Artificial Intelligence

Topic 1

Introduction

- ◇ What is AI?
- ◇ Contributions to AI
- ◇ History of AI
- ◇ Modern AI

Reading: Russel and Norvig, Chapter 1

1.1 AI in the Media - the glitz and glamour

- ◇ sci-fi
  — Kubric, Spielberg,...
- ◇ "science" programs
  — "Towards 2000"
- ◇ news/current affairs
  — Kasparov
- ◇ advertisements
  — washing machines, TVs, cars,...
  ⋮

Don't believe a word you hear!

(...without proof)

1.2 The AI Literature

"[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak+McDermott, 1985)
"The study of how to make computers do things at which, at the moment, people are better" (Rich+Knight, 1991)	"The branch of computer science that is concerned with the automation of intelligent behavior" (Luger+Stubblefield, 1993)

Views of AI fall into four categories:

Thinking humanly	Thinking rationally
Acting humanly	Acting rationally

1.3 Thinking humanly: cognitive modelling

- determine how humans think
- develop theory of human mind — psychological experiments
- model theory using computer programs

  eg. General Problem Solver (GPS) [Newel & Simon, 1961]

Requires scientific theories of internal activities of the brain

- What level of abstraction? "Knowledge" or "circuits"?
- How to validate?

1. Predicting and testing behavior of human subjects (top-down)
   ⇒ *Cognitive Science*
2. Direct identification from neurological data (bottom-up)
   ⇒ *Cognitive Neuroscience*

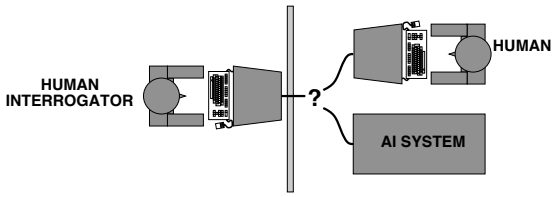
1.4 Acting humanly: The Turing test

Alan Turing (1950) "Computing machinery and intelligence":

♢ "Can machines think?" ⟶ "Can they behave intelligently?"

intelligence = ability to act indistinguishably from a human in cognitive tasks

♢ Operational test for intelligent behavior ⇒ Turing Test



– human interrogates computer via teletype
– passes test if human cannot tell if there's a human or computer at the other end

What else might today's turing test include. . . ?

1.4 Acting humanly: The Turing test

♢ Predicted that by 2000, a machine might have a 30% chance of fooling a lay person for 5 minutes

♢ Anticipated all major arguments against AI in following 50 years

♢ Suggested major components of AI: knowledge, reasoning, language understanding, learning

Problem: Turing test is not reproducible, constructive, or amenable to mathematical analysis

1.5 Thinking rationally: Laws of Thought

Began with Greeks in 4th century BC (1st century BE)

— e.g. Aristotle's logical *syllogisms*

All men are mortal,
Socrates is a man,
therefore Socrates is mortal

Several Greek schools developed various forms of logic: notation and rules of derivation for thoughts;

— may or may not have proceeded to the idea of mechanization

1.5 Thinking rationally: Laws of Thought

Direct line through mathematics and philosophy to modern AI, e.g.

- Boole (1815–1864, *Laws of Thought* 1854, *Boolean logic*)
- Frege (1848–1925, *Begriffsschrift* 1879, *first-order logic*)
- Hilbert (1862–1943), (*Hilbert systems*)
- Russell & Whitehead (*Principia Mathematica* 1918)
- Tarski (1902–1983, *Tarski semantics* 1933)

Problems:

1. Normative (or prescriptive) rather than descriptive
2. Not all intelligent behavior is mediated by logical deliberation

1.6 Acting rationally

$\Rightarrow$ act in such a way to achieve goals, given beliefs

(Doesn't necessarily involve thinking—e.g., blinking reflex.
Is a thermostatically controlled heater "intelligent"?)

Define *agent* — entity that percieves surroundings and acts accordingly

AI = study and construction of *rational agents*.

AI as intelligent agent design

- incorporates aspects from the other three approaches
- currently the dominant view of AI

1.7 An Engineering Viewpoint

"a bunch of methodologies, inspired by socio-biological analogy, for getting machines to do stuff"

2. Pre-history — Contributions to AI

Philosophy (428 BC $\rightarrow$ present)

- logic, methods of reasoning
- mind as a physical system
- foundations of learning, language, rationality

Mathematics (c. 800 $\rightarrow$ present)

- formal representation and proof
- computation, (un)decidability, (in)tractability
- probability

Psychology (1879 $\rightarrow$ present)

- perception and motor control
- cognitive neuroscience
- learning (reinforcement)

2. Pre-history — Contributions to AI

Computing (1940 $\rightarrow$ present)

- provision of programmable machines
- algorithms
- declarative languages (PROLOG and LISP)
- neural computing

Linguistics (1957 $\rightarrow$ present)

- language theory: grammar, semantics

3. The Chequered History of AI

In the beginning...

1943 McCulloch & Pitts: Boolean circuit model of brain
on/off neurons, corresponding to propositions
even suggested networks could learn!
⇒ forerunner of symbolic and connectionist traditions

1950 Turing's "Computing Machinery and Intelligence"
Turing and Shannon writing chess programs
— with no computers!

1951 Minsky & Edmonds, first neural net computer, SNARK

1950s Newell & Simon's Logic Theorist (machine coded by hand!)
"*We have created a computer program capable of thinking non-numerically, and thereby solved the venerable mind-body problem.*" — Simon
Able to prove theorems in Russell and Whitehead's *Principia Mathematica*
— even came up with a shorter proof!

1956 McCarthy's Dartmouth meeting: "Artificial Intelligence"

3. The Chequered History of AI

"Look, Ma, no hands!" era

1952 Samuel's checker playing programs
— eventually tournament level
⇒ disproved computers can only do what they are told

1958 McCarthy
— defined LISP ⇒ dominant AI programming language
— he and others invented time-sharing ⇒ birth of DEC
— published "Programs with Common Sense"
⇒ defined hypothetical program Advice Taker
AI program that includes general knowledge, axioms
forerunner to knowledge representation and reasoning today

1959 McCarthy & Hayes "Philisophical Investigations from the Standpoint of AI" ⇒ KR, reasoning, planning,...

1961 Newell and Simon's General Problem Solver (GPS)
imitate human problem solving — "thinking humanly"

3. The Chequered History of AI

1965 Robinson's *resolution principle*
complete theorem proving algorithm for 1st-order logic
made PROLOG possible

1971 STRIPS — practical logic-based planning system
SHAKEY — integration of logical reasoning and physical activity

3. The Chequered History of AI

Falling off the bike

Bold predictions prove elusive...

♢ Lack of domain knowledge

eg. machine translation

US goverment funding to translate Russian to English after Sputnik in 1957

Initially thought syntactic transformations using grammar and electronic dictionary

"the spirit is willing but the flesh is weak"

"*the vodka is good but the meat is rotten*"

1966 report — no immediate prospect of success

All government funding cancelled

3. The Chequered History of AI

♢ Intractibility — early solutions did not scale up!

development of computational complexity theory and NP-completeness

program finds solution in principle $\not\rightarrow$ *has any of the mechanisms needed to find it in practice*

difficulties of combinatorial explosion one of main criticisms in Lighthill Report in 1973

⇒ British government decision to end support for AI research in all but 2 universities

♢ limitations of basic AI structures

1969 Minsky and Papert's book *Perceptrons*

— two-input perceptron could not be trained to recognise when its two inputs were different (*exclusive-or* problem)

⇒ research funding for neural nets all but disappeared

3. The Chequered History of AI

AI goes specialist

1969 Buchanan et al, "Heuristic DENDRAL: a program for generating explanatory hypotheses in organic chemistry."
Arguably first *knowledge-intensive* system.
Later incorporated McCarthy's ADVICE TAKER approach
— clean separation of knowledge from reasoning.
⇒ birth of *expert systems*.

1976 MYCIN — diagnosis of blood infections.
$\sim$ 450 rules. Performed as well as some experts, better than junior doctors.
No theoretical model — acquired rules from interviewing experts.
Early attempt to deal with *uncertainty*.

3. The Chequered History of AI

1979 Duda et al., PROSPECTOR
Probabilistic reasoning system — recommended drilling at geological site that proved to contain large molybdenum deposit!!

1970s Recognition that language understanding also required knowledge and a means of using it. (Charniak)
"*There is no such thing as syntax.*" — Schank

1973 Woods' LUNAR system — allowed geologists to ask questions in English about Apollo's rock samples.
— first NLP program used by others for real work.

3. The Chequered History of AI

Expert systems industry booms. . .

1982 McDermott's R1 began operation at DEC
— first successful commercial expert system.
Helped configure orders for new systems — saved estimated $40 million a year.

1988 DEC: 40 deployed expert systems
Du Pont: 100 in use, 500 in development, est $100m year

Increased demand for AI languages — eg Prolog

1980s Japanese ambitious "Fifth Generation" project.
"Prolog machines" — millions of inferences per sec.

US funding increased accordingly.
British Alvey report reinstated funding cut by Lighthill report (under new name "Intelligent Knowledge-Based Systems")

Boom in Expert System development tools, dedicated Lisp workstations (eg Symbolics), etc. . .

few million sales in 1980 $\longrightarrow$ $2 billion in 1988

3. The Chequered History of AI

and busts. . . ?

∼1986 Recognition of limitations and some disillusionment
— buying expert system shell and filling it with rules not enough.

1985–95 Neural networks return to popularity
— rediscovery of back-propagation learning algorithm.
Brooks' insects.
Symbolic vs sub-symbolic argument intensifies!
(battles for funding)

⇒ predictions of "AI winter".

4. Modern AI

AI matures (at least a bit!)

- Better understanding of difficulty!
- Increase in technical and theoretical depth.
- Recognition (by most) of the need for both symbolic and sub-symbolic approaches, working together.
- Resurgence and incorporation of probabilistic and decision-theoretic methods.

⇒ emphasis on solid foundations and a more wholistic approach

The new environment!

- Fast distributed hardware
- New languages (eg Java), distributed programs
- Increased communications (WWW).
- New possibilities, eg ALife, GAs,. . .
- Advances in understanding of biological and neural systems (biologically-inspired computing)
- New applications, eg Web agents, Mars rovers, . . .

⇒ emphasis on intelligent "agents", incorporating a range of AI technologies

Artificial Intelligence

Topic 2

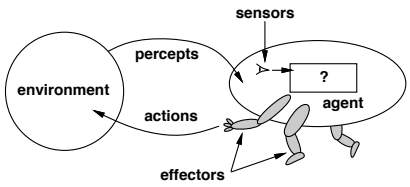
Intelligent Agents

- ◇ What are agents?
- ◇ Rational agents
- ◇ Agent functions and programs
- ◇ Types of agents

Reading: Russell and Norvig, Chapter 2

1. What are agents?

agent — *perceives* environment through *sensors*
— *acts* on the environment through *effectors*



Examples. . .

Percepts	light, sound, solidity, . . .
Sensors	
human	eyes, ears, skin, . . .
robot	infra-red detectors, cameras, microphone, accelerometers, . . .
Effectors	
human	hands, legs, voice, . . .
robot	grippers, wheels, speakers, . . .
Actions	pickup, throw, speak, . . .

2. Rational agents

Recall: rational agent tries to...
— "do the right thing"
— act to achieve goals

The "right thing" can be specified by a *performance measure* defining a numerical value for any environment history

Rational action: whichever action maximizes the expected value of the performance measure given the percept sequence to date

Rational $\neq$ omniscient
Rational $\neq$ clairvoyant
Rational $\neq$ successful

3. Agent functions and programs

An agent can be completely specified by an *agent function* mapping percept sequences to actions

(In principle, one can supply each possible sequence to see what it does. Obviously, a lookup table would usually be immense.)

One agent function (or a small equivalence class) is rational

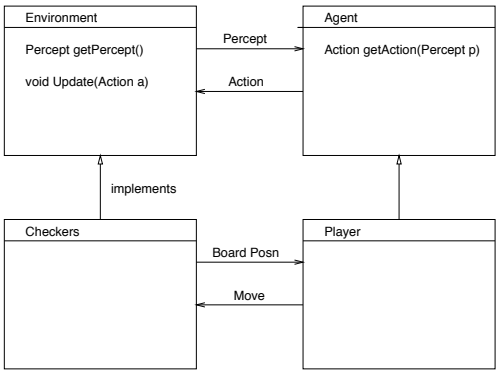
Aim: find a way to *implement* the rational agent function *concisely* and *efficiently*

An *agent program* implements an agent function: takes a single percept as input, keeps internal state, returns an action:

function SKELETON-AGENT(*percept*) **returns** action
static: *memory*, the agent's memory of the world
memory ← UPDATE-MEMORY(*memory*, *percept*)
action ← CHOOSE-BEST-ACTION(*memory*)
memory ← UPDATE-MEMORY(*memory*, *action*)
return *action*

In OO-speak $\cdots \Rightarrow$

3.1 Agents in Java




```
class Player implements Agent {
   ...

   Action getAction(Percept percept) {
     Move myNextMove;
     // my selection algorithm
     return myNextMove;
   }
   ...

}
```


4. Types of Agents

An *agent program* accepts percepts, combines them with any stored knowledge, and selects actions.

A *rational agent* will choose actions so as to maximise some *performance measure*. (In practice try to achieve "good' performance.)

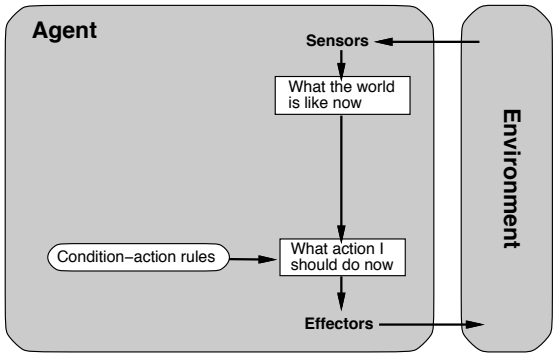
Four basic types in order of increasing generality:

– simple reflex agents
– reflex agents with state
– goal-based agents
– utility-based agents

4.1 Simple Reflex Agents

$\Rightarrow$ choose responses using *condition-action* rules (or *production rules*).

if *you see the car in front's brake lights* **then** *apply the brakes*



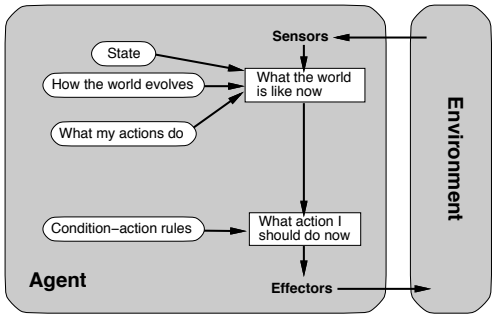
Some researches claim this is how simple life-forms (eg. insects) behave.

4.2 Agents that keep track of the world

While simply reacting to the current state of the world is adequate in some circumstances, most intelligent action requires more knowledge to work from:

- memory of the past
- knowledge about the effects of actions — how the world evolves
- requires internal *state*

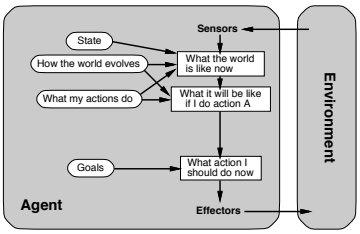
eg. You notice someone ahead signal to the bus. You know that this will cause the bus driver to stop (ideally), and conclude that you should change lanes.



4.3 Goal-based agents

Reacting to evolving world may keep you from crashing your car, but it doesn't tell you where to go!

Intelligent beings act in such a way as to try to achieve some *goals*.



Some goals are pretty simple. eg. Star Trek — "*to go where no-one has gone before*"

Some are more complex and require *planning* to achieve them. eg. Star Wars — *to defeat the Empire* — find a potential Jedi knight, ship him off to see Yoda, teach him to use the force, etc

Planning is a fundamental problem of AI. Usually requires *search* through the available actions to find an appropriate sequence.

Some people even say AI *is* really search!

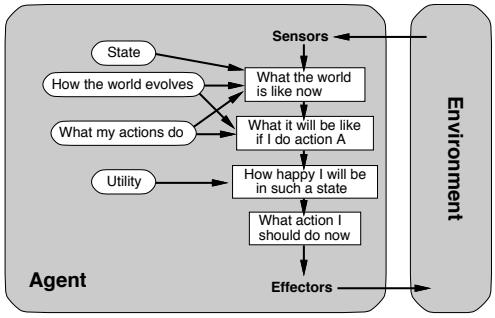
4.4 Utility-based agents

Two views:

1. a successful agent is only required to satisfy objective goals (emotions are a hindrance)
2. subjective measures such as happiness, security (safety), etc are important for success

eg. Luke could maximise his material benefits by turning to the dark side of the force, but he'd be very unhappy :-(

Researchers call this measure of happiness *utility* (since it sounds more scientific)



5. Next...

- Conceptualising the environment
  — states, actions (operators), goals
- The fundamental skills of an intelligent agent
  — problem solving and search

Artificial Intelligence

Topic 3

Problem Solving and Search

- ◇ Problem-solving and search
- ◇ Search algorithms
- ◇ Uninformed search algorithms
  - breadth-first search
  - uniform-cost search
  - depth-first search
  - iterative deepening search
  - bidirectional search

Reading: Russell and Norvig, Chapter 3

1. Problem Solving and Search

Seen that an intelligent agent has:

- knowledge of the *state* of the "world"
- a notion of how actions or *operations* change the world
- some *goals*, or states of the world, it would like to bring about

Finding a sequence of operations that changes the state of the world to a desired goal state is a *search problem* (or basic *planning problem*).

Search algorithms are the cornerstone of AI

In this section we see some examples of how the above is encoded, and look at some common *search strategies*.

1.1 States, Operators, Graphs and Trees

state — description of the world at a particular time
— impossible to describe the whole world
— need to abstract those attributes or properties that are important.

Examples

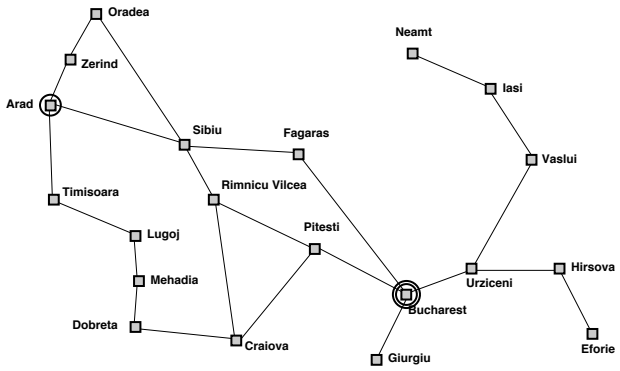
Example 1: Say we wish to drive from Arad to Bucharest. First we "discretise" the problem:

states — map of cities + our location
operators — drive from one city to the next
start state — driver located at Arad
goal state — driver located at Bucharest

Find solution:
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

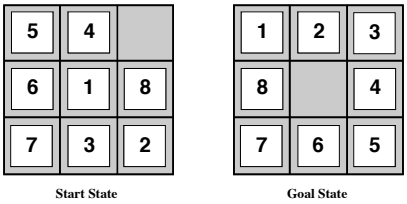
1.1 States, Operators, Graphs and Trees

The states and operators form a *graph* — states form *nodes*, operators form *arcs* or *edges*...



1.1 States, Operators, Graphs and Trees

Example 2: The *eight puzzle*



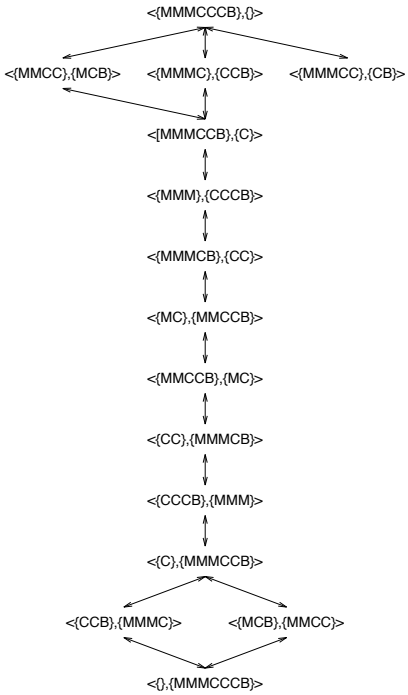
states — description of the positions of the numbered squares
operators — some alternatives...
(a) move a numbered square to an adjacent place, or
(b) move blank left, right, up or down — far fewer operators

1.1 States, Operators, Graphs and Trees

Example 3: *Missionaries and Cannibals*

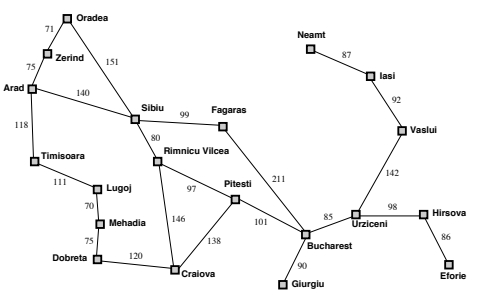
start state — 3 missionaries, 3 cannibals, and a boat that holds two people, on one side of river
goal state — all on other side
states — description of legal configurations (ie. where no-one gets eaten) of where the missionaries, cannibals, and boat are
operators — state changes possible using 2-person boat

1.1 States, Operators, Graphs and Trees



1.2 Operator costs

Graphs may also contain the *cost* of getting from one node to another (ie. associated with each operator, or each arc).



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

cost of path = sum of the costs of arcs making up the path

Usually concerned not just with finding a path to the goal, but finding a cheap path.

shortest path problem — find the cheapest path from a start state to a goal state

Where operator costs are not given, all operators are assumed to have unit cost.

1.3 Problem formulation

A *problem* is defined by four items:

initial state e.g., "at Arad"

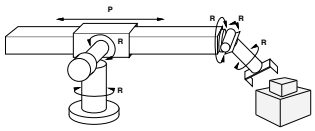
operators (or *successor function* $S(x)$)
e.g., Arad → Zerind Arad → Sibiu etc.

goal test, can be
explicit, e.g., $x =$ "at Bucharest"
implicit, e.g., "world peace"

path cost (additive)
e.g., sum of distances, number of operators executed, etc.

A *solution* is a sequence of operators leading from the initial state to a goal state.

1.4 Example: robotic assembly



states??: real-valued coordinates of
robot joint angles
parts of the object to be assembled

operators??: continuous motions of robot joints

goal test??: complete assembly *with no robot included!*

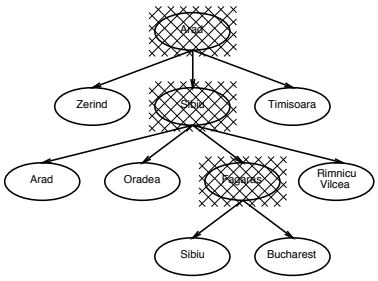
path cost??: time to execute

2. Search algorithms

Basic idea:
offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. *expanding* states)

function GENERAL-SEARCH(*problem, strategy*) **returns** a solution, or failure
initialize the search tree using the initial state of *problem*
loop do
if there are no candidates for expansion
then return failure
choose a leaf node for expansion according to *strategy*
if the node contains a goal state
then return the corresponding solution
else expand node and add resulting nodes to the search tree
end

2.1 General search example



2.2 General search in more detail

General search algorithm

Given a *start state* $s_0 \in S$, a *goal function* $g(s) \rightarrow \{true, false\}$, and an (optional) *terminal condition* $t(s) \rightarrow \{true, false\}$:

Initialise a set $U = \{s_0\}$ of *unvisited* nodes containing just the start state, and an empty set $V = \{\}$ of *visited* nodes.

1. If U is empty halt and report no goal found.
2. Select, according to some (as yet undefined) *strategy*, a node s from U.
3. (Optional) If $s \in V$ discard s and repeat from 1.
4. If $s(g) = true$ halt and report goal found.
5. (Optional) If $t(s) = true$ discard s and repeat from 1.
6. Otherwise move s to the set V, and add to U all the nodes reachable from s. Repeat from 1.

Step 3 is an *occurs check* for cycles.

- Some search strategies will still work without this
  $\Rightarrow$ trade-off — work to check if visited vs work re-searching same nodes again.
- Others may cycle forever.

With these cycles removed, the graph becomes a *search tree*.

2.3 Comparing search strategies

The *search strategy* is crucial — determines in which order the nodes are expanded. Concerned with:

completeness — does the strategy guarantee finding a solution if there is one
time complexity — how long does it take
space complexity — how much memory is needed to store states
optimality — does it guarantee finding the best solution

Time and space complexity are often measured in terms of

b — maximum *branching factor* of the search tree
d — *depth* of the *least-cost solution*
m — *maximum depth* of the state space (may be ∞)

2.4 Implementation of search algorithms

Many strategies can be implemented by placing unvisited nodes in a *queue* (Step 6) and always selecting the next node to expand from the front of the queue (Step 2)

$\Rightarrow$ the way the children of expanded nodes are placed in the queue determines the search strategy.

Many different strategies have been proposed. We'll look at some of the most common ones...

3. Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional search

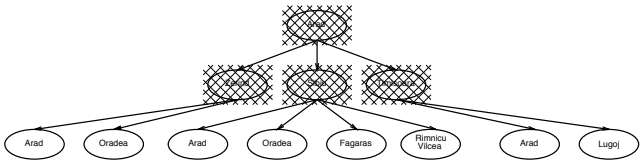
Later we will look at *informed* stategies that use additional information.

3.1 Breadth-first search

Expand shallowest unexpanded node

Implementation:

QUEUEINGFN = put successors at end of queue



⇒ *expand all nodes at one level before moving on to the next*

3.1 Breadth-first search

Complete? Yes (if b is finite)

Time? $O(1 + b + b^2 + b^3 + \ldots + b^d) = O(b^d)$, i.e., exponential in d

Space? $O(b^d)$ (all leaves in memory)

Optimal? *Yes (if cost = 1 per step); not optimal in general*

Space is the big problem; can easily generate nodes at 1MB/sec so 24hrs = 86GB.

Good example of computational explosion. . .

Assume branching factor of 10, 1000 nodes/sec, 100 bytes/node.

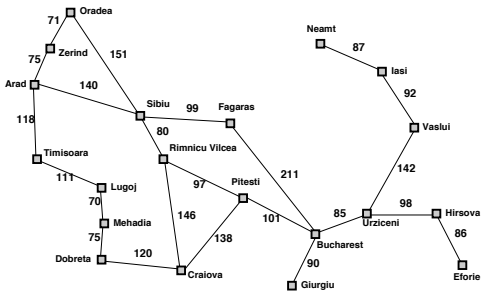
Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Welcome to AI!

3.2 Uniform-cost search

Problem: Varying cost operations

eg. Romania with step costs in km. . .



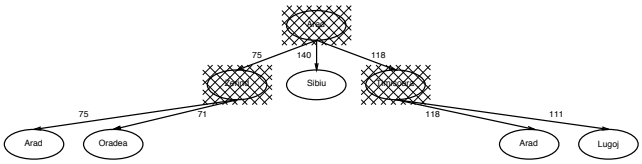
Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

3.2 Uniform-cost search

Expand least total cost unexpanded node

Implementation:

QUEUEINGFN = insert in order of increasing path cost



3.2 Uniform-cost search

Complete? Yes (if step cost ≥ 0)

Time? # of nodes with path cost $g \leq$ cost of optimal solution

Space? # of nodes with $g \leq$ cost of optimal solution

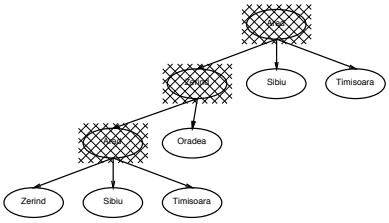
Optimal? *Yes (if step cost ≥ 0)*

3.3 Depth-first search

Follow one path until you can go no further, then backtrack to last choice point and try another alternative.

Implementation:

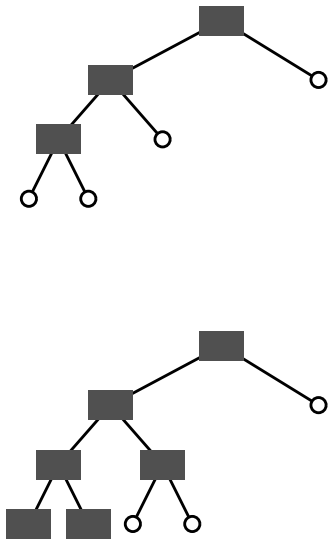
QUEUEINGFN = insert successors at front of queue (or use *recursion* — "queueing" performed automatically by internal stack)

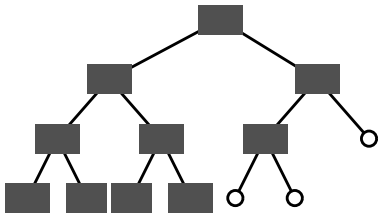


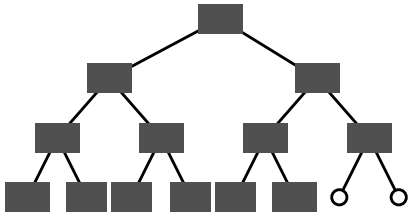
Occurs check or terminal condition needed to prevent infinite cycling.

3.3 Depth-first search

Finite tree example. . .







3.3 Depth-first search

Complete? No: fails in infinite-depth spaces. — complete for finite tree (in particular, require cycle check).

Time? $O(b^m)$.

— may do very badly if m is much larger than d

— may be much faster than breadth-first if solutions are dense

Space? $O(bm)$, i.e., linear space!

Optimal? No

Space performance is big advantage.

Time, completeness and optimality can be big disadvantages.

3.4 Depth-limited search

= depth-first search with depth limit l

Implementation:

Nodes at depth l have no successors.

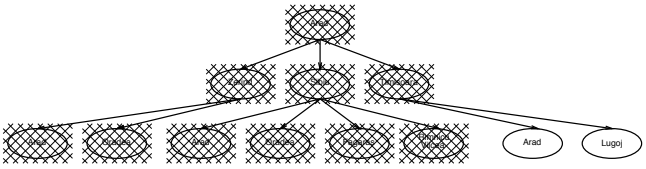
Can be implemented by our *terminal condition*.

Sometimes used to apply depth-first to infinite (or effectively infinite) search spaces. Take "best" solution found with limited resources. (See Game Playing. . .)

Also in. . .

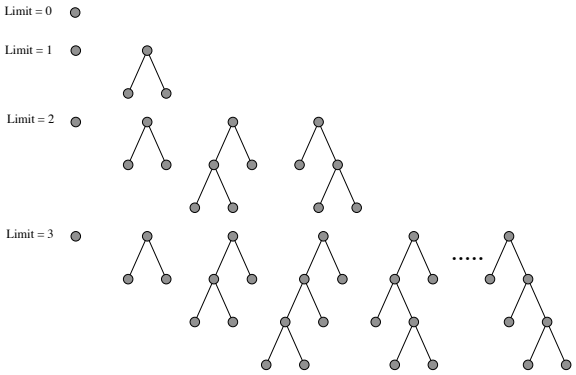
3.5 Iterative deepening search

"Probe" deeper and deeper (often bounded by available resources).



3.5 Iterative deepening search

Summary view. . .



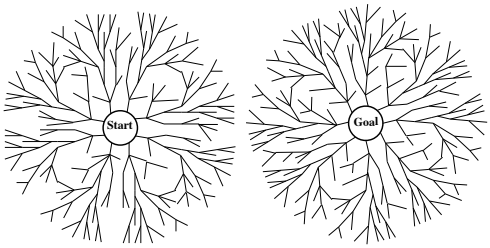
3.5 Iterative deepening search

Complete? Yes
Time? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d \rightarrow O(b^d)$
Space? $O(bd)$
Optimal? *Yes, if step cost = 1*
Can also be modified to explore uniform-cost tree

How do the above compare with

- breadth-first?
- depth-first?

3.6 Bidirectional search



Tends to expand fewer nodes than unidirectional, but raises other difficulties — eg. how long does it take to check if a node has been visited by other half of search?

4. Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored.

- Abstraction reveals states and operators.
- Evaluation by goal or utility function.
- Strategy implemented by queuing function (or similar).

Variety of uninformed search strategies. . .

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.

Artificial Intelligence

Topic 4

Informed search algorithms

- ◇ Best-first search
- ◇ Greedy search
- ◇ A* search
- ◇ Admissible heuristics
- ◇ Memory-bounded search
- ◇ IDA*
- ◇ SMA*

Reading: Russell and Norvig, Chapter 4, Sections 1–3

1. Informed (or best-first) search

Recall uninformed search:

- select nodes for expansion on basis of distance from start
- uses only information contained in the graph
- no indication of distance to go!

Informed search:

- select nodes on basis of some estimate of *distance to goal*!
- requires additional information — *evaluation function*, or *heuristic rules*
- choose "best" (most promising) alternative ⇒ *best-first search*.

Implementation:

QUEUEINGFN = insert successors in decreasing order of desirability

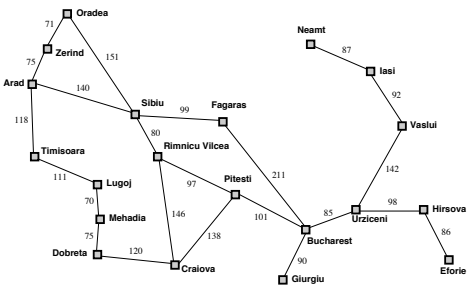
Examples:

- greedy search
- A* search

2. Greedy search

Assume we have an estimate of the distance to the goal.

For example, in our travelling to Bucharest problem, we may know straight-line distances to Bucharest...



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

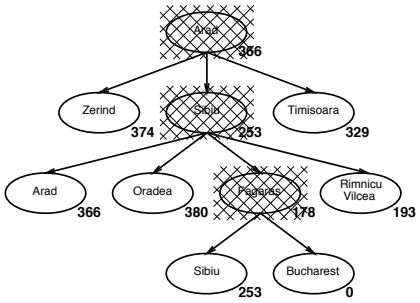
Greedy search always chooses to visit the candidate node with the smallest estimate

⇒ that which *appears* to be closest to goal

Evaluation function $h(n)$ (heuristic)
= estimate of cost from n to $goal$

E.g., $h_{\mathrm{SLD}}(n)$ = straight-line distance from n to Bucharest

2. Greedy search



2. Greedy search

Complete? No, in general. e.g. can get stuck in loops,
Iasi → Neamt → Iasi → Neamt →
Complete in finite space with repeated-state checking

Time? $O(b^m)$, but a good heuristic can give dramatic improvement

Space? $O(b^m)$—keeps all nodes in memory

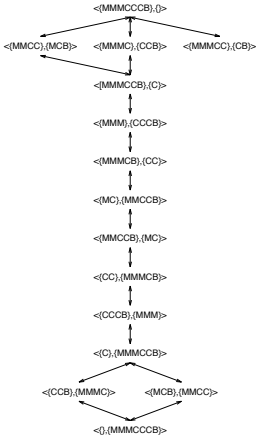
Optimal? *No*

2.1 Means-ends Analysis

Example of greedy search. (Used in SOAR problem solver.)

Heuristic: Pick operations that reduce as much as possible the "difference" between the intermediate state and goal state.

eg. Missionaries and cannibals



Indicates best choice in all states except for ⟨{MC},{MMCCB}⟩ and ⟨{MMCCB},{MC}⟩ (though no other choices).

3. A* search

Greedy search minimises estimated cost to goal, and thereby (hopefully) reduces search cost, but is neither optimal nor complete.

Uniform-cost search minimises path cost so far and is optimal and complete, but is costly.

Can we get the best of both worlds. . . ?

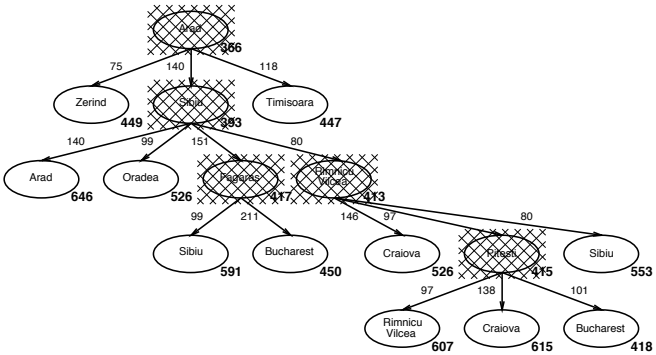
Yes! Just add the two together to get *estimate of total path length* of solution as our evaluation function. . .

Evaluation function

$$f(n) = g(n) + h(n)$$

$g(n)$ = cost so far to reach n
$h(n)$ = estimated cost to goal from n
$f(n)$ = estimated total cost of path through n to goal

3. A* search



3. A* search

A heuristic h is *admissible* iff

$$h(n) \leq h^*(n) \quad \text{for all } n$$

where $h^*(n)$ is the *true* cost from n.

i.e. $h(n)$ *never overestimates*

e.g., $h_{\text{SLD}}(n)$ never overestimates the actual road distance

Can prove:

if $h(n)$ is admissible, $f(n)$ provides a complete and optimal search!

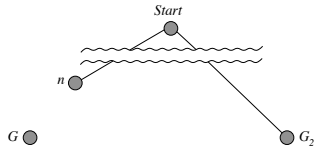
$\Rightarrow$ called A* search.

3.1 Optimality of A*

Theorem: A* search is optimal

Proof

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.



$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\ &\geq g(n) + h(n) && \text{since } h \text{ is admissible} \\ &= f(n) \end{aligned}$$

Since $f(G_2) > f(n)$, A* will not select G_2 for expansion

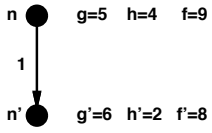
3.2 Monotonicity and the pathmax equation

To get a more intuitive view, we consider the f-values along any path.

For most admissible heuristics, f-values increase *monotonically* (see Romania problem).

For some admissible heuristics, f may be nonmonotonic — ie it may *decrease* at some points.

e.g., suppose n' is a successor of n



But $f' = 8$ is redundant!

$f(n) = 9 \Rightarrow$ true cost of a path through n is ≥ 9

$\Rightarrow$ true cost of a path through n' is ≥ 9

Pathmax modification to A*:

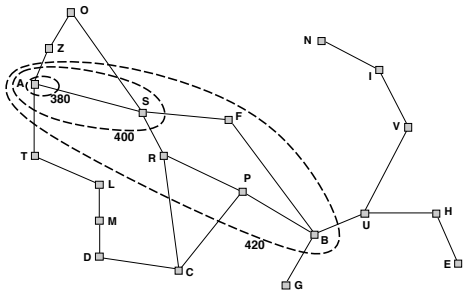
$$f(n') = max(g(n') + h(n'), f(n))$$

With pathmax, f is always increases monotonically $\cdots \rightsquigarrow$

3.3 Contours

Lemma: A* (with pathmax) expands nodes in order of increasing f value

Gradually adds "f-contours" of nodes (cf. breadth-first/uniform-cost adds layers or "circles" — A* "stretches" towards goal)



If f^* is cost of optimal solution path:

- A* expands all nodes with $f(n) < f^*$
- A* expands some nodes with $f(n) = f^*$

Can see intuitively that A* is complete and optimal.

3.4 Properties of A*

Complete Yes, unless there are infinitely many nodes with $f \leq f(G)$
Time Exponential in [relative error in h × length of soln.]
Space Keeps all nodes in memory (see below)
Optimal Yes—cannot expand f_{i+1} until f_i is finished

Among optimal algorithms of this type A* is *optimally efficient*!

ie. no other algorithm is guaranteed to expand fewer nodes.

"Proof"

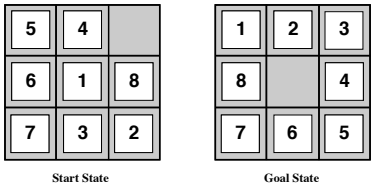
Any algorithm that does not expand all nodes in each contour may miss an optimal solution.

4. Admissible heuristics

Straight line distance is an obvious heuristic for distance planning. What about other problems?

This section ⇒ examine heuristics in more detail.

E.g., two heuristics for the 8-puzzle:



$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)

$h_1(S)$ =??

$h_2(S)$ =??

Are both admissible?

4.1 Measuring performance

Quality of heuristic can be characterised by *effective branching factor* $b*$.

Assume:

- A* expands N nodes
- solution depth d

b^* is branching factor of uniform tree, depth d with N nodes:

$$N = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

- tends to remain fairly constant over problem instances
- can be determined empirically
  ⇒ fairly good guide to heuristic performance

a good heuristic would have b^* close to 1

4.1 Measuring performance

Example

Effective branching factors for iterative deepening search and A* with h_1 and h_2 (averaged over 100 randomly generated instances of 8-puzzle for each solution length):

	Search Cost			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

- informed better than uninformed
- h_2 better than h_1

Is h_2 always better than h_1?

4.2 Dominance

Yes!

We say h_2 *dominates* h_1 if $h_2(n) \geq h_1(n)$ for all n (both admissible).

dominance $\Rightarrow$ *better efficiency*

— h_2 will expand fewer nodes on average than h_1

"Proof"

A* will expand all nodes n with $f(n) < f^*$.

$\Rightarrow$ A* will expand all nodes with $h(n) < f^* - g(n)$

But $h_2(n) \geq h_1(n)$ so all nodes expanded with h_2 will also be expanded with h_1 (h_1 may expand others as well).

always better to use an (admissible) heuristic function with higher values

4.3 Inventing heuristics — relaxed problems

- How can we come up with a heuristic?
- Can the computer do it automatically?

A problem is *relaxed* by reducing restrictions on operators

cost of exact solution of a relaxed problem is often a good heuristic for original problem

Example

- if the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then $h_1(n)$ gives the shortest solution
- if the rules are relaxed so that a tile can move to *any adjacent square*, then $h_2(n)$ gives the shortest solution

Note: Must also ensure heuristic itself is not too expensive to calculate.

Extreme case: perfect heuristic can be found by carrying out search on original problem.

4.4 Automatic generation of heuristics

If problem is defined in suitable formal language $\Rightarrow$ may be possible to construct relaxed problems automatically.

eg. 8-puzzle operator description

A is adjacent to B & B is blank $\rightarrow$ can move from A to B

Relaxed rules

A is adjacent to B $\rightarrow$ can move from A to B

B is blank $\rightarrow$ can move from A to B

can move from A to B

ABSOLVER (Prieditis 1993)

- new heuristic for 8-puzzle better than any existing one
- first useful heuristic for Rubik's cube!

5. Memory-bounded Search

Good heuristics improve search, but many problems are still too hard.

Usually memory restrictions that impose a hard limit.

(eg. recall estimates for breadth-first search

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

)

This section — algorithms designed to save memory.

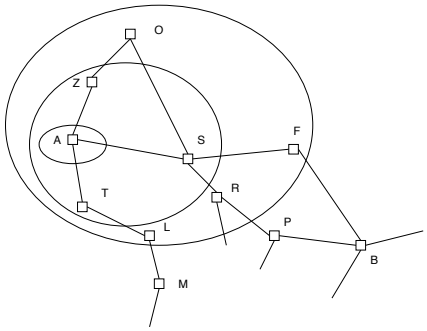
- IDA*
- SMA*

5.1 Iterative Deepening A* (IDA*)

Recall uninformed search

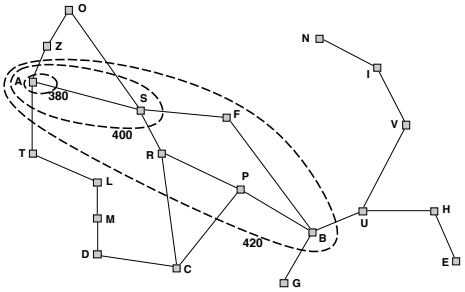
- uniform-cost/breadth-first search
  + completeness, optimality
  − exponential space usage
- depth-first
  + linear space usage
  − incomplete, suboptimal

Solution: iterative-deepening ⇒ explores "uniform-cost trees", or "contours", using linear space.



5.1 Iterative Deepening A* (IDA*)

Can we do the same with A*?



Contours more directed, but same technique applies!

Modify depth-limited search to use f-cost limit, rather than depth limit ⇒ *IDA**

Complete? *Yes* (with admissible heuristic)
Optimal? *Yes* (with admissible heuristic)
Space? *Linear* in path length
Time? *?*

5.1 Iterative Deepening A* (IDA*)

Time complexity of IDA*

Depends on number of different values f can take on.

- small number of values, few iterations
  eg. 8-puzzle
- many values, many iterations
  eg. Romania example, each state has different heuristic ⇒ only one extra town in each contour

Worst case: A* expands N nodes, IDA* goes through N iterations

$$1 + 2 + \cdots + N \Rightarrow O(N^2)$$

A solution: increase f-cost limit by fixed amount ϵ in each iteration

⇒ returns solutions at worst ϵ worse than optimal

Called *ϵ-admissible*.

IDA* was first memory-bounded optimal heuristic algorithm and solved many practical problems.

5.2 Simplified memory-bounded A* (SMA*)

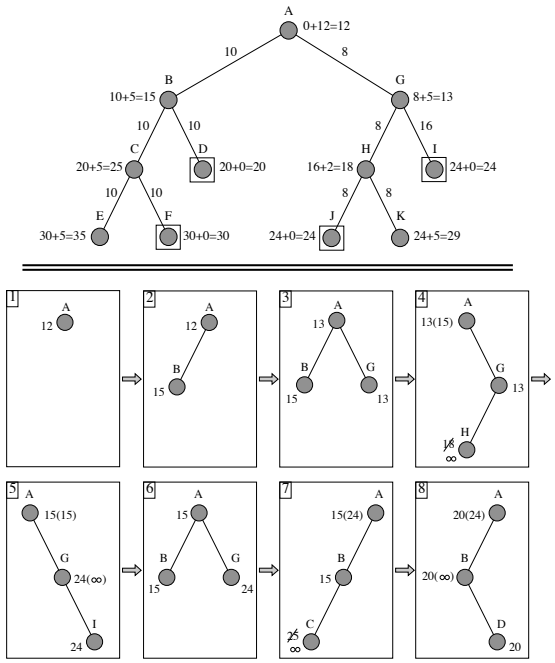
- *Uses all available memory*.
- Complete if available memory is sufficient to store the *shallowest* solution path.
- Optimal if available memory is sufficient to store the *shallowest* solution path. Otherwise best solution given available memory.

How does it work?

- Needs to generate successor nodes but no memory left ⇒ drop, or *"forget"*, least promising nodes.
- Keep record of best f-cost of forgotten nodes in ancestor.
- Only regenerate nodes if all more promising options are exhausted.

Example (values of forgotten nodes in parentheses) · · · ⇒

5.2 Simplified memory-bounded A* (SMA*)



5.2 Simplified memory-bounded A* (SMA*)

- Solves significantly more difficult problems than A*.
- Performs well on highly-connected state spaces and real-valued heuristics on which A* has difficulty.
- Susceptible to continual "switching" between candidate solution paths.
  ie. *limit in memory can lead to intractible computation time*

Artificial Intelligence

Topic 5

Game playing

- ◇ broadening our world view — dealing with incompleteness
- ◇ why play games?
- ◇ perfect decisions — the MINIMAX algorithm
- ◇ dealing with resource limits
  — evaluation functions
  — cutting off search
- ◇ alpha-beta pruning
- ◇ game-playing agents in action

Reading: Russell and Norvig, Chapter 5

1. Broadening our world view

We have assumed we are dealing with world descriptions that are:

complete — all necessary information about the problem is available to the search algorithm

deterministic — effects of actions are uniquely determined

Real-world problems are rarely complete and deterministic. . .

Sources of Incompleteness

sensor limitations — not possible to gather enough information about the world to completely know its state — includes the future!

intractability — full state description is too large to store, or search tree too large to compute

Sources of (Effective) Nondeterminism

- humans, the weather, stress fractures, dice, . . .

Aside. . .

Debate: incompleteness ↔ nondeterminism

1.1 Approaches for Dealing with Incompleteness

contingency planning

- build all possibilities into the plan
- may make the tree very large
- can only guarantee a solution if the number of contingencies is finite and tractable

interleaving or adaptive planning

- alternate between planning, acting, and sensing
- requires extra work during execution — planning cannot be done in advance (or "off-line")

strategy learning

- learn, from looking at examples, strategies that can be applied in any situation
- must decide on parameterisation, how to evaluate states, how many examples to use, . . . black art??

2. Why Play Games?

- abstraction of real world
- well-defined, clear state descriptions
- limited operations, clearly defined consequences

but!

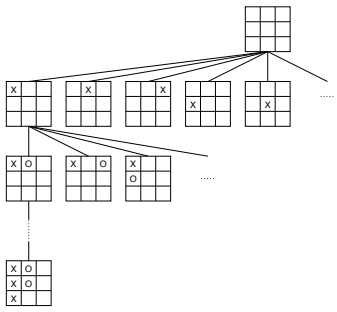
- provide a mechanism for investigating many of the real-world issues outlined above
  ⇒ more like the real world than examples so far

Added twist — the domain contains hostile agents (also making it like the real world. . . ?)

2.1 Examples

Tractable Problem with Complete Information

Noughts and crosses (tic-tac-toe) for control freaks — you get to choose moves for both players!



Stop when you get to a goal state.

- What uninformed search would you select? How many states visited?
- What would be an appropriate heuristic for an informed search? How many states visited?

2.1 Examples

Tractable Contingency Problem

Noughts and crosses — allow for all the oponents moves. (Oponent is non-deterministic.)

How many states?

Intractable Contingency Problem

Chess

- average branching factor 35, approx 50 operations
  ⇒ search tree has about 35^{100} nodes (although only about 10^{40} different legal positions)!
- cannot solve by brute force, must use other approaches, eg.
  – interleave time- (or space-) limited search with moves
    ⇒ this section
    * algorithm for perfect play (Von Neumann, 1944)
    * finite horizon, approximate evaluation (Zuse, 1945; Shannon, 1950; Samuel, 1952–57)
    * pruning to reduce costs (McCarthy, 1956)
  – learn strategies that determine what to do based on some aspects of the current position
    ⇒ later in the course

3. Perfect Decisions — MINIMAX Algorithm

Perfect play for deterministic, perfect-information games

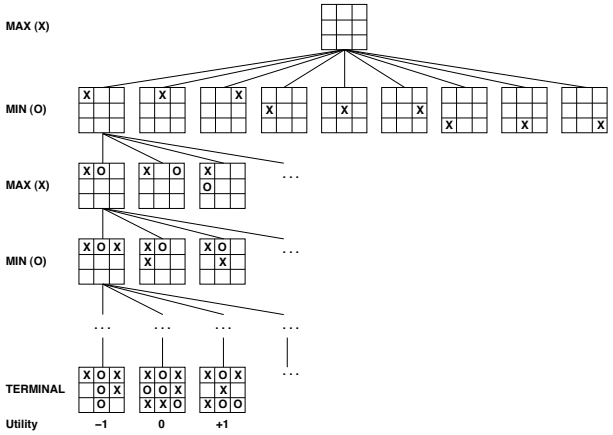
- two players, MAX and MIN, both try to win
- MAX moves first
  $\Rightarrow$ can MAX find a strategy that *always wins?*

Define a game as a kind of search problem with:

- initial state
- set of legal moves (operators)
- terminal test — is the game over?
- utility function — how good is the outcome for each player?

eg. Tic-tac-toe — can MAX choose a move that always results in a terminal state with a utility of $+1$?

3. Perfect Decisions — MINIMAX Algorithm

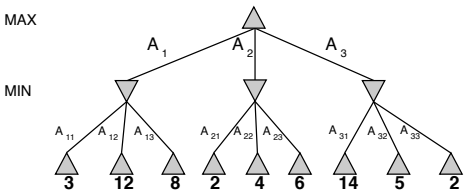


Even for this simple game the search tree is large.

Try an even simpler game...

3. Perfect Decisions — MINIMAX Algorithm

eg. Two-ply (made-up game)



(one move deep, two *ply*)

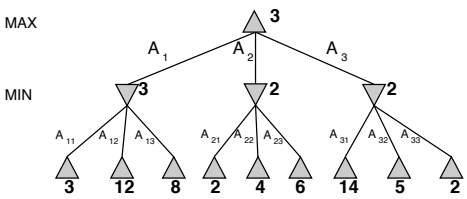
- MAX's aim — maximise utility of terminal state
- MIN's aim — minimise it
- what is MAX's optimal strategy, *assuming* MIN makes the best possible moves?

3. Perfect Decisions — MINIMAX Algorithm

function MINIMAX-DECISION(*game*) **returns** *an operator*
  for each *op* **in** OPERATORS[*game*] **do**
    VALUE[*op*] ← MINIMAX-VALUE(APPLY(*op*, *game*), *game*)
  end
  return the *op* with the highest VALUE[*op*]

---

function MINIMAX-VALUE(*state*, *game*) **returns** *a utility value*
  if TERMINAL-TEST[*game*](*state*) **then**
    return UTILITY[*game*](*state*)
  else if MAX is to move in *state* **then**
    return the highest MINIMAX-VALUE of SUCCESSORS(*state*)
  else
    return the lowest MINIMAX-VALUE of SUCCESSORS(*state*)



3. Perfect Decisions — MINIMAX Algorithm

Complete Yes, if tree is finite (chess has specific rules for this)
Optimal Yes, against an optimal opponent. Otherwise??
Time complexity $O(b^m)$
Space complexity $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
$\Rightarrow$ exact solution completely infeasible

Resource limits

Usually time: suppose we have 100 seconds, explore 10^4 nodes/second
$\Rightarrow$ $\underline{10^6}$ nodes per move

Standard approach:

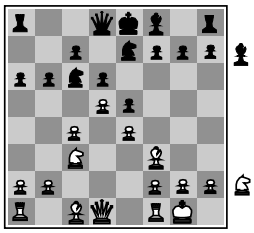
- *cutoff test*
  e.g., depth limit (perhaps add *quiescence search*)
- *evaluation function*
  = estimated desirability of position

4. Evaluation functions

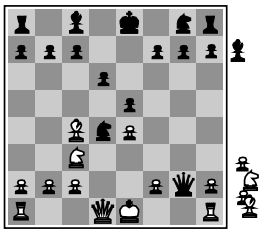
Instead of stopping at terminal states and using utility function, cut off search and use a heuristic *evaluation function*.

Chess players have been doing this for years...

simple — 1 for pawn, 3 for knight/bishop, 5 for rook, etc
more involved — centre pawns, rooks on open files, etc



Black to move
White slightly better



White to move
Black winning

Can be expressed as *linear weighted sum* of *features*

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

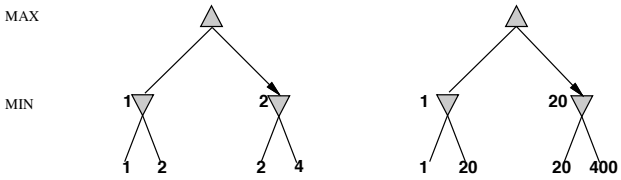
e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (number of black queens)

4.1 Quality of evalation functions

Success of program depends *critically* on quality of evalutation function.

- agree with utility function on terminal states
- time efficient
- reflect chances of winning

Note: Exact values don't matter



Behaviour is preserved under any *monotonic* transformation of EVAL

Only the order matters:
payoff acts as an *ordinal utility* function

5. Cutting off search

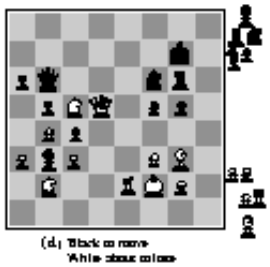
Options...

- fixed depth limit
- iterative deepening (fixed time limit) — more robust

Problem — inaccuracies of evaluation function can have disastrous consequences.

5.1 Non-quiescence problem

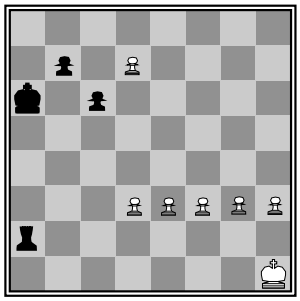
Consider chess evaluation function based on material advantage. White's depth limited search stops here...



Looks like a win to white — actually a win to black.

Want to stop search and apply evaluation function in positions that are *quiescent*. May perform *quiescence search* in some situations — eg. after capture.

5.2 Horizon problem



Black to move

Win for white, but black may be able to chase king for extent of its depth-limited search, so does not see this. Queening move is "pushed over the horizon".

No general solution.

6. Alpha-beta pruning

Consider MINIMAX with reasonable evaluation function and quiescent cut-off. Will it work in practice?

Assume can search approx 5000 positions per second. Allowed approx 150 seconds per move. Order of 10^6 positions per move.

$$b^m = 10^6, \quad b = 35 \quad \Rightarrow \quad m = 4$$

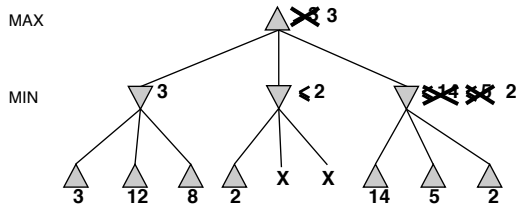
4-ply lookahead is a hopeless chess player!

4-ply ≈ human novice
8-ply ≈ typical PC, human master
12-ply ≈ Deep Blue, Kasparov

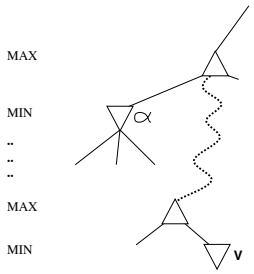
But do we need to search all those positions? Can we eliminate some before we get there — *prune* the search tree?

One method is *alpha-beta* pruning...

6.1 α–β pruning example



6.2 Why is it called α–β?



α is the best value (to MAX) found so far off the current path

If V is worse than α, MAX will avoid it $\Rightarrow$ prune that branch

Define β similarly for MIN

6.3 The α–β algorithm

Basically MINIMAX + keep track of α, β + prune

function MAX-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*
  inputs: *state*, current state in game
    game, game description
    α, the best score for MAX along the path to *state*
    β, the best score for MIN along the path to *state*
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\alpha \leftarrow$ MAX(α, MIN-VALUE(*s*, *game*, α, β))
    if $\alpha \geq \beta$ **then return** β
  end
  return α

---

function MIN-VALUE(*state*, *game*, α, β) **returns** the minimax value of *state*
  if CUTOFF-TEST(*state*) **then return** EVAL(*state*)
  for each *s* **in** SUCCESSORS(*state*) **do**
    $\beta \leftarrow$ MIN(β, MAX-VALUE(*s*, *game*, α, β))
    if $\beta \leq \alpha$ **then return** α
  end
  return β

6.4 Properties of α–β

Pruning *does not* affect final result

Good move ordering improves effectiveness of pruning

With "perfect ordering," time complexity $= O(b^{m/2})$
  $\Rightarrow$ *doubles* depth of search
  $\Rightarrow$ can easily reach depth 8 and play good chess

Perfect ordering is unknown, but a simple ordering (captures first, then threats, then forward moves, then backward moves) gets fairly close.

Can we learn appropriate orderings? $\Rightarrow$ *speedup learning*

(Note complexity results assume idealized tree model:

- nodes have same branching factor b
- all paths reach depth limit d
- leaf evaluations randomly distributed

Ultimately resort to empirical tests.)

7. Game-playing agents in practice

Games that don't include chance

Checkers: Chinook became world champion in 1994 after 40-year-reign of human world champion Marion Tinsley (who retired due to poor health). Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match (not a World Championship) in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

7. Game-playing agents in practice

Games that include an element of chance

Dice rolls increase b: 21 possible rolls with 2 dice

Backgammon $\approx$ 20 legal moves (can be 6,000 with 1-1 roll)

$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, probability of reaching a given node shrinks
$\Rightarrow$ value of lookahead is diminished

α–β pruning is much less effective

TDGammon uses depth-2 search + very good Eval
$\approx$ world-champion level

8. Summary

Games are fun to work on! (and can be addictive)

They illustrate several important points about AI

♢ problems raised by
— incomplete knowledge
— resource limits

♢ perfection is unattainable $\Rightarrow$ must approximate

Games are to AI as grand prix racing is to automobile design

Artificial Intelligence

Topic 6

Agents that Learn

♢ why learn?
♢ general model of learning agents
♢ inductive learning
♢ learning decision trees

Reading: Russell & Norvig, Chapter 18

1. Why Learn?

So far, all intelligence comes from the designer:

- time consuming for designer
- *restricts capabilities of agent*

Learning agents can:

- act autonomously
- deal with unknown environments
- synthesise rules/patterns from large volumes of data
- handle complex data
- improve their own performance

2. General Model of Learning Agents

Idea: percepts used not just for acting, but for improving future performance.

Four basic components. . .

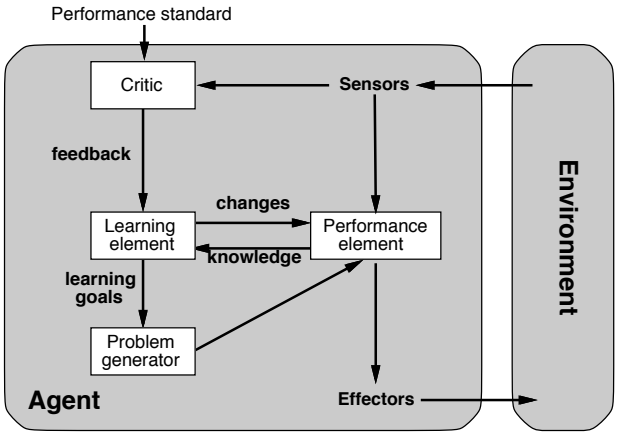
Performance element
— responsible for selecting actions for good agent performance
— agent function considered previously: percepts →actions

Learning element
— responsible for improving performance element
— requires feedback on how the agent is doing

Critic element
— responsible for providing feedback
— comparison with objective performance standard (outside agent)

Problem generator
— responsible for generating new experience
— requires *exploration* — taking suboptimal actions

2. General Model of Learning Agents



E.g. taxi driver agent

performance element — Lets take Winthrop Ave, I know it works.

problem generator — Nah, lets try Mounts Bay Rd for a change. You never know, it may be quicker.

critic — Wow, it was 5 mins quicker, and what a lovely view!

learning element — Yeh, in future we'll take Mounts Bay Rd.

2.1 The Learning Element

Two separate goals:

1. Improve *outcome* of performance element
   — how good is the solution
2. Improve *time performance* of performance element
   — how fast does it reach a solution
   — known as *speedup learning*

Learning systems may work on one or both tasks.

Design of a learning element is affected by four issues:

1. the components of the performance element to be improved
2. representation of those components
3. feedback available
4. prior information available

2.1 The Learning Element

Components of the performance element

Can use:

1. direct mapping from states to actions
2. means to infer properties (model) of the world from percept sequence
3. information on how the world evolves
4. information about how actions change the world
5. utility information about desirability of states
6. action-value information about desirability of actions in states
7. goals whose achievement maximises utility

Each component can be learned, given appropriate feedback.

E.g. our taxi driver agent

- Mounts Bay Rd has a nicer view (5)
- Taking Mounts Bay Rd → arrive more quickly (4,7)
- If travelling from Perth to UWA, always take Mounts Bay Rd (6)

2.1 The Learning Element

Representation of components

Examples:

- utility functions, eg linear weighted polynomials for game-playing agent
- logical sentences for reasoning agent
- probabilistic descriptions, eg belief networks for decision-theoretic agent

Available feedback

supervised learning
— agent provided with both inputs and correct outputs
— usually by a "teacher"

reinforcement learning
— agent chooses actions
— receives some *reward* or *punishment*

unsupervised learning
— no hint about correct outputs
— can only learn relationships between percepts

2.1 The Learning Element

Prior knowledge

- agent begins with a *tabula rasa* (empty slate)
- agent makes use of background knowledge

In practice learning is hard $\Rightarrow$ use background knowledge if available.

Learning = Function approximation

All components of performance element can be described mathematically by a function. eg.

- how the world evolves: f: state $\mapsto$ state
- goal: f: state $\mapsto \{0,1\}$
- utilities: f: state $\mapsto [-\infty, \infty]$
- action values: f: (state,action) $\mapsto [-\infty, \infty]$

all learning can be seen as learning a function

3. Inductive Learning

Assume f is function to be learned.

Learning algorithm supplied with sample inputs and corresponding outputs $\Rightarrow$ supervised learning

Define *example* (or *sample*): pair $(x, f(x))$

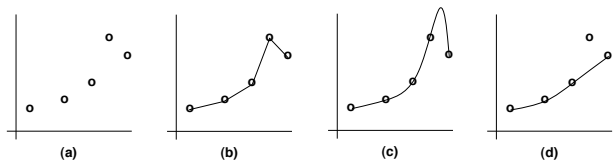
Task:

given a set of examples of f, return a function h that approximates f.

h is called a *hypothesis*

Task is called *pure inductive inference* or *induction*.

Many choices for h



A preference for one over another is called a *bias*.

3. Inductive Learning

Example: Simple reflex agent

Sample is a pair (*percept*,*action*)

eg. *percept* — chess board position

action — best move supplied by friendly grandmaster

Algorithm:

global *examples* $\leftarrow \{\}$

---

function REFLEX-PERFORMANCE-ELEMENT(*percept*) **returns** an action

if (*percept*, *a*) **in** *examples* **then return** *a*
else
$h \leftarrow$ INDUCE(*examples*)
return *h*(*percept*)

---

procedure REFLEX-LEARNING-ELEMENT(*percept*, *action*)
inputs: *percept*, feedback percept
action, feedback action

examples $\leftarrow$ *examples* $\cup \{$(*percept*, *action*)$\}$

3. Inductive Learning

Many variations possible, eg.

- *incremental learning* — learning element updates h with each new sample
- *reinforcement* — agent receives feedback on quality of action

Many possible representations of h — choice of representation is critical

- type of learning algorithm that can be used
- whether learning problem is feasible
- expressiveness vs tractability

4. Learning Decision Trees

Decision tree

- input — description of a situation
  — abstracted by set of *properties*, *parameters*, *attributes*, *features*,...
- output — boolean (yes/no) decision
  — can also be thought of as defining a *categorisation*, *classification* or *concept*
  $\Rightarrow$ set of situations with a positive response

$$f\colon \text{ situation} \mapsto \{0,1\}$$

We consider

1. decision trees as performance elements
2. inducing decision trees (learning element)

4.1 Decision Trees as Performance Elements

Example

Problem: decide whether to wait for a table at a restaurant

Aim: provide a definition, expressed as a decision tree, for the goal concept "Will Wait"

First step: identify attributes

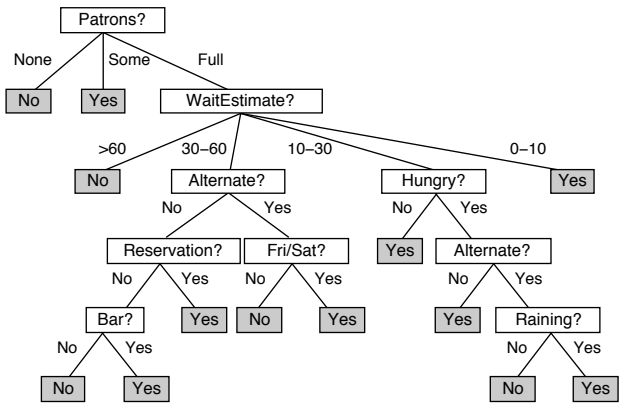
— what factors are necessary to make a rational decision

For example:

1. alternative nearby?
2. bar?
3. Friday/Saturday?
4. hungry?
5. patrons?
6. price?
7. raining?
8. reservation?
9. type of food?
10. estimated wait?

Example of a decision tree $\cdots\rightsquigarrow$

4.1 Decision Trees as Performance Elements



Choice of attributes is *critical* $\Rightarrow$ determines whether an appropriate function can be learned ("garbage-in, garbage-out")

No matter how good a learning algorithm is, it will fail if appropriate features cannot be identified ($cf.$ neural nets)

In real world problems feature selection is often the hardest task!

(black art?)

4.1 Decision Trees as Performance Elements

Expressiveness of Decision Trees

- limited to propositional (boolean) problems
  - cannot input arbitrary number of restaurants and ask to choose
  - cannot handle continuous information
- fully expressive within class of propositional problems
- may be exponentially large w.r.t. no. inputs

Number of trees (size of hypothesis space)

n attributes

$\Downarrow$

2^n combinations of inputs

$\Downarrow$

2^{2^n} different functions

eg. 6 attributes, approx. 2×10^{19} functions to choose from

$\Rightarrow$ lots of work for learning element!

4.2 Decision Tree Induction

Terminology...

example — $(\{attributes\}, value)$
positive example — $value = true$
negative example — $value = false$
training set — set of examples used for learning

Example training set:

Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

Goal: find a decision tree that agrees with all the examples in the training set

4.2 Decision Tree Induction

Trivial Solution

Branch on each attribute in turn, until you reach a distinct leaf for each example.

Problems

- tree is bigger than needed — does not find *patterns* that "summarise" or "simplify" information
- cannot answer for examples that haven't been seen $\Rightarrow$ cannot *generalise*

Two sides of the same coin!

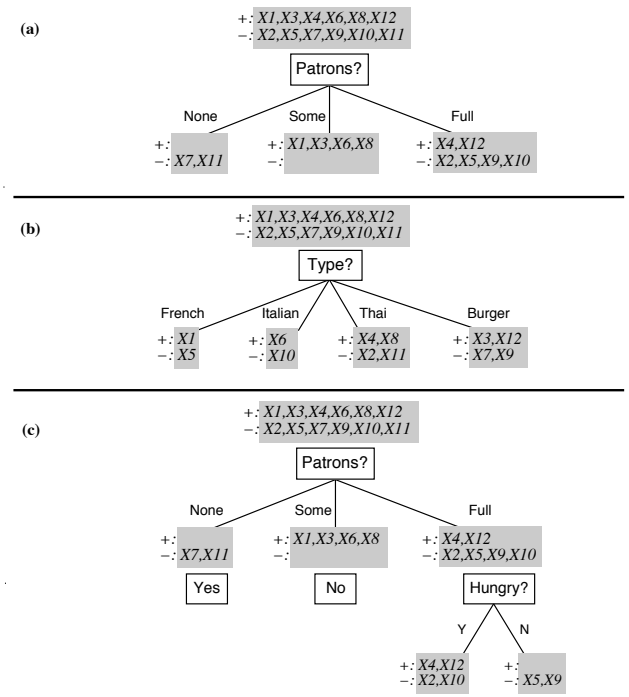
> *Ockham's razor: the most likely hypothesis is the simplest one that is consistent with the examples*

Finding smallest tree is intractable, but can use heuristics to generate reasonably good solution.

Basic idea: recursively select the most "important", or *discriminating*, attribute for successive branch points

(*discriminating* — formally requires *information theory*, but humans do it intuitively $\Rightarrow$ sort the "men from the boys", "sheep from the lambs", "oats from the chaff"...)

4.2 Decision Tree Induction



4.2 Decision Tree Induction

Recursive algorithm

Base cases

- remaining examples all +ve or all −ve — stop and label "yes" or "no"
- no examples left — no relevant examples have been seen, return majority in parent node
- no attributes left — problem: either data is inconsistent ("noisy", or nondeterministic) or attributes chosen were insufficient to adequately discriminate (start again, or use majority vote)

Recursive case

- both +ve and −ve examples — choose next most discriminating attribute and repeat

4.2 Decision Tree Induction

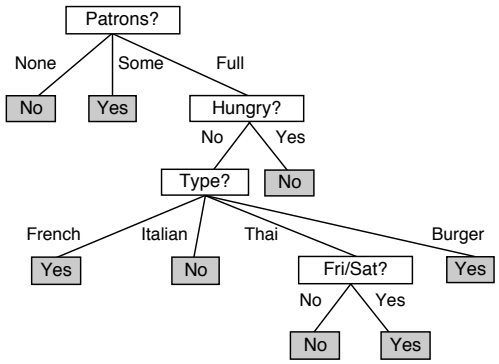
Algorithm...

function DECISION-TREE-LEARNING(*examples*, *attributes*, *default*)
returns a decision tree
  inputs: *examples*, set of examples
    attributes, set of attributes
    default, default value for the goal predicate

  if *examples* is empty **then return** *default*
  else if all *examples* have the same classification
    then return the classification
  else if *attributes* is empty
    then return MAJORITY-VALUE(*examples*)
  else
    best ← CHOOSE-ATTRIBUTE(*attributes*, *examples*)
    tree ← a new decision tree with root test *best*
    for each value v_i of *best* **do**
      $examples_i$ ← {elements of *examples* with *best* = v_i}
      subtree ← DECISION-TREE-LEARNING($examples_i$,
        attributes − *best*, MAJORITY-VALUE(*examples*))
      add a branch to *tree* with label v_i and subtree *subtree*
    end
    return *tree*

4.2 Decision Tree Induction

Applied to our training set gives:



Note: different to original tree, despite using data generated from that tree. Is the tree (hypothesis) wrong?

- *No — with respect to seen examples!*
  In fact more concise, and highlights new patterns.
- *Probably — w.r.t. unseen examples...*

4.3 Assessing Performance

good hypothesis = predicts the classifications of unseen examples

To assess performance we require further examples with known outcomes — *test set*

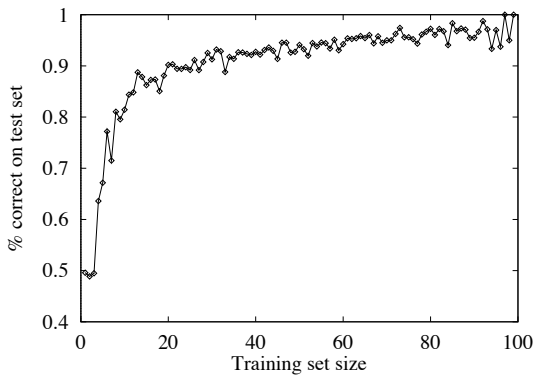
Usual methodology:

1. Collect large set of examples.
2. Divide into two disjoint sets: *training set* and *test set*.
3. Apply learning algorithm to training set to generate hypothesis H.
4. Measure performance of H (% correct) on test set.
5. Repeat for different training sets of different sizes.

Plot prediction quality against training set size...

4.3 Assessing Performance

Learning curve (or "Happy Graph")



4.4 Practical Uses of Decision-Tree Learning

Despite representational limitations, decision-tree learning has been used successfully in a wide variety of applications. eg. . .

Designing Oil Platform Equipment

GASOIL — BP, deployed 1986 (Michie)

— designs complex gas-oil separation systems for offshore oil platforms

— attributes include relative proportions of gas, oil and water, flow rate, pressure, density, viscosity, temperature and susceptibility to waxing

— largest commercial expert system in the world at that time

— approx. 2500 rules

— building by hand would have taken ~ 10 person-years

— descision-tree learning applied to database of existing designs

$\Rightarrow$ developed in 100 person-days

— outperformed human experts

— said to have saved BP millions of dollars!

4.4 Practical Uses of Decision-Tree Learning

Learning to Fly

C4.5 — 1992 (Sammut *et al*)

— one approach — learn correct mapping from state to action

— Cessna on flight simulator

— training: 3 skilled human pilots, assigned flight plan, 30 times each

— training example for each action taken by pilot $\Rightarrow$ 90000 examples

— 20 state variables

— decision tree generated and fed back into simulator to fly plane

— Results: flies *better* than its teachers!

$\Rightarrow$ generalisation process "cleans out" mistakes by teachers

Artificial Intelligence

Topic 7

Sequential Decision Problems

- ◇ Introduction to sequential decision problems
- ◇ Value iteration
- ◇ Policy iteration
- ◇ Longevity in agents

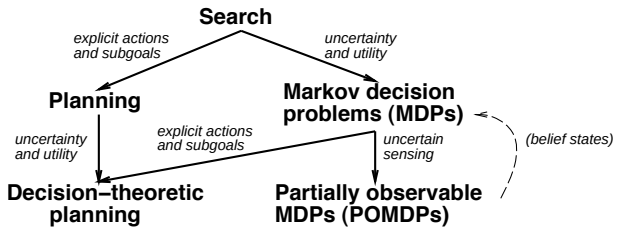
Reading: Russell and Norvig, Chapter 17, Sections 1–3

1. Sequential decision problems

Previously concerned with single decisions, where utility of each action's outcome is known.

This section — *sequential decision problems*
— utility depends on a sequence of decisions

Sequential decision problems which include utilities, uncertainty, and sensing, generalise search and planning problems. . .



1.1 From search algorithms to policies

Sequential decision problems in known, accessible, deterministic domains

tools — search algorithms
outcome — sequence of actions that leads to good state

Sequential decision problems in *uncertain* domains

tools — techniques originating from control theory, operations research, and decision analysis
outcome — *policy*

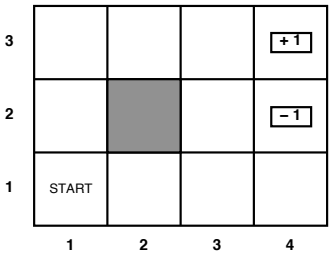
policy = set of state-action "rules"

- tells agent what is the best (MEU) action to try in any situation
- derived from utilities of states

This section is about finding optimal policies.

1.2 From search algorithms to policies – example

Consider the environment:



Problem

Utilities only known for terminal states

⇒ even for deterministic actions, depth-limited search fails!

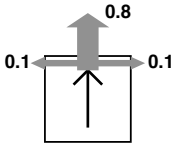
Utilities for other states will depend on sequence (or *environment history*) that leads to terminal state.

1.2 From search algorithms to policies – example

Indeterminism

deterministic version — each action (N,S,E,W) moves one square in intended direction (bumping into wall results in no change)

stochastic version
— actions are unreliable. . .



transition model — probabilities of actions leading to transitions between states

$M^a_{ij} \equiv P(j|i,a)$ = probability that doing a in i leads to j

Cannot be certain which state an action leads to ($cf.$ game playing).

⇒ generating sequence of actions in advance then executing unlikely to succeed

1.2 From search algorithms to policies – example

Policies

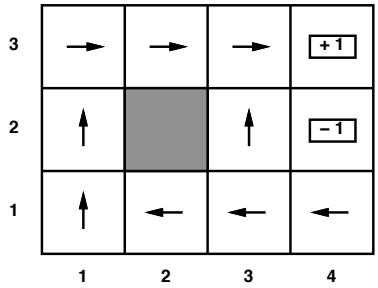
But, if

- we know what state we've reached (accessible)
- we can calculate best action for each state

$\Rightarrow$ *always know what to do next!*

Mapping from states to actions is called a *policy*

eg. Optimal policy for step costs of 0.04...



Note: small step cost $\Rightarrow$ conservative policy (eg. state (3,1))

1.2 From search algorithms to policies – example

Expected Utilities

Given a policy, can calculate expected utilities...

3	0.812	0.868	0.912	+1
2	0.762		0.660	−1
1	0.705	0.655	0.611	0.388
	1	2	3	4

Aim is therefore not to find action sequence, but to find optimal policy — ie. policy that maximises expected utilities.

1.2 From search algorithms to policies – example

Policy represents agent function explicitly

utility-based agent $\mapsto$ *simple reflex agent!*

function SIMPLE-POLICY-AGENT(*percept*) **returns** an *action*
  static: *M*, a transition model
    U, a utility function on environment histories
    P, a policy, initially unknown
  if *P* is unknown **then** *P* $\leftarrow$ the optimal policy given *U*, *M*
  return *P*[*percept*]

Problem of calculating an optimal policy in an accessible, stochastic environment with a known transition model is called a *Markov decision problem*.

Markov property — transition probabilities from a given state depend only on the state (not previous history)

How can we calculate optimal policies...?

2. Value Iteration

Basic idea:

- calculate utility of each state $U(state)$
- use state utilities to select optimal action

Sequential problems usually use an *additive* utility function ($cf.$ path cost in search problems):

$$\begin{aligned} U([s_1, s_2, \ldots, s_n]) &= R(s_1) + R(s_2) + \cdots + R(s_n) \\ &= R(s_1) + U([s_2, \ldots, s_n]) \end{aligned}$$

where $R(i)$ is *reward* in state i (eg. +1, -1, -0.04).

Utility of a *state* (a.k.a. its *value*):

$U(s_i) =$ expected sum of rewards until termination assuming optimal actions

Difficult to express mathematically. Easier is recursive form...

expected sum of rewards = current reward
  + expected sum of rewards after taking best action

2.1 Dynamic programming

Bellman equation (1957)

$$U(i) = R(i) + \max_a \sum_j M_{ij}^a U(j)$$

eg.

$$\begin{aligned} U(1,1) = &-0.04 \\ &+ \max\{0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), && \textit{up} \\ &\quad 0.9U(1,1) + 0.1U(1,2) && \textit{left} \\ &\quad 0.9U(1,1) + 0.1U(2,1) && \textit{down} \\ &\quad 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1)\} && \textit{right} \end{aligned}$$

One equation per state $= n$ nonlinear equations in n unknowns

Given utilities of the states, choosing best action is just maximum expected utility (MEU) — choose action such that the expected utility of the immediate successors is highest.

$$policy(i) = \arg\max_a \sum_j M_{ij}^a U(j)$$

Proven optimal (Bellman & Dreyfus, 1962).

How can we solve

$$U(i) = R(i) + \max_a \sum_j M_{ij}^a U(j)$$

2.2 Value iteration algorithm

Idea

- start with arbitrary utility values
- update to make them locally consistent with Bellman eqn.
- repeat until "no change"

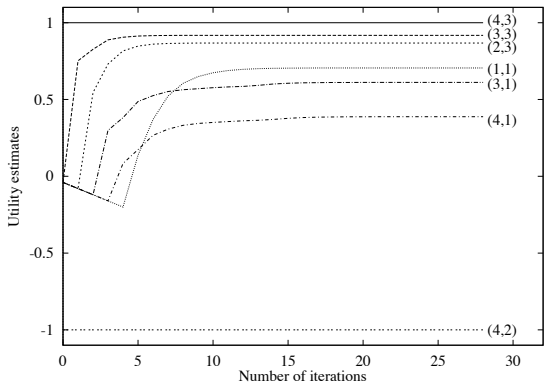
Everywhere locally consistent ⇒ global optimality

function VALUE-ITERATION(M, R) **returns** a utility function
  inputs: M, a transition model
    R, a reward function on states
  local variables: U, utility function, initially identical to R
    U', utility function, initially identical to R

  repeat
    $U \leftarrow U'$
    for each state i **do**
      $U'[i] \leftarrow R[i] + \max_a \Sigma_j M_{ij}^a U[j]$
    end
  until CLOSE-ENOUGH(U, U')
  return U

Applying to our example· · · ⇝

2.2 Value iteration algorithm



3	0.812	0.868	0.912	+1
2	0.762		0.660	−1
1	0.705	0.655	0.611	0.388
	1	2	3	4

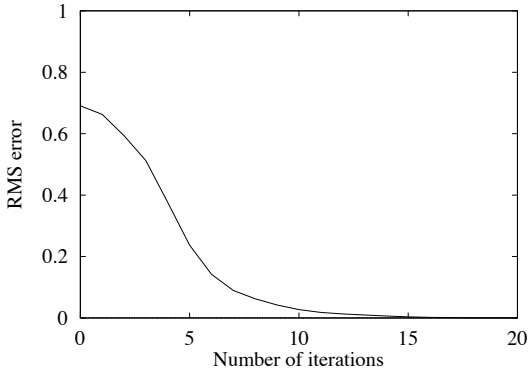
2.3 Assessing performance

Under certain conditions utility values are guaranteed to converge.

Do we require convergence?

Two measures of progress:

1. RMS (root mean square) Error of Utitily Values

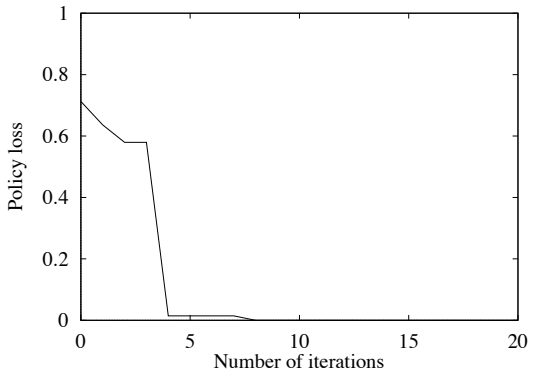


2.3 Assessing performance

2. Policy Loss

Actual utility values less important than the policy they imply

$\Rightarrow$ measure difference between expected utility obtained from policy and expected utility from optimal policy



Note: policy is optimal before RMS error converges.

3. Policy iteration

- policies may not be highly sensitive to *exact* utility values
  $\Rightarrow$ may be less work to iterate through policies than utilities!

Policy Iteration Algorithm

$\pi \leftarrow$ an arbitrary initial policy
repeat until no change in π
  compute utilities given π (*value determination*)
  update π as if utilities were correct (i.e., local MEU)

function POLICY-ITERATION(M, R) **returns** a policy
  inputs: M, a transition model
    R, a reward function on states
  local variables: U, a utility function, initially identical to R
    P, a policy, initially optimal with respect to U
  repeat
    $U \leftarrow$ VALUE-DETERMINATION(P, U, M, R)
    unchanged? $\leftarrow$ true
    for each state i **do**
      if $\max_a \sum_j M^a_{ij}\, U[j] > \sum_j M^{P[i]}_{ij}\, U[j]$ **then**
        $P[i] \leftarrow \arg\max_a \sum_j M^a_{ij}\, U[j]$
        unchanged? $\leftarrow$ false
  until *unchanged?*
  return P

3.1 Value determination

$\Rightarrow$ simpler than value iteration since action is fixed

Two possibilities:

1. Simplification of value iteration algorithm.

$$U'(i) \leftarrow R(i) + \sum_j M^{\pi(i)}_{ij} U(j)$$

May take a long time to converge.

2. Direct solution.

$$U(i) = R(i) + \sum_j M^{\pi(i)}_{ij} U(j) \qquad \text{for all } i$$

i.e., n simultaneous *linear* equations in n unknowns, solve in $O(n^3)$ (eg. Gaussian elimination)

Can be most efficient method for small state spaces.

4. What if I live forever?

Agent continues to exist — using the additive definition of utilities

- $U(i)$s are infinite!
- value iteration fails to terminate

How should we compare two infinite lifetimes?

How can we decide what to do?

One method: *discounting*

Future rewards are discounted at rate $\gamma \leq 1$

$$U([s_0, \ldots s_\infty]) = \Sigma_{t=0}^{\infty} \gamma^t R(s_t)$$

Intuitive justification:

1. purely pragmatic
   - smoothed version of limited horizons in game playing
   - smaller γ, shorter horizon
2. model of animal and human preference behaviour
   - *a bird in the hand is worth two in the bush!*
   - eg. widely used in economics to value investments

Artificial Intelligence

Topic 8

Reinforcement Learning

- ◇ passive learning in a known environment
- ◇ passive learning in unknown environments
- ◇ active learning
- ◇ exploration
- ◇ learning action-value functions
- ◇ generalisation

Reading: Russell & Norvig, Chapter 20, Sections 1–7.

1. Reinforcement Learning

Previous learning examples

- supervised — input/output pairs provided
  eg. chess — given game situation and best move

Learning can occur in much less generous environments

- no examples provided
- no model of environment
- no utility function
  eg. chess — try random moves, gradually build model of environment and opponent

Must have *some* (absolute) feedback in order to make decision.

eg. chess — comes at end of game

⇒ called *reward* or *reinforcement*

Reinforcement learning — use rewards to learn a successful agent function

1. Reinforcement Learning

Harder than supervised learning

eg. reward at end of game — which moves were the good ones?

. . . but . . .

only way to achieve very good performance in many complex domains!

Aspects of reinforcement learning:

- *accessible* environment — states identifiable from percepts
  inaccessible environment — must maintain internal state
- model of environment known or learned (in addition to utilities)
- rewards only in terminal states, or in any states
- rewards components of utility — eg. dollars for betting agent or hints — eg. "nice move"
- *passive learner* — watches world go by
  active learner — act using information learned so far, use problem generator to explore environment

1. Reinforcement Learning

Two types of reinforcement learning agents:

utility learning

- agent learns utility function
- selects actions that maximise expected utitility

Disadvantage: must have (or learn) model of environment — need to know where actions lead in order to evaluate actions and make decision

Advantage: uses "deeper" knowledge about domain

Q-learning

- agent learns *action-value* function
  — expected utility of taking action in given state

Advantage: no model required

Disadvantage: shallow knowledge

— cannot look ahead

— can restrict ability to learn

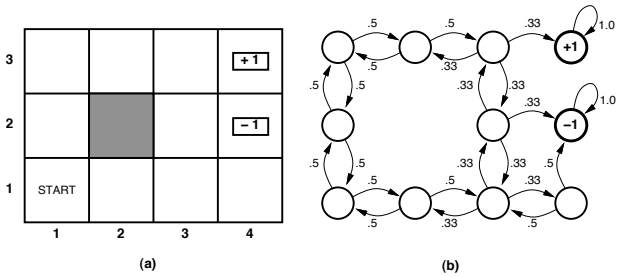
We start with utility learning. . .

2. Passive Learning in a Known Environment

Assume:

- accessible environment
- effects of actions known
- actions are selected *for* the agent $\Rightarrow$ *passive*
- known model M_{ij} giving probability of transition from state i to state j

Example:



(a) environment with utilities (rewards) of terminal states
(b) transition model M_{ij}

Aim: *learn utility values for non-terminal states*

2. Passive Learning in a Known Environment

Terminology

Reward-to-go = sum of rewards from state to terminal state

additive utilitly function: utility of sequence is sum of rewards accumulated in sequence

Thus for additive utility function and state s:

expected utility of s = expected reward-to-go of s

Training sequence eg.

(1,1) →(2,1) →(3,1) →(3,2) →(3,1) →(4,1) →(4,2) [-1]
(1,1) →(1,2) →(1,3) →(1,2) →· · · →(3,3) →(4,3) [1]
(1,1) →(2,1) →· · · →(3,2) →(3,3) →(4,3) [1]

Aim: use *samples* from training sequences to *learn* (an approximation to) expected reward for all states.

ie. generate an *hypothesis* for the utility function

Note: similar to sequential decision problem, except rewards initially unknown.

2.1 A generic passive reinforcement learning agent

Learning is iterative — successively update estimates of utilities

function PASSIVE-RL-AGENT(e) **returns** an action
  static: U, a table of utility estimates
    N, a table of frequencies for states
    M, a table of transition probabilities from state to state
    percepts, a percept sequence (initially empty)
  add e to *percepts*
  increment N[STATE[e]]
  $U \leftarrow$ UPDATE(U, e, *percepts*, M, N)
  if TERMINAL?[e] **then** *percepts* $\leftarrow$ the empty sequence
  return the action *Observe*

Update

- after transitions, or
- after complete sequences

update function is one key to reinforcement learning

Some alternatives · · · ⇝

2.2 Naïve Updating — LMS Approach

From Adaptive Control Theory, late 1950s

Assumes:

observed rewards-to-go → actual expected reward-to-go

At end of sequence:

- calculate (observed) reward-to-go for each state
- use observed values to update utility estimates

eg, utility function represented by table of values — maintain running average. . .

function LMS-UPDATE(U, e, *percepts*, M, N) **returns** an updated U
  if TERMINAL?[e] **then** *reward-to-go* $\leftarrow 0$
    for each e_i **in** *percepts* (starting at end) **do**
      reward-to-go $\leftarrow$ *reward-to-go* + REWARD[e_i]
      U[STATE[e_i]] $\leftarrow$ RUNNING-AVERAGE(U[STATE[e_i]], *reward-to-go*, N[STATE[e_i]])
    end

2.2 Naïve Updating — LMS Approach

Exercise

Show that this approach minimises *mean squared error (MSE)* (and hence *root mean squared (RMS)* error) w.r.t. observed data.

That is, the hypothesis values x_h generated by this method minimise

$$\sum_i \frac{(x_i - x_h)^2}{N}$$

where x_i are the sample values.

For this reason this approach is sometimes called the *least mean squares (LMS)* approach.

In general wish to learn utility function (rather than table).

Have examples with:

- input value — state
- output value — observed reward

$\Rightarrow$ *inductive learning problem!*

Can apply any techniques for inductive function learning — linear weighted function, neural net, etc...

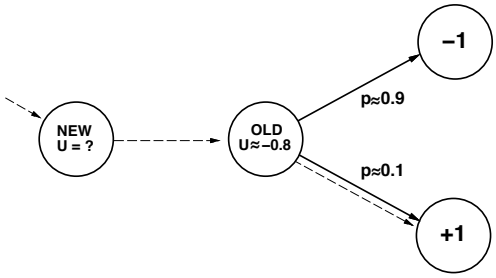
2.2 Naïve Updating — LMS Approach

Problem:

LMS approach ignores important information

$\Rightarrow$ interdependence of state utilities!

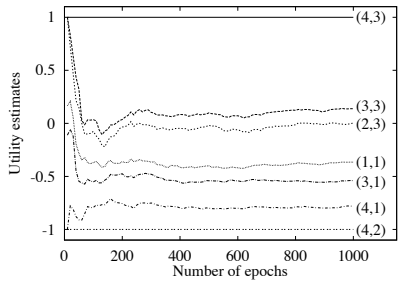
Example (Sutton 1998)

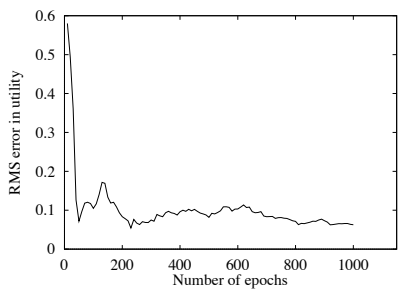


New state awarded estimate of $+1$. Real value ~ -0.8.

2.2 Naïve Updating — LMS Approach

Leads to slow convergence...





2.3 Adaptive Dynamic Programming

Take into account relationship between states...

utility of a state = probability weighted average of its successors' utilities + its own reward

Formally, utilities are described by set of equations:

$$U(i) = R(i) + \sum_j M_{ij} U(j)$$

(passive version of Bellman equation — no maximisation over actions)

Since transition probabilities M_{ij} known, once enough training sequences have been seen so that all reinforcements $R(i)$ have been observed:

- problem becomes well-defined *sequential decision problem*
- equivalent to *value determination* phase of policy iteration

$\Rightarrow$ above equation can be solved exactly

2.3 Adaptive Dynamic Programming

3	−0.0380	0.0886	0.2152	+1
2	−0.1646		−0.4430	−1
1	−0.2911	−0.0380	−0.5443	−0.7722
	1	2	3	4

Refer to learning methods that solve utility equations using dynamic programming as *adaptive dynamic programming (ADP)*.

Good benchmark, but intractable for large state spaces

eg. backgammon: 10^{50} equations in 10^{50} unknowns

2.4 Temporal Difference Learning

Can we get the best of both worlds — use contraints without solving equations for all states?

$\Rightarrow$ use observed transitions to adjust locally in line with constraints

$$U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i))$$

α is *learning rate*

Called *temporal difference (TD) equation* — updates according to *difference* in utilities between *successive* states.

Note: compared with

$$U(i) = R(i) + \sum_j M_{ij} U(j)$$

— only involves observed successor rather than all successors

However, average value of $U(i)$ converges to correct value.

Step further — replace α with function that decreases with number of observations

$\Rightarrow$ $U(i)$ *converges to correct value* (Dayan, 1992).

Algorithm $\cdots \rightsquigarrow$

2.4 Temporal Difference Learning

function TD-UPDATE(*U*, *e*, *percepts*, *M*, *N*) **returns** utility table *U*

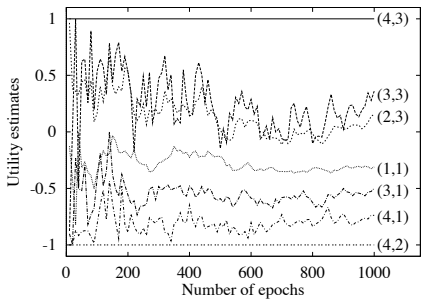
  if TERMINAL?[*e*] **then**
    U[STATE[*e*]] ← RUNNING-AVERAGE(*U*[STATE[*e*]], REWARD[*e*], *N*[STATE[*e*]])
  else if *percepts* contains more than one element **then**
    e′ ← the penultimate element of *percepts*
    i, *j* ← STATE[*e*′], STATE[*e*]
    U[i] ← *U*[i] + α(*N*[i])(REWARD[*e*′] + *U*[j] - *U*[i])

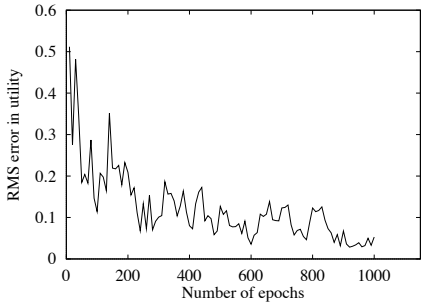
Example runs $\cdots \rightsquigarrow$

Notice:

- values more eratic
- RMS error significantly lower than LMS approach after 1000 epochs

2.4 Temporal Difference Learning





3. Passive Learning, Unknown Environments

- LMS and TD learning don't use model directly
  $\Rightarrow$ operate unchanged in unknown environment
- ADP requires estimate of model
- All utility-based methods use model for action selection

Estimate of model can be updated during learning by observation of transitions

- each percept provides input/output example of transition function

eg. for tabular representation of M, simply keep track of percentage of transitions to each neighbour

Other techniques for learning stochastic functions — not covered here.

4. Active Learning in Unknown Environments

Agent must decide which actions to take.

Changes:

- agent must include *performance element* (and *exploration element*) $\Rightarrow$ choose action
- model must incorporate probabilities *given action* — M_{ij}^a
- constraints on utilities must take account of choice of action

  $$U(i) = R(i) + \max_a \sum_j M_{ij}^a U(j)$$

  (Bellman's equation from sequential decision problems)

Model Learning and ADP

- Tabular representation — accumulate statistics in 3 dimensional table (rather than 2 dimensional)
- Functional representation — input to function includes action taken

ADP can then use value iteration (or policy iteration) algorithms

$\cdots \rightsquigarrow$

4. Active Learning in Unknown Environments

function ACTIVE-ADP-AGENT(e) **returns** an action
  static: U, a table of utility estimates
    M, a table of transition probabilities from state to state for each action
    R, a table of rewards for states
    percepts, a percept sequence (initially empty)
    last-action, the action just executed
  add e to *percepts*
  R[STATE[e]] $\leftarrow$ REWARD[e]
  $M \leftarrow$ UPDATE-ACTIVE-MODEL(M, *percepts*, *last-action*)
  $U \leftarrow$ VALUE-ITERATION(U, M, R)
  if TERMINAL?[e] **then** *percepts* $\leftarrow$ the empty sequence
  last-action $\leftarrow$ PERFORMANCE-ELEMENT(e)
  return *last-action*

Temporal Difference Learning

Learn model as per ADP.

Update algorithm...?

No change! Strange rewards only occur in proportion to probability of strange action outcomes

$$U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i))$$

5. Exploration

How should performance element choose actions?

Two outcomes:

- gain rewards on current sequence
- observe new percepts for learning, and improve rewards on future sequences

trade-off between immediate and long-term good

— not limited to automated agents!

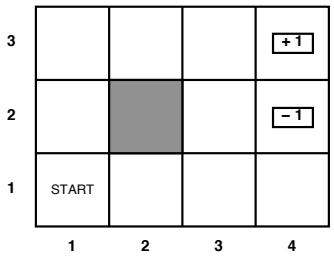
Non trivial

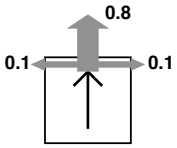
- too conservative $\Rightarrow$ get stuck in a rut
- too inquisitive $\Rightarrow$ inefficient, never get anything done

eg. taxi driver agent

5. Exploration

Example





Two extremes:

whacky — acts randomly in hope of exploring environment
- ⇒ learns good utility estimates
- ⇒ never gets better at reaching positive reward

greedy — acts to maximise utility given current estimates
- ⇒ finds a path to positive reward
- ⇒ never finds optimal route

Start whacky, get greedier?

Is there an optimal exploration policy?

5. Exploration

Optimal is difficult, but can get close...

— give weight to actions that have not been tried often, while tending to avoid low utilities

Alter constraint equation to assign higher utility estimates to relatively unexplored action-state pairs

⇒ optimistic "prior" — initially assume everything is good.

Let

$U^+(i)$ — optimistic estimate
$N(a,i)$ — number of times action a tried in state i

ADP update equation

$$U^+(i) \leftarrow R(i) + \max_a f(\sum_j M^a_{ij} U^+(j), N(a,i))$$

where $f(u,n)$ is *exploration function*.

Note U^+ (not U) on r.h.s. — propagates tendency to explore from sparsely explored regions through densely explored regions

5. Exploration

$f(u,n)$ determines *trade-off* between "greed" and "curiosity"

⇒ should increase with u, decrease with n

Simple example

$$f(u,n) = \begin{cases} R^+ & \text{if } n < N_e \\ u & \text{otherwise} \end{cases}$$

where R^+ is optimistic estimate of best possible reward, N_e is fixed parameter

⇒ try each state at least N_e times.

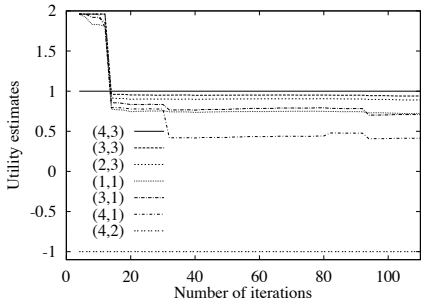
Example for ADP agent with $R^+ = 2$ and $N_e = 5 \cdots \rightsquigarrow$

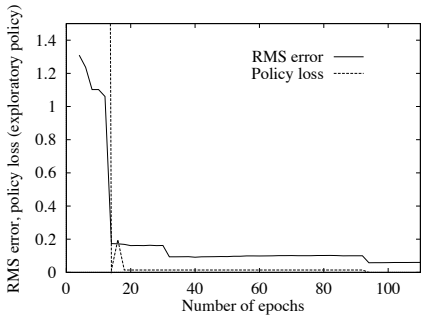
Note policy converges on optimal very quickly

(wacky — best policy loss ≈ 2.3
greedy — best policy loss ≈ 0.25)

Utility estimates take longer — after exploratory period further exploration only by "chance"

5. Exploration





6. Learning Action-Value Functions

Action-value functions

- assign expected utility to taking action a in state i
- also called *Q-values*
- allow decision-making without use of model

Relationship to utility values

$$U(i) = \max_a Q(a, i)$$

Constraint equation

$$Q(a, i) = R(i) + \sum_j M_{ij}^a \max_{a'} Q(a', j)$$

Can be used for iterative learning, but need to learn model.

Alternative $\Rightarrow$ temporal difference learning

TD Q-learning update equation

$$Q(a, i) \leftarrow Q(a, i) + \alpha(R(i) + \max_{a'} Q(a', j) - Q(a, i))$$

6. Learning Action-Value Functions

Algorithm:

function Q-LEARNING-AGENT(e) **returns** an action
  static: Q, a table of action values
    N, a table of state-action frequencies
    a, the last action taken
    i, the previous state visited
    r, the reward received in state i

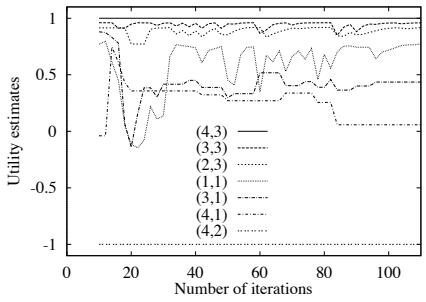
  j $\leftarrow$ STATE[e]
  if i is non-null **then**
    $N[a, i] \leftarrow N[a, i] + 1$
    $Q[a, i] \leftarrow Q[a, i] + \alpha(r + \max_{a'} Q[a', j] - Q[a, i])$
  if TERMINAL?[e] **then**
    $i \leftarrow$ null
  else
    $i \leftarrow j$
    $r \leftarrow$ REWARD[e]
  $a \leftarrow \arg\max_{a'} f(Q[a', j], N[a', j])$
  return a

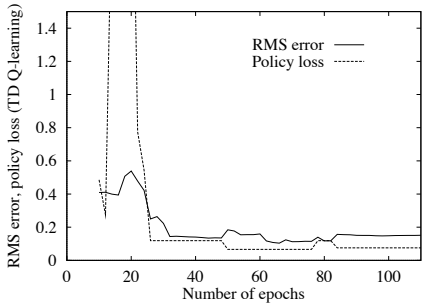
Example $\cdots \rightsquigarrow$

Note: slower convergence, greater policy loss

Consistency between values not enforced by model.

6. Learning Action-Value Functions





7. Generalisation

So far, algorithms have represented hypothesis functions as tables — *explicit representation*

eg. state/utility pairs

OK for small problems, impractical for most real-world problems.

eg. chess and backgammon $\rightarrow 10^{50}$ – 10^{120} states.

Problem is not just storage — *do we have to visit all states to learn?*

Clearly humans don't!

Require *implicit representation* — compact representation, rather than storing value, allows value to be calculated

eg. weighted linear function of features

$$U(i) = w_1 f_1(i) + w_2 f_2(i) + \cdots + w_n f_n(i)$$

From say 10^{120} states to 10 weights $\Rightarrow$ *whopping compression!*

But more importantly, returns estimates for unseen states

$\Rightarrow$ *generalisation!!*

7. Generalisation

Very powerful. eg. from examining 1 in 10^{44} backgammon states, can learn a utility function that can play as well as any human.

On the other hand, may fail completely...

hypothesis space must contain a function close enough to actual utility function

Depends on

- type of function used for hypothesis
  eg. linear, nonlinear (neural net), etc
- chosen features

Trade off:

larger the hypothesis space

$\Rightarrow$ *better likelihood it includes suitable function*, but

$\Rightarrow$ *more examples needed*

$\Rightarrow$ *slower convergence*

7. Generalisation

And last but not least...