ie/~luzs/t/cs7032/abstractarch-notes.pdf`
[3]`http://robocode.sf.net/`



```
package sflf;
import robocode.*;
//import java.awt.Color;

/**
 * Suicidal - a robot by (your name here)
 */
public class Suicidal extends Robot
{
    /**
     * run: Suicidal's default behavior
     */
    public void run() {
        // After trying out your robot, try uncommenting the import
        // and the next line:
        //setColors(Color.red,Color.blue,Color.green);
        while(true) {
            // Replace the next 4 lines with any behavior you would
            ahead(100);
            turnGunRight(360);
```

Figure 3.1: Robocode battlefield and editor windows

3.3 Robot creation in (some) detail

You normally create robots by sub-classing Robocode classes `Robot` or `AdvancedRobot`. You may have noticed that `Robot` is the class used in the first robot you've just created. It provides basic functionality such as methods that cause the robot to move, turn, etc as well as "events" (i.e. methods that will be called back by the simulator) when the robot is hit by a bullet, hits another robot etc. `AdvancedRobot` provides a bit more flexibility by allowing you to define non-blocking behaviours, handle the agents event queue explicitly, create and register your own events, and write to the filesystem. For the moment we will deal only with `Robot`.

The robot consists of three parts: the vehicle, the gun and the radar, as shown in Fig 3.2. `Robot` defines the following (blocking) methods for moving those parts:

- `turnRight(double degree)` and `turnLeft(double degree)` turn the robot by a specified degree.

- `ahead(double distance)` and `back(double distance)` move the robot by the specified pixel distance; these two methods are completed if the robot hits a wall or another robot.

- `turnGunRight(double degree)` and `turnGunLeft(double degree)` turn the gun, independent of the vehicle's direction.

- `turnRadarRight(double degree)` and `turnRadarLeft(double degree)` turn the radar on top of the gun, independent of the gun's direction (and the vehicle's direction).

When the vehicle is turned, the direction of the gun (and radar) will also move, unless you call one of the `setAdjust...` methods to indicate otherwise. See the help pages[4] for further details.

The following methods will be called by the simulator when certain events arise:

[4] `http://robocode.sourceforge.net/help/index.html`

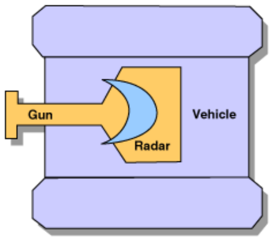


Figure 3.2: Anatomy of the robot

- `ScannedRobotEvent`. Handle the ScannedRobotEvent by overriding the `onScannedRobot()` method; this method is called when the radar detects a robot.
- `HitByBulletEvent`. Handle the HitByBulletEvent by overriding the `onHitByBullet()` method; this method is called when the robot is hit by a bullet.
- `HitRobotEvent`. Handle the HitRobotEvent by overriding the onHitRobot() method; this method is called when your robot hits another robot.
- `HitWallEvent`. Handle the HitWallEvent by overriding the onHitWall() method; this method is called when your robot hits a wall.

Finally, you can fire a bullet by calling `fire(double power)` and `fireBullet(double power)`. `power` controls the amount of energy to be spent on a particular shot.

3.4 Exercises

3.4.1 The environment

Describe the robocode environment and characterise it in terms of the environment properties taxonomy given in the lecture notes (p. 3)[5] task environment description (Russell and Norvig, 2003, p. 45).

3.4.2 Create your own robot

Study the Robocode API[6] and subclass Robot or AdvancedRobot to implement your robot. Time permitting, we will run a short tournment next week with your robots as contestants, after which some of you will be invited to describe your implementations in terms of the concepts we've discussed in the past weeks (see next section).

3.4.3 Describe the task environment and robot

1. Describe your robot's task environment in terms of PEAS (Performance measure, Environment, Actuators and Sensors). See lecture notes.

[5] `http://www.cs.tcd.ie/~luzs/t/cs7032/abstractarch-notes.pdf#page=3`
[6] `http://robocode.sourceforge.net/docs/index.html`

2. Describe your robot and the enrironment it operates in as a logic-based (aka knowledge-based) agent architecture.

   Specifically, you should describe the environment S, the events the agent is capable of perceiving, the actions (A) it is capable of performing, the environmental configurations that trigger certain actions and the environmental changes (if any) that they bring about. Facts, states and actions may be described with PROLOG clauses or predicate logic expressions, for instance.

   What sort of abstract architecture would most adequately describe your robot? Why? What are the main difficulties in describing robots this way?

3.4.4 PROLOG Robots (optional)

In the course's software repository[7] you will find a (gzip'd) tarball containing the source code of an earlier version of the robocode simulator (1.0.7, the first open-source release) which I have modified so that it allows you to write your robocode declaratively, in Prolog. The main modifications are in `robocode/robolog/*.java`, which include a new class manager and loader adapted to deal with Prolog code (identified by extension `.pro`). The class manager (`RobologClassManager`) assigns each Prolog robot a `LogicalPeer` which does the interface between the simulator and Prolog "theories". Some Prolog code (`robocode/robolog/*.pro`) handle the other end of the interface: `actioninterface.pro` contains predicates which invoke action methods on `Robot` objects, and `environinterface.pro` predicates that retrieve information from the battlefield. All these predicates will be available to your robots. A simple logical robot illustrating how this works can be found in `bindist/robots/log/LogBot.pro`.

The archive contains a `Makefile` which you can use to compile the code and generate a 'binary' distribution analogous to the pre-installed robocode environment (`make bindist`). Alternatively, you can download a pre-compiled version[8] from the course website.

Build a Prolog robot. Do you think there are advantages in being able to specify your robot's behaviour declaratively? If so, perhaps you could consider adapting 'robolog' to the latest version of robocode and contributing the code to the project.

[7] `http://www.cs.tcd.ie/~luzs/t/cs7032/sw/robolog-0.0.1-1.0.7.tar.gz`
[8] `http://www.cs.tcd.ie/~luzs/t/cs7032/sw/robolog-0.0.1-1.0.7-bin.tar.gz`

4 Utility functions and Concrete architectures: deductive agents

Abstract Architectures (ctd.)

- An agent's behaviour is encoded as its *history*:

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \ldots$$

- Define:
  - H^A: set of all histories which end with an *action*
  - H^S: histories which end in *environment states*
- A *state transformer* function $\tau : H^A \to \wp(S)$ represents the effect an agent has on an environment.
- So we may represent environment dynamics by a triple:

$$Env =< S, s_0, \tau >$$

- And similarly, agent dynamics as

$$Ag : H^S \to A$$

The architecture described in this slide and in (Wooldridge, 2002) appears to be an attempt to address some limitations of the formalisation described in the Abstract Architecture Notes. In that model, *action* and *env* — describing respectively the agent's choice of action given an environment, and possible changes in the environment given an action — are regarded as "primitives" (in the sense that these are the basic concepts of which the other concepts in the architecture are derived). As the framework strives to be general enough to describe a large variety of agent systems, no assumptions are made about causal connections between actions and changes in the environment. Furthermore, although the history of environmental changes is assumed to be available to the agent as it chooses an action (recall the function's signature $action : S^* \to A$), no specific structure exists which encodes action history.

The new model proposed here builds on efforts originating in AI (Genesereth and Nilsson, 1987, ch 14) and theoretical computer science (Fagin et al., 1995, p 154) and apparently attempts to provide an account of both action and environment history. The primitive used for describing an agent's behaviour is no longer an action fuction, as defined above, but a *history* (or *run*, as in (Fagin et al., 1995)):

$$h : s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{u-1}} s_u \xrightarrow{a_u} \ldots$$

If one defines:

- H^A: the set of all histories which end with an *action*
- H^S: the histories which end in *environment states*

Then a *state transformer* function $\tau : H^A \rightarrow \wp(S)$ represents the way the environment changes as actions are performed. (Or would it be the effect an agent has on its environment? Note that the issue of causality is never satisfactorily addressed), and the "environment dynamics" can be described as a triple:

$$Env =< S, s_0, \tau > \tag{4.1}$$

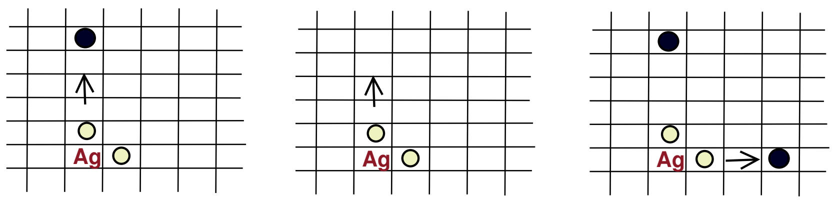
and the action function redefined as $action : H^S \rightarrow A$.

Although this approach seems more adequate, (Wooldridge, 2002) soon abandons it in favour of a variant the first one when *perception* is incorporated to the framework (i.e. the action function becomes $action : P^* \rightarrow A$).

Utility functions

- Problem: how to "tell agents what to do"? (when exhaustive specification is impractical)
- Decision theory (see (Russell and Norvig, 1995, ch. 16)):
  - associate *utilities* (a performance measure) to states:
    $$u : S \rightarrow \mathbb{R}$$
  - Or, better yet, to *histories*:
    $$u : H \rightarrow \mathbb{R}$$

Example: The Tileworld



- The *utility* of a course of action can be given by:
  $$u(h) = \frac{\text{number of holes filled in h}}{\text{number of holes that appeared in h}}$$
- When the utility function has an upper bound (as above) then we can speak of *optimal agents*.

Optimal agents

- Let $P(h|Ag, Env)$ denote the probability that h occurs when agent Ag is placed in Env.
- Clearly
$$\sum_{h \in H} P(h|Ag, Env) = 1 \tag{4.2}$$
- An *optimal agent* Ag_{opt} in an environment Env will maximise expected utility:
$$Ag_{opt} = \arg \max_{Ag} \sum_{h \in H} u(h) P(h|Ag, Env) \tag{4.3}$$

From abstract to concrete architectures

- Moving from abstract to concrete architectures is a matter of further specifying *action* (e.g. by means of algorithmic description) and choosing an underlying form of representation.
- Different ways of specifying the *action* function and representing knowledge:
  - Logic-based: decision function implemented as a theorem prover (plus control layer)
  - Reactive: (hierarchical) condition $\rightarrow$ action rules
  - BDI: manipulation of data structures representing *Beliefs*, *Desires* and *Intentions*
  - Layered: combination of logic-based (or BDI) and reactive decision strategies

Logic-based architectures

- AKA *Deductive Architectures*
- Background: symbolic AI
  - Knowledge representation by means of logical formulae
  - "Syntactic" symbol manipulation
  - Specification in logic $\Rightarrow$ executable specification
- "Ingredients":
  - Internal states: sets of (say, first-order logic) formulae
    - $\Delta = \{temp(roomA, 20), heater(on), ...\}$
  - Environment state and perception,
  - Internal state seen as a set of beliefs
  - Closure under logical implication ($\Rightarrow$):
    - $closure(\Delta, \Rightarrow) = \{\varphi | \varphi \in \Delta \vee \exists \psi . \psi \in \Delta \wedge \psi \Rightarrow \varphi\}$
  - (is this a reasonable model of an agent's beliefs?)

About whether logical closure (say, first-order logic) corresponds to a reasonable model of an agent's beliefs, the answer is likely no (at if we are talking about human-like agents). One could, for instance, know all of axioms Peano's axioms[1] for natural numbers and still not know whether Goldbach's conjecture[2] (that all even numbers greater than 2 is the sum of two primes).

Representing deductive agents

- We will use the following objects:
  - L: a set of sentences of a logical system
    - As defined, for instance, by the usual wellformedness rules for first-order logic
  - $D = \wp(L)$: the set of *databases* of L
  - $\Delta_0, ..., \Delta_n \in D$: the agent's internal states (or *beliefs*)
  - $\models_\rho$: a deduction relation described by the deduction rules ρ chosen for L: We write $\Delta \models_\rho \varphi$ if $\varphi \in closure(\Delta, \rho)$

Describing the architecture

- A logic-based architecture is described by the following structure:

$$\boxed{Arch_L =< L, A, P, D, action, env, see, next >} \tag{4.4}$$

- The update function consists of additions and removals of facts from the current database of internal states:
  - $next : D \times P \to D$
    - *old*: removal of "old" facts
    - *new*: addition of new facts (brought about by *action*)

Pseudo-code for *action*

```
1.   function $action(\Delta : D)$ : A
2.   begin
3.       for each $a \in A$ do
4.           if $\Delta \models_\rho do(a)$ then
5.               return $a$
6.           end if
7.       end for
8.       for each $a \in A$ do
9.           if $\Delta \not\models_\rho \neg do(a)$ then
10.              return $a$
11.          end if
12.      end for
13.      return $noop$
14.  end function $action$
```

[1]http://en.wikipedia.org/wiki/Peano_axioms

[2]http://unsolvedproblems.org/index_files/Goldbach.htm

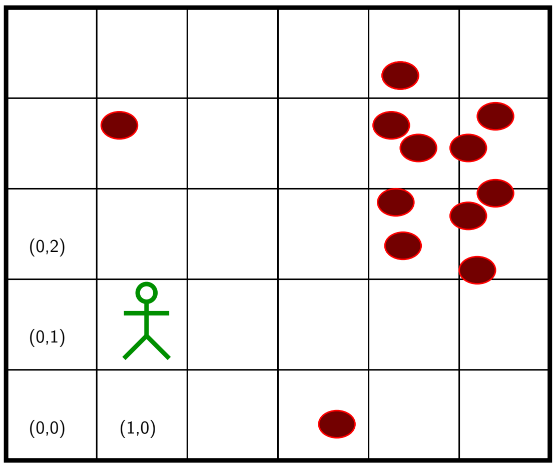
Environment and Belief states

- The environment change function, *env*, remains as before.
- The belief database update function could be further specified as follows

$$\boxed{next(\Delta, p) = (\Delta \setminus old(\Delta)) \cup new(\Delta, p)} \tag{4.5}$$

where $old(\Delta)$ represent beliefs no longer held (as a consequence of *action*), and $new(\Delta, p)$ new beliefs that follow from facts perceived about the new environmental conditions.

Example: The Vacuum world



(Russell and Norvig, 1995; Weiss, 1999)

Describing The Vacuum world

- Environment
  - perceptual input: $dirt, null$
  - directions: (facing) $north, south, east, west$
  - position: coordinate pairs (x, y)
- Actions
  - $move\_forward, turn\_90^o\_left, clean$
- Perception:

$$P = \{\{dirt(x, y), in(x, y), facing(d), ...\}, ...\}$$

Deduction rules

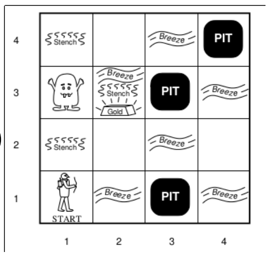
- Part of the decision function
- Format (for this example):
  - $P(...) \Rightarrow Q(...)$
- PROLOG fans may think of these rules as *Horn Clauses*.
- Examples:
  - $in(x, y) \wedge dirt(x, y) \Rightarrow do(clean)$
  - $in(x, 2) \wedge \neg dirt(x, y) \wedge facing(north) \Rightarrow do(turn)$
  - $in(0, 0) \wedge \neg dirt(x, y) \wedge facing(north) \Rightarrow do(forward)$
  - ...

Updating the internal state database

- $next(\Delta, p) = (\Delta \setminus old(\Delta)) \cup new(\Delta, p)$
- $old(\Delta) = \{(P(t_1, ..., t_n) | P \in \{in, dirt, facing\} \wedge P(t_1, ..., t_n) \in \Delta\}$
- $new(\Delta, p)$:
  - update agent's position,
  - update agent's orientation,
  - etc

The "Wumpus World"

- See (Russell and Norvig, 2003, section 7.2)
- BTW, Ch 7 is available online (last accessed Oct 2012) at `http://aima.cs.berkeley.edu/newchap07.pdf`



Shortcomings of logic-based agents

- Expressivity issues: problems encoding percepts (e.g. visual data) etc
- *Calculative rationality* in dynamic environments
- Decidability issues
- Semantic elegance vs. performance:
  - loss of "executable specification"
  - weakening the system vs. temporal specification
- etc

Existing (??) logic-based systems

- MetameM, Concurrent MetameM: specifications in temporal logic, model-checking as inference engine (Fisher, 1994)
- CONGOLOG: Situation calculus
- Situated automata: compiled logical specifications (Kaelbling and Rosenschein, 1990)
- AgentSpeak, ...
- (see (Weiss, 1999) or (Wooldridge, 2002) for more details)

BDI Agents

- Implement a combination of:
  - deductive reasoning (*deliberation*) and
  - planning (*means-ends reasoning*)

Planning

- Planning formalisms describe actions in terms of (sets of) *preconditions* (P_a), *delete lists* (D_a) and *add lists* (A_a):

$$< P_a, D_a, A_a >$$

- E.g. Action encoded in STRIPS (for the "block's world" example):

```
Stack(x,y):
    pre: clear(y), holding(x)
    del: clear(y), holding(x)
    add: armEmpty, on(x,y)
```

Planning problems

- A planning problem $< \Delta, O, \gamma >$ is determined by:
  - the agent's *beliefs* about the initial environment (a set Δ)
  - a set of operator descriptors corresponding to the actions available to the agent:

$$O = \{< P_a, D_a, A_a > | a \in A\}$$

  - a set of formulae representing the *goal/intention* to be achieved (say, γ)
- A plan $\pi =< a_i, ..., a_n >$ determines a sequence $\Delta_0, ..., \Delta_{n+1}$ where $\Delta_0 = \Delta$ and $\Delta_i = (\Delta_{i-1} \setminus D_{a_i}) \cup A_{a_i}$, for $1 \leq i \leq n$.

Suggestions

- Investigate the use of BDI systems and agents in games.
- See, for instance, (Norling and Sonenberg, 2004) which describe the implementation of interactive BDI characters for Quake 2.
- And (Wooldridge, 2002, ch. 4), for some background.
- There are a number of BDI-based agent platforms around. ‘Jason’, for instance, seems interesting:

  `http://jason.sourceforge.net/`

5 Reactive agents & Simulation

Concrete vs. abstract architectures

- Different ways of specifying the *action*
  - Logic-based: decision function implemented as a theorem prover (plus control layer)
  - *Reactive: (hierarchical) condition $\rightarrow$ action rules*
  - BDI: manipulation of data structures representing *Beliefs*, *Desires* and *Intentions*
  - Layered: combination of logic-based (or BDI) and reactive decision strategies

Stimuli and responses

- Behaviour: the product of an agent's interaction with its environment
- Intelligence: patterns that *emerge* from the interactions triggered by different behaviours
- Emergence: The transition from *local feedback* (human designed) and *global feedback* (product of agent autonomy).

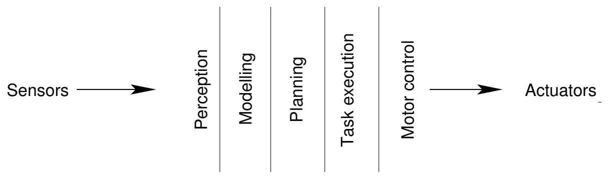
A typical scenario

- Multiple goals
  - sometimes conflicting or inconsistent
- Multiple sensors
  - dealing with varied, sometimes inconsistent readings
- Robustness and fault tolerance
  - w.r.t. loss of agents

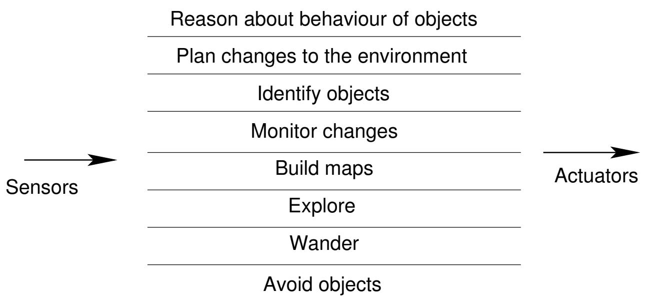
- Additivity
  - the more sensors and capabilities, the more processing power the agent needs

Comparing action models

- Cognitive Agent action model



- Reactive Agent action model

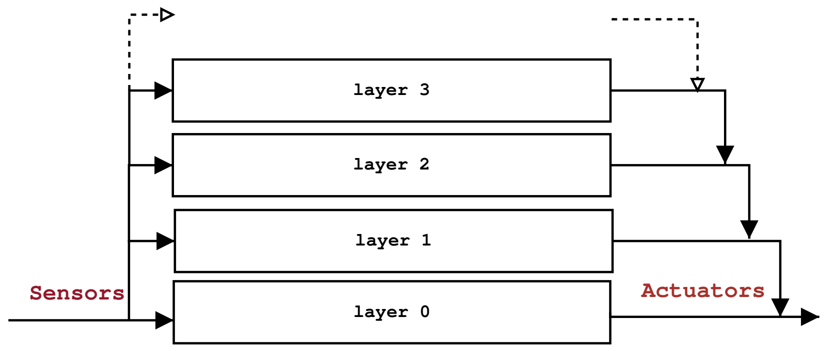


Some reactive architectures

- Situated rules:
  - PENGI (Chapman and Agre, 1987)
- *Subsumption architecture*
  - (Brooks, 1986)
- Competing tasks (Maes, 1989)
- Eco Agents (Drogoul and Ferber, 1992)
- Neural nets??
- ...

A simple reactive architecture

- The subsumption diagram



A Formalisation

- Situated Rules represented as pairs $< c, a >$ (*behaviours*)
  - The set of all possible behaviours is then: $Beh = \{< c, a > | c \in P \wedge a \in A\}$
- The subsumption hierarchy will be represented by a total ordering, $\prec$, on the *behaviour relation*, $R \subseteq Beh$: $\prec \subseteq R \times R$
  - We say that "b inhibits b'" if $b \prec b'$

A reactive decision function

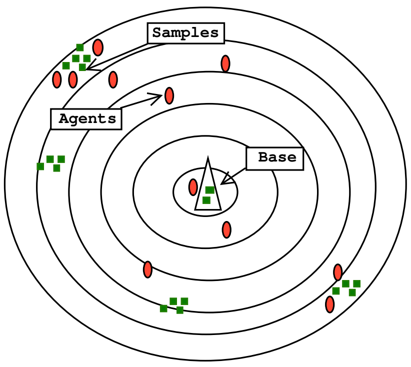
```
1.  function $action(p : P)$ : A
2.  var $fired : \wp(R)$
3.  begin
4.      $fired := \{\langle c, a\rangle | \langle c, a\rangle \in R \wedge p \in c\}$
5.      for each $\langle c, a\rangle \in fired$ do
6.          if $\neg(\exists \langle c', a'\rangle \in fired \wedge$
                $\langle c', a'\rangle \prec \langle c, a\rangle)$
7.          then return $a$
8.          end if
9.      end for
10.     return $noop$
11. end function $action$
```

Time complexity

- For the "naive" algorithm above...
  - $action() = O(n^2)$, where $n = max(|R|, |P|)$
- N.B.: Complexity for *each* agent
- (In practice, one can often do better than $O(n^2)$, low time complexity being one of the main selling points of reactive architectures.)

Example: collective problem solving

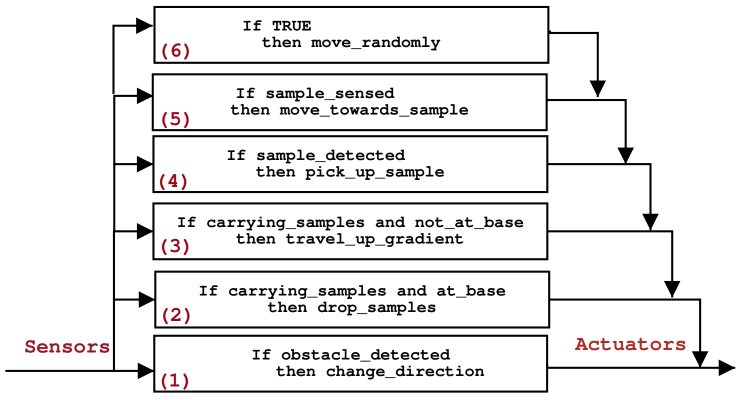
- Case study: Foraging Robots (Steels, 1990; Drogoul and Ferber, 1992):



- Constraints:
  - No message exchange
  - No agent maps
  - obstacles
  - gradient field
  - clustering of samples

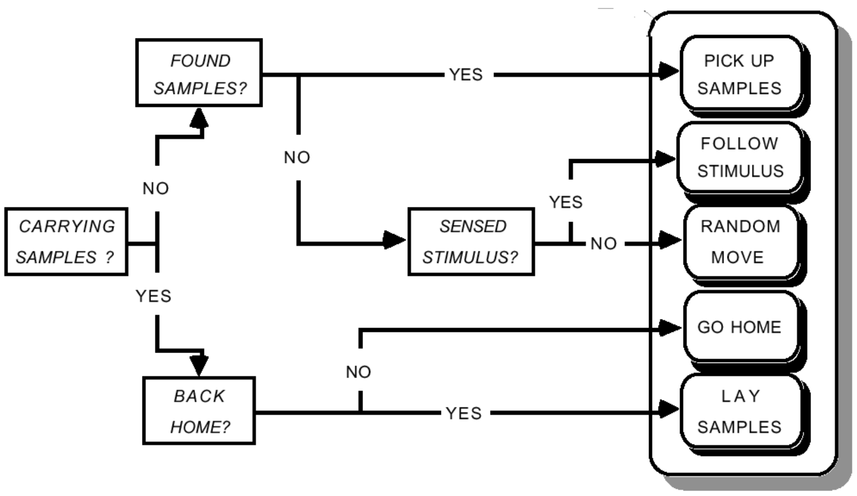
Simple collecting behaviour

- Subsumption ordering: (1) $\prec$ (2) $\prec$ (3) $\prec$ (4) $\prec$ (5) $\prec$ (6)


If TRUE then move_randomly (6)
If sample_sensed then move_towards_sample (5)
If sample_detected then pick_up_sample (4)
If carrying_samples and not_at_base then travel_up_gradient (3)
If carrying_samples and at_base then drop_samples (2)
If obstacle_detected then change_direction (1)
Sensors
Actuators

Behaviour diagram

- Same rules (roughly), as described in (Drogoul and Ferber, 1992)



Can we improve on this design?

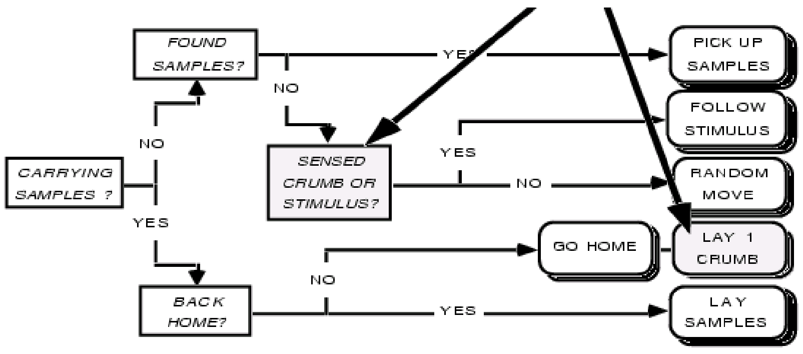
- What is the problem with the proposed system?
  - Too many random trips when samples in clusters.
- Try Indirect communication: “bread crumbs”, ant pheromones, etc
- Replace rules (3) and (5) by:

```
(3’) If carrying samples
       and not_at_base
     then drop_crumb
       and travel_up_gradient
```

```
(5’) If sample_sensed
       or crumb_sensed
     then
       move_towards_sample_or_crumb
```

Improved architecture I

- After replacing (3) and (5):



Further improvement?

- What is the long-term effect of laying bread crumbs? Is there room for improvement?
- Change (3') into:

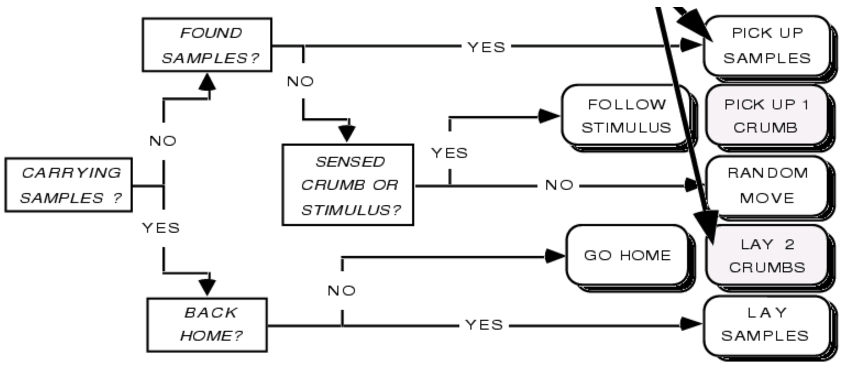
```
(3") If carrying_samples
        and not_at_base
       then drop_2_crumbs and
            travel_up_gradient
```

- and add the following:

```
(7) If crumb_detected
      then pick_up_1_crumb
```

Improved architecture II

- The subsumption ordering becomes
  - $(1) \prec (2) \prec (3'') \prec (4) \prec (7) \prec (5') \prec (6)$



Advantages of this approach

- Low time (and space) complexity
- Robustness
- Better performance
  - (near optimal in some cases)
  - problems when the agent population is large

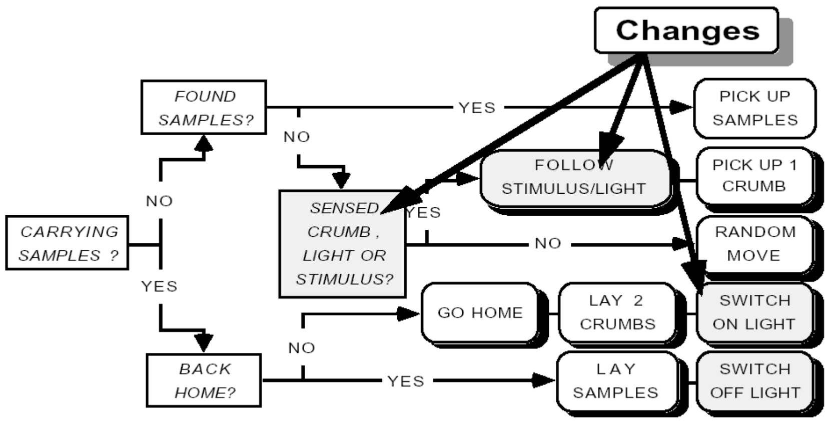
Agent chains

- New rules ("docker robots") (Drogoul and Ferber, 1992)

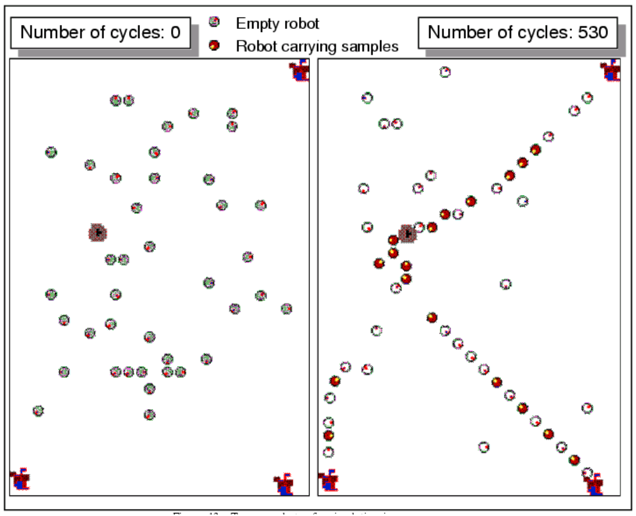
```
(a) If not_carrying_sample and
       travelling_down_gradient and
       detected_sample_carrying_agent
    then pick_up_other_agents_sample and
         travel_up_gradient

(b) If carrying_sample and
       travelling_up_gradient and
       detected_empty_agent
    then deliver_sample_to_other_agent and
         travel_down_gradient
```

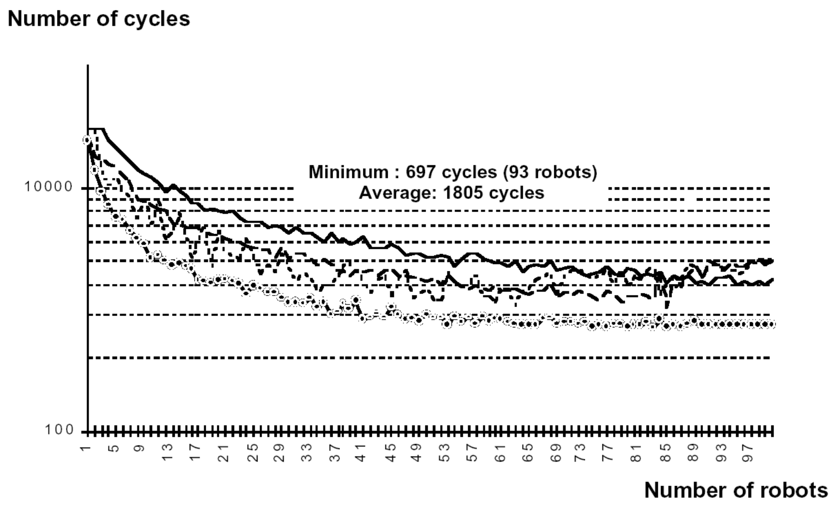
Behaviour diagram



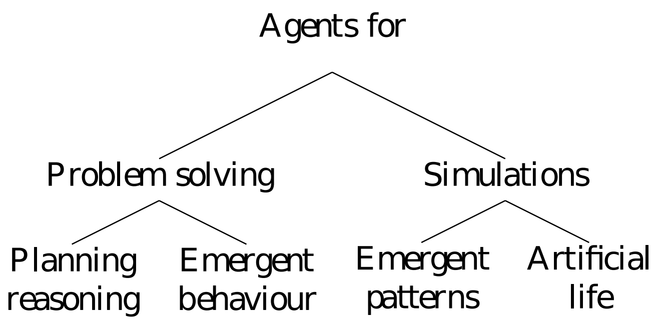
A simulation



How do the different architectures perform?



Simulation Agents and Reactive Architectures



Why use agents in simulations

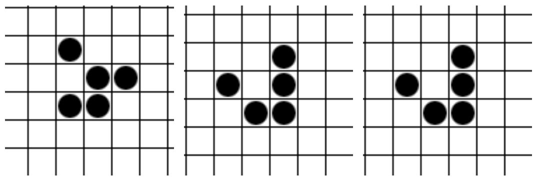
- challenges for "traditional" modelling methods:
  - how to extrapolate a model to situations where the assumptions on which it is based (described through ODEs or PDEs, for instance) no longer hold;
  - how to *explain* the macroscopic properties of the systems modelled;
  - how handle heterogenity;
  - how to handle discontinuity...

Background

- A workshop (Langton, 1989)
- Nonlinear models and complex systems:
  - A few phenomena which resist linearisation: plant growth, weather, traffic flow, stock market crashes, intelligence, ...
- "Understanding by building":
  - Individual-based Modelling in Biology (population biology, ecology, ...)
  - principles of intelligent behaviour
  - and practical applications

Applications to games

- Artificial Life: cellular automata and the "Game of Life"



- Tamagotchi, The Sims^{tm}, etc
- Distributed search and problem solving (e.g. for path-finding)

Examples: Conway's Life

- A "zero-player" game invented by John H Conway in the 70s.
- Rules:
  1. Any live cell with fewer than two live neighbours dies, as if caused by under-population.
  2. Any live cell with two or three live neighbours lives on to the next generation.

3. Any live cell with more than three live neighbours dies, as if by overcrowding.
4. Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

- (Demo in emacs...)

Entertaining examples: flocking behaviour

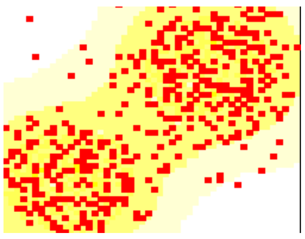
- Craig Reynolds page at Sony Entertainment: `http://www.red3d.com/cwr/boids/`
- (demo Video and applets)
- Steering behaviours:
  - *Separation*: steer to avoid crowding local flockmates
  - *Alignment*: steer towards the average heading of local flockmates
  - *Cohesion*: steer to move toward the average position of local flockmates

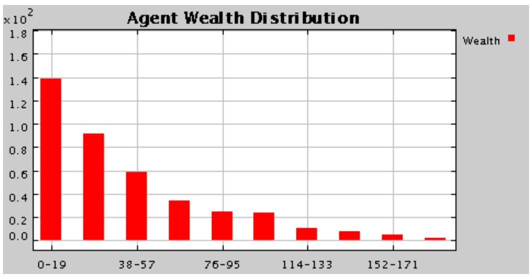
Emergence in agent-based simulations

- Emergence in complex systems:

  Stable macroscopic patterns arising from the local interaction of agents.

- Example: Skewed "wealth" distribution in (Epstein and Axtell, 1996, ch 2)





Advantages of agent-based modelling

- Simplicity: shift from mathematical descriptions of entire systems to rule based specifications of agent behaviour.
- Implementation of complex boundary conditions: in agent-based simulations, environments with irregular shape are not more complex to model than regular ones.
- Inherent parallelism: no code changes when porting to parallel architectures

- Adequacy to modelling of small populations
- Realism (??)

NB: Complex boundary conditions are handled in traditional (e.g. PDE) modelling by methods such as multigrid, at the price of complicated complications in ensuring consistency of the conditions in the embedded problems.

Disdvantages of agent-based modelling

- Memory and processing speed might constrain the size of the agent population in the model
- Difficulties in exploring the parameter space, if the simulation comprises a large number of rules
- Understanding how simple local behaviour gives rise to complex global behaviour is not always an easy task; if a model captures too much of the complexity of the world, it may become just as difficult to understand as the world itself.
- "Noise" introduced by the model or its implementation might give rise to phenomena not present in the real system

Agent modelling toolkits

- Swarm, RePast, StarLogo, Ascape, ...
- What they provide
  - mechanisms for managing resource allocation
  - a schedule
  - basic environment topography
  - graphics, (media handling etc)
  - a scientific computing library
  - basic statistics
  - Usually no built-in agent semantics
- Agent development library: `mason`: "a fast, discrete-event multiagent simulation library core in Java" (Luke et al., 2005).

Applications of agent-based modelling

- Sociological models (e.g. (Epstein and Axtell, 1996))
- Biological simulations
  - Insect societies
  - bacterial growth
  - forest dynamics

- Molecule interaction in artificial chemistry
- Traffic simulations
- Computer networks (see `http://www.crd.ge.com/~bushsf/ImperishNets.html`, for instance)

RePast: A (Pure) Java Simulator

- Repast is an acronym for *REcursive Porous Agent Simulation Toolkit.*

  > *"Our goal with Repast is to move beyond the representation of agents as discrete, self-contained entities in favor of a view of social actors as permeable, interleaved and mutually defining, with cascading and recombinant motives."*
  >
  > *From the Repast web site*

- ??????

Two simulations: 1 - Mouse Traps

- A demonstration of "nuclear fission":
  - Lay a bunch of mousetraps on the floor in a regular grid, and load each mousetrap with two ping pong balls.
  - Drop one ping pong ball in the middle...
- A discrete-event simulation that demonstrates the dynamic scheduling capabilities of Repast
- The agent programmer defines:
  - an agent (`MouseTrap`)
  - a model (`MouseTrapModel`)

Two simulations: 2 - Heat Bugs

- "(...) an example of how simple agents acting only on local information can produce complex global behaviour".
- Agents: `HeatBugs` which absorb and expel heat
- Model: `HeatBugsModel` has a spatial property, heat, which diffuses and evaporates over time. (green dots represent `HeatBugs`, brighter red represents warmer spots of the world.)
- A `HeatBug` has an ideal temperature and will move about the space attempting to achieve this ideal temperature.

6 Practical: Multiagent Simulations in Java

6.1 Goals

In this lab we will take a look at a Java-based simulation environment called Repast[1], run two simple demos, learn the basics of how to build a simulation, and analyse and modify a simulation of economic agents (Epstein and Axtell, 1996).

6.2 Setting up the environment

The simulation software you will need may have already been installed on the machines in the IET Lab. If not, please download *Repast*, version 3[2] from the course's software repository[3]. And uncompress it to a suitable directory (folder) in your user area. In the rest of this document we will refer to the location of the Repast directory (folder) on these machines as `REPAST_HOME`.

The libraries you will need in order to compile and run the simulations are located in `REPAST_HOME/` and `REPAST_HOME/lib/`. These include simulation-specific packages (such as `repast.jar`), CERN's scientific computing libraries (`colt.jar`), chart plotting libraries (`plot.jar`, `jchart.jar`), etc.

The source code for the original demos is located in `REPAST_HOME/demo/`. You will find modified versions of some of the demos for use in this lab[4] on the course website. Just download and uncompress them into the same folder where you uncompressed repast3.1.tgz, so that your directory tree will look like this:

```
--|
  |
  |-- repast3.1/
  |
  |-- simlab-0.8/ --|
                    |--- bugs/
```

[1]Repast is an acronym for *REcursive Porous Agent Simulation Toolkit.*

[2]Newer versions might work as well, but I haven't tested them.

[3]`https://www.scss.tcd.ie/~luzs/t/cs7032/sw/repast3.1.tgz`

[4]`http://www.scss.tcd.ie/~luzs/t/cs7032/sw/simlab-0.8.tar.gz`

```
|--- mousetraps/
|--- sscape-2.0-lab/
```

The demos can be run via the shell scripts (Unix) or batch files (Windows) provided. These scripts are set so that they will "find" the repast libraries in the above directory structure. If you uncompress repast to a different location, you will have to set the `REPAST_HOME` variable manually in these scripts so that it points to the location of repast on your machine.

Documentation for the simulation (REPAST) and scientific computing (COLT) API's is available in javadoc format in `REPAST_HOME/docs/{api,colt}`, respectively. Further information on how to build and run simulations in Repast can be found in

`REPAST_HOME/docs/how_to/how_to.html`.

6.3 Running two demos

We will start by running two demos originally developed to illustrate the *Swarm* simulation environment and ported to Repast:

Mouse trap: a simple discrete-event simulation the illustrates the dynamic scheduling capabilities of Repast (and Swarm, the simulation system for which it was first developed). The source code contains detailed commentary explaining how the simulator and model work. Please take a close look at it.

Heatbugs: an example of how simple agents acting only on local information can produce complex global behaviour. Each agent in this model is a 'HeatBug'. The world (a 2D torus) has a spatial property, heat, which diffuses and evaporates over time. The environmental heat property is controlled in `HeatSpace.java`[5]. The world is displayed on a `DisplaySurface`[6] which depicts agents as green dots and warmer spots of the environment as brighter red cells. The class `HeatBugsModel`[7] sets up the environment, initialise the simulation parameters (which you can change via the GUI) distributes the agents on it and schedules actions to be performed at each time 'tick'. Agents release a certain amount of heat per iteration. They have an ideal temperature, and assess at each step their level of 'unhappiness' with their temperature at that step. Locally, since no agent can produce enough heat on his own to make them happy, agents seek to minimise their individual unhappiness by moving to cells that will make them happier in the immediate future. Although all agents act purely on local 'knowledge', globally, the system can be seen as a distributed optimisation algorithm that minimises average unhappiness. The original code has been modified so that it plots the average unhappiness for the environment, showing how it decreases over time.

[5] uchicago/src/repastdemos/heatBugs/HeatSpace.java
[6] uchicago.src.sim.gui.DisplaySurface
[7] uchicago.src.repastdemos.heatBugs.HeatBugsModel

In order to run a demo, simply, `cd` to its directory (created by uncompressing the lab archive), and use `run.sh` (or `run.bat`). If you want to modify the simulations, and use `compile.sh`[8] (or `compile.bat`) to recompile the code.

The purpose of running these simulations is to get acquainted with the simulation environment and get an idea of what it can do. So, feel free to experiment with different parameters, options available through the GUI, etc. Details on how to use the GUI can be found in the How-To documentation:

```
REPAST_HOME/how_to/simstart.html
```

6.4 Simulating simple economic ecosystems

Now we will explore the *SugarScape* demo in some detail. The demo is a partial implementation of the environment described in chapter 2 of (Epstein and Axtell, 1996). The growth rule G_α (page 23), the movement rule M (page 25), and the replacement rule R (PAGE 32) have been implemented.

A slightly modified version of the original demo is available in the archive you downloaded for the first part of this lab, in `sscape-2.0-lab/`. Shell scripts `compile.sh` and `run.sh` contain the `classpath` settings you will need in order to compile and run the simulation. Remember that **if you are not using the repast3.1 distribution provided** or **if the distribution has not been uncompressed as described above**, you will need to modify the scripts (.bat files) so that variable `REPAST_HOME` points to the right place.

6.4.1 Brief description of the sscape simulation

At the top level, the directory `sscape-2.0-lab/*` contains, in addition to the scripts, an (ASCII) description of the "sugar landscape": `sugarspace.pgm`. Open this file with your favourite editor and observe that it is filled with digits (0-4) which represent the distribution of "food" on the environment.

The source code for the simulation is in `sugarscape/*.java`[9]. All Repast simulations must contain a *model* class, which takes care of setting up the simulation environment, and at least one *agent* class. The file `SugarModel.java` contains an implementation of a model class, whereas `SugarAgent.java` and `SugarSpace.java` implement simple agents. Note that in this simulation, as is often the case, the environment is also conceptualised as an agent, whose only behaviour in this case is described by the growth rule.

6.4.2 Exercises

These are some ideas for exercises involving the sscape simulation. Please answer (at least) **three** of the questions below and submit your answers by the end of next week.

1. Modify the topography of the environment by changing `sugarscape.pgm` (copy the original file into, say, `sugarscape.old` first). Make four hills instead of two, or a valley, etc. What effect does the new shape of the world have on the system dynamics?

[8] For emacs+JDEE users, there is a `prj.el` file in `bugs` which you may find useful.
[9] uchicago/src/sim/sugarscape/*.java

2. What causes the uneven wealth distribution we see in the original simulation? Is it the agent death and replacement rule R? Is it the uneven distribution of food in `sugarspace.pgm`? Is it the variance in maximum age, or metabolism or vision? Pick one or two parameters of the model with the replacement rule and see how their variance affects the wealth distribution.

3. Modify `SugarModel.java` so that it also displays a time series of agent population. How does the population vary over time, if the default parameters are used? Why?

4. Reset the initial parameters so that the initial agent population decreases, asymptotically approaching the environment's carrying capacity (you might want to change the replacement rule and maximum death age, for instance). What carrying capacity (Epstein and Axtell, 1996, see pages 30-31) values do you obtain for the original maximum values of agent *metabolism* and *vision*?

5. Do an experiment on how the final carrying capacity of the system varies with the initial population. Write down a hypothesis, then design an experiment to test that hypothesis by varying the initial population size. Draw a graph of the result, evaluate the hypothesis, and draw conclusions[10].

6. Do an experiment to discover the effects of maximum metabolism and vision on agent survival.

[10]Exerxise suggested by N. Minar in an introduction to his implementation of the SugarScape in the SWARM toolkit (Minar, , now apparently offline)

7 Swarm intelligence: Ant Colony Optimisation

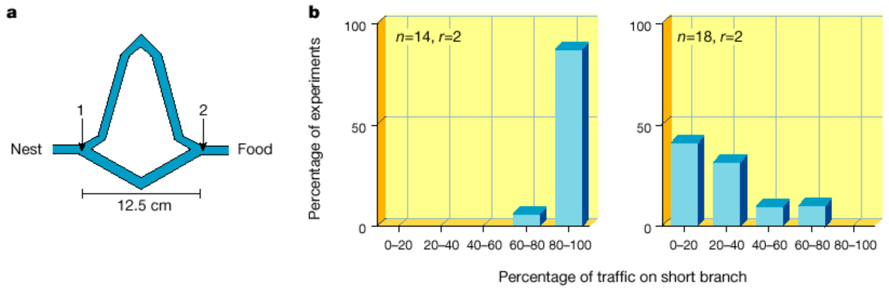
Simulating problem solving?

- Can simulation be used to improve distributed (agent-based) problem solving algorithms?
- Yes: directly, in a supervised fashion (e.g. Neural Nets "simulators")
- But also, indirectly, via exploration & experimental parameter tuning
- Case Study: Ant Colony Optimisation Heuristics (Dorigo et al., 1996; Dorigo and Di Caro, 1999)
- See also (Bonabeau et al., 2000) for a concise introduction (* recommended reading)

More specifically, the case study presented in these notes illustrate an application of ant algorithms to the symmetric variant of the *traveling salesman problem*. Ant algorithms have, as we will see, been employed in a number of optimization problems, including network routing and protein folding. The techniques presented here aim for generality and certain theoretical goals rather than performance in any of these particular application areas. For a good overview of variants, optimizations and heuristics, please consult (Dorigo and Di Caro, 1999).

Biological Inspiration

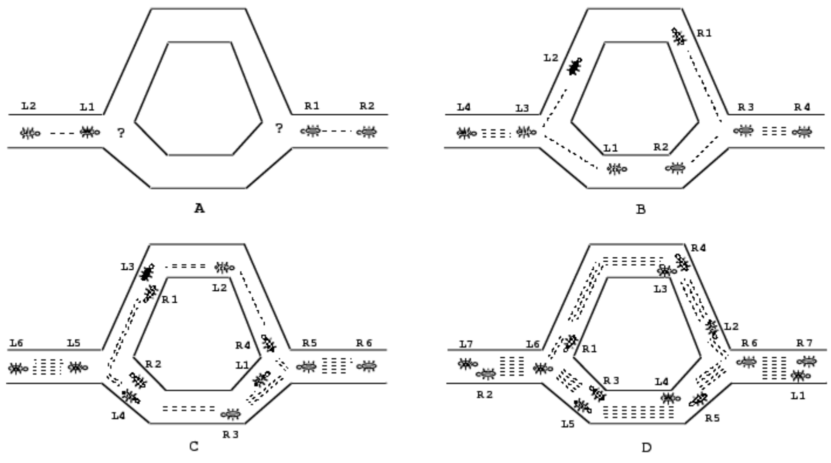
- Real ants foraging behaviour



- long branch is r times longer than the short branch.
- left graph: branches presented simultaneously.
- right graph: shorter branch presented 30 mins. later

The gaphs show results of an actual experiment using real ants (*Linepithema humile*). The role of pheromone plays is clear: it reinforces the ants' preferences for a particular solution. If regarded as a system, the ant colony will "converge" to a poor solution if the short branch is presented too late (i.e. if the choice of the long branch is reinforced too strongly by the accummulated pheromone).

Pheromone trails



A simple model to fit the observed ant behaviour could be based on the probabilities of choices available to ants at the branching point, e.g. the probability that the $m+1^{th}$ ant to come to the branching point will choose the upper path is given by

$$P_u(m) = \frac{(U_m + k)^h}{(U_m + k)^h + (L_m + k)^h} \tag{7.1}$$

where U_m is the number of ants the chose the upper path, L_m is the number of ants the chose the lower path and h, k are parameters which allow experimenters to tune the model to the observed behaviour, via Monte Carlo simulations.

Ants solve TSP

- The Travelling Salesman Problem:
  - Let $N = \{a, ..., z\}$ be a set of cities, $A = \{(r, s) : r, s \in V\}$ be the edge set, and $\delta(r, s) = \delta(s, r)$ be a cost measure associated with edge $(r, s) \in A$.
  - *TSP* is the problem of finding a minimal cost closed tour that visits each city once.
  - If cities $r \in N$ are given by their coordinates (x_r, y_r) and $\delta(r, s)$ is the Euclidean distance between r and s, then we have an *Euclidean TSP.*
  - If $\delta(r, s) \neq \delta(s, r)$ for at least one edge (r, s) then the TSP becomes an asymmetric TSP (ATSP).

TSP is known to be NP-hard, with a search space of size $\frac{(n-1)!}{2}$.

A Simple Ant Algorithm for TSP

```
Initialize
while ( !End_condition )
{ /*  call these 'iterations'  */
  position each ant on a node
  while (! all_ants_built_complete_solution)
  { /* call these 'steps' */
    for (each ant)
    {
      ant applies a state transition rule to
      incrementally build a solution
    }
  }
  apply local (per ant) pheromone update rule
  apply global pheromone update rule
}
```

Characteristics of Artificial Ants

- Similarities with real ants:
  - They form a colony of cooperating individuals
  - Use of pheromone for *stigmergy* (i.e. "stimulation of workers by the very performance they have achieved" (Dorigo and Di Caro, 1999))
  - Stochastic decision policy based on *local* information (i.e. no lookahead)

Dissimilarities with real ants

- Artificial Ants in Ant Algorithms keep internal state (so they wouldn't qualify as purely reactive agents either)
- They deposit amounts of pheromone directly proportional to the quality of the solution (i.e. the length of the path found)
- Some implementations use lookahead and backtracking as a means of improving search performance

Environmental differences

- Artificial ants operate in discrete environments (e.g. they will "jump" from city to city)
- Pheromone evaporation (as well as update) are usually problem-dependant
- (most algorithms only update pheromone levels after a complete tour has been generated)

ACO Meta heuristic

```
acoMetaHeuristic()
   while( !terminationCriteriaSatisfied )
    { /* begin scheduleActivities */
       antGenerationAndActivity()
       pheromoneEvaporation()
       daemonActions()            # optional #
    } /* end scheduleactivities */
```

```
antGenerationAndActivity()
   while(availableResources)
    {
       scheduleCreationOfNewAnt()
       newActiveAnt()
    }
```

Ant lifecycle

```
newActiveAnt()
  initialise()
 M := updateMemory()
 while(currentState ≠ targetState) {
  A := readLocalRoutingTable()
  P := transitionProbabilities(A,M,constraints)
  next := applyDecisionPolicy(P,constraints)
  move(next)
  if (onlineStepByStepPheromoneUpdate) {
     depositPheromoneOnVisitedArc()
       updateRoutingTable()
  }
  M := updateInternalState()
 } /* end while */
 if (delayedPheromoneUpdate) {
     evaluateSolution()
     depositPheromoneOnAllVisitedArcs()
     updateRoutingTable()
 }
```

How do ants decide which path to take?

- A *decision table* is built which combines pheromone and distance (cost) information:
- $A_i = [a_{ij}(t)]_{|N_i|}$, for all j in N_i, where:

$$a_{ij}(t) = \frac{[\tau_{ij}(t)]^\alpha [\eta_{ij}]^\beta}{\sum_{l \in N_i} [\tau_{il}(t)]^\alpha [\eta_{il}]^\beta} \tag{7.2}$$

- N_i: set of nodes accessible from i
- $\tau_{ij}(t)$: pheromone level on edge (i, j) at iteration t
- $\eta_{ij} = \frac{1}{\delta_{ij}}$

Relative importance of pheromone levels and edge costs

- The decision table also depends on two parameters:
  - α: the weight assigned to pheromone levels
  - β: the weight assigned to edge costs
- Probability that ant k will chose edge (i, j):

$$P_{ij}^k(t) = \begin{cases} a_{ij}(t) & \text{if } j \text{ has not been visited} \\ \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

The values given in (7.3) aren't strictly speaking correct, since part of the probability mass that would have been assigned to visited nodes is lost. The following variant avoids the problem:

$$P_{ij}^k(t) = \frac{a_{ij}(t)}{\sum_{l \in N_i^k} a_{il}(t)} \tag{7.4}$$

(for links to non-visited nodes) where $N_i^k \subseteq N_i$ is the set edges still to be travelled which incide from i. When all edges remain to be travelled, i.e. $N_i^k = N_i$, the probability that any edge will be visited next is simply given by the initial decision table:

$$\begin{aligned} P_{ij}^k(t) &= \frac{a_{ij}(t)}{\sum_{l \in N_i^k} a_{il}(t)} = \frac{a_{ij}(t)}{\sum_{l \in N_i} a_{il}(t)} \\ &= \frac{a_{ij}(t)}{\frac{\sum_{l \in N_i}[\tau_{il}(t)]^\alpha[\eta_{il}]^\beta}{\sum_{l \in N_i}[\tau_{il}(t)]^\alpha[\eta_{il}]^\beta}} = a_{ij}(t) \end{aligned}$$

Pheromone updating (per ant)

- Once all ants have completed their tours, each ant k lays a certain quantity of pheromone on each edge it visited:

$$\Delta\tau_{ij}^k(t) = \begin{cases} \frac{1}{L^k(t)} & \text{if } (i, j) \in T^k, \\ 0 & \text{otherwise} \end{cases} \tag{7.5}$$

- $T^k(t)$: ant k's tour
- $L^k(t)$: length of $T^k(t)$

Pheromone evaporation and (global) update

- Let

$$\Delta\tau_{ij}(t) = \sum_{k=1}^{m} \Delta\tau_{ij}^k(t)$$

  - where m is the number of ants,

- and $\rho \in (0, 1]$ be a *pheromone decay coefficient*
- The (per iteration) *pheromone update rule* is given by:

$$\tau_{ij}(t) = (1 - \rho)\tau_{ij}(t') + \Delta\tau_{ij}(t) \tag{7.6}$$

where $t' = t - 1$

Parameter setting

- How do we choose appropriate values for the following parameters?
  - α: the weight assigned to pheromone levels NB: if $\alpha = 0$ only edge costs (lengths) will be considered
  - β: the weight assigned to edge costs NB: if $\beta = 0$ the search will be guided exclusively by pheromone levels
  - ρ: the evaporation rate
  - m: the number of ants

Exploring the parameter space

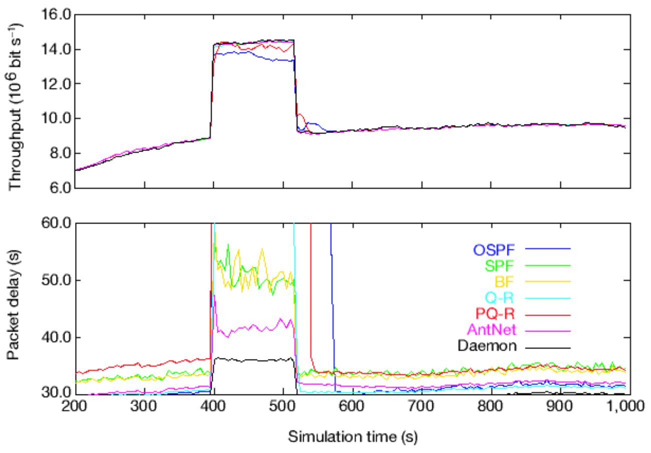
- Different parameters can be "empirically" tested using a multiagent simulator such as Repast
- Vary the problem space
- Collect statistics
- Run benchmarks
- etc

Performance of ACO on TSP

- Ant algorithms perform better than the best known evolutionary computation technique (Genetic Algorithms) on the Asymmetric Travelling Salesman Problem (ATSP)
- ... and practically as well as Genetic Algorithms and Tabu Search on standard TSP
- Good performance on other problems such as Sequential Ordering Problem, Job Scheduling Problem, and Vehicle Routing Problem

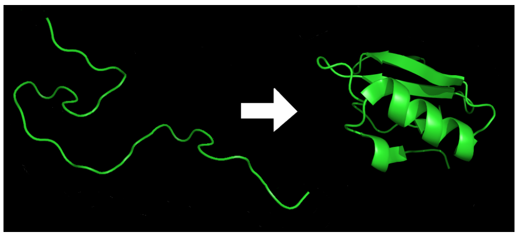
Other applications

- Network routing (Caro and Dorigo, 1998) and load balancing (Heusse et al., 1998)



Application: protein folding in the HP model

- Motivation: an important, unsolved problem in molecular biology and biophysics;

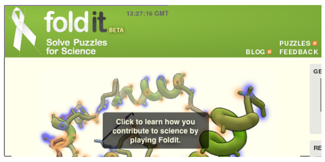


- Protein structure, not sequence, determines function.
- When folding conditions are met, protein will fold (repeatably) into a native state.

Practical applications

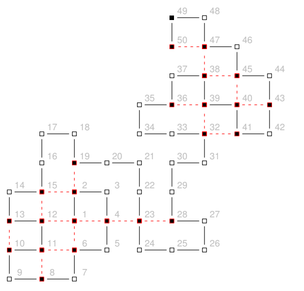
- Knowing how folding works is important:
    - Learn how to design synthetic polymers with particular properties
    - Treat disease (many diseases attributed to protein mis-folding, e.g. Alzheimer disease, cystic fibrosis, mad cow disease etc)
- Why model?
    - Deducing 3-d structure from protein sequence using experimental methods (such as x-ray diffraction studies or nuclear magnetic resonance [NMR]) is highly costly in terms of labour, skills and machinery.
- A connection with (serious) games:

– Fold.it: A Protein folding game...


foldit BETA
Solve Puzzles for Science
PUZZLES
BLOG
FEEDBACK
Click to learn how you contribute to science by playing Foldit.

ACO for Protein Folding: Motivation

- Levinthal's paradox: how can proteins 'find' their correct configuration (structure) when there is an astronomical number of configurations they can assume?



Total energy of this embedding = −19 ; theoretical lower bound = −25

- The HP (hydrophobic-hydrophilic, or nonpolar-polar) model (Dill et al., 1995) restricts the degrees of freedom (for folding) but...

- (Crescenzi et al., 1998; Berger and Leighton, 1998) independently proved folding in simple models (such as the 2D and 3D lattice HP model) to be NP-complete.

A possible ACO approach

- There are many ways to implement the HP model; one might adopt the following (Roche & Luz, unpublished):

- *Problem data*: A binary string of length n, representing chain of amino acid residues that make up a protein.

- *Decision tree*: From a given position (in the problem data), valid choices are Left, Right and Forward (orientation) – except where a dead-end is encountered (ie. tour must be self avoiding)

- *Environment*:Pheromone represents a memory of fitness for possible decisions.

  – e.g. a 2-dimensional array of size $[n-1][3]$, representing the respective desirabilities for the different orientations (left,right,forward) from a given position in the problem data.

- *Tour*: The set of moves (orientations) made by an ant in deriving a partial or complete solution.

Ant behaviour

- State transition rule: How *ant selects next* move (left,right,forward)
  - *Exploitation*:
    - Greedy algorithm involve a lookahead ($r \geq 2$) to gauge local benefit of choosing a particular direction (benefit measured in terms of additional HH contacts)
    - Combine greedy algorithm with 'intuition' (pheromone), to derive overall cost measure
  - *Exploration*: Pseudo-random (often biased) selection of an orientation (help avoid local minima)
  - *Constraints*:
    - Resultant tour must be self avoiding.
    - Look-ahead could help detect unpromising folding (e.g. two elements can be topological neighbours only when the number of elements between them is even etc)

Pheromone update

- *Local update*: (optional)
  - Can follow same approach as with TSP, by applying pheromone decay to edges as those edges are visited, to promote the exploration of different paths by subsequent ants during this iteration.
- *Global update*:
  - Can also follow same approach as with TSP: Apply pheromone decay (evaporation!) to each edge in the decision tree (negative factor)
  - Selection of 'best' candidate (best cost) and reinforcement of pheromone on edges representing this decision
  - Typically global updating rule will be iteration-best or global-best

Results

- Competitive with best-known algorithms (MCMC, GA, etc) in terms of best conformations found
- Can be slow and more resource-intensive.
- May be possible to combine ACO with techniques useb by approximation algorithms (Hart and Istrail, 1995)
- Alternative approaches based on ACO have been proposed (Shmygelska and Hoos, 2003).

8 Practical: ACO simulation for TSP in Repast

8.1 Goals

In this practical you will implement an Ant Colony Optmisation (ACO) algorithm (Dorigo and Di Caro, 1999) for the Travelling Salesman Problem, and visualise its performance through Repast.

8.2 Some code

In the course web site you will find some Java code for this practical[1]. This code[2] should provide you with the basics for implementing an ACO simulation using Repast, namely: the basic model for the simulation (`TSAModel`[3]), a map (`TSAWorld`[4]) of the `TSACity`[5]s to be visited, backed by a `Object2DTorus`[6] and set by `TSAModel` to be displayed onto an `Object2DDisplay`[7]. The display includes methods for drawing the nodes (as green squares) the total amount of pheromone on each path/link (in white) and the best path (in red). The link drawing method is set so that the more pheromone a link contains the more clearly the link will be drawn.

You will also find a placeholder class for `TravellingSalesAnt`[8] which you will modify in order to implement the ACO meta-heuristic. The code contains documentation explaining how the simulation is structured.

The libraries you will need in order to compile and run this simulation are distributed under the repast3 release and have been included in this lab's tar file in their .jar packaging for convenience. They include simulation-specific packages (such as `repastj.jar`), CERN's scientific computing libraries (`colt.jar`), plotting libraries (`plot.jar`) and the trove high performance collections library.

[1]`http://www.scss.tcd.ie/~luzs/t/cs7032/sw/tsa-0.7.tar.gz`

[2]The package is named "sim.tsa", short for "Travelling Sales Ant", after a project by Armin Buch, a former student.

[3]sim.tsa.TSAModel

[4]sim.tsa.TSAWorld

[5]sim.tsa.TSACity

[6]uchicago.src.sim.space.Object2DTorus

[7]uchicago.src.sim.gui.Object2DDisplay

[8]sim.tsa.TravellingSalesAnt

Emacs/JDEE users will find a `prj.el`[9] file which can be used for compilation and running of the simulation under JDEE with a minor modification to the setting of `jde-global-classpath`.

Bourne shell scripts and Windows "batch files" have been included for compilation (`compile.sh` and `compile.bat`) and stand-alone running of the simulation (`run.sh`, `run.bat`). They refer only to libraries included in the tar file and should be easily translated into equivalent .bat files (i.e. by replacing colons by semi-colons and forward slashes by backslashes etc).

Although the above described code template is provided, you are free to implement your own ACO algorithm for TSP from scratch. You may even use a different simulation platform, such as MASON[10], which offers similar functionality to Repast.

8.3 Exercises

From the list of exercises below, please attempt exercise 1, plus **one** of the remaining exercises of your choice.

1. Implement the behaviour of a an active ant (see `newActiveAnt()` in the lecture notes) by modifying (implementing, in fact) methods step() and step2() in `TravellingSalesAnt.java`.

2. Modify the model so that the program allows you to save and load city maps (i.e. specific instances of TSP) so that you can compare the performance of the algorithm across several runs under different parameter settings. A database of symmetrical TSP (and related problems) and their best known tours can be found at the Univeity of Heidelberg's TSPLIB[11].

3. Experiment with different parameters settings (e.g. number of ants, distance weight, pheromone weight, amount of pheromone) and report briefly on how the algorithm performs.

4. `TSAModel` differs from the algorithm describe in class in that `TSAModel` currently updates pheromone only after all ants have finished their tour. Try modifying the code so that each ant leaves its pheromone trail immediately after it completes its tour. What effect does that have on the system?

5. Can you devise a heuristic for identifying unpromising solutions and dynamically tuning the ACO parameters in order to enable the algorithm to explore a wider solution space?

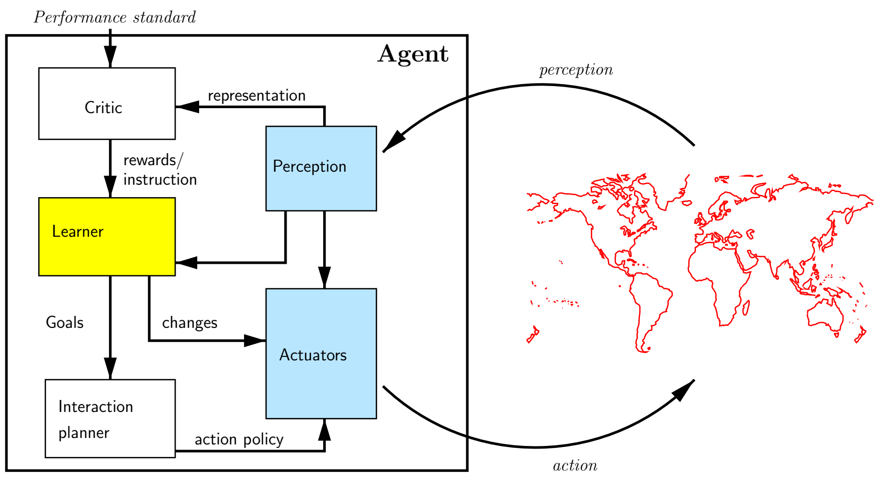
8.3.1 Delivering the assignment

Please submit this assignemt (as usual) through blackboard, by the deadline specified in the blackboard assignment submission page. Any text should be submitted either as plain text or in PDF format. No MSWord files, please. Code and text should be submitted as a single file in a compressed archive format (.tar.gz or .zip).

[9] sim/tsa/prj.el
[10] `http://cs.gmu.edu/~eclab/projects/mason/`
[11] `http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`

9 Learning Agents & Games: Introduction

Learning in agent architectures



Machine Learning for Games

- Reasons to *use* Machine Learning for Games:
  - Play against, and *beat human players* (as in board games, DeepBlue etc)
  - Minimise *development effort* (when developing AI components); avoid the *knowledge engineering bottleneck*
  - Improve the *user experience* by adding variability, *realism*, a sense that artificial characters evolve, etc.

Some questions

- What is *(Machine) Learning*?
- What *can* Machine Learning really *do* for us?

- What *kinds of techniques* are there?
- How do we *design* machine learning systems?
- What's different about *reinforcement learning*?
- Could you give us some *examples*?
  - YES:
  - *Draughts* (checkers)
  - *Noughts & crosses* (tic-tac-toe)

Defining "learning"

- ML has been studied from various perspectives (AI, control theory, statistics, information theory, ...)
- From an AI perspective, the general definition is formulated in terms of agents and tasks. E.g.:

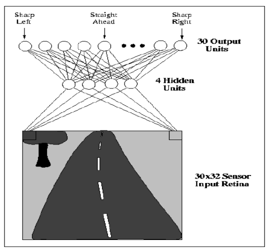
> [An agent] is said to *learn* from *experience* E with respect to some class of *tasks* T and *performance measure* P, if its performance at tasks in T, as measured by P, *improves* with E.

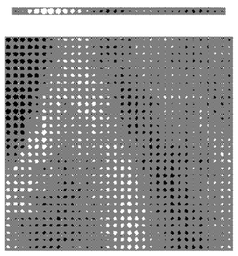
(Mitchell, 1997, p. 2)

- Statistics, model-fitting, ...

Some examples

- Problems too *difficult to program* by hand

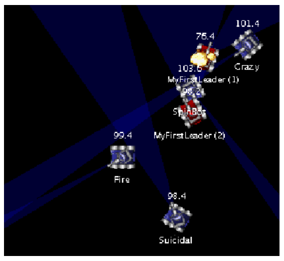




(ALVINN (Pomerleau, 1994))

Data Mining

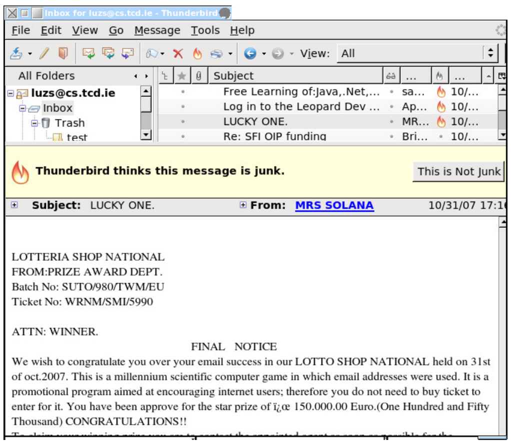
```
Name:      Corners     Name:      Corners     Name:      Corners
Bearing:   100         Bearing:   40          Bearing:   20
Velocity:  20          Velocity:  20          Velocity:  20
Energy:    30          Energy:    20          Energy:    20
Heading:   90          Heading:   90          Heading:   90
...                    ...                    ...
----------------------------------------------------------------->
t0                     t1                     t2               time
```



```
if Name = Corners & Energy < 25
then
   turn(91 - (Bearing - const)
   fire(3)
```

User interface agents

- *Recommendation* services,
- Bayes spam *filtering*
- JIT information *retrieval*


Inbox for luzs@cs.tcd.ie - Thunderbird
File Edit View Go Message Tools Help
View: All
All Folders
luzs@cs.tcd.ie
Inbox
Trash
test
Subject
This is Not Junk

```
Free Learning of:Java,.Net,...   sa...   10/...
Log in to the Leopard Dev ...    Ap...   10/...
LUCKY ONE.                       MR...   10/...
Re: SFI OIP funding              Bri...  10/...
```

Thunderbird thinks this message is junk.

Subject: LUCKY ONE. From: MRS SOLANA 10/31/07 17:1

LOTTERIA SHOP NATIONAL
FROM:PRIZE AWARD DEPT.
Batch No: SUTO/980/TWM/EU
Ticket No: WRNM/SMI/5990

ATTN: WINNER.

FINAL NOTICE

We wish to congratulate you over your email success in our LOTTO SHOP NATIONAL held on 31st of oct.2007. This is a millennium scientific computer game in which email addresses were used. It is a promotional program aimed at encouraging internet users; therefore you do not need to buy ticket to enter for it. You have been approve for the star prize of ï¿œ 150.000.00 Euro.(One Hundred and Fifty Thousand) CONGRATULATIONS!!

Designing a machine learning system

- Main design decisions:
  - *Training* experience: How will the system access and use *data*?
  - *Target function*: What exactly should be learned?
  - *Hypothesis representation*: How will we represent the *concepts* to be learnt?
  - Inductive *inference*: What specific *algorithm* should be used to learn the target concepts?

Types of machine learning

- How will the system be exposed to its *training experience*?
  - *Direct* or *indirect* access:

    - indirect access: *record* of past experiences, databases, corpora
    - direct access: *situated agents* $\rightarrow$ reinforcement learning
  - Source of feedback ("teacher"):
    - *supervised* learning
    - *unsupervised* learning
    - mixed: *semi-supervised* ("transductive"), *active learning*, ...

The hypothesis space

- The *data* used in the induction process need to be *represented uniformly.* E.g.:
  - representation of the opponent's behaviour as *feature vectors*
- The choice of representation constrains the space of available hypotheses (*inductive bias*).
- Examples of inductive bias:
  - assume that positive and negative instances can be separated by a (hyper) plane
  - assume that feature co-occurrence does not matter (*conditional independence assumption* by Naïve Bayes classifiers)
  - assume that the *current state* of the environment *summarises* environment history (*Markov property*)

Determining the target function

- The goal of the learning algorithm is to induce an *approximation* $\hat{f}$ of a target function f
- In *supervised learning*, the target function is assumed to be specified through *annotation of training data* or some form of *feedback.*
- Examples:
  - a *collection of texts* categorised by subject $f : D \times S \rightarrow \{0, 1\}$
  - a database of *past games*
  - user or *expert feedback*
- In *reinforcement learning* the agent will learn an *action selection policy* (as in $action : S \rightarrow A$)

Deduction and Induction

- Deduction: from general premises to a concludion. E.g.:
  - $\{A \rightarrow B, A\} \vdash B$
- Induction: from instances to generalisations

- Machine learning algorithms produce models that generalise from *instances* presented to the algorithm
- But all (useful) learners have some form of *inductive bias*:
  - In terms of *representation*, as mentioned above,
  - But also in terms of their preferences in *generalisation procedures*. E.g:
    - prefer simpler hypotheses, or
    - prefer shorter hypotheses, or
    - incorporate domain (expert) knowledge, etc etc

Given an function $\hat{f} : X \to C$ trained on a set of instances D_c describing a concept c, we say that the *inductive bias* of $\hat{f}$ is a minimal set of assertions B, such that for any set of instanced X:

$$\forall x \in X(B \wedge D_c \wedge x \vdash \hat{f}(x))$$

This should not be confused with *estimation bias*, which is a quantitity (rather than a set of assumptions) which quantifies the average "loss" due to misclassification. Together with *variance*, this quantitity determines the level of *generalisation error* of a learner (Domingos, 2012).

Choosing an algorithm

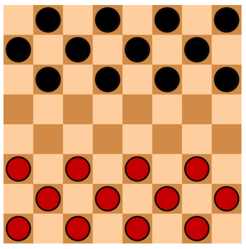
- Induction task as *search* for a *hypothesis* (or model) that fits the data and sample of the *target function* available to the learner, in a large space of hypotheses
- The choice of *learning algorithm* is conditioned to the choice of *representation*
- Since the target function is not completely accessible to the learner, the algorithm needs to operate under the *inductive learning assumption* that:

  an approximation that performs well over a *sufficiently large* set of instances will *perform well* on unseen data

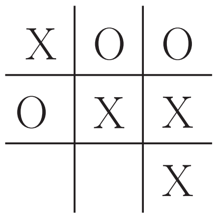
- *Computational Learning Theory* addresses this question.

Two Games: examples of learning

- *Supervised* learning: draughts/checkers (Mitchell, 1997)



- *Reinforcement* learning: noughts and crosses (Sutton and Barto, 1998)



- Task? (*target function*, data *representation*) *Training* experience? *Performance* measure?

A target for a draughts learner

- Learn.... $f : Board \rightarrow Action$ or $f : Board \rightarrow \mathbb{R}$
- But how do we label (evaluate) the *training experience*?
- Ask an expert?
- Derive values from a rational strategy:
  - if b is a final board state that is *won*, then $f(b) = 100$
  - if b is a final board state that is *lost*, then $f(b) = -100$
  - if b is a final board state that is *drawn*, then $f(b) = 0$
  - if b is a *not a final state* in the game, then $f(b) = f(b')$, where b' is the best final board state that can be achieved starting from b and *playing optimally* until the end of the game.
- How feasible would it be to implement these strategies?
  - Hmmmm... *Not feasible...*

Hypotheses and Representation

- The choice of *representation* (e.g. logical formulae, decision tree, neural net architecture) constrains the *hypothesis* search space.
- A representation scheme: linear combination of board *features*:

$$\begin{aligned}\hat{f}(b) \quad = \quad & w_0 + w_1 \cdot bp(b) + w_2 \cdot rp(b) + w_3 \cdot bk(b) \\ & + w_4 \cdot rk(b) + w_5 \cdot bt(b) + w_6 \cdot rt(b)\end{aligned}$$

- where:
  - $bp(b)$: number of black pieces on board b
  - $rp(b)$: number of red pieces on b
  - $bk(b)$: number of black kings on b
  - $rk(b)$: number of red kings on b
  - $bt(b)$: number of red pieces threatened by black
  - $rt(b)$: number of black pieces threatened by red

Training Experience

- Some notation and distinctions to keep in mind:
  - $f(b)$: the true *target function*
  - $\hat{f}(b)$: the *learnt function*
  - $f_{train}(b)$: the *training value* (obtained, for instance, from a training set containing instances and its corresponding training values)
- Problem: How do we *obtain* training values?
- A *simple rule* for obtaining (*estimating*) training values:
  - $f_{train}(b) \leftarrow \hat{f}(Successor(b))$

Remember that what the agent really needs is an action *policy*, $\pi : B \rightarrow A$, to enable it to choose a an action a given a board configuration b. We are assuming that the agent implements a policy $\pi(s)$ which selects the action that maximises the value of the successor state of b, $f(b')$.

How do we learn the weights?

Algorithm 9.1: Least Means Square

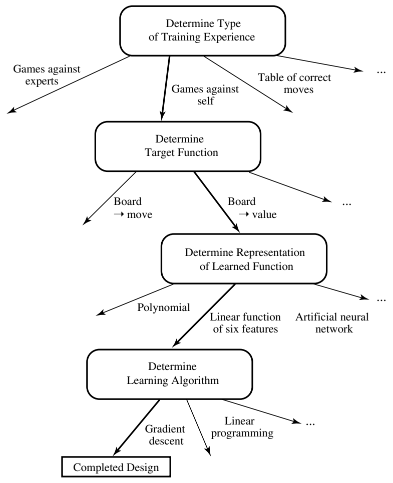
```
LMS(c: learning rate)
  for each training instance $< b, f_{train}(b) >$
    do
      compute $error(b)$ for current approximation
      (i.e. using current weights):
            $error(b) = f_{train}(b) - \hat{f}(b)$
      for each board feature $t_i \in \{bp(b), rp(b), \ldots\}$,
        do
          update weight $w_i$:
            $w_i \leftarrow w_i + c \times t_i \times error(b)$
        done
    done
```

LMS minimises the squared error between training data and current approx.:$E \equiv \sum_{\langle b, f_{train}(b)\rangle \in \mathcal{D}} (f_{train}(b) - \hat{f}(b))^2$ Notice that if $error(b) = 0$ (i.e. target and approximation match) no weights change. Similarly, if or $t_i = 0$ (i.e. feature t_i doesn't occcur) the corresponding weight doesn't get updated. This weight update rule can be shown to perform a gradient descent search for the minimal squared error (i.e. weight updates are proportional to $-\nabla E$ where $\nabla E = [\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots])$.

That the LMS weight update rule implements gradient descent can be seen by differentiating ∇E:

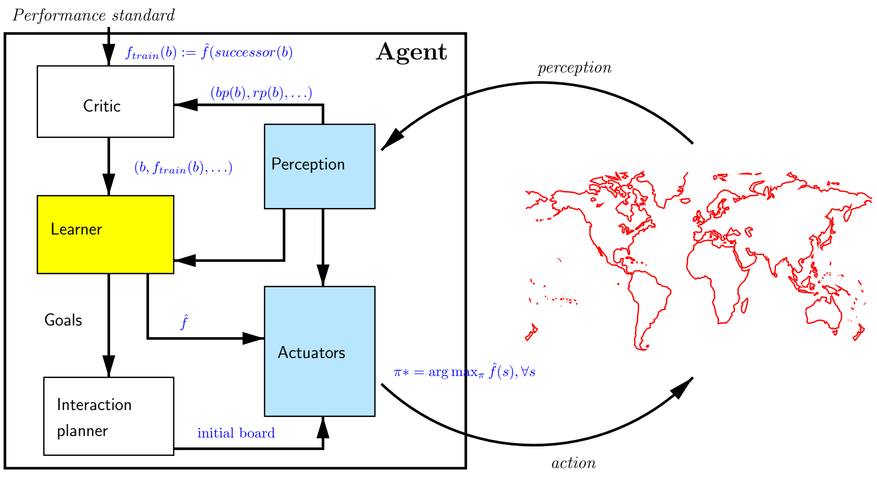
$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial \sum [f(b) - \hat{f}(b)]^2}{\partial w_i} \\
&= \frac{\sum \partial [f(b) - \hat{f}(b)]^2}{\partial w_i} \\
&= \sum 2 \times [f(b) - \hat{f}(b)] \times \frac{\partial}{\partial w_i}[f(b) - \hat{f}(b)] \\
&= \sum 2 \times [f(b) - \hat{f}(b)] \times \frac{\partial}{\partial w_i}[f(b) - \sum_{i}^{|\mathcal{D}|} w_j t_j] \\
&= -\sum 2 \times error(b) \times t_j
\end{aligned}$$

Design choices: summary


Determine Type of Training Experience
Games against experts
Games against self
Table of correct moves
...
Determine Target Function
Board → move
Board → value
...
Determine Representation of Learned Function
Polynomial
Linear function of six features
Artificial neural network
...
Determine Learning Algorithm
Gradient descent
Linear programming
...
Completed Design

(from (Mitchell, 1997)) These are some of the decisions involved in ML design. A number of other practical factors, such as evaluation, avoidance of "overfitting", feature engineering, etc. See (Domingos, 2012) for a useful introduction, and some machine learning "folk wisdom".

The Architecture instantiated



Reinforcement Learning

- What is *different* about reinforcement learning:
  - Training experience (data) obtained through *direct interaction* with the environment;
  - Influencing the *environment*;
  - *Goal-driven* learning;
  - Learning of an *action policy* (as a first-class concept)
  - Trial and error approach to search:
    - *Exploration* and *Exploitation*

Basic concepts of Reinforcement Learning

- The *policy*: defines the learning agent's way of behaving at a given time:

$$\pi : S \to A$$

- The (immediate) *reward function*: defines the goal in a reinforcement learning problem:

$$r : S \to \mathbb{R}$$

often identified with timesteps: $r_0, \ldots, r_n \in \mathbb{R}$

- The (long term) *value function*: the total amount of reward an agent can expect to accumulate over the future:

$$V : S \to \mathbb{R}$$

- A *model* of the environment

Theoretical background

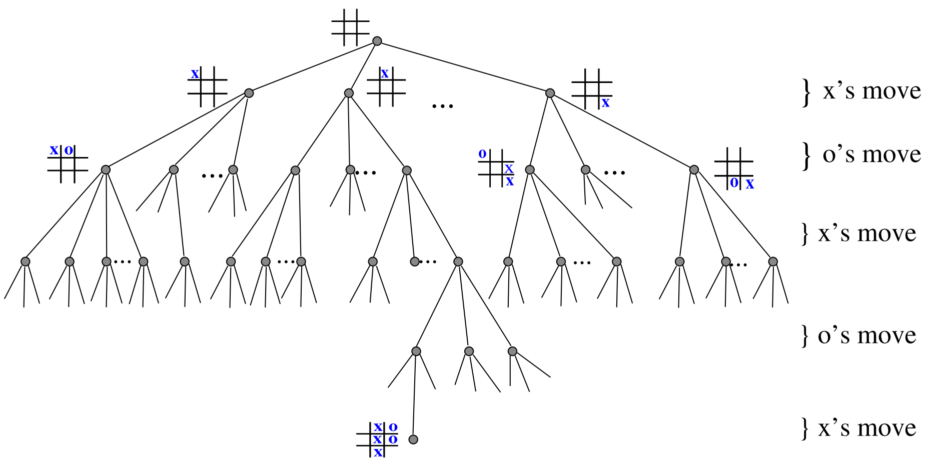
- *Engineering*: "optimal control" (dating back to the 50's)
  - Markov Decision Processes (*MDPs*)
  - Dynamic programming
- *Psychology*: learning by trial and error, animal learning. Law of effect:
  - learning is *selectional* (genetic methods, for instance, are selectional, but not associative) and
  - *associative* (supervised learning is associative, but not selectional)
- *AI*: TD learning, Q-learning

Law of effect: "Of several responses made to the same situation, those which are accompanied or closely followed by satisfaction to the animal will, other things being equal, be more firmly connected with the situation, so that, when it recurs, they will be more likely to recur; those which are accompanied or closely followed by discomfort to the animal will, other things being equal, have their connections with that situation weakened, so that, when it recurs, they will be less likely to occur. The greater the satisfaction or discomfort, the greater the strengthening or weakening of the bond." (Thorndike, 1911, p. 244, quoted by (Sutton and Barto, 1998))

The *selectional* aspect means that the learner will select from a large pool of complete policies, overlooking individual states (e.g. the contribution of a particular move to winning the game).

The *associative* aspect means that the learner associates action (move) to a value but does not select (or compare) policies as a whole.

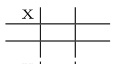
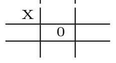
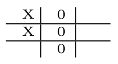
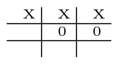
Example: Noughts and crosses



Possible solutions: *minimax* (assume a perfect opponent), *supervised learning* (directly search the space of policies, as in the previous example), *reinforcement learning* (our next example).

A Reinforcement Learning strategy

- Assign *values* to *each possible game state* (e.g. the probability of winning from that state):

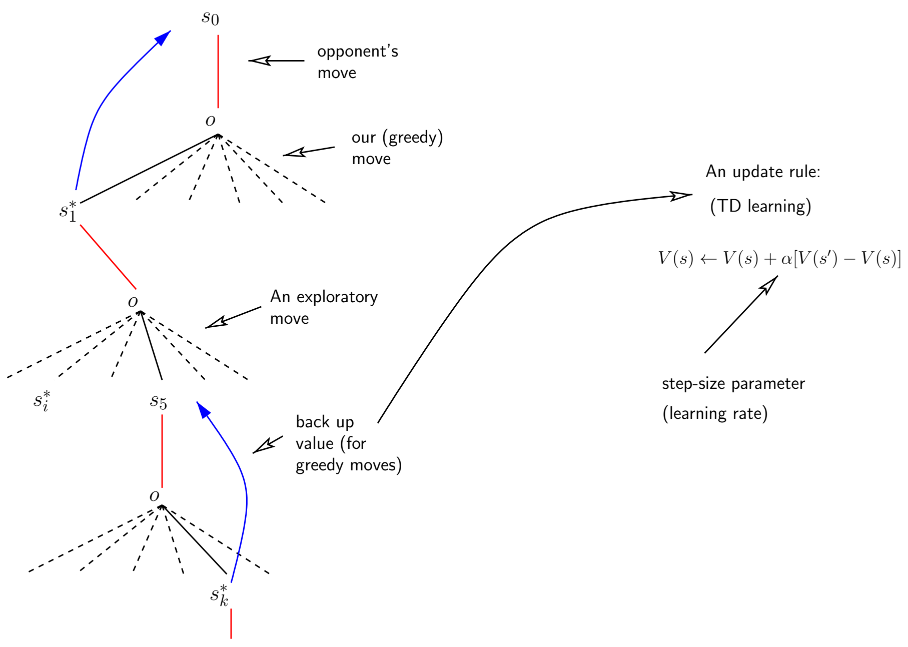
state	$V(s)$	outcome
$s_0 =$ 	0.5	??
$s_1 =$ 	0.5	??
⋮		
$s_i =$ 	0	loss
⋮		
$s_n =$ 	1	win

Algorithm 9.2: TD Learning

```
While learning
   select move by
      looking ahead 1 state
      choose next state $s$:
        if\= exploring
           pick $s$ at random
        else
           $s = \arg\max_s V(s)$
```

N.B.: *exploring* could mean, for instance, pick a *random next state* 10% of the time.

How to update state values



Some nice properties of this RL algorithm

- For a *fixed opponenent*, if the parameter that controls learning rate (α) is reduced properly over time, *converges to the true probabilities of winning* from each state (yielding an optimal policy)
- If α isn't allowed to reach zero, the system will *play well against opponents that alter their game* (slowly)
- Takes into account what happens *during* the game (unlike supervised approaches)

What was not illustrated

- RL also applies to *situations where there isn't a clearly defined adversary* ("games against nature")
- RL also applies to *non-episodic problems* (i.e. rewards can be received at any time not only at the end of an episode such as a finished game)
- RL *scales up well* to games where the search space is (unlike our example) truly vast.
  - See (Tesauro, 1994), for instance.
- *Prior knowledge* can also be incorporated
- *Look-ahead* isn't always required

10 Practical: Survey of game AI competitions

10.1 Goals

To learn about a Game AI competition (preferably one the is co-located with an AI or games conference). Give a short presentation describing the competition. The final course project will involve creating an entry for a competition to be chosen among the ones presented in class. The presentations can be done by groups of 3 students.

10.2 What to do

1. Start by exploring some game-AI conference websites, such as the The IEEE Conference on Computational Intelligence and Games[1], The IEEE World Congress on Computational Intelligence[2], or the AI and Interactive Digital Entertainment Conference[3], to see which competitions they host, how often they are held, requirements for participation etc. There are also web sites that compile lists of such competitions.

2. Choose a game competition to present, and email me your choice and the composition of your group. Groups should present different game competitions, so choices will be recorded more or less on a 1st-come-1st-served basis.

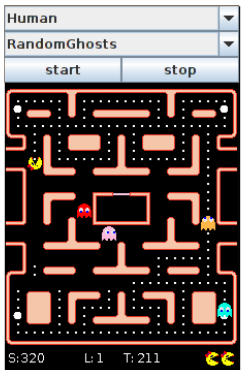
3. Below is a list of possible games you might like to take a look at. Bear in mind that the deadlines for most competitions have already passed, so your survey will basically look at past competitions (including the papers written by participants on the techniques they used). The expectation is that the competition will run again in 2014. The list below is by no means eshaustive, and is probably a bit out of date, so please feel free to search for Game AI competitions (here's a sample search[4]).

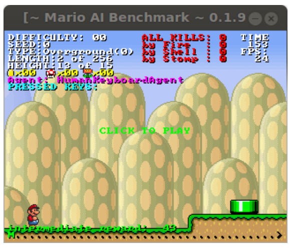
[1] `https://duckduckgo.com/?q=IEEE+Conference+on+Computational+Intelligence+and+Games`

[2] `http://ieee-wcci2014.org/`

[3] `http://aiide.org`

[4] `https://duckduckgo.com/?q=Game+AI+competitions`





- AI Birds.org: an Angry Birds AI Competition[5] heal at at the International Joint Conference on AI (IJCAI).
- Simulated Car Racing Championship[6]: a competition based on the open racing car simulator (TORCS).
- MarioAI Competition[7]: A tribute to the famous game, consisting of different tracks: game play, level generation and Turing test. Each track could count as a separate competition, so different groups could present different tracks.
- The The 2K BotPrize[8]: a competition based on UT2004 and the agent development toolkit POGAMUT.
- The Ms Pac-Man vs Ghost Team Competition[9]: another tribute, in Java. There are two tracks which can be described separately.
- The StarCraft RTS AI Competition[10]: based on the popular real time strategy game.
- ...

10.2.1 Presentation

You will give a short (< 5 minute) presentation describing your chosen competition next Tuesday. Your presentation could include, for instance, a bried demo of the game showing a sample agent (most AI game platforms include a "baseline" sample agent) playing the game.

Since the main Course Project's topic will be based on one of these game competitions, your presentation should also include details on:

1. Which platforms (Linux, Windows, MacOS, etc) the game software runs on,
2. Which programming languages are supported for agent development,
3. Whether there are additional software requirements or constraints, licensing issues (e.g. is all the software required open-source or are there proprietary software that need to be installed?) third-party libraries etc,
4. How difficult it is to get the game server and development environment set up and ready for agent development,

[5]http://aibirds.org/
[6]http://scr.geccocompetitions.com/
[7]http://www.marioai.org
[8]http://www.botprize.org/
[9]http://www.pacman-vs-ghosts.net/
[10]http://webdocs.cs.ualberta.ca/~cdavid/starcraftaicomp/

5. Your assessment of how interesting the game/competition is in terms of allowing us to explore AI techniques,
6. Whether there will be a competition next year (so we can perhaps enter it!)

Many of the game competitions above have been described in journal and conference papers (e.g. (Karakovskiy and Togelius, 2012), (Genesereth et al., 2005), (Shaker et al., 2011), (Loiacono et al., 2010), (Rohlfshagen and Lucas, 2011)). There are also a number of papers describing various AI approaches employed by participants in past competitions (e.g. these[11]). Feel free to explore.

[11] `http://aibirds.org/symposium-on-ai-in-angry-birds/accepted-papers.html`

11 Evaluative feedback

Learning and Feedback

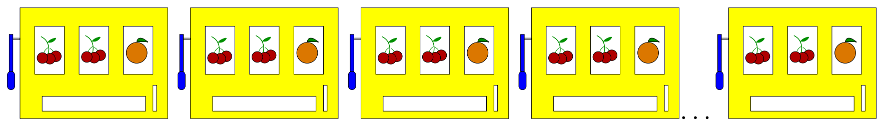
- Consider the *ACO algorithm*: how does the system *learn*?
- *Contrast* that form of "learning" with, say, our system that learns to play *draughts*, or to a system that learns to filter out *spam mail*.
- The RL literature often contrasts *instruction* and *evaluation*.
- *Evaluation* is a key component of Reinforcement learning systems:
  - Evaluative feedback is *local* (it indicates how good an action is) but not whether it is the best action possible
  - This creates a need for *exploration*

Associative vs. non-associative settings

- In general, learning is both
  - *selectional*: i.e. actions are selected by trying different alternatives and comparing their effects,
  - *associative*: i.e. the actions selected are associated to particular situations.
- However, in order to study evaluative feedback in detail it is convenient to *simplify things* and consider the problem from a *non-associative* perspective.

Learning from fruit machines

- The *n-armed bandit* setting:



- Choice of n actions (which yield numerical rewards drawn from a stationary probability distribution)
  - each action selection called a *play*
- Goal: *maximise* expected (long term) *total reward* or *return*.
  - Strategy: concentrate *plays* on the *best levers*.

How to find the best plays

- Let's *distinguiguish* between:
  - *reward*, the immediate outcome of a play, and
  - *value*, the expected (mean) reward of a play
- But how do we *estimate* values?
- We could *keep a record of rewards* $r_1^a, \ldots, r_{k_a}^a$ for each chosen action a and estimate the value of choosing a at time t as

$$Q_t(a) = \frac{r_1^a + r_2^a + \cdots + r_{k_a}^a}{k_a} \tag{11.1}$$

But how do we choose an action?

- We could be *greedy* and *exploit* our knowledge of $Q_t(a)$ to choose at any time t the play with the highest estimated value.
- But this strategy will tend to neglect the estimates of non-greedy actions (which might have produced greater total reward in the long run).
- One could, alternatively, *explore* the space of actions by choosing, from time to time, a non-greedy action

Balancing exploitation and exploration

- The goal is to approximate the *true value*, $Q(a)$ for each action
- $Q_t(a)$, given by equation (11.1) will *converge* to $Q(a)$ as $k_a \to \infty$ (Law of Large Numbers)
- So balancing exploration and exploitation is necessary...
- but *finding the right balance can be tricky*; many approaches rely on strong assumptions about the underlying distributions
- We will present a simpler alternative...

A simple strategy

- The *simplest strategy*: choose action a^* so that:

$$a^* = \arg\max_a Q_t(a) \qquad \text{(greedy selection)} \qquad (11.2)$$

- A better simple strategy: ϵ-greedy methods:
  - With probability $1 - \epsilon$, exploit; i.e. *choose* a^* according with (11.2)
  - The rest of the time (probability ϵ) choose an action at *random*, uniformly
  - A suitably *small* ϵ guarantees that all actions get *explored sooner or later* (and the *probability of selecting the optimal action* converges to greater than $1 - \epsilon$)

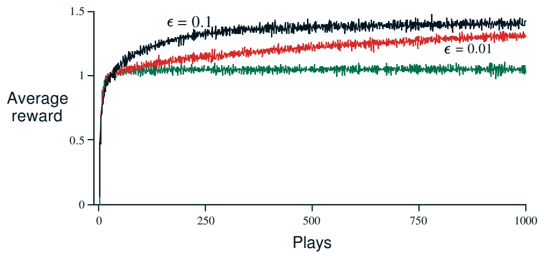
Exploring exploration at different rates

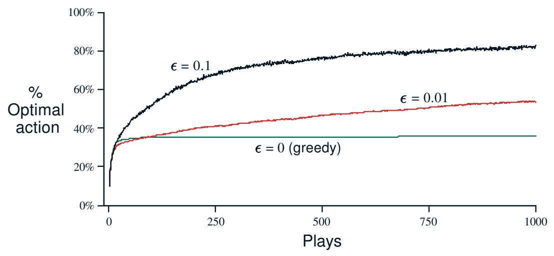
- The "10-armed testbed" (Sutton and Barto, 1998):
  - *2000 randomly generated* n-armed bandit tasks with $n = 10$.
  - For each action, a, expected *rewards* $Q(a)$ (the "true" expected values of choosing a) are selected from a normal (Gaussian) probability distribution $N(0, 1)$
  - Immediate rewards $r^a_1, \ldots, r^a_k$ are similarly selected from $N(Q(a), 1)$

```
nArmedBanditTask <- function(n=10, s=2000)
  {
    qtrue <- rnorm(n)
    r <- matrix(nrow=s,ncol=n)
    for (i in 1:n){
      r[, i] <- rnorm(s,mean=qtrue[i])
    }
    return(r)
  }
```

Some empirical results

- Averaged results of 2000 rounds of 1000 plays each:





Softmax methods

- *Softmax action selection* methods grade action probabilities by estimated values.
- Commonly use *Gibbs, or Boltzmann, distribution.* I.e. choose action a on the t-th play with probability

$$\pi(a) = \frac{e^{Q_t(a)/\tau}}{\sum_{b \in A} e^{Q_t(b)/\tau}} \tag{11.3}$$

- where τ is a positive parameter called the *temperature*, as in *simulated annealing* algorithms.

High temperatures cause the actions to be nearly equiprobable. Low temperatures cause a greater difference in selection probability for actions that differ in their value estimates. Softmax action selection becomes the same as greedy action selection as $\tau \to 0$.

Keeping track of the means

- Maintaining a record of means by recalculating them after each play can be a source of inefficiency in evaluating feedback. E.g.:

```
## inefficient way of selecting actions
selectAction <- function(r, e=0) {
  n <- dim(r)[2] # number of arms (cols in r)
  if (sample(c(T,F),1,prob=c(e,1-e))) {
    return(sample(1:n,1)) # explore if e > 0 and chance allows
  }
  else { # all Q_t(a_i), i \in [1,n]
    Qt <- sapply(1:n, function(i)mean(r[,i], na.rm=T))
    ties <- which(Qt == max(Qt))
    ## if there are ties in Q_t(a); choose one at random
    if (length(ties)>1)
      return(sample(ties, 1))
    else
      return(ties)
  }
}
```

Incremental update

- Do we really need to store all rewards? NO
- Let Q_k be the mean of the first k rewards for an action,
- We can express the next mean value as

$$\begin{aligned} Q_{k+1} &= \frac{1}{k+1}\sum_{i=1}^{k+1} r_i \\ &= \ldots \\ &= Q_k + \frac{1}{k+1}[r_{k+1} - Q_k] \end{aligned} \tag{11.4}$$

- Equation (11.4) is part of family of formulae in which the *new estimate* (Q_{k+1}) is the old estimate (Q_k) adjusted by the *estimation error* ($r_{k+1} - Q_k$) scaled by a *step parameter* ($\frac{1}{k+1}$, in this case)
- (Recall the LMS weight update step described in the introduction to machine learning).

The complete derivation of (11.4) is:

$$\begin{aligned} Q_{k+1} &= \frac{1}{k+1}\sum_{i=1}^{k+1} r_i \\ &= \frac{r_{k+1} + \sum_{1=1}^{k} r_i}{k+1} \\ &= \frac{r_{k+1} + kQ_k}{k+1} \\ &= \frac{r_{k+1} + kQ_k + Q_k - Q_k}{k+1} \\ &= \frac{r_{k+1} - Q_k}{k+1} + \frac{Q_k(k+1)}{k+1} \\ &= Q_k + \frac{1}{k+1}[r_{k+1} - Q_k] \end{aligned}$$

Non-stationary problems

- What happens if the “n-armed bandit” changes over time? How do we keep track of a changing mean?
- An approach: weight recent rewards more heavily than past ones.
- So, our update formula (11.4) could be modified to

$$Q_{k+1} = Q_k + \alpha[r_{k+1} - Q_k] \tag{11.5}$$

  where $0 < \alpha \leq 1$ is a constant
- This gives us Q_k as an *exponential, recency-weighted average* of past rewards and the initial estimate

Recency weighted estimates

- Given (11.5), Q_k can be rewritten as:

$$\begin{aligned} Q_k &= Q_{k-1} + \alpha[r_k - Q_{k-1}] \\ &= \alpha r_k + (1-\alpha)Q_{k-1} \\ &= \dots \\ &= (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i \end{aligned} \tag{11.6}$$

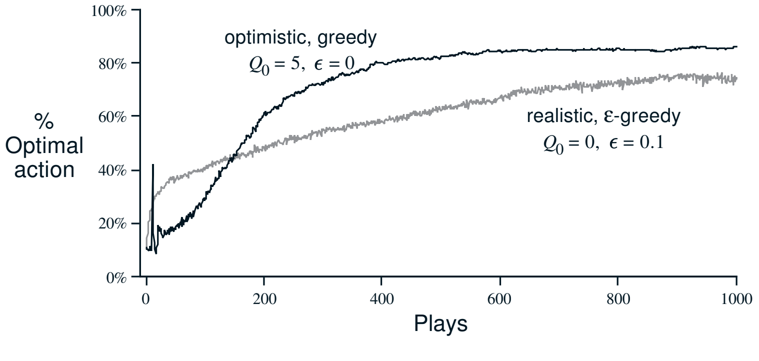
- Note that $(1-\alpha)^k + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} = 1$. I.e. weights sum to 1.
- The weight $\alpha(1-\alpha)^{k-i}$ decreases exponentially with the number of intervening rewards

Picking initial values

- The above methods are *biased* by Q_0
  - For sample-average, bias disappears once all actions have been selected
  - For recency-weighted methods, bias is *permanent* (but decreases over time)
- One can supply *prior knowledge* by picking the right values for Q_0
- One can also choose *optimistic initial values* to encourage exploration *(Why?)*

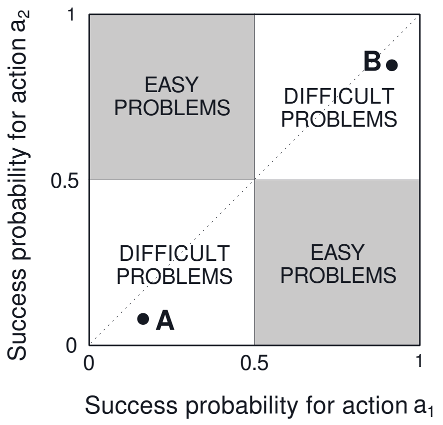
The effects of optimism

- With Q_0's set initially high (for each action)q, the learner will be *disappointed* by actual rewards, and sample all actions many times before converging.
- E.g.: Performance of *Optimistic* $(Q_0 = 5, \forall a)$ *vs Realistic* $(Q_0 = 0, \forall a)$ strategies for the 10-armed testbed



Evaluation vs Instruction

- RL searches the *action space* while SL searchs the *parameter space.*
- *Binary*, 2-armed bandits:
  - two actions: a_1 and a_2
  - two rewards: *success* and *failure* (as opposed to numeric rewards).
- *SL*: select the *action* that returns *success most often.*
- Stochastic case (a *problem* for SL?):



Two stochastic SL schemes (learning automata)

- $L_{r\text{-}p}$, *Linear reward-penalty*: choose the action as in the naive SL case, and keep track of the successes by updating an estimate of the probability of choosing it in future.
- Choose $a \in A$ with *probability* $\pi_t(a)$
- Update the probability *of the chosen action* as follows:

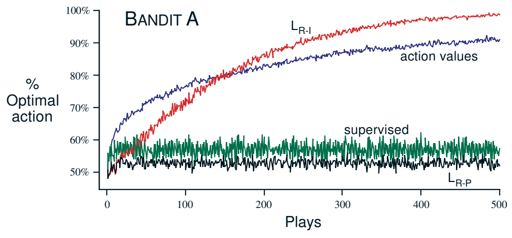
$$\pi_{t+1}(a) = \begin{cases} \pi_t(a) + \alpha(1 - \pi_t(a)) & \text{if success} \\ \pi_t(a)(1-\alpha) & \text{otherwise} \end{cases} \tag{11.7}$$

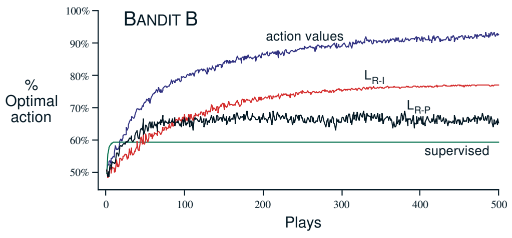
- All *remaining actions* $a_i \neq a$ are adjusted proportionally:

$$\pi_{t+1}(a_i) = \begin{cases} \pi_t(a_i)(1-\alpha) & \text{if } a \textit{ succeeds} \\ \frac{\alpha}{|A|-1} + \pi_t(a_i)(1-\alpha) & \text{otherwise} \end{cases} \tag{11.8}$$

- $L_{r\text{-}i}$, *Linear reward-inaction*: like $L_{r\text{-}p}$ but update probabilities *only in case of success.*

Performance: instruction vs evaluation



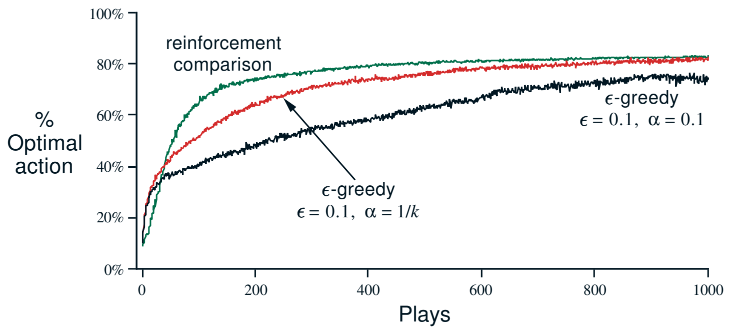


- See (Narendra and Thathachar, 1974), for a survey of learning automata methods and results.

Reinforcement comparison

- Instead of estimates of values for each action, keep an estimate of overall reward level $\bar{r}_t$:
$$\bar{r}_{t+1} = \bar{r}_t + \alpha[r_{t+1} - r_t] \tag{11.9}$$
- and *action preferences* $p_t(a)$ w.r.t. reward estimates:
$$p_{t+1}(a) = p_t(a) + \beta[r_t - \bar{r}_t] \tag{11.10}$$
- The *softmax function* can then be used as a PMF for action selection:
$$\pi_t(a) = \frac{e^{p_t(a)}}{\sum_{b \in A} e^{p_t(b)}} \tag{11.11}$$
- ($0 < \alpha \leq 1$ and $\beta > 0$ are step-size parameters.)

How does reinforcement comparison compare?

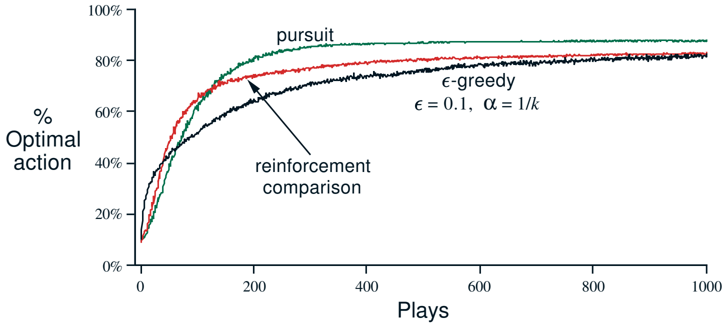


- Reinforcement comparison ($\alpha = 0.1$) vs action-value on the 10-armed testbed.

Pursuit methods

- maintain both action-value estimates and action preferences,
- preferences continually "pursue" greedy actions.
- E.g.: for greedy action $a^* = \arg\max_a Q_{t+1}(a)$, increase its selection probability:
$$\pi_{t+1}(a^*) = \pi_t(a^*) + \beta[1 - \pi_t(a^*)] \tag{11.12}$$
- while decreasing probabilities for all remaining actions $a \neq a^*$
$$\pi_{t+1}(a) = \pi_t(a) + \beta[0 - \pi_t(a)] \tag{11.13}$$

Performance of pursuit methods



- Pursuit ($\alpha = 1/k$ for Q_t update, $\pi_0(a) = 1/n$ and $\beta = 0.01$) versus reinforcement comparison ($\alpha = 0.1$) vs action-value on the 10-armed testbed.

Further topics

- Associative search:
  - suppose we have many different bandit tasks and the learner is presented with a different one at each play.
  - Suppose each time the learner is given a clue as to the which task it is facing...
  - In such cases, the best strategy is a combination of *search* for the best actions and *association* of actions to situations
- Exact algorithms for computation of *Bayes optimal* way to balance exploration and exploitation exist (Bellman, 1956), but are intractable.

12 Practical: Evaluating Evaluative Feedback

12.1 Goals

To replicate the experiments with the 10-armed testbed (Sutton and Barto, 1998) presented in class[1]. To investigate different settings of the exploration vs exploitation dilemma. And to model a simple game as a MDP. The MDP modelling part is unrelated to the R code given.

12.2 Some code

In the course web site you will find some code for this practical[2]. This file contains functions that create an n-armed bandit task (`makeNArmedBanditTask`) and run the ϵ-greedy experiment described in the notes (`runExperiment1`).

The code is written in R, a programming environment for data analysis and graphics which is freely available from the R project website[3] (Venables and Smith, 2004). R has an Algol-like syntax which should make it relatively easy to learn the basics. The code provided does not use any of the advanced features of the language, so it should be straightforward to extend and modify it. A short introduction to R[4] is available through the course website. Only question 2 requires some coding. However, you may use a programming language of your choice, or simply answer question 2 using pseudo-code.

The R code provided implements the basics of evaluative feedback as described in our lecture notes:

- function `makeNArmedBanditTask(n,s)` creates a matrix of `s` by `n` randomly sampled from normal distributions which represents `s` plays on `n` machines ("arms")

- function `evalLearn(dset, e, tmax, q0)` "learns" to evaluate feedback for `dset` (e.g. a 10-armed dataset generated as described above) by iterating up to `tmax` times, with greed factor `e`.

[1] `http://www.scss.tcd.ie/~luzs/t/cs7032/1/efeedback-notes.pdf`
[2] `http://www.scss.tcd.ie/~luzs/t/cs7032/1/sw/evfeedback.R`
[3] `http://cran.r-project.org/`
[4] `http://www.scss.tcd.ie/~luzs/t/cs7032/1/bib/R-intro.pdf`

- `selectAction(avgr, e)` selects an action that gives maximal reward or an exploratory action, depending on the "greed" parameter `e` and `updateEstimate(old,target,step)` updates the average rewards efficiently.

In addition, a sample function `runExperiment1(...)` is provided which uses the above described code to run the first evaluative feedback simulation described in the notes and plot the results.

12.3 Exercises

Do the following exercises, to be handed in next Tuesday:

1. run the initial evaluative feedback experiment for different settings of the "greed" parameter (ϵ), number of runs etc. Plot and compare the results. NB: `\runExperiment1()` does all that is needed, all you have to do is pass different parameters.

2. Extend or modify the R code given (or re-implement it in your language of choice) so as to run at least one of the other settings described in the notes: softmax action selection, optimistic initial estimates, reinforcement comparison, etc. Alternatively (if you are not comfortable enough with R) you can simply describe the algorithm in pseudo-code.

3. Describe the robocode game as a Markov Decision process along the lines seen in the last lectures. In particular, outline a scheme for describing:

   - the battle environment as a set of states (note that you will typically need to simplify the state to a few variable, as we illustrated, for instance, in our simplification of the draughts game).
   - actions and
   - a reward structure (for a robocode learner)

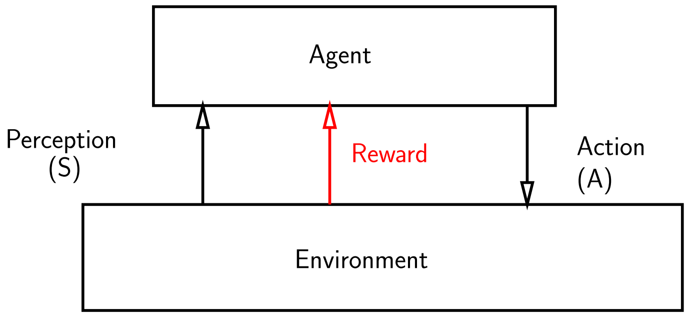
13 Learning Markov Decision Processes

Outline

- Reinforcement Learning problem as a *Markov Decision Process* (MDP)
- Rewards and returns
- Examples
- The *Bellman Equations*
- Optimal *state-* and *action-value functions* and Optimal Policies
- *Computational* considerations

The Abstract architecture revisited (yet again)

- Add the ability to *evaluate* feedback:

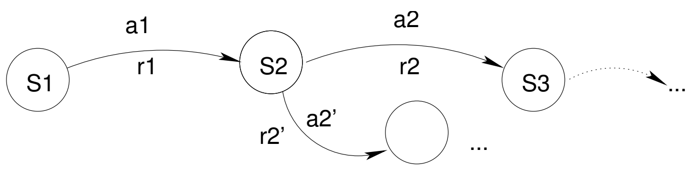


- How to represent *goals*?

Interaction as a Markov decision process

- We start by *simplifying action* (as in purely reactive agents):
  - $action : S \to A$ (**New notation: action* $\overset{def}{=} \pi$)

  - $env : S \times A \to S$ (*New notation:* $env \overset{def}{=} \delta$)
- at each discrete time agent *observes state* $s_t \in S$ and *chooses action* $a_t \in A$
- then *receives* immediate *reward* r_t
- and *state changes* to s_{t+1} (deterministic case)



Levels of abstraction

- *Time* steps *need not be fixed* real-time intervals.
- *Actions* can be *low level* (e.g., voltages to motors), or *high level* (e.g., accept a job offer), *mental* (e.g., shift in focus of attention), *etc.*
- *States* can be low-level *sensations*, or they can be *abstract*, symbolic, based on memory, or subjective (e.g., the state of being surprised or lost).
- An RL agent is *not like a whole animal* or *robot.*
  - The *environment* encompasses everything the agent *cannot change arbitrarily.*
- The *environment* is not necessarily unknown to the agent, only *incompletely controllable.*

Specifying goals through rewards

- The *reward hypothesis* (Sutton and Barto, 1998, see):

  $\boxed{\text{All of what we mean by \textit{goals} and purposes can be well thought of \textit{as the maximization of the cumulative sum} of a received scalar signal (\textit{reward}).}}$

- Is this *correct*?
- Probably not: but *simple*, surprisingly flexible and *easily disprovable*, so it makes scientific sense to explore it before trying anything more complex.

Some examples

- Learning to play a game (e.g. *draughts*):
  - +1 for *winning*, −1 for *losing*, 0 for *drawing*
  - (similar to the approach presented previously.)
- Learning *how to escape* from a maze:

  - set the reward to *zero until it escapes*
  - and $+1$ when it does.

- Recycling robot: $+1$ *for each recyclable container* collected, -1 *if container isn't recyclable*, 0 for wandering, -1 for bumping into obstacles etc.

Important points about specifying a reward scheme

- the reward signal *is* the place to specify *what the* agent's *goals are* (given that the agent's high-level goal is always to maximise its rewards)
- the reward signal *is not* the place to specify *how* to achieve such goals
- *Where* are *rewards computed* in our agent/environment diagram?
- Rewards and goals are *outside the agent's direct control*, so they it makes sense to assume they are *computed by the environment*!

From rewards to returns

- We define *(expected) returns* (R_t) to formalise the notion of rewards received in the long run.
- The simplest case:

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T \tag{13.1}$$

  where $r_{t+1}, \ldots$ is the *sequence of rewards* received *after* time t, and T is the final time step.

- What sort of *agent/environment* is this definition most *appropriate* for?
- Answer: *episodic* interactions (which *break* naturally *into subsequences*; e.g. a game of chess, trips through a maze, etc).

Non-episodic tasks

- Returns should be defined differently for *continuing* (aka *non-episodic*) tasks (i.e. $T = \infty$).
- In such cases, the idea of *discounting* comes in handy:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{13.2}$$

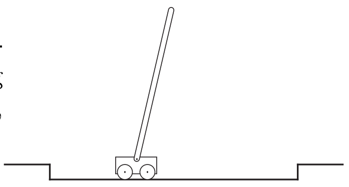
  where $0 \leq \gamma \leq 1$ is the *discount rate*

- Is this sum *well defined*?
- One can thus specify *far-sighted* or *myopic* agents by *varying the discount rate* γ.

If the rewards r_k are bounded from above by some constant c and $\gamma < 1$, then the infinite sum in (13.2) is well defined, since it is an infinite sum of a sequence bounded in absolute value by a decreasing geometric prograssion γc.

The pole-balancing example

- Task: keep the pole *balanced* (beyond a critical angle) *as long as possible*, without hitting the ends of the track (Michie and Chambers, 1968)



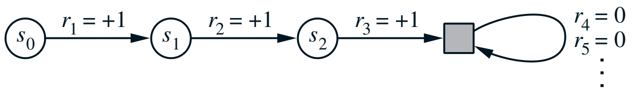
- Modelled as an *episodic task*:
  - reward of +1 *for each step* before failure ⇒ R_t = *number of steps* before failure
- Can *alternatively* be modelled *as a continuing task*:
  - "reward" of -1 *for failure* and 0 for other steps ⇒ $R_t = -\gamma^k$ for k steps before failure

What would happen if we treated pole-balancing as an episodic task but also used discounting, with all rewards zero except for -1 upon failure?

What would be the likely outcome of the maze task described in slide 185 if it were treated as an episodic task?

Episodic and continuing tasks as MDPs

- Extra formal *requirements* for describing *episodic and continuing* tasks:
  - need to *distinguish episodes* as well as *time steps* when referring to states: ⇒ $s_{t,i}$ for time step t of episode i (we often omit the episode index, though)
  - need to be able to *represent interaction* dynamics so that R_t can be defined as sums *over finite or infinite numbers of terms* [equations (13.1) and (13.2)]
- Solution: represent termination as an *absorbing state*:



- and making $R_t = \sum_{k=0}^{T-t-1} \gamma^k r_{t+k+1}$ (where we could have $T = \infty$ or $\gamma = 1$, but not both)

MDPs, other ingredients

- We assume that a reinforcement learning task has the *Markov Property*:

$$P(s_{t+1} = s', r_{t+1} = r | s_t, a_t, r_t, \dots r_1, s_0, a_0) = P(s_{t+1} = s', r_{t+1} = r | s_t, a_t) \quad (13.3)$$

  for all states, rewards and histories.
- So, to specify a *RL task* as an *MDP* we need:
  - to specify S and A

- and $\forall s, s' \in S, a \in A$:
  - *transition probabilities*:

$$\mathcal{P}^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a)$$

  - and *rewards* $\mathcal{R}^a_{ss'}$, Where a reward could be specified as an average over transitions from s to s' when the agent performs action a

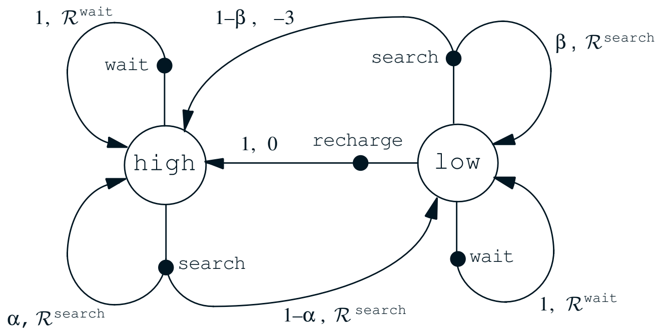
$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

The recycling robot revisited

- At *each step*, robot has to *decide* whether it should (1) actively *search for a can*, (2) *wait* for someone to bring it a can, or (3) *go to home base* and recharge.
- Searching is *better but runs down the battery*; if it runs out of power while searching, has to be rescued (which is bad).
- Decisions made on basis of *current energy level*: high,low.
- Rewards = *number of cans collected* (or -3 *if robot needs to be rescued* for a battery recharge and 0 while recharging)

As a state-transition graph

- $S = \{high, low\}$, $A = \{search, wait, recharge\}$
- $\mathcal{R}^{search}$ = expected *no. of cans collected* while *searching*
- $\mathcal{R}^{wait}$ = expected no. of cans collected while *waiting* ($\mathcal{R}^{search} > \mathcal{R}^{wait}$)



Value functions

- RL is (almost always) based on estimating *value functions* for states, i.e. how much *return* an agent can expect to obtain *from a given state.*
- We can the define the *state-value function* under policy π as the the *expected return* when *starting in s* and *following* π *thereafter*:

$$V^\pi(s) = E_\pi\{R_t | s_t = s\} \tag{13.4}$$

- Note that this implies averaging over probabilities of reaching future states, that is, $P(s_{t+1} = s'|s_t = s, a_t = a)$ over all t.
- We can also generalise the action function (*policy*) to $\pi(s, a)$, returning the *probability of taking action a* while in state s, which implies also averaging over actions.

The action-value function

- we can also define an *action-value function* to give the *value of taking action a* in state s under a policy π:

$$Q^\pi(s, a) = E_\pi\{R_t|s_t = s, a_t = a\} \tag{13.5}$$

  where $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$.

- Both v^π and Q^π can be *estimated*, for instance, through simulation (*Monte Carlo methods*):
  - for each *state s visited* by following π, keep *an average $\hat{V}^\pi$ of returns* received from that point on.
  - $\hat{V}^\pi$ *approaches V^π* as the number of times s is visited approaches ∞
  - Q^π can be estimated similarly.

The Bellman equation

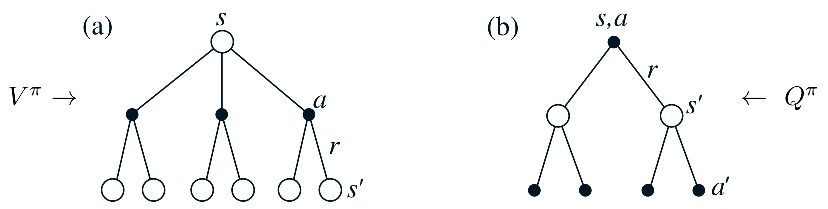
- Value functions *satisfy* particular *recursive relationships.*
- For any policy π and any state s, the following *consistency condition* holds:

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t|s_t = s\} \\ &= E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s\} \\ &= E_\pi\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_t = s\} \\ &= \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma E_\pi\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2}|s_{t+1} = s'\}] \\ &= \sum_a \pi(s, a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \qquad (13.6) \end{aligned}$$

The value function for π is the unique solution for the *set of linear equations* (one per state) represented by (13.6).

Backup diagrams

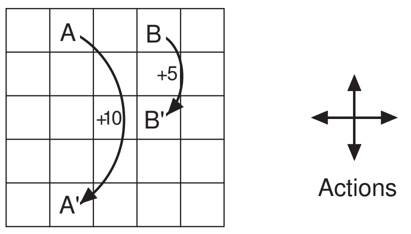
- The *Bellman equation* for V^π (13.6) expresses a relationship between the value of a *state* and the value of *its successors.*
- This can be depicted through *backup diagrams*



- representing *transfers of value information back to a state* (or a state-action pair) *from its successor states* (or state-action pairs).

An illustration: The GridWorld

- Deterministic *actions* (i.e. $\mathcal{P}^a_{ss'} = 1$ for all s, s', a such that s' is reachable from s through a; or 0 otherwise);
- *Rewards*: $\mathcal{R}^a = -1$ if a would move agent off the grid, otherwise $\mathcal{R}^a = 0$, *except for* actions from states A and B.



(a)

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

(b)

- Diagram (b) shows the *solution of the set of equations* (13.6), for equiprobable (i.e. $\pi(s,\uparrow) = \pi(s,\downarrow) = \pi(s,\leftarrow) = \pi(s,\rightarrow) = .25$, for all s) random *policy* and $\gamma = 0.9$

One can easily verify that equation (13.6) holds for, say, the state of being in cell $(2,2)$ (top-left cell is $(1,1)$) of the grid (call it s and the states above, below, left and right of it s_a, s_b, s_l and s_r, respectively) represented by the cell with value 3.0 in the diagram above (second row, second column):

$$\begin{aligned}
V^\pi(s) &= \pi(s,\uparrow)\mathcal{P}^{\uparrow}_{ss_a} \times [\mathcal{R}^{\uparrow}_{ss_a} + \gamma V^\pi(s_a)] \\
&\quad +\pi(s,\downarrow)\mathcal{P}^{\downarrow}_{ss_b} \times [\mathcal{R}^{\downarrow}_{ss_b} + \gamma V^\pi(s_b)] \\
&\quad +\pi(s,\leftarrow)\mathcal{P}^{\leftarrow}_{ss_l} \times [\mathcal{R}^{\downarrow}_{ss_l} + \gamma V^\pi(s_l)] \\
&\quad +\pi(s,\rightarrow)\mathcal{P}^{\rightarrow}_{ss_r} \times [\mathcal{R}^{\rightarrow}_{ss_r} + \gamma V^\pi(s_r)] \\
&= .25 \times 1 \times [0 + \gamma(V^\pi(s_a) + V^\pi(s_b) + V^\pi(s_l) + V^\pi(s_r))] \\
&= .25 \times 1 \times (0 + .9(8.8 + 0.7 + 1.5 + 2.3) \\
&= 2.995 \\
&\approx 3
\end{aligned}$$

As an exercise, check that the state valued 4.4 (top row) also obeys the consistency constraint under policy π.

Optimal Value functions

- For *finite MDPs*, policies can be *partially ordered*: $\pi \geq \pi'$ iff $V^{\pi}(s) \geq V^{\pi'}(s), \quad \forall s \in S$
- There are always one or more policies that are better than or equal to all the others. These are the *optimal policies*, denoted π^*.
- The Optimal policies share *the same*
  - *optimal state-value* function: $V^*(s) = max_{\pi} V^{\pi}(s), \quad \forall s \in S$ and
  - *optimal action-value* function: $Q^*(s,a) = max_{\pi} Q^{\pi}(s,a), \quad \forall s \in S$ and $a \in A$

Bellman optimality equation for V*

- The value of a state under an optimal policy *must equal the expected return for the best action* from that state:

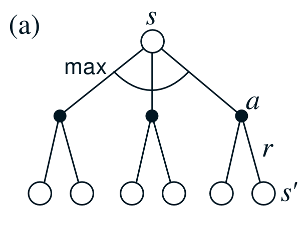
$$\begin{aligned}
V^*(s) &= \max_{a \in \mathcal{A}(s)} Q^*(s,a) \\
&= \max_a E_{\pi^*}\{R_t | s_t = s, a_t = a\} \\
&= \max_a E_{\pi^*}\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} | s_t = s, a_t = a\} \\
&= \max_a E_{\pi^*}\{r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a\} && (13.7) \\
&= \max_{a \in \mathcal{A}(s)} \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')] && (13.8)
\end{aligned}$$

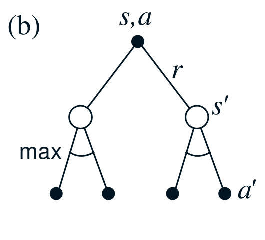
Bellman optimality equation for Q*

- *Analogously* to V^*, we have:

$$\begin{aligned}
Q^*(s,a) &= E\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') | s_t = s, a_t = a\} && (13.9) \\
&= \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s', a')] && (13.10)
\end{aligned}$$

- V^* and Q^* are the *unique solutions* of these systems of equations.

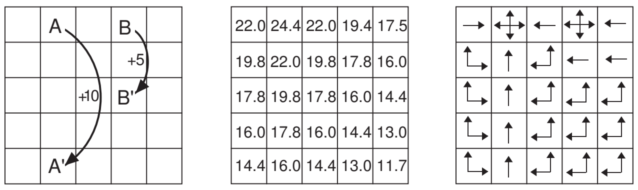




From optimal value functions to policies

- Any policy that is *greedy with respect to V^** is an optimal policy.
- Therefore, a *one-step-ahead search* yields the long-term optimal actions.

- Given Q^*, all the agent needs to do is *set* $\pi^*(s) = \arg\max_a Q^*(s,a)$.



22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

a) gridworld b) V^* c) π^*

Knowledge and Computational requirements

- Finding an optimal policy by *solving the Bellman Optimality Equation* requires:
  - accurate *knowledge of environment* dynamics,
  - the *Markov Property.*
- Tractability:
  - *polynomial* in number of states (via dynamic programming)...
  - ...but *number of states* is often very *large* (e.g., backgammon has about 10^{20} states).
  - So *approximation algorithms* have a role to play
- Many RL methods can be understood as *approximately solving the Bellman Optimality Equation.*

14 Solving MDPs: basic methods

The basic background & assumptions

- Environment is a *finite MDP* (i.e. A and S are finite).
- MDP's dynamics defined by *transition probabilities*:

$$\mathcal{P}^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a)$$

- and expected *immediate rewards*,

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

- Goal: to *search for* good *policies* π
- Strategy: use *value functions* to structure search:

$$\begin{aligned} V^*(s) &= \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V*(s')] \quad \text{or} \\ Q^*(s,a) &= \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s',a')] \end{aligned}$$

Overview of methods for solving Bellman's equations

- *Dynamic programming*:
  - well-understood mathematical properties...
  - ...but require a complete and accurate model of the environment
- *Monte Carlo* (simulation methods):
  - conceptually simple
  - no model required...
  - ...but unsuitable for incremental computation
- *Temporal difference* methods
  - also require no model;
  - suitable for incremental computation...
  - ... but mathematically complex to analyse

Dynamic programming

- Basic Idea: *"sweep" through S* performing a *full backup* operation on each s.
- A few different methods exist. E.g.:
  - *Policy Iteration* and
  - *Value Iteration*.
- The building blocks:
  - *Policy Evaluation*: how to compute V^π for an arbitrary π.
  - *Policy Improvement*: how to compute an improved π given V^π.

Policy Evaluation

- The task of computing V^π for an arbitrary π is known as the *prediction problem.*
- As we have seen, a state-value function is given by

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t|s_t = s\} = E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')] \end{aligned}$$

- a system of $|S|$ linear equations in $|S|$ unknowns (the state values $V^\pi(s)$)

Iterative Policy evaluation

- Consider the sequence of approximations $V_0, \dots V^\pi$.
- Choose V_0 arbitrarily and set each successive approximation accommodation to the Bellman equation:

$$\begin{aligned} V_{k+1}(s) &\leftarrow E_\pi\{r_{t+1} + \gamma V_k(s_{t+1})|s_t = s\} \\ &\leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')] \end{aligned} \tag{14.1}$$

- "Sweeps": $\begin{array}{ccccccccccc} V_0 & \rightarrow & V_1 & \rightarrow & \dots & \rightarrow & V_k & \rightarrow & V_{k+1} & \dots & \rightarrow & V^\pi \\ & \uparrow & & \uparrow & & \uparrow & & \uparrow & & & \uparrow & \end{array}$

Iterative Policy Evaluation Algorithm

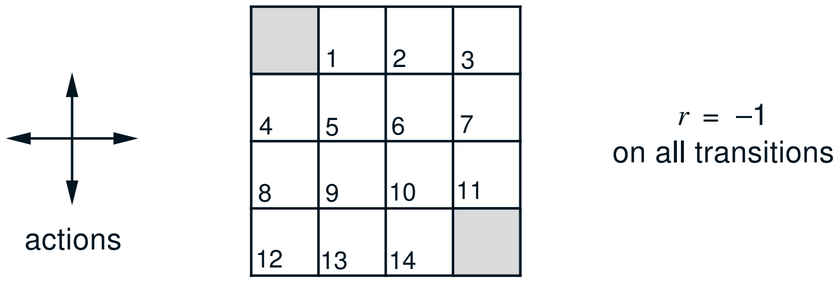
```
Initialisation:
  for (each $s \in S$)
    $V(s) \leftarrow 0$

IPE($\pi$)                    /* $\pi$: policy to be evaluated */
  repeat
    $\Delta \leftarrow 0$
    $V_k \leftarrow V$
    for (each $s \in S/\{s_{terminal}\}$)
      $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')]$
      $\Delta \leftarrow \max(\Delta, |V_k(s) - V(s)|)$
  until $\Delta < \theta$                /* $\theta > 0$: a small constant */
  return $V$                               /* $V \approx V^\pi$ */
```

- NB: alternatively one could evaluate V^π in place (i.e usnig a single vector V to store all values and update it directly).

An example: an episodic GridWorld

- Rewards of -1 until terminal state (shown in grey) is reached
- Undiscounted episodic task:



Policy Evaluation for the GridWorld

- Iterative evaluation of V_k for equiprobable random policy π:

$V_0 \rightarrow$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$V_1 \rightarrow$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$V_2 \rightarrow$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$V_3 \rightarrow$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$V_{10} \rightarrow$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$V_\infty \rightarrow$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Policy Improvement

- Consider the following: how would the expected return change for a policy π if instead of following $\pi(s)$ for a given state s we choose an action $a \neq \pi(s)$?
- For this setting, the value would be:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a\} \\ &= \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^\pi(s_{t+1})] \end{aligned}$$

- So, a should be preferred iff $Q^\pi(s,a) > V^\pi(s)$

Policy improvement theorem

If choosing $a \neq \pi(s)$ implies $Q^\pi(s,a) \geq V^\pi(s)$ for a state s, then the policy π' obtained by choosing a every time s is encoutered (and following π otherwise) is at least as good as π (i.e. $V^{\pi'}(s) \geq V^\pi(s)$). If $Q^\pi(s,a) > V^\pi(s)$ then $V^{\pi'}(s) > V^\pi(s)$

- If we apply this strategy to all states to get a new greedy policy $\pi'(s) = \arg\max_a Q^\pi(s,a)$, then $V^{\pi'} \geq V^\pi$
- $V^{\pi'} = V^\pi$ implies that

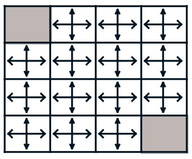
$$V^{\pi'}(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')]$$

  which is... a form of the Bellman optimality equation.
- Therefore $V^\pi = V^{\pi'} = V^*$

Improving the GridWorld policy

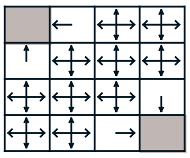
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$\longrightarrow$



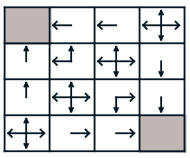
0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$\longrightarrow$



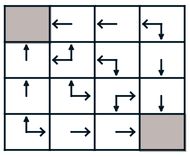
0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$\longrightarrow$



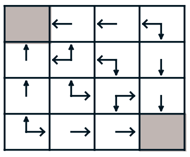
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$\longrightarrow$



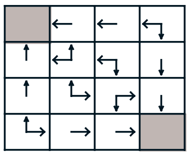
0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$\longrightarrow$



0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

$\longrightarrow$



Putting them together: Policy Iteration

$$\pi_0 \xrightarrow{eval} V_{\pi_0} \xrightarrow{improve} \pi_1 \xrightarrow{e} V_{\pi_1} \xrightarrow{i} \cdots \xrightarrow{i} \pi^* \xrightarrow{e} V^*$$

```
Initialisation:
  for all $s \in S$
    $V(s) \leftarrow$ an arbitrary $v \in \mathbb{R}$

Policy_Improvement($\pi$):
  do
    stable($\pi$) $\leftarrow$ true
    $V \leftarrow$ IPE($\pi$)
    for each $s \in S$
      $b \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
      if ( $b \neq \pi(s)$)
        stable($\pi$) $\leftarrow$ false
  while (not stable($\pi$))
  return $\pi$
```

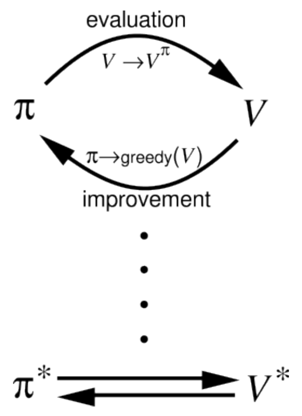
Other DP methods

- *Value Iteration*: evaluation is stopped after a *single sweep* (one backup of each state). The backup rule is then:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')]$$

- *Asynchronous DP*: back up the values of states *in any order*, using whatever values of other states happen to be available.
  - On problems with *large state spaces*, asynchronous DP methods are often preferred

Generalised policy iteration



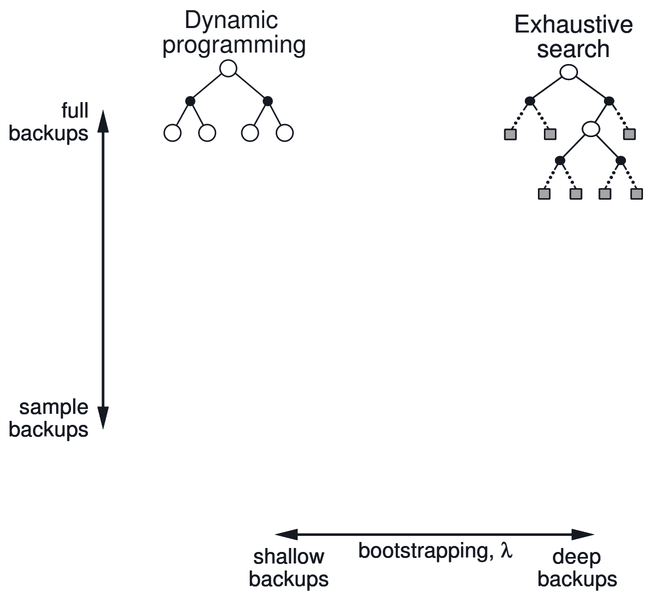
Value Iteration

- Potential computational savings over Policy Iteration in terms of policy evaluation

```
Initialisation:
  for all $s \in S$
    $V(s) \leftarrow$ an arbitrary $v \in \mathbb{R}$

Value Iteration($\pi$):
  repeat
    $\Delta \leftarrow 0$
    for each $s \in S$
      $v \leftarrow V(s)$
      $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  until $\Delta < \theta$
  return deterministic $\pi$ s.t.
      $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
```

Summary of methods



Monte Carlo Methods

- Complete knowledge of environment is not necessary
- Only *experience* is required
- Learning can be *on-line* (no model needed) or through *simulated experience* (model only needs to generate sample transitions.
  - In both cases, learning is based on *averaged sample returns.*
- As in DP, one can use an *evaluation-improvement* strategy.
- Evaluation can be done by keeping averages of:
  - *Every Visit* to a state in an episode, or
  - of the *First Visit* to a state in an episode.

Estimating value-state functions in MC

The *first visit policy evaluation* method: [2em]

```
FirstVisitMC($\pi$)
Initialisation:
  $V \leftarrow$ arbitrary state values
  $Returns(s) \leftarrow$ empty list of size $|S|$

Repeat
  Generate an episode $E$ using $\pi$
  For each $s$ in $E$
     R $\leftarrow$ return following the fisrt occurrence of $s$
     Append R to $Returns(s)$
     $V(s) \leftarrow mean(Returns(s))$
```

Example

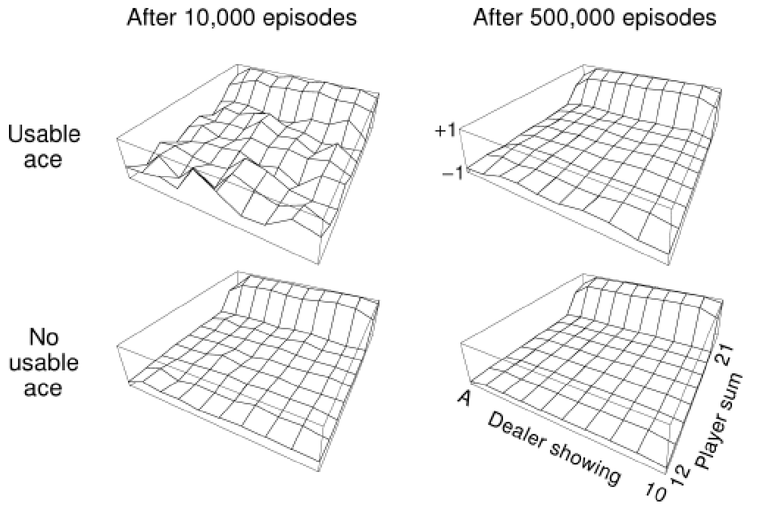
Evaluate the policy described below for *blackjack* (Sutton and Barto, 1998, section 5.1)

- Actions: stick (stop receiving cards), hit (receive another card)
- Play against dealer, who has a fixed strategy ('hit' if *sum* < 17; 'stick' otherwise).
- You win if your card sum is greater than the dealer's without exceeding 21.
- States:
  - current sum (12-21)
  - dealers showing card (ace-10)
  - do I have a useable ace (can be 11 without making sum exceed 21)?
- Reward: +1 for winning, 0 for a draw, -1 for losing
- Policy: Stick if my sum is 20 or 21, else hit

Note: We are assuming sampling with replacement (something like an infinite card deck).

Question: What is the total number of states?

MC value function for blackjack example

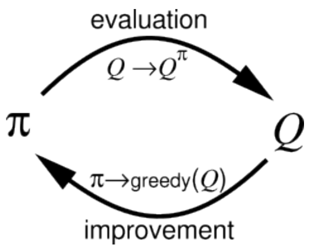


- Question: compare MC to DP in estimating the value function. Do we know the the *environment*? The *transition probabilities*? The *expected returns* given each state and action? What does the MC backup diagram look like?

Unlike DP, MC explores entire episodes, exploring a single choice of successor state at a time. Therefore the algorithm doesn't need to know the transition probabilities. The rewards are sampled by averaging over a large number of episodes. MC does not bootstrap; that is, unlike DP it doesn't update estimates of the value of a state based on the estimated value of its successor.

Monte Carlo control

- Monte Carlo *version* of DP policy iteration:



- *Policy improvement* theorem applies:

$$\begin{aligned}
Q^{\pi_k}(s, \pi_{k+1}(s)) &= Q^{\pi_k}(s, \arg\max_a Q^{\pi_k}(s,a)) \\
&= \max_a Q^{\pi_k}(s,a) \\
&\geq Q^{\pi_k}(s,a) \\
&= V^{\pi_k}(s)
\end{aligned}$$

MC policy iteration (exploring starts)

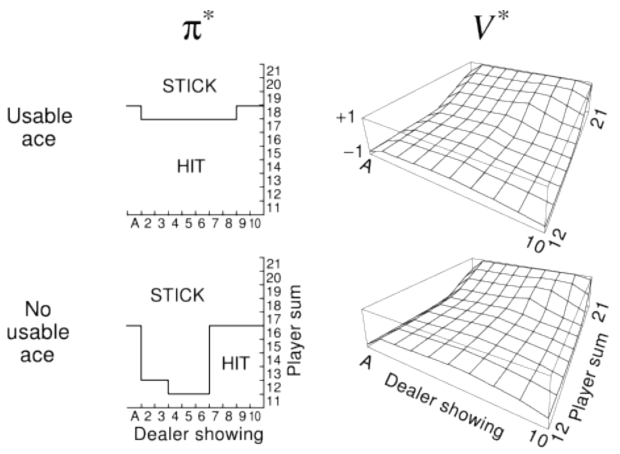
- As with DP, we have *evaluation-improvement* cycles.
- Learn Q^* (if no model is available)
- One must make sure that each state-action pair can be a *starting pair* (with probability > 0).

```
MonteCarloES()
Initialisation , $\forall s \in S, a \in A$:
  $Q(s,a) \leftarrow$ arbitrary; $\pi(s) \leftarrow$ arbitrary
  $Returns(s,a) \leftarrow$ empty list of size $|S|$

Repeat until stop-criterion met
  Generate an episode $E$ using $\pi$ and exploting starts
  For each $(s,a)$ in $E$
     R $\leftarrow$ return following the fisrt occurrence of $s,a$
     Append R to $Returns(s,a)$
     $Q(s,a) \leftarrow mean(Returns(s,a))$
  For each $s$ in $E$
     $\pi(s) \leftarrow \arg\max_a Q(s,a)$
```

Optimal policy for blackjack example

- Optimal *policy* found by *MonteCarloES* for the blackjack example, and its *state-value* function:



On- and off- policy MC control

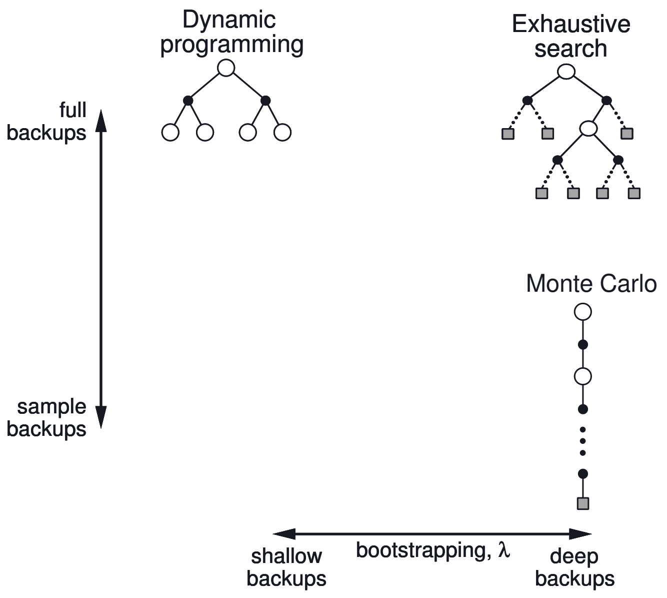
- *MonteCarloES assumes* that all *states* are *observed an infinite number of times* and *episodes* are generated with *exploring starts*
  - For an analysis of convergence properties, see (Tsitsiklis, 2003)
- On-policy and off-policy methods *relax these assumptions* to produce practical algorithms
- On-policy methods use a given policy and *ϵ-greedy strategy* (see lecture on Evaluative Feeback) to generate episodes.
- Off-policy methods *evaluate a policy* while generating an episode through *a different policy*

On-policy control

```
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $Returns(s,a) \leftarrow$ empty list
    $\pi \leftarrow$ an arbitrary $\varepsilon$-soft policy

Repeat forever:
    (a) Generate an episode using $\pi$
    (b) For each pair $s, a$ appearing in the episode:
            $R \leftarrow$ return following the first occurrence of $s, a$
            Append $R$ to $Returns(s,a)$
            $Q(s,a) \leftarrow$ average$(Returns(s,a))$
    (c) For each $s$ in the episode:
            $a^* \leftarrow \arg\max_a Q(s,a)$
            For all $a \in \mathcal{A}(s)$:
            $\pi(s,a) \leftarrow \begin{cases} 1-\varepsilon+\varepsilon/|\mathcal{A}(s)| & \text{if } a = a^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq a^* \end{cases}$
```

Summary of methods



15 Temporal Difference Learning

Recall background & assumptions

- Environment is a *finite MDP* (i.e. A and S are finite).
- MDP's dynamics defined by *transition probabilities*:

$$\mathcal{P}^a_{ss'} = P(s_{t+1} = s' | s_t = s, a_t = a)$$

- and expected *immediate rewards*,

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\}$$

- Goal: to *search for* good *policies* π
- *DP* Strategy: use *value functions* to structure search:

$$\begin{aligned} V^*(s) &= \max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V^*(s')] \quad \text{or} \\ Q^*(s,a) &= \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma \max_{a'} Q^*(s',a')] \end{aligned}$$

- *MC* strategy: *expand each episode* and keep averages of returns per state.

Temporal Difference (TD) Learning

- TD is a combination of ideas from Monte Carlo methods and DP methods:
  - TD can learn directly from raw experience without a model of the environment's dynamics, like MC.
  - TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (i.e. they "bootstrap"), like DP
- Some widely used TD methods: *TD(0), TD(λ)*, *Sarsa*, *Q-learning*, Actor-critic, R-Learning

Prediction & Control

- Recall *value function estimation* (prediction) for DP and MC:
  - Updates *for DP*:

$$\begin{aligned} V^\pi(s) &= E_\pi\{R_t|s_t = s\} \\ &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s\} \end{aligned}$$

  - Update rule *for MC*:

$$V(s_t) \leftarrow V(s_t) + \alpha[R_t - \gamma V(s_t)]$$

(Q: why are the value functions above merely *estimates*?)

Prediction in TD(0)

- Prediction (update of *estimates of* V) for the TD(0) method is done as follows:

$$V(s_t) \leftarrow V(s_t) + \alpha[\underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{1-step estimate of } R_t} - V(s_t)] \tag{15.1}$$

- No need to *wait* until the end of the episode (as in MC) to update V
- No need to *sweep* across the possible successor states (as in DP).

Tabular TD(0) value function estimation

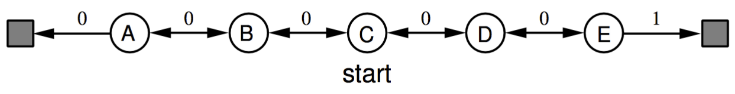
- Use *sample backup* (as in MC) *rather than full backup* (as in DP):

Algorithm 15.1: TD(0)

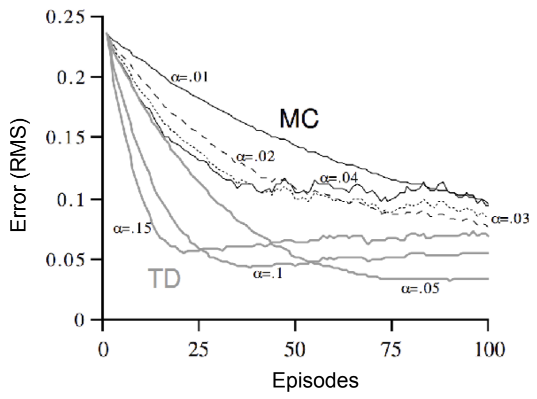
```
TabularTD0($\pi$)
  Initialisation , $\forall s \in S$:
  $V(s) \leftarrow$ arbitrary;
  $\alpha \leftarrow$ learning constant
  $\gamma \leftarrow$ discount factor

  For each episode $E$ until stop-criterion met
    Initialise $s$
    For each step of $E$
       $a \leftarrow \pi(s)$
       Take action $a$;
       Observe reward $r$ and next state $s'$
       $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
       $s \leftarrow s'$
    until $s$ is a terminal state
```

Example: random walk estimate



- $\mathcal{P}^a_{ss'} = .5$ for all non-terminal states
- $\mathcal{R}^a_{ss'} = 0$ except for terminal state on the right which is 1.
- Comparison between MC and TD(0) (over 100 episodes)



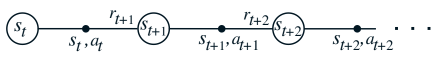
Focusing on action-value function estimates

- The *control* (algorithm) part.
- *Exploration* Vs. *exploitation* trade-off
- *On-policy* and *off-policy* methods
- Several variants, e.g.:
  - *Q-learning*: off-policy method (Watkins and Dayan, 1992)
  - Modified Q-learning (aka *Sarsa*): on-policy method

Sarsa

- Transitions from non-terminal states *update* Q as follows:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \qquad (15.2)$$



(for *terminal states* s_{t+1}, assign $Q(s_{t+1}, a_{t+1}) \leftarrow 0$)

Algorithm 15.2: Sarsa

```
Initialise $Q(s,a)$ arbitrarily
for each episode
  initialise $s$
  choose $a$ from $s$ using $\pi$ derived from $Q$ /* e.g. $\epsilon$-greedy */
  repreat (for each step of episode)
    perform $a$, observe $r, s'$
    choose $a'$ from $s'$ using $\pi$ derived from $Q$ /* e.g. $\epsilon$-greedy */
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
    $a \leftarrow a'$
  until $s$ is terminal state
```

Q-learning

- An *off-policy method*: approximate Q^* independently of the policy being followed:

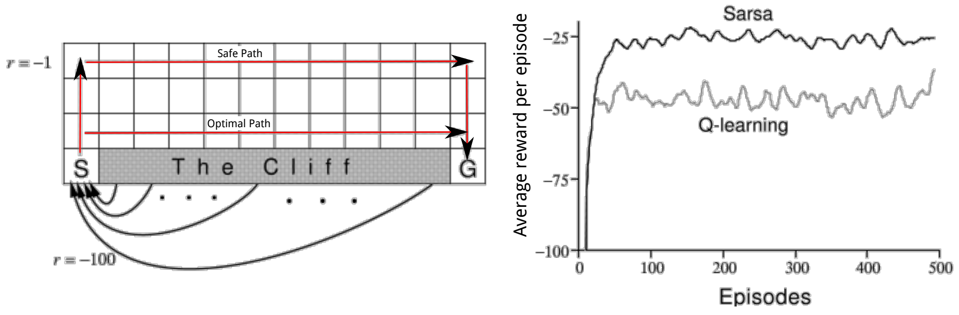
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{15.3}$$

Algorithm 15.3: Q-Learning

```
Initialise $Q(s,a)$ arbitrarily
for each episode
  initialise $s$
  repreat (for each step of episode)
    choose $a$ from $s$ using $\pi$ derived from $Q$ /* e.g. $\epsilon$-greedy */
    perform $a$, observe $r, s'$
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
    $s \leftarrow s'$
  until $s$ is terminal state
```

A simple comparison

- A comparison between *Q-Learning* and *SARSA* on the "cliff walking" problem (Sutton and Barto, 1998)



- Why does Q-learning find the optimal path? Why is its average reward per episode worse than SARSA's?

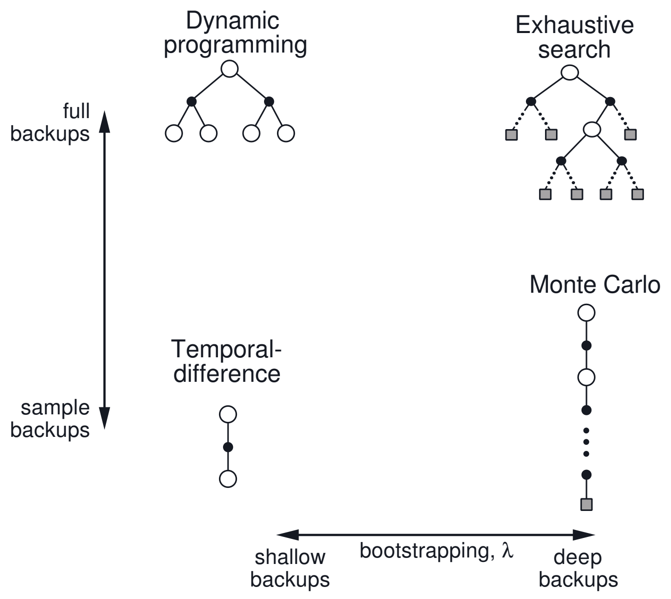
Convergence theorems

- Convergence, in the following sense:

> Q converges in the limit to the optimal Q^* function, provided the system can be modelled as a deterministc Markov process, r is bounded and π visit every action-state pair infinitely often.

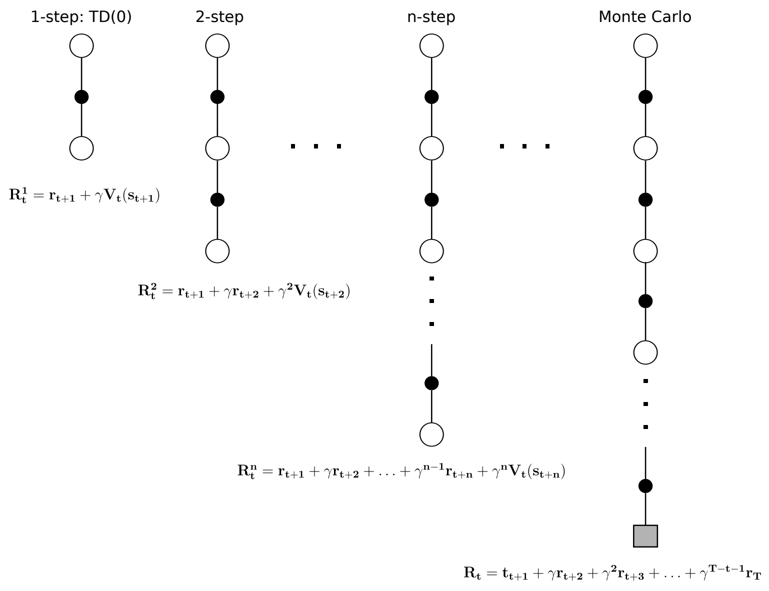
- Has been proved for the above described TD methods: Q-Learning (Watkins and Dayan, 1992), Sarsa and TD(0) (Sutton and Barto, 1998). General proofs based on stochastic approximation theory can be found in (Bertsekas and Tsitsiklis, 1996).

Summary of methods



TD(λ)

- Basic Idea: if with *TD(0)* we backed up our estimates based on *one step ahead*, why not generalise this to include *n-steps* ahead?


1-step: TD(0)
2-step
n-step
Monte Carlo
$R_t^1 = r_{t+1} + \gamma V_t(s_{t+1})$
$R_t^2 = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})$
$R_t^n = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$
$R_t = t_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T$

Averaging backups

- Consider, *for instance*, averaging 2- and 4-step backups:

$$R_t^{\mu} = 0.5R_t^2 + 0.5R_t^4$$

- TD(λ)is a method for averaging *all* n-step backups.
  - Weight by λ^{n-1} $(0 \le \lambda \le 1)$:

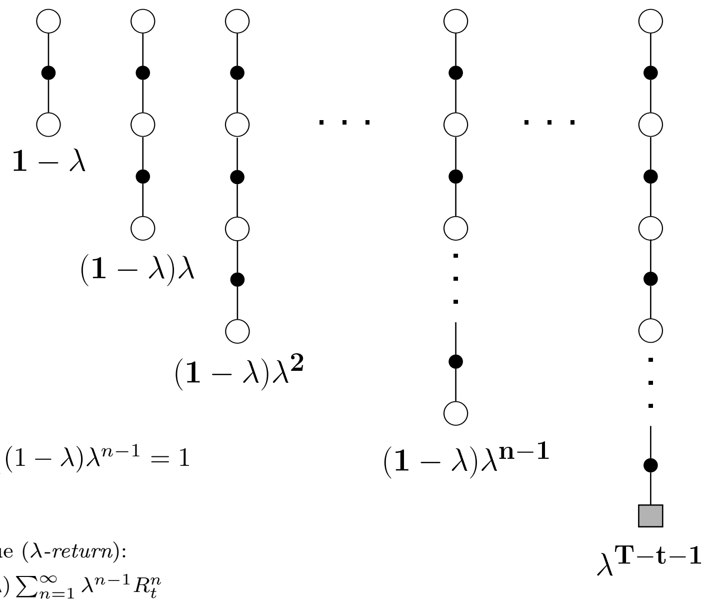
$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^n$$

  - *Backup* using λ-return:

$$\Delta V_t(s_t) = \alpha[R_t^{\lambda} - V_t(s_t)]$$

- (Sutton and Barto, 1998) call this the *Forward View of TD(λ)*

TD(λ) backup structure


$\mathbf{1-\lambda}$
$(\mathbf{1-\lambda})\lambda$
$(\mathbf{1-\lambda})\lambda^{\mathbf{2}}$
$(\mathbf{1-\lambda})\lambda^{\mathbf{n-1}}$
$\lambda^{\mathbf{T-t-1}}$

$\lim_{n\to\infty} \sum_{i=1}^{n}(1-\lambda)\lambda^{n-1} = 1$

Backup value (λ-*return*):

$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^n$

or

$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^n + \lambda^{T-t-1} R_t$

Implementing TD(λ)

- We need a way to accummulate the effect of the *trace-decay parameter* λ
- Eligibility traces:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) + 1 & \text{if} s = s_t \\ \gamma\lambda e_{t-1}(s) & \text{otherwise.} \end{cases} \tag{15.4}$$

- TD error for state-value prediction is:

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \tag{15.5}$$

- Sutton and Barto call this the *Backward View of TD(λ)*

Equivalence

- It can be shown (Sutton and Barto, 1998) that:

The Forward View of TD(λ) and the Backward View of TD(λ) are equivalent.

Tabular TD(λ)

Algorithm 15.4: On-line TD(λ)

```
Initialise $V(s)$ arbitrarily
           $e(s) \leftarrow 0$ for all $s \in S$
for each episode
  initialise $s$
  repeat (for each step of episode)
    choose $a$ according to $\pi$
    perform $a$, observe $r, s'$
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    for all $s$
       $V(s) \leftarrow V(s) + \alpha\delta e(s)$
       $e(s) \leftarrow \gamma\lambda e(s)$
    $s \leftarrow s'$
  until $s$ is terminal state
```

- Similar algorithms can be implemented for *control* (Sarsa, Q-learning), using eligibility traces.

Control algorithms: SARSA(λ)

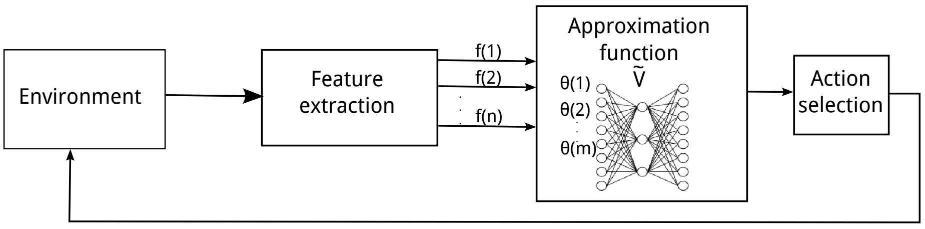
Algorithm 15.5: SARSA(λ)

```
Initialise $Q(s,a)$ arbitrarily
           $e(s,a) \leftarrow 0$ for all $s \in S$ and $a \in A$
for each episode
  initialise $s, a$
  repeat (for each step of episode)
    perform $a$, observe $r, s'$
    choose $a', s'$ according to $Q$ (e.g. $\epsilon-$greedy
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s,a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    for all $s$
       $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
       $e(s) \leftarrow \gamma\lambda e(s)$
    $s \leftarrow s'$ $a \leftarrow a'$
  until $s$ is terminal state
```

16 Combining Dynamic programming and approximation architectures

Combining TD and function approximation

- Basic idea: use *supervised learning* to provide an approximation of the value function for TD learning
- The approximation architecture is should *generalise* over (possibly *unseen) states*



- In a sense, it *groups states* into equivalence classes (wrt value)

Why use approximation architectures

- To cope with the *curse of dimensionality*
- by generalising over *states*
  - Note that the algorithms we have seen so far (DP, TD, Sarsa, Q-learning) all use tables to store states (or state-action tuples)
  - This works well if the number of states is relatively small
  - But it doesn't scale up very well
- (We have already seen *examples* of approximation architectures: the draughts player, the examples in the neural nets lecture.)

Gradient descent methods

- The *LMS algorithm* use for draghts illustrates a gradient descent method
  - (to approximate a linear function)
- *Goal*: to learn the *parameter* vector

$$\overrightarrow{\theta_t} = (\theta_t(1), \theta_t(2), \theta_t(3), \dots, \theta_t(m)) \tag{16.1}$$

  by adjusting them at each iteration towards *reducing the error*:

$$\begin{aligned}\vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha\nabla_{\vec{\theta}_t}(V^\pi(s_t) - V_t(s_t))^2 && (16.2)\\ &= \vec{\theta}_t + (V^\pi(s_t) - V_t(s_t))\alpha\nabla_{\vec{\theta}_t}V_t(s_t) && (16.3)\end{aligned}$$

  where V_t is a smooth, differentiable function of $\vec{\theta}_t$.

Backward view and update rule

- The problem with (16.2) is that the target value (V^π) is typically not available.
- Different methods replace their estimates for this value function:
  - So Monte Carlo, for instance, would use the return R_t
  - And the $TD(\lambda)$ method uses R_t^λ:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha(R_t^\lambda - V_t(s_t))\nabla_{\vec{\theta}_t}V_t(s_t) \tag{16.4}$$

  - The *backward view* is given by:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t \tag{16.5}$$

    where $\vec{e}_t$ is a vector of eligibility traces (one for each component of $\vec{\theta}_t$), updated by

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t}V_t(s_t) \tag{16.6}$$

Value estimation with approximation

Algorithm 16.1: On-line gradient descent TD(λ)

```
Initialise $\vec{\theta}$ arbitrarily
           $\vec{e} \leftarrow 0$
           $s \leftarrow$ initial state of episode
repeat (for each step of episode)
  choose $a$ according to $\pi$
  perform $a$, observe $r, s'$
  $\delta \leftarrow r + \gamma V(s') - V(s)$
  $\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{\theta}}V(s)$
  $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
  $s \leftarrow s'$
until $s$ is terminal state
```

- Methods commonly used to compute the *gradients* $\nabla_{\vec{\theta}}V(s)$:
  - error *back-propagation* (multilayer NNs), or by
  - *linear approximators* (for value functions of the form $V_t(s) = (\vec{\theta}_t)^T\vec{f} = \sum_{i=1}^n \theta_t(i)f(i)$. (where $(\vec{\theta}_t)^T$ denotes the transpose of $\vec{\theta}_t$)

Control with approximation

- The general (*forward view*) update rule for action-value prediction (by gradient descent) can be written:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha(R_t^\lambda - Q_t(s_t, a_t))\nabla_{\vec{\theta}_t} Q_t(s_t, a_t) \tag{16.7}$$

  (recal that V_t is determined by $\vec{\theta}_t$)

- So the *backward view* can be expressed as before:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t \tag{16.8}$$

  where

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t, a_t) \tag{16.9}$$

An algorithm: gradient descent Q-learning

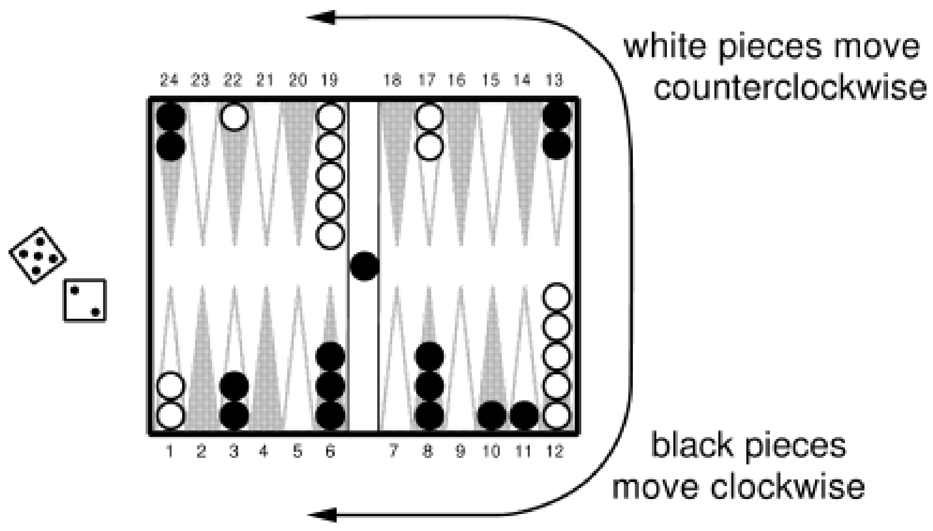
Algorithm 16.2: Linear Gradient Descent Q(λ)

```
Initialise $\theta$ arbitrarily
for each episode
  $\vec{e} \leftarrow \vec{0}$; initialise $s, a$
  $\mathcal{F}_a \leftarrow$ set of features in $s, a$
  repeat (for each step of episode)
    for all $i \in \mathcal{F}_a$ : $e(i) \leftarrow e(i) + 1$
    perform $a$, observe $r, s$
    $\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$
    for all $a \in A$
       $\mathcal{F}_a \leftarrow$ set of features in $s, a$
       $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    $\delta \leftarrow \delta + \gamma \max_a Q_a$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
    with probability $1 - \epsilon$
       for all $a \in A$
         $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
       $a \leftarrow \arg\max_a Q_a$
       $\vec{e} \leftarrow \gamma\lambda\vec{e}$
    else
       $a \leftarrow$ a random action
       $\vec{e} \leftarrow 0$
  until $s$ is terminal state
```

A Case Study: TD-Gammon

- 15 white and 15 black pieces on a board of 24 locations, called points.

- Player rolls 2 dice and can move 2 pieces (or same piece twice)


white pieces move counterclockwise
black pieces move clockwise
24 23 22 21 20 19 18 17 16 15 14 13
1 2 3 4 5 6 7 8 9 10 11 12

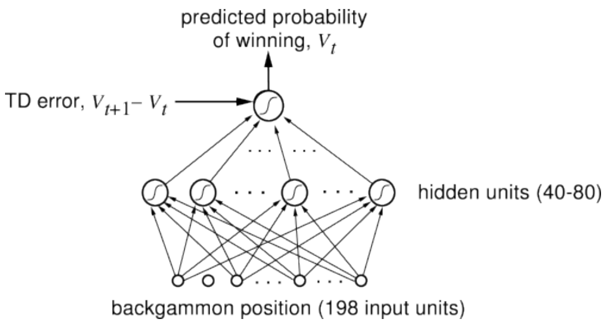
- Goal is to move pieces to last quadrant (for white that's 19-24) and then off the board
- A player can "hit" any opposing single piece placed on a point, causing that piece to be moved to the "bar"
- Two pieces on a point block that point for the opponent
- + a number of other complications

Game complexity

- 30 pieces, 26 locations
- Large number of *actions* possible from a given state (up to 20)
- Very large number of possible *states* (10^{20})
- Branching factor of about 400 (so difficult to apply heuristics)
- *Stochastic* environment (next state depends on the opponent's move) but *fully observable*

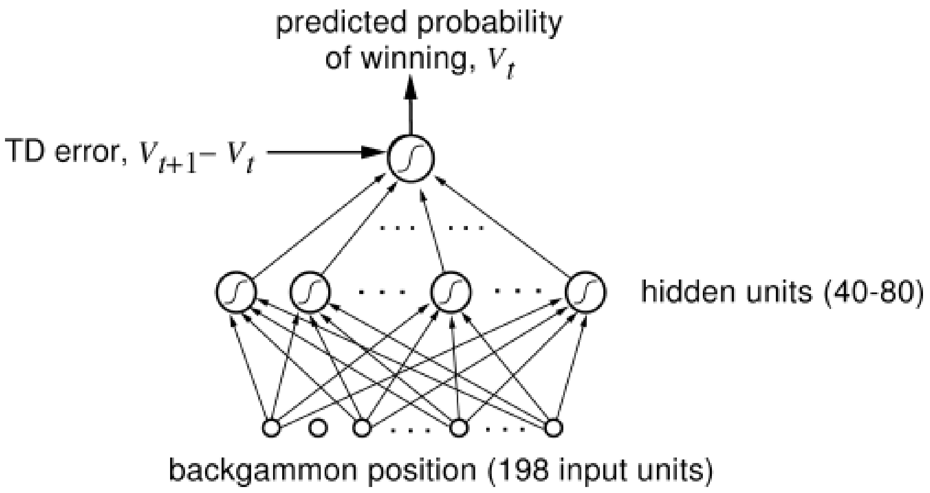
TD-Gammon's solution

- $V_t(s)$ meant to estimate the *probability of winning* from any state s
- *Rewards*: 0 for all stages, except those on which the game is won
- Learning: non-linear form of TD(λ)
  - like the Algorithm presented above, using a multilayer neural network to compute the gradients


predicted probability of winning, V_t
TD error, $V_{t+1} - V_t$
hidden units (40-80)
backgammon position (198 input units)

State representation in TD-Gammon

- Representation involved little domain knowledge
- 198 input features:


predicted probability of winning, V_t
TD error, $V_{t+1} - V_t$
hidden units (40-80)
backgammon position (198 input units)

- For each point on the backgammon board, *four units* indicated the number of white pieces on the point (see (Tesauro, 1994) for a detailed description of the encoding used)
- (4 (white) + 4 (black)) x 24 points = 192 units
- 2 units encoded the number of white and black pieces on the bar
- 2 units encoded the number of black and white pieces already successfully removed from the board
- 2 units indicated in a binary fashion whether it was white's or black's turn to move.

TD-Gammon learning

- Given state (position) representation, the network computed its estimate in the way described in lecture 10.

  - Output of hidden unit j given by a sigmoid function of the weighted sum of inputs i
$$h(j) = \sigma(\sum_i w_{ij} f(i)) \tag{16.10}$$
  - Computation from hidden to output units is analogous to this

- TD-Gammon employed TD(λ) where the eligibility trace updates (equation (16.9),
$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t)$$
were computed by the back-propagation procedure

- TD-Gammon set $\gamma = 1$ and rewards to zero, except on winning, so TD error is usually $V_t(s_{t+1}) - V_t(s_t)$

TD-Gammon training

- Training data obtained by playing against itself
- Each game was treated as an episode
- Non-linear TD applied incrementally (i.e. after each move)
- Some results (according to (Sutton and Barto, 1998))

Program	Hidden Units	Training Games	Opponents	Results
TD-Gam 0.0	40	300,000	other programs	tied for best
TD-Gam 1.0	80	300,000	Robertie, Magriel, ...	-13 pts / 51 games
TD-Gam 2.0	40	800,000	various Grandmasters	-7 pts / 38 games
TD-Gam 2.1	80	1,500,000	Robertie	-1 pt / 40 games
TD-Gam 3.0	80	1,500,000	Kazaros	+6pts / 20 games

17 Final Project

17.1 Background

This end-of-course project will account for 70% of your overall grade in CS7032. The project can be done individually or in teams of two students.

The project will be assessed through a report and code. In some cases, a demonstration might also be appropriate and will be arranged in advance. The report must be structured as an academic paper and contain, minimally: (1) a precise statement of the problem addressed by the project, (2) the approach you adopted to solving it, (3) a discussion of you results (whether positive or negative), (4) the main difficulties you encountered, (5) your assessment of the usefulness of an AI technique in addressing the type of problem tackled by your project and (6) references to the relevant literature. In addition, if you worked as a team, you must include a separate section describing the contribution of each team member.

17.2 Goals

You should implement an agent (or controller for a team of agents, as the case might be) to play one of the recommended Game AI competitions. Although we should ultimately be aiming to produce an agent (or agents) which can in principle (say, after some extra work after the assignment itself is finished) be entered into a competition, some competitions may be too complex for a complete and competitive agent to be implemented within the time frame of the project. In such cases, it is acceptable for you to focus on an interesting aspect or sub-problem of the game in question (e.g. learning targeting or move strategies in combat-style games, learning the best choice of shooting technique in Angry Birds (e.g. (Jutzeler et al., 2013)), learning defense actions in fighting games (e.g. (Yamamoto et al., 2014)), etc).

17.3 Assessment

The project will be assessed and marked in terms of: (a) clarity of problem description, including adequate (not excessive) background information, (b) quality of the technical execution, bearing in mind that we are not looking for an unbeatable agent but one that illustrates clearly and meaningfully some of the techniques discussed in the course (such as problem specification in terms of agent architectures, simulation and multi-agent problem solving, supervised and reinforcement learning, etc), (c) clarity in the assessment of results, discussion of the effectiveness of the techniques explored, and conclusions drawn from the project work, (d) quality of the presentation and (e) description and assessment of the contributions of each team member (if you choose to work in a team).

The amount of effort you should put into the project should be commensurable with the amount of work put into preparing for an exam.

17.4 Delivering the assignment

The deadline for submission of the project report (and code) is Monday, 26th January. The project must be submitted through blackboard, as with the other assignments.

Please format the report (preferably in LaTeX) according to the ACM's conference proceedings style template[1]. The report/paper should normally not exceed 10 pages in the (two column) ACM format.

17.5 Possible choices

You may choose one of the following competitions:

1. Fighting AI,
2. AI Birds,
3. Starcraft or
4. Vindinium

These options are recommended based on the following criteria: your expressed preferences, availability (and freedom) of code, interest of the AI community in general (as reflected in published papers and competitions scheduled for next year's AI conferences), simplicity of the domain (so that you can focus on the AI).

17.5.1 Resources

Surveys and links to the competitions, papers etc can be found at the course website[2].

[1]`http://www.acm.org/sigs/publications/proceedings-templates#aL1`

[2]`https://www.scss.tcd.ie/~luzs/t/cs7032/gamesaisurvey-teams.html`

17.6 Questions?

Feel free to email me.

Bibliography

Bellman, R. (1956). A problem in the sequential design of experiments. *Sankhya*, 16:221–229.

Berger, B. and Leighton, T. (1998). Protein folding in the hydrophobic-hydrophilic (hp) is np-complete. In *Proceedings of the second annual international conference on Computational molecular biology*, pages 30–39. ACM Press.

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming.* Athena Scientific, Belmont.

Bonabeau, E., Dorigo, M., and Theraulaz, G. (2000). Inspiration for optimization from social insect behaviour. *Nature*, 4006.

Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23.

Caro, G. D. and Dorigo, M. (1998). Antnet: Distributed stigmergetic control for communications networks. *Journal of Artificial Intelligence Research*, 9:317–365.

Cassell, J. (2000). Embodied conversational interface agents. *Communications of the ACM*, 43(4):70–78.

Chapman, D. and Agre, P. E. (1987). Pengi: An implementation of a theory of activity. Technical report, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts. Subm. to the Cognitive Modeling/Science Section of AAAI-87.

Crescenzi, P., Goldman, D., Papadimitriou, C., Piccolboni, A., and Yannakakis, M. (1998). On the complexity of protein folding (abstract). In *Proceedings of the second annual international conference on Computational molecular biology*, pages 61–62. ACM Press.

Dill, K. A., Bromberg, S., Yue, K., Fiebig, K. M., Yee, D. P., Thomas, P. D., and Chan, H. S. (1995). Principles of protein folding – A perspective from simple exact models. *Protein Science*, 4(4):561–602.

Domingos, P. (2012). A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87.

Dorigo, M. and Di Caro, G. (1999). The ant colony optimization meta-heuristic. In Corne, D., Dorigo, M., and Glover, F., editors, *New Ideas in Optimization*, pages 11–32. McGraw-Hill, London.

Dorigo, M., Maniezzo, V., and Colorni, A. (1996). The Ant System: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics Part B: Cybernetics*, 26(1):29–41.

Drogoul, A. and Ferber, J. (1992). From tom thumb to the dockers: Some experiments with foraging robots. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 451–459, Honolulu, Hawaii.

Epstein, J. and Axtell, R. (1996). *Growing Artificial Societies: Social Science From The Bottom Up*. MIT Press.

Fagin, R., Halpern, J. Y., Vardi, M. Y., and Moses, Y. (1995). *Reasoning about knowledge*. MIT Press.

Genesereth, M., Love, N., and Pell, B. (2005). General game playing: Overview of the AAAI competition. *AI magazine*, 26(2):62.

Genesereth, M. and Nilsson, N. (1987). *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, CA.

Hart, W. E. and Istrail, S. (1995). Fast protein folding in the hydrophobic-hydrophilic model within three-eights of optimal. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 157–168. ACM Press.

Heusse, M., Guerin, S., Snyers, D., and Kuntz, P. (1998). Adaptive agent-driven routing and load balancing in communication networks. *Adv. Complex Systems*, 1:237–254.

Jutzeler, A., Katanic, M., and Li, J. J. (2013). Managing luck: A multi-armed bandits meta-agent for the angry birds competition. In *AI Birds Competition*.

Karakovskiy, S. and Togelius, J. (2012). The mario ai benchmark and competitions. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):55–67.

Langton, C. G. (1989). *Artificial Life: Proceedings of the First Int. Workshop on the Synthesis and Simulation of Living Systems*. Addison-Wesley.

Loiacono, D., Lanzi, P., Togelius, J., Onieva, E., Pelta, D., Butz, M., Lonneker, T., Cardamone, L., Perez, D., Sáez, Y., et al. (2010). The 2009 simulated car racing championship. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(2):131–147.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). Mason: A multiagent simulation environment. *Simulation*, 81(7):517–527.

McCarthy, J. (1979). Ascribing mental qualities to machines. Technical Memo 326, Stanford University AI Lab.

McCarthy, J. (1998). Partial formalizations and the lemmings game.

Michie, D. and Chambers, R. A. (1968). BOXES: An experiment in adaptive control. In Dale, E. and Michie, D., editors, *Machine Intelligence 2*, pages 137–152. Oliver and Boyd, Edinburgh.

Millington, I. (2006). *Artificial Intelligence for Games.* Morgan Kaufmann.

Minar, N. Implementation of SugarScape in Swarm. http://www.swarm.org/community-contrib.html.

Mitchell, T. M. (1997). *Machine Learning.* McGraw-Hill.

Narendra, K. and Thathachar, M. (1974). Learning automata - a survey. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-4(4):323–334.

Norling, E. and Sonenberg, L. (2004). Creating interactive characters with bdi agents. In *Proceedings of the Australian Workshop on Interactive Entertainment IE'04*.

Pomerleau, D. A. (1994). *Neural Network Perception for Mobile Robot Guidance.* Kluwer, Dordrecht, Netherlands.

Rohlfshagen, P. and Lucas, S. (2011). Ms pac-man versus ghost team CEC 2011 competition. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 70–77. IEEE.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence. A Modern Approach.* Prentice-Hall, Englewood Cliffs.

Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence. A Modern Approach.* Prentice-Hall, Englewood Cliffs, 2nd edition.

Shaker, N., Togelius, J., Yannakakis, G., Weber, B., Shimizu, T., Hashiyama, T., Sorenson, N., Pasquier, P., Mawhorter, P., Takahashi, G., et al. (2011). The 2010 Mario AI championship: Level generation track. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(4):332–347.

Shmygelska, A. and Hoos, H. H. (2003). An improved ant colony optimisation algorithm for the 2D HP protein folding problem. In Xiang, Y. and Chaib-draa, B., editors, *Advances in Artificial Intelligence: 16th Conference of the Canadian Society for Computational Studies of Intelligence*, volume 2671 of *Lecture Notes in Computer Science*, pages 400–417, Halifax, Canada. Springer-Verlag.

Shneiderman, B. and Maes, P. (1997). Direct manipulation vs interface agents. *interactions*, 4(6):42–61.

Simon, H. A. (1981). *The Sciences of the Artificial.* MIT Press, Cambridge, Massachusetts, second edition.

Steels, L. (1990). Cooperation between distributed agents through self organization. In Demazeau, Y. and Müller, J. P., editors, *Decentralised AI, Proceedings of MAAMAW '89*, pages 175–196, Amsterdam. Elsevier.

Sutton, R. S. and Barto, A. G. (1998). *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA.

Tesauro, G. (1994). TD-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219.

Tsitsiklis, J. N. (2003). On the convergence of optimistic policy iteration. *The Journal of Machine Learning Research*, 3:59–72.

Venables, W. N. and Smith, D. M. (2004). *An Introduction to R*. The R Development Core Team.

Watkins, C. J. C. H. and Dayan, P. (1992). Technical note Q-learning. *Machine Learning*, 8:279–292.

Weiss, G. (1999). *Multiagent Systems*. MIT Press, Cambridge.

Wooldridge, M. (2002). *An Introduction to MultiAgent Systems*. John Wiley & Sons.

Wooldridge, M. (2009). *An Introduction to MultiAgent Systems*. John Wiley & Sons, 2nd edition.

Yamamoto, K., Mizuno, S., Chu, C. Y., and Thawonmas, R. (2014). Deduction of fighting-game countermeasures using the k-nearest neighbor algorithm and a game simulator. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–5. IEEE.