cs/~shavlik/cs540.html #homeworks

L-45

CS 540 - Introduction to Artificial Intelligence (Fall 2000)

This page contains information specific to Prof. Shavlik's section of CS 540 (Fall 2000).
Click here for general information about CS 540.

Instructor: Jude Shavlik

```
Office:        6357 CS & Stats Building
Email:         shavlik@cs.wisc.edu
Office Hours:  Monday 3:15-4:00pm, Friday 2:45-3:45pm, and by appointment (send e
```

Teaching Assistant: Mario Pi

```
Office:        1331 CS & Stats Building
Email:         mariopi@cs.wisc.edu
Office Hours:  Tuesday 11-12am and Thursday 2:15-3:15pm, and by appointment (send
```

Additional Information

- Prerequisite: CS 367 or consent of instructor
- Meeting schedule: MWF 11-11:50am, 2535 Engineering Hall
- Discussion Section: Mon 2:25-3:15pm, 1325 CS & Stats Building
- Textbook: *Artificial Intelligence: A Modern Approach*
  by Russell and Norvig, Prentice Hall, Englewood Cliffs, N.J., 1995 (corrections to the text).
- Class email alias: `cs540-1list` (archive of what was sent out).

Table of Contents

Course Overview and Requirements

This course provides an introduction to artificial intelligence. Topics covered include knowledge representation, heuristic search, game playing, deductive reasoning, reasoning under uncertainty, planning, learning, philosophical foundations, and (time permitting) language understanding and computer vision.

The work in the course will consist of 5-6 homework assignments (about one every two weeks), a midterm exam, and a final exam. Your programms will be partially automatically graded, so *they must be written to run on the instructional Unix machines (the Nova's)*. Two or three of the homework assignments will involve programming tasks that are to be done in Java. (A page of useful Java-related links and tips is available.)

Homeworks will count for 40% of the grade, the 'midterm' exam for 25%, and the final for 35%. *Quality* class participation will have an impact on borderline cases. The course will be graded on the conventional (A-F) system.

Homeworks

- HW5 - Miscellaneous Topics
  Due 5pm 12/11/00 (no accepted after 5pm 12/13/00 in order to allow enough time for grading and so that a solution can be posted a few days before the exam, which is 12/18/00)

  Solution: MS Word | PDF | Grading Sheet

- HW4 - Natural Language Processing, Case-Based Reasoning, and Bayesian Reasoning
  Due 5pm 11/22/00 (not accepted after 5pm 11/29/00)

  Solution (for Problems 1-4): MS Word | PDF | Grading Sheet

- HW3 - Representing and Reasoning about Knowledge using Logic
  Due 5pm 11/3/00 (not accepted after 5pm 11/10/00)

  Solution: MS Word | PDF | Grading Sheet

- HW 2 - Searching for Solutions
  Due 5pm 10/16/00 (not accepted after 5pm 10/23/00)

  FAQ

  Solution: MS Word | PDF | Grading Sheet

- HW 1 - Learning Decision Trees from Training Examples
  Due 5pm 10/2/00 (not accepted after 5pm 10/9/00)

  FAQ

lution (for Problem 1): PDF | Grading Sheet

Academic Misconduct

All examinations, programming assignments, and written homeworks must be done *individually*. Cheating and plagiarism will be dealt with in accordance with University procedures (see the Academic Misconduct Guide for Students). Hence, for example, code for programming assignments must not be developed in groups, nor should code be shared. You are encouraged to discuss with your peers, the TAs or the instructor ideas, approaches and techniques broadly, but not at a level of detail where specific implementation issues are described by anyone. If you have any questions on this, please ask the instructor before you act.

- **Late policy on HWs**
  - HWs are due at *5 pm*. Turn in HWs during class or in Shavlik's mailbox (5th floor of CS building).
  - Each student will have FIVE "free" late days for use over the semester, for delays due to illness, especially hectic weeks, interview trips, etc. Once these are exhausted, there will be a penalty of 10 points per day (measured 5pm-to-5pm; weekends and official university holidays are free).
  - To make the TA's job tractable, no HWs will be accepted more than *one week* late.

- Here is a good command to use for printing out files of code (do a `man print` for more information):

```
print -landscape -Plaser HW1.java
```

Reading Assignments

Assigned December 11, 2000:

Page 463 of the *first* edition of the textbook (this material also appears, unchanged, in the NEW Chapter 16) and Chapters 26 & 27 of Russell & Norvig.

Assigned December 4, 2000:

Sections 20.1, 20.5, 20.6, 20.8, 20.9 (and skim rest of chapter) of Russell & Norvig.

Assigned November 10, 2000:

Prof. Craven's lecture notes on Information Retrieval (in Powerpoint). He will be giving a guest lecture on Friday, Nov. 17.

Sections 10.1 and 10.5 plus Chapter 19 (except Section 19.6) of Russell & Norvig.

Assigned November 3, 2000:

Case-based reasoning on-line tutorial notes (in Powerpoint) *or* an alternate tutorial (in HTML)

Assigned November 1, 2000:

The *new* Chapter 15 and *new* Sections 16.1 and 16.2 plus Section 16.4 to the middle of page 363 of the forthcoming second edition of Russell & Norvig

Assigned October 23, 2000:

Chapter 9 (except pages 281-289) and pages 302-303 (Unification algorithm), Norvig

Assigned October 13, 2000:
Chapter 7 and Chapter 8 through page 228 plus Section 8.6 of Russell & Norvig

Assigned October 6, 2000:
Chapters 5 & 6 of Russell & Norvig

Assigned September 20, 2000:
Chapters 3 & 4 of Russell & Norvig (Skim Sections 3.7 and 4.3)

Assigned September 6, 2000:
Chapters 1 & 2 and Sections 18.1-18.4 of Russell & Norvig

Exam Schedule

- Midterm: October 25 (7:15-9:15pm, Room 1240 CS&Stats)
  ONE (8.5x11) page of notes and a calculator allowed.
- Final (cumulative, though with emphasis on material covered since midterm): Monday, December 18 (2:45-4:45pm, Room 1221 CS&Stats)
  TWO (8.5x11) pages of notes and a calculator allowed.

Previous Exams (postscript unless otherwise noted)

- Midterm (2000, MS Word) | Midterm (2000, pdf) | Final (2000, MS Word)
  Histogram of midterm grades
- Midterm (1999) | Final (1999, pdf) | Final (1999, MS Word)
- Exam 1 (1998) | Exam 2 (1998) | Final (1998)
- Exam 1 (1995) | Exam 2 (1995) | Final (1995)
- Exam 1 (1994) | Exam 2 (1994) | Final (1994)
- Exam 1 (1992) | Exam 2 (1992) | Final (1992)

Some General AI Articles and Sites

- Our collection of on-line AI demos
- *AI Magazine*
- History and Promise of AI (by D. Waltz)
- Report on 21st Century Intelligent Systems (by B. Grosz and R. Davis)
- The Role of Intelligent Systems in the National Information Infrastructure (by D. Weld)
- Additional External AI Resources

This page was created by shavlik@cs.wisc.edu

Computer Sciences Department
University of Wisconsin - Madison

INFORMATION -PEOPLE -GRADS -UNDERGRADS -RESEARCH -RESOURCES

5355a Computer Sciences and Statistics ~ 1210 West Dayton Street, Madison, WI 53706
cs@cs.wisc.edu ~ voice: 608-262-1204 ~ fax: 608-262-9777

CS 540 Lecture Notes

- Introduction (January 22-24)
- Intelligent Agents (January 26)
- Machine Learning (January 29 - February 5)
- Uninformed Search (February 5-9)
- Informed Search (February 12-14)
- Game Playing (February 16-21)
- Logic (February 23-26)
- First-Order Logic (February 26 - March 7)
- Logical Reasoning Systems (March 7-21)
- Neural Networks (March 23 - April 4)
- Genetic Algorithms (April 6)
- Reasoning under Uncertainty (April 9-20)
- Computer Vision (April 23-27)
- Lecture notes on face recognition (2-up, 3MB pdf)
- Face recognition using eigenfaces, M. Turk and A. Pentland, *Proc. Computer Vision and Pattern Recognition Conference*, 1991, pp. 586-591.
- Voxel Coloring slides (4-up, 2.2MB pdf)
- Planning (April 30 - May 2)
- Partial-Order Planning (May 4-7)
- Speech Recognition (May 9)
- Tutorial on Speech Recognition and Hidden Markov Models by E. Fosler-Lussier (optional)

540 Lecture Notes Spring 2001

Introduction (Chapter 1)

What is AI?

- One answer: A field that focuses on developing techniques to enable computer systems to perform activities that are considered intelligent (in humans and other animals).
- Another answer: "It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.

→ "Intelligence is the computational part of the ability to achieve goals in the world." -- J. McCarthy

See also more of John McCarthy's views on this.

Goals of AI

→ • Replicate human intelligence

"AI is the study of complex information processing problems that often have their roots in some aspect of biological information processing. The goal of the subject is to identify solvable and interesting information processing problems, and solve them." -- David Marr

→ • Solve knowledge-intensive tasks

"AI is the design, study and construction of computer programs that behave intelligently." -- Tom Dean

→ • Intelligent connection of perception and action

AI not centered around representation of the world, but around action in the world. Behavior-based intelligence. (see Rod Brooks in the movie *Fast, Cheap and Out of Control*)

→ • Enhance human-human, human-computer and computer-computer interaction/communication

Computer can sense and recognize its users, see and recognize its environment, respond visually and audibly to stimuli. New paradigms for interacting productively with computers using speech, vision, natural language, 3D virtual reality, 3D displays, more natural and powerful user interfaces, etc. (See, for example, projects in Microsoft's "Advanced Interactivity and Intelligence" group.)

Some Application Areas of AI

- **Game Playing**
  Deep Blue Chess program beat world champion Gary Kasparov

9

- **Speech Recognition**
  PEGASUS spoken language interface to American Airlines' EAASY SABRE reseration system, which allows users to obtain flight information and make reservations over the telephone. The 1990s has seen significant advances in speech recognition so that limited systems are now successful.
- **Computer Vision**
  Face recognition programs in use by banks, government, etc. The ALVINN system from CMU autonomously drove a van from Washington, D.C. to San Diego (all but 52 of 2,849 miles), averaging 63 mph day and night, and in all weather conditions. Handwriting recognition, electronics and manufacturing inspection, photointerpretation, baggage inspection, reverse engineering to automatically construct a 3D geometric model.
- **Expert Systems**
  Application-specific systems that rely on obtaining the knowledge of human experts in an area and programming that knowledge into a system.
  - **Diagnostic Systems**
    Microsoft Office Assistant in Office 97 provides customized help by decision-theoretic reasoning about an individual user. MYCIN system for diagnosing bacterial infections of the blood and suggesting treatments. Intellipath pathology diagnosis system (AMA approved). Pathfinder medical diagnosis system, which suggests tests and makes diagnoses. Whirlpool customer assistance center.
  - **System Configuration**
    DEC's XCON system for custom hardware configuration. Radiotherapy treatment planning.
  - **Financial Decision Making**
    Credit card companies, mortgage companies, banks, and the U.S. government employ AI systems to detect fraud and expedite financial transactions. For example, AMEX credit check. Systems often use learning algorithms to construct profiles of customer usage patterns, and then use these profiles to detect unusual patterns and take appropriate action.
  - **Classification Systems**
    Put information into one of a fixed set of categories using several sources of information. E.g., financial decision making systems. NASA developed a system for classifying very faint areas in astronomical images into either stars or galaxies with very high accuracy by learning from human experts' classifications.
- **Mathematical Theorem Proving**
  Use inference methods to prove new theorems.
- **Natural Language Understanding**
  AltaVista's translation of web pages. Translation of Catepillar Truck manuals into 20 languages. (Note: One early system translated the English sentence "The spirit is willing but the flesh is weak" into the Russian equivalent of "The vodka is good but the meat is rotten.")
- **Scheduling and Planning**
  Automatic scheduling for manufacturing. DARPA's DART system used in Desert Storm and Desert Shield operations to plan logistics of people and supplies. American Airlines rerouting contingency planner. European space agency planning and scheduling of spacecraft assembly, integration and verification.

Some AI "Grand Challenge" Problems

- Translating telephone
- Accident-avoiding car
- Aids for the disabled
- Smart clothes
- Intelligent agents that monitor and manage information by filtering, digesting, abstracting

- Tutors
- Self-organizing systems, e.g., that learn to assemble something by observing a human do it.

A Framework for Building AI Systems

- **Perception**
  Intelligent biological systems are physically embodied in the world and experience the world through their sensors (senses). For an autonomous vehicle, input might be images from a camera and range information from a rangefinder. For a medical diagnosis system, perception is the set of symptoms and test results that have been obtained and input to the system manually. Includes areas of vision, speech processing, natural language processing, and signal processing (e.g., market data and acoustic data).
- **Reasoning**
  Inference, decision-making, classification from what is sensed and what the internal "model" is of the world. Might be a neural network, logical deduction system, Hidden Markov Model induction, heuristic searching a problem space, Bayes Network inference, genetic algorithms, etc. Includes areas of knowledge representation, problem solving, decision theory, planning, game theory, machine learning, uncertainty reasoning, etc.
- **Action**
  Biological systems interact within their environment by actuation, speech, etc. All behavior is centered around actions in the world. Examples include controlling the steering of a Mars rover or autonomous vehicle, or suggesting tests and making diagnoses for a medical diagnosis system. Includes areas of robot actuation, natural language generation, and speech synthesis.

Some Fundamental Issues for Most AI Problems

- **Representation**
  Facts about the world have to be represented in some way, e.g., mathematical logic is one language that is used in AI. Deals with the questions of *what* to represent and *how* to represent it. How to structure knowledge? What is explicit, and what must be inferred? How to encode "rules" for inferencing so as to find information that is only implicitly known? How to deal with incomplete, inconsistent, and probabilistic knowledge? Epistemology issues (what kinds of knowledge are required to solve problems).

  Example: "The fly buzzed irritatingly on the window pane. Jill picked up the newspaper." Inference: Jill has malicious intent; she is not intending to read the newspaper, or use it to start a fire, or ...

  Example: Given 17 sticks in 3 x 2 grid, remove 5 sticks to leave exactly 3 squares.
- **Search**
  Many tasks can be viewed as searching a very large problem space for a solution. For example, Checkers has about 10^{40} states, and Chess has about 10^{120} states in a typical games. Use of heuristics (meaning "serving to aid discovery") and constraints.
- **Inference**
  From some facts others can be inferred. Related to search. For example, knowing "All elephants have trunks" and "Clyde is an elephant," can we answer the question "Does Clyde hae a trunk?" What about "Peanuts has a trunk, is it an elephant?" Or "Peanuts lives in a tree and has a trunk, is it an elephant?" Deduction, abduction, non-monotonic reasoning, reasoning under uncertainty.
- **Learning**

Inductive inference, neural networks, genetic algorithms, artificial life, evolutionary approaches.

- **Planning**
  Starting with general facts about the world, facts about the effects of basic actions, facts about a particular situation, and a statement of a goal, generate a strategy for achieving that goals in terms of a sequence of primitive steps or actions.

Design Methodology and Goals

- **Engineering Goal**: Develop concepts, theory and practice of building intelligent machines. Emphasis on system building.
- **Science Goal**: Develop concepts, mechanisms and vocabulary to understand biological intelligent behavior. Emphasis on understanding intelligent behavior.

Alternatively, methodologies can be defined by choosing (1) the goals of the computational model, and (2) the basis for evaluating performance of the system:

1 Think like humans "cognitive science" Ex. GPS	**2** Think rationally => formalize inference process "laws of thought"
3 Act like humans Ex. ELIZA Turing Test	**4** Act rationally "satisficing" methods

- **Box 1**
  "Cognitive science" approach - Focus not just on behavior and I/O, look at reasoning process. Computational model should reflect "how" results were obtained. GPS (General Problem Solver): Goal not just to produce humanlike behavior (like ELIZA), but to produce a sequence of steps of the reasoning process that was similar to the steps followed by a person in solving the same task.

- **Box 2**
  The goal is to formalize the reasoning process, producing a system that contains logical inference mechanisms that are provably correct, and guarantee finding an optimal solution. This brings up the question: How do we represent information that will allow us to do inferences like the following one? "Socrates is a man. All men are mortal. Therefore Socrates is mortal." -- Aristotle

  Problem : Not all problems can be solved just by reasoning and inferences.

- **Box 3**
  Behaviorist approach. Not interested in how you get results, just the similarity to what human results are. Exemplified by the **Turing Test** (Alan Turing, 1950): 3 rooms contain a

person, a computer, and an interrogator. The interrogator can communicate with the other 2 by teletype (to avoid requiring that the machine imitate the appearance or voice of the person). The interrogator tries to determine which is the person and which is the machine. The machine tries to fool the interrogator into believing that it is the human, and the person also tries to convince the interrogator that it is the human. If the machine succeeds in fooling the interrogator, we conclude that the machine is intelligent.

The Turing test is a one-sided test in that a machine that passes the test should certainly be considered intelligent, but a machine could still be considered intelligent without knowing enough about humans to imitate humans.

The Loebner Prize is one version of the Turing Test that is awarded annually to the "most human computer."

ELIZA: A program that simulated a psychotherapist interacting with a patient and successfully passed the Turing Test. Some extracts from the test:

```
Patient:    Men are all alike.
ELIZA:      In what way?
Patient     They're always bugging us about something or other.
ELIZA:      Can you think of a specific example?
Patient:    Well, my boyfriend made me come here.
ELIZA:      Your boyfriend made you come here?
Patient:    He says I'm depressed much of the time.
ELIZA:      I am sorry you are depressed.
...
```

Problems: ELIZA and other similar programs stressed simple syntactic analysis and generation of sentences. They used pattern matching with known sentences. They used pattern matching with known vocabulary and key words with templates of sentences to generate. For example,

if sentence = "* mother *" then respond with "Tell me about your family."

Note that even with simple syntactic style, ELIZA managed to fool people. Purely behavioral-based approach can be simulated without a deeper understanding or true "intelligence."

- **Box 4**
  For a given set of inputs, tries to generate an appropriate output that is not necessarily correct but gets the job done. Rational and sufficient ("satisficing" methods, not "optimal").

Most of AI work falls into Boxes 2 and 4. These don't rely on tests that correspond to human performance.

Symbols versus Signals

Most of AI built on an information processing model called a "Physical-Symbol System" (PSS) (Newell and Simon). Symbols usually correspond to objects in the environment. Symbols are physical patterns that can occur as components of an expression or symbol structure. A PSS is a collection of symbol structures plus processes that operate (i.e., create, modify, reproduce) expressions to produce other expressions. Hence, a PSS produces over time an evolving collection of symbol structures.

=> AI is the enterprise of constructing physical-symbol system that can reliably pass the Turing Test [or whatever your performance test is].

Physical-Symbol System Hypothesis (Newell and Simon, 1976): A physical-symbol system has the necessary and sufficient means for general intelligent action.

==> Intelligence is a functional property and is completely independent of any physical embodiment.

==> Develop structural/functional theory of intelligence, i.e., what are the mechanisms, physical or formal structures, which form the basis of intelligent behavior.

An alternative, less-symbolic paradigm: Neural Networks

---

540 Lecture Notes Spring 2001

Intelligent Agents (Chapter 2)

What is an Intelligent Agent?

- One definition: An (intelligent) **agent** perceives it environment via sensors and acts rationally upon that environment with its effectors. Hence, an agent gets percepts one at a time, and maps this percept sequence to actions.
- Another definition: An agent is a computer software system whose main characteristics are situatedness, autonomy, adaptivity, and sociability.

Agent Characteristics

- Situatedness
  The agent receives some form of sensory input from its environment, and it performs some action that changes its environment in some way. Examples of environments: the physical world and the Internet.
- Autonomy
  The agent can act without direct intervention by humans or other agents and that it has control over its own actions and internal state.
- Adaptivity
  The agent is capable of (1) reacting flexibly to changes in its environment; (2) taking goal-directed initiative (i.e., is pro-active), when appropriate; and (3) learning from its own experience, its environment, and interactions with others.
- Sociability
  The agent is capable of interacting in a peer-to-peer manner with other agents or humans.

Examples of Agents

Agent Type	Percepts	Actions	Goals	Environment
Bin-Picking Robot	Images	Grasp objects; Sort into bins	Parts in correct bins	Conveyor belt
Medical Diagnosis	Patient symptoms, tests	Tests and treatments	Healthy patient	Patient & hospital
Excite's Jango product finder	Web pages	navigate web, gather relevant products	Find best price for a product	Internet
Webcrawler Softbot	Web pages	Follow links, pattern matching	Collect info on a subject	Internet
Financial forecasting software	Financial data	Gather data on companies	Pick stocks to buy & sell	Stock market, company reports

How to Evaluate an Agent's Behavior/Performance?

- Rationality => Need a performance measure to say how well a task has been achieved. An ideal rational agent should, for each possible percept sequence, do whatever actions will

maximize its performance measure based on (1) the percept sequence, and (2) its built-in and acquired knowledge. Hence includes information gathering, not "rational ignorance."

- Types of objective performance measures: false alarm rate, false dismissal rate, time taken, resources required, effect on environment, etc.
- Examples: Benchmarks and test sets, Turing test (there is no homunculus!)

Approaches to Agent Design

1. **Simple Reflex Agent**
   - Table lookup of percept-action pairs defining all possible condition-action rules necessary to interact in an environment
   - Problems
     - Too big to generate and to store (Chess has about 10^{120} states, for example)
     - No knowledge of non-perceptual parts of the current state
     - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
     - Looping: Can't make actions conditional

2. **Reflex Agent with Internal State**
   - Encode "internal state" of the world to remember the past as contained in earlier percepts
   - Needed because sensors do not usually give the entire state of the world at each input, so perception of the environment is captured over time. "State" used to encode different "world states" that generate the same immediate percept.
   - Requires ability to represent change in the world; one possibility is to represent just the latest state, but then can't reason about hypothetical courses of action
   - Example: Rodney Brooks's Subsumption Architecture
     Main idea: build complex, intelligent robots by decomposing behaviors into a hierarchy of skills, each completely defining a complete percept-action cycle for one very specific task. For example, avoiding contact, wandering, exploring, recognizing doorways, etc. Each behavior is modeled by a finite-state machine with a few states (though each state may correspond to a complex function or module). Behaviors are loosely-coupled, asynchronous interactions.

3. **Goal-Based Agent**
   - Choose actions so as to achieve a (given or computed) goal= a description of a desirable situation
   - Keeping track of the current state is often not enough--- need to add goals to decide which situations are good
   - Deliberative instead of reactive
   - May have to consider long sequences of possible actions before deciding if goal is achieved--- involves consideration of the future, "what will happen if I do...?"

4. **Utility-Based Agent**
   - When there are multiple possible alternatives, how to decide which one is best?
   - A goal specifies a crude distinction between a happy and unhappy state, but often need a more general performance measure that describes "degree of happiness"

- Utility function U: State --> Reals
  indicating a measure of success or happiness when at a given state
- Allows decisions comparing choice between conflicting goals, and choice between likelihood of success and importance of goal (if achievement is uncertain)

Last modified September 3, 1998

Machine Learning (Chapter 18.1 - 18.4)

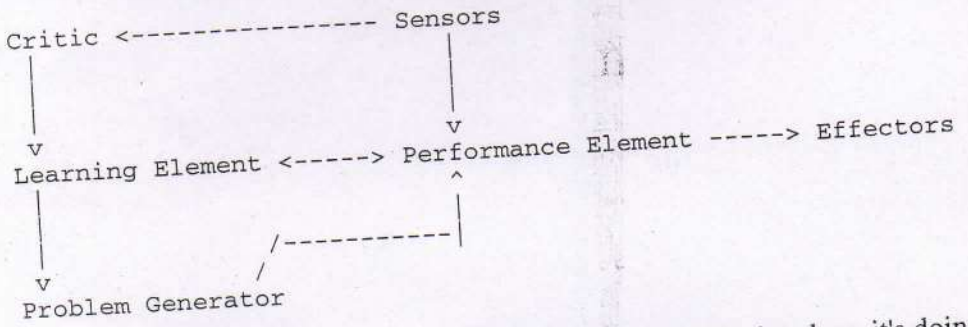
What is Learning?

- "Learning denotes changes in a system that ... enable a system to do the same task more efficiently the next time." --Herbert Simon
- "Learning is constructing or modifying representations of what is being experienced." --Ryszard Michalski
- "Learning is making useful changes in our minds." --Marvin Minsky

Why do Machine Learning?

- Understand and improve efficiency of human learning
  For example, use to improve methods for teaching and tutoring people, as done in CAI -- Computer-aided instruction
- Discover new things or structure that is unknown to humans
  Example: Data mining
- Fill in skeletal or incomplete specifications about a domain
  Large, complex AI systems cannot be completely derived by hand and require dynamic updating to incorporate new information. Learning new characteristics expands the domain or expertise and lessens the "brittleness" of the system

Components of a Learning System



- Learning Element makes changes to the system based on how it's doing
- Performance Element is the agent itself that acts in the world
- Critic tells the Learning Element how it is doing (e.g., success or failure) by comparing with a fixed standard of performance
- Problem Generator suggests "problems" or actions that will generate new examples or experiences that will aid in training the system further

We will concentrate on the Learning Element

Evaluating Performance

Several possible criteria for evaluating a learning algorithm:

- Predictive accuracy of classifier

- Speed of learner
- Speed of classifier
- Space requirements

Most common criterion is **predictive accuracy**

Major Paradigms of Machine Learning

- **Rote Learning**
  One-to-one mapping from inputs to stored representation. "Learning by memorization." Association-based storage and retrieval.
- **Induction**
  Use specific examples to reach general conclusions
- **Clustering**
- **Analogy**
  Determine correspondence between two different representations
- **Discovery**
  Unsupervised, specific goal not given
- **Genetic Algorithms**
- **Reinforcement**
  Only feedback (positive or negative reward) given at end of a sequence of steps. Requires assigning reward to steps by solving the credit assignment problem--which steps should receive credit or blame for a final result?

The Inductive Learning Problem

- Extrapolate from a given set of examples so that we can make accurate predictions about future examples.
- **Supervised versus Unsupervised learning**
  Want to learn an unknown function $f(\mathbf{x}) = \mathbf{y}$, where **x** is an input example and **y** is the desired output. Supervised learning implies we are given a set of (**x**, **y**) pairs by a "teacher." Unsupervised learning means we are only given the **x**s. In either case, the goal is to estimate f.
- **Concept learning**
  Given a set of examples of some concept/class/category, determine if a given example is an instance of the concept or not. If it is an instance, we call it a **positive example**. If it is not, it is called a **negative example**.
- **Problem: Supervised Concept Learning by Induction**
  Given a **training set** of positive and negative examples of a concept, construct a description that will accurately classify whether future examples are positive or negative. That is, learn some good estimate of function f given a training set {**(x1, y1), (x2, y2), ..., (xn, yn)**} where each **yi** is either + (positive) or - (negative).

Inductive Bias

- Inductive learning is an inherently conjectural process because any knowledge created by generalization from specific facts cannot be proven true; it can only be proven false. Hence, inductive inference is **falsity preserving**, not truth preserving.
- To generalize beyond the specific training examples, we need constraints or **biases** on what f is best. That is, learning can be viewed as searching the **Hypothesis Space** H of possible f functions.
- A bias allows us to choose one f over another one

- A completely unbiased inductive algorithm could only memorize the training examples and could not say anything more about other unseen examples.
- Two types of biases are commonly used in machine learning:
  - **Restricted Hypothesis Space Bias**
    Allow only certain types of *f* functions, not arbitrary ones
  - **Preference Bias**
    Define a metric for comparing *f*s so as to determine whether one is better than another

Inductive Learning Framework

- Raw input data from sensors are preprocessed to obtain a **feature vector**, **x**, that adequately describes all of the relevant features for classifying examples.
- Each **x** is a list of (attribute, value) pairs. For example,

  x = `(Person = Sue, Eye-Color = Brown, Age = Young, Sex = Female)`

  The number of attributes (also called features) is fixed (positive, finite). Each attribute has a fixed, finite number of possible values.

- Each example can be interpreted as a *point* in an *n*-dimensional **feature space**, where *n* is the number of attributes.

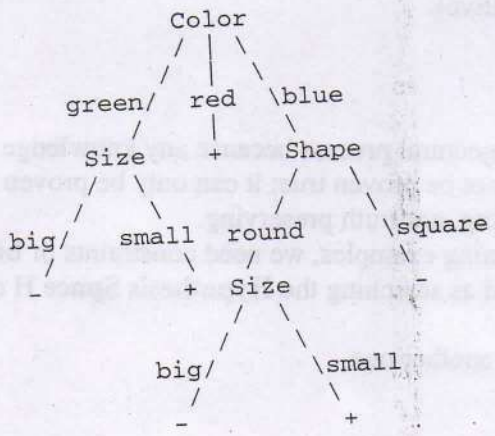
Inductive Learning by Nearest-Neighbor Classification

One simple approach to inductive learning is to save each training example as a point in Feature Space, and then classify a new example by giving it the same classification (+ or -) as its **nearest neighbor in Feature Space**.

The problem with this approach is that it doesn't necessarily generalize well if the examples are not "clustered."

Inductive Concept Learning by Learning Decision Trees

- Goal: Build a decision tree for classifying examples as positive or negative instances of a concept
- Supervised learning, batch processing of training examples, using a preference bias
- A **decision tree** is a tree in which each non-leaf node has associated with it an attribute (feature), each leaf node has associated with it a classification (+ or -), and each arc has associated with it one of the possible values of the attribute at the node where the arc is directed from. For example,



- Preference Bias: **Ockham's Razor**: The simplest explanation that is consistent with all observations is the best. Here, that means the smallest decision tree that correctly classifies all of the training examples is best.
- Finding the provably smallest decision tree is an NP-Hard problem, so instead of constructing the absolute smallest tree that is consistent with all of the training examples, construct one that is pretty small.
- Decision Tree Construction using a Greedy Algorithm
  - Algorithm called ID3 or C5.0, originally developed by Quinlan (1987)
  - Top-down construction of the decision tree by recursively selecting the "best attribute" to use at the current node in the tree. Once the attribute is selected for the current node, generate children nodes, one for each possible value of the selected attribute. Partition the examples using the possible values of this attribute, and assign these subsets of the examples to the appropriate child node. Repeat for each child node until all examples associated with a node are either all positive or all negative.

Algorithm

```
function decision-tree-learning(examples, attributes, default)
  ;; examples is a list of training examples
  ;; attributes is a list of candidate attributes for the
  ;;    current node
  ;; default is the default value for a leaf node if there
  ;;    are no examples left
  if empty(examples) then return(default)
  if same-classification(examples) then return(class(examples))
  if empty(attributes) then return(majority-classification(examples))
  best = choose-attribute(attributes, examples)
  tree = new node with attribute best
  foreach value v of attribute best do
    v-examples = subset of examples with attribute best = v
    subtree = decision-tree-learning(v-examples, attributes - best,
                  majority-classification(examples))
    add a branch from tree to subtree with arc labeled v
  return(tree)
```

- How to Choose the Best Attribute for a Node?

Some possibilities:

- Random: Select any attribute at random
- Least-Values: Choose the attribute with the smallest number of possible values
- Most-Values: Choose the attribute with the largest number of possible values
- Max-Gain: Choose the attribute that has the largest expected information gain. In other words, try to select the attribute that will result in the smallest expected size of the subtrees rooted at its children.

The C5.0 algorithm uses the Max-Gain method of selecting the best attribute.

Information Gain Method for Selecting the Best Attribute

Use **information theory** to estimate the size of the subtrees rooted at each child, for each possible attribute. That is, try each attribute, evaluate and pick the best one.

- How much (expected) work is required to guess which element I am thinking of in a set S of size |S|?

$\log_2|S|$

That is, at each step we can ask a yes/no question that eliminates at most 1/2 of the elements remaining. Call this value the *information value* of being told which element it is without having to guess it.

- Given S = P union N, where P and N are two disjoint sets, how hard is it to guess which element I am thinking of in S?

  if x in P, then $\log_2|P| = \log_2 p$ questions needed, where p = |P|
  if x in N, then $\log_2|N| = \log_2 n$ questions needed, where n = |N|

  So, the expected number of questions that have to be asked is:

  $$(Pr(x \text{ in } P) * \log_2 p) + (Pr(x \text{ in } N) * \log_2 n)$$

  or, equivalently,

  $$(p/(p+n))\ \log_2 p + (n/(p+n))\ \log_2 n$$

- So, how much expected work is required to guess which element I am thinking of in a set S after I am told whether the element is in P or N?

  $$I(P,N) = \log_2|S| - (|P|/|S|\ \log_2|P|) - (|N|/|S|\ \log_2|N|)$$

  or, equivalently,

  $$I(\%P, \%N) = -(\%P\ \log_2 \%P) - (\%N\ \log_2 \%N)$$

  where %P = |P|/|S| = p/(p+n) (% positive examples in S), and %N = |N|/|S| = n/(p+n) (% negative examples in S).

  I measures the **information content** in bits (i.e., number of yes/no questions that must be asked) associated with a set S of examples, which consists of the subset P of positive examples and subset N of negative examples.

  Note: 0 <= I(P,N) <= 1, where 0 => no information, and 1 => maximum information.

- Example: Perfect Balance (Maximum Disorder) in S:

  Half the examples in S are positive and half are negative. Hence, %P = %N = 1/2. So,

  $$\begin{aligned} I(1/2, 1/2) &= -1/2\ \log_2 1/2 - 1/2\ \log_2 1/2 \\ &= -1/2\ (\log_2 1 - \log_2 2) - 1/2\ (\log_2 1 - \log_2 2) \\ &= -1/2\ (0 - 1) - 1/2\ (0 - 1) \\ &= 1/2 + 1/2 \\ &= 1 \quad \Rightarrow \text{information content is large} \end{aligned}$$

- Example: Perfect Homogeneity in S:

  Say all of the examples in S are positive and none are negative. Then, %P = 1, and %N = 0.

So,

$$I(1,0) = -1 \log_2 1 - 0 \log_2 0$$
$$= -0 - 0$$
$$= 0 \Rightarrow \text{information content is low}$$

Low information content is desirable in order to make the smallest tree because low information content means that most of examples are classified the SAME, and therefore we would expect that the rest of the tree rooted at this node will be quite small to differentiate between the two classifications.

Now, measure the **information gained** by using a given attribute. That is, measure the difference in the information content of a node and the information content after a node splits up the examples based on a selected attribute's possible values. To do this, we need a measure of the information content after "splitting" a node's examples into its children based on a hypothesized attribute.

Given a node with a set of examples S = P union N, and an hypothesized attribute A that has m possible values, define **Remainder(A)** as the weighted sum of the information content of each subset of the examples that are associated with each child node as imposed by possible values of the attribute. More specifically, let

S_i = subset of S with value i, i=1,...,m
P_i = subset of S_i that are +
N_i = subset of S_i that are -
$q_i = |S_i|/|S|$ = % of examples on branch i
$\%P_i = |P_i|/|S_i|$ = % of + examples on branch i
$\%N_i = |N_i|/|S_i|$ = % of - examples on branch i

$$\text{Remainder}(A) = \sum_{i=1}^{m} q_i \, I(\%P_i, \%N_i)$$

So, Remainder(A) is a weighted sum of the information content at each child node generated by that attribute. It measures the total "disorder" or "inhomogeneity" of the children nodes. 0 <= Remainder(A) <= 1.

Now, measure the **gain** from using the attribute test at the current node, defined by:

$$\text{Gain}(A) = I(\%P, \%N) - \text{Remainder}(A)$$

The best attribute at a node is now defined as the attribute A with maximum Gain(A) of all the possible attributes that can be used at the node. Since at a given node I(%P, %N) is constant, this is equivalent to selecting the attribute A with minimum Remainder(A).

Example

Consider the following six training examples, where each example has three attributes: color, shape and size. Color has three possible values: red, green and blue. Shape has two possible values: square and round. Size has two possible values: big and small.

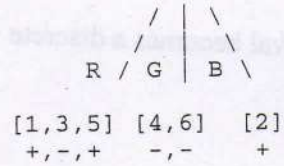
Example	Color	Shape	Size	Class
1	red	square	big	+
2	blue	square	big	+
3	red	round	small	-
4	green	square	small	-
5	red	round	big	+
6	green	square	big	-

- **Which is best attribute for the root node of decision tree?**

```
Remainder(color) = 3/6 I(2/3,1/3) + 1/6 I(1/1,0/1) + 2/6 I(0/2,2/2)
                   |       |       |       |       |       |
                   |       |       |       1 of 6 is blue  2 of 6 are green
                   |       |       1 of the red is negative
                   |       2 of the red are positive
                   3 out of 6 are red
                 = 1/2 * (-2/3 log2 2/3  - 1/3 log2 1/3)
                     + 1/6 * (-1 log2 1 - 0 log2 0)
                     + 2/6 * (-0 log2 0 - 1 log2 1)
                 = 1/2 * (-2/3(log2 2 - log2 3) - 1/3(log2 1 - log2 3))
                     + 1/6 * 0
                     + 2/6 * 0
                 = 1/2 * (-2/3(1 - 1.58) - 1/3(0 - 1.58))
                 = 1/2 * 0.914
                 = 0.457

Gain(color) = I(3/6, 3/6) - Remainder(color)
            = 1.0 - 0.457
            = 0.543

Remainder(shape) = 4/6 I(2/4, 2/4) + 2/6 I(1/2, 1/2)
                 = 4/6 * 1.0 + 2/6 * 1.0
                 = 1.0

Gain(shape) = I(3/6, 3/6) - Remainder(shape)
            = 1.0 - 1.0
            = 0.0

Remainder(size) = 4/6 I(3/4, 1/4) + 2/6 I(0/2, 2/2)
                = 0.541

Gain(size) = I(3/6, 3/6) - Remainder(size)
           = 1.0 - 0.541
           = 0.459
```

Max(0.543, 0.0, 0.459) = 0.543, so color is best. Make the root node's attribute color and partition the examples for the resulting children nodes as shown:

```
color
```



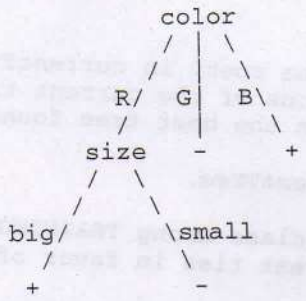
The children associated with values green and blue are uniform, containing only - and + examples, respectively. So make these children leaves with classifications - and +, respectively.

- **What is the best attribute for the red child node?**

Now recurse on red child node, containing three examples, [1,3,5], and two remaining attributes, [shape, size].

```
Remainder(shape) = 1/3 I(1/1, 0/1) + 2/3 I(1/2, 1/2)
                 = 1/3 * 0 + 2/3 * 1
                 = 0.667

Gain(shape) = I(2/3, 1/3) - .667
            = .914 - .667
            = 0.247

Remainder(size) = 2/3 I(2/2, 0/2) + 1/3 I(0/1, 1/1)
                = 2/3 * 0 + 1/3 * 0
                = 0

Gain(size) = I(2/3, 1/3) - 0
           = 0.914
```

Max(.247, .914) = .914, so make size the attribute at this node. It's children are uniform in their classifications, so the final decision tree is:



Case Studies

Many case studies have shown that decision trees are at least as accurate as human experts. For example, one study for diagnosing breast cancer had humans correctly classifying the examples 65% of the time, and the decision tree classified 72% correct.

British Petroleum designed a decision tree for gas-oil separation for offshore oil platforms. Replaced a rule-based expert system.

Cessna designed an airplane flight controller using 90,000 examples and 20 attributes per example.

Extensions of the Decision Tree Learning Algorithm

- **Real-valued data**

Select a set of thresholds defining intervals; each interval becomes a discrete value of the attribute

- **Noisy data and Overfitting**
  There are many kinds of "noise" that could occur in the examples:
  - Two examples have the same attribute, value pairs, but different classifications
  - Some values of attributes are incorrect because of errors in the data acquisition process or the preprocessing phase
  - The classification is wrong (e.g., + instead of -) because of some error
  - Some attributes are irrelevant to the decision-making process. For example, the color of a die is irrelevant to its outcome.

The last problem, irrelevant attributes, can result in **overfitting** the training example data. For example if the hypothesis space has many dimensions because there are a large number of attributes, then we may find meaningless regularity in the data that is irrelevant to the true, important, distinguishing features. Fix by pruning lower nodes in the decision tree. For example, if Gain of the best attribute at a node is below a threshold, stop and make this node a leaf rather than generating children nodes.

One way to address the overfitting problem in decision-tree induction is to use a tuning set in conjunction with a **pruning algorithm**. The following is a greedy algorithm for doing this.

```
Let bestTree = the tree produced by C5.0 on the TRAINING set
Let bestAccuracy = the accuracy of bestTree on the TUNING set
Let progressMade = true

while (progressMade) // Continue as long as improvement on TUNING SET
{
    Set progressMade = false
    Let currentTree = bestTree

    For each interiorNode N (including the root) in currentTree
    {   // Consider various pruned versions of the current tree
        // and see if any are better than the best tree found so far

        Let prunedTree be a copy of currentTree,
        except replace N by a leaf node
        whose label equals the majority class among TRAINING set
        examples that reached node N (break ties in favor of '-')

        Let newAccuracy = accuracy of prunedTree on the TUNING set

        // Is this pruned tree an improvement, based on the TUNE set?
        // When a tie, go with the smaller tree (Occam's Razor).
        If (newAccuracy >= bestAccuracy)
        {
          bestAccuracy = newAccuracy
          bestTree = prunedTree
          progressMade = true
        }
    }
}
return bestTree
```

- **Generation of rules**
  Each path, from the root to a leaf, corresponds to a rule where all of the decisions leading to the leaf define the antecedent to the rule, and the consequent is the classification at the leaf node. For example, from the tree above we could generate the rule:

```
if color = red and size = big then +
```

By constructing a rule for each path to a leaf yields an interpretation of what the tree means.

- **Setting Parameters**
  Some learning algorithms require setting learning parameters. Parameters must be set without looking at the test data! One method: Tuning Sets.

  Using Tuning Sets for Parameter Setting
  1. Partition data in Training set and Test set. Then partition Training set into Train set and Tune set.
  2. For each candidate parameter value, generate decision tree using the Train set
  3. Use Tune set to evaluate error rates and determine which parameter value is best
  4. Compute new decision tree using selected parameter values and entire Training set

- **Cross-Validation for Experimental Validation of Performance**
  1. Divide all examples into N disjoint subsets, $E = E_1, E_2, ..., E_N$
  2. For each i = 1, ..., N do
     - Test set = E_i
     - Training set = $E - E_i$
     - Compute decision tree using Training set
     - Determine performance accuracy P_i using Test set
  3. Compute N-fold cross-validation estimate of performance = $(P_1 + P_2 + ... + P_N)/N$

  One special case of interest called **Leave-1-Out** where N-fold cross-validation uses N = number of examples. Good when the number of examples available is small (less than about 100)

Summary

- One of the most widely used learning methods in practice
- Can out-perform human experts in many problems
- Strengths: Fast; simple to implement; can convert result to a set of easily interpretable rules; empirically valid in many commercial products; handles noisy data
- Weaknesses: "Univariate" splits/partitioning using only one attribute at a time so limits types of possible trees; large decision trees may be hard to understand; requires fixed-length feature vectors; non-incremental (i.e., batch method).

Last modified January 31, 2001

Uninformed Search (Chapter 3)

Building Goal-Based Agents

To build a goal-based agent we need to answer the following questions:

1. What is the **goal** to be achieved?

   Could describe a situation we want to achieve, a set of properties that we want to hold, etc. Requires defining a "goal test" so that we know what it means to have achieved/satisfied our goal.

   This is a hard part that is rarely tackled in AI, usually assuming that the system designer or user will specify the goal to be achieved. Certainly psychologists and motivational speakers always stress the importance of people establishing clear goals for themselves as the first step towards solving a problem. What are *your* goals???

2. What are the **actions**?

   Quantify all of the primitive actions or events that are sufficient to describe all necessary changes in solving a task/goal. No uncertainty associated with what an action does to the world. That is, given an action (also called an **operator** or **move**) and a description of the current state of the world, the action completely specifies (1) if that action CAN be applied to the current world (i.e., is it applicable and legal), and (2) what the exact state of the world will be after the action is performed in the current world (i.e., we don't need any "history" information to be able to compute what the new world looks like).

   Note also that actions can all be considered as discrete events that can be thought of as occurring at an instant of time. That is, the world is in one situation, then an action occurs and the world is now in a new situation. For example, if "Mary is in class" and then performs the action "go home," then in the next situation she is "at home." There is no representation of a point in time where she is neither in class nor at home (i.e., in the state of "going home").

   The number of operators needed depends on the representation used in describing a state (see below). For example, in the 8-puzzle, we could specify 4 possible moves for each of the 8 tiles, resulting in a total of 4*8=32 operators. On the other hand, we could specify four moves for the "blank" square and there would need to be only 4 operators.

3. What information is necessary to encode about the the world to sufficiently describe all relevant aspects to solving the goal? That is, what knowledge needs to be represented in a **state description** to adequately describe the current **state** or **situation** of the world?

   The size of a problem is usually described in terms of the number of states that are possible. For example, in Tic-Tac-Toe there are about 3^9 states. In Checkers there are about 10^40 states. Rubik's Cube has about 10^19 states. Chess has about 10^120 states in a typical game.

We will use the **Closed World Assumption**: All necessary information about a problem domain is available in each percept so that each state is a complete description of the world. There is no incomplete information at any point in time.

What's in a state is the *knowledge representation problem*. That is, we must decide what information from the raw percept data is relevant to keep, and what form the data should be represented in so as to make explicit the most important features of the data for solving the goal. Is the color of the boat relevant to solving the Missionaries and Cannibals problem? Is sunspot activity relevant to predicting the stock market? *What* to represent is a very hard problem that is usually left to the system designer to specify. *How* to represent domain knowledge is a topic that will be treated later in the course.

Related to this is the issue of what **level of abstraction** or detail to describe the world. Too fine-grained and we'll "miss the forest for the trees." Too coarse-grained and we'll miss critical details for solving the problem.

The number of states depends on the representation and level of abstraction chosen. For example, in the Remove-5-Sticks problem, if we represent the individual sticks, then there are 17-choose-5 possible ways of removing 5 sticks. On the other hand, if we represent the "squares" defined by 4 sticks, then there are 6 squares initially and we must remove 3 squares, so only 6-choose-3 ways of removing 3 squares.

Examples

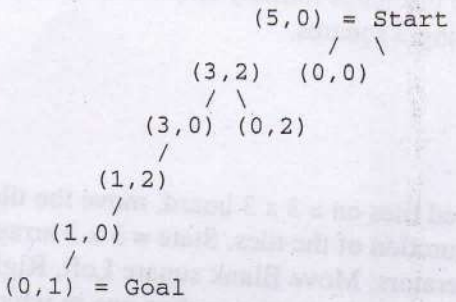
- **8-Puzzle**
  Given an initial configuration of 8 numbered tiles on a 3 x 3 board, move the tiles in such a way so as to produce a desired goal configuration of the tiles. State = 3 x 3 array configuration of the tiles on the board. Operators: Move Blank square Left, Right, Up or Down. (Note: this is a more efficient encoding of the operators than one in which each of four possible moves for each of the 8 distinct tiles is used.) Initial State: A particular configuration of the board. Goal: A particular configuration of the board.

- **Missionaries and Cannibals**
  There are 3 missionaries, 3 cannibals, and 1 boat that can carry up to two people on one side of a river. Goal: Move all the missionaries and cannibals across the river. Constraint: Missionaries can never be outnumbered by cannibals on either side of the river, or else the missionaries are killed. State = configuration of missionaries and cannibals and boat on each side of the river. Operators: Move boat containing some set of occupants across the river (in either direction) to the other side.

- **Cryptarithmetic**
  Find an assignment of digits (0, ..., 9) to letters so that a given arithmetic expression is true. For example, SEND + MORE = MONEY Note: In this problem, unlike the two above, the solution is NOT a sequence of actions that transforms the initial state into the goal state, but rather the solution is simply finding a goal node that includes an assignment of digits to each of the distinct letters in the given problem.

- **Remove 5 Sticks**
  Given the following configuration of sticks, remove exactly 5 sticks in such a way that the remaining configuration forms exactly 3 squares.

- **Water Jug Problem**
  Given a 5-gallon jug and a 2-gallon jug, with the 5-gallon jug initially full of water and the 2-gallon jug empty, the goal is to fill the 2-gallon jug with exactly one gallon of water.
  - State = (x,y), where x = number of gallons of water in the 5-gallon jug and y is gallons in the 2-gallon jug
  - Initial State = (5,0)
  - Goal State = (*,1), where * means any amount
  - Operators
    - (x,y) -> (0,y) ; empty 5-gal jug
    - (x,y) -> (x,0) ; empty 2-gal jug
    - (x,2) and x<=3 -> (x+2,0) ; pour 2-gal into 5-gal
    - (x,0) and x>=2 -> (x-2,2) ; pour 5-gal into 2-gal
    - (1,0) -> (0,1) ; empty 5-gal into 2-gal
  - State Space (also called the Problem Space)



Formalizing Search in a State Space

- A **state space** is a *graph*, (V, E), where V is a set of **nodes** and E is a set of **arcs**, where each arc is *directed* from a node to another node
- Each **node** is a data structure that contains a state description *plus* other information such as the parent of the node, the name of the operator that generated the node from that parent, and other bookkeeping data
- Each **arc** corresponds to an instance of one of the operators. When the operator is applied to the state associated with the arc's source node, then the resulting state is the state associated with the arc's destination node
- Each arc has a fixed, positive cost associated with it corresponding to the cost of the operator
- Each node has a set of *successor nodes* corresponding to all of the legal operators that can be applied at the source node's state. The process of **expanding a node** means to generate all of the successor nodes and add them and their associated arcs to the state-space graph
- One or more nodes are designated as **start nodes**

- A **goal test** predicate is applied to a state to determine if its associated node is a **goal node**
- A **solution** is a sequence of operators that is associated with a *path* in a state space from a start node to a goal node
- The **cost of a solution** is the sum of the arc costs on the solution path
- State-space search is the process of searching through a state space for a solution by *making explicit a sufficient portion of an implicit state-space graph to include a goal node*. Hence, initially V={S}, where S is the start node; when S is expanded, its successors are generated and those nodes are added to V and the associated arcs are added to E. This process continues until a goal node is found
- Each node implicitly or explicitly represents a **partial solution path** (and cost of the partial solution path) from the start node to the given node. In general, from this node there are many possible paths (and therefore solutions) that have this partial path as a prefix

State-Space Search Algorithm

```
function general-search(problem, QUEUEING-FUNCTION)
  ;; problem describes the start state, operators, goal test, and
  ;;   operator costs
  ;; queueing-function is a comparator function that ranks two states
  ;; general-search returns either a goal node or "failure"

  nodes = MAKE-QUEUE(MAKE-NODE(problem.INITIAL-STATE))
  loop
     if EMPTY(nodes) then return "failure"
     node = REMOVE-FRONT(nodes)
     if problem.GOAL-TEST(node.STATE) succeeds
        then return node
     nodes = QUEUEING-FUNCTION(nodes, EXPAND(node, problem.OPERATORS))
     ;; Note: The goal test is NOT done when nodes are generated
     ;; Note: This algorithm does not detect loops
  end
```

Key Procedures to be Defined

- EXPAND
  Generate all successor nodes of a given node
- GOAL-TEST
  Test if state satisfies all goal conditions
- QUEUEING-FUNCTION
  Used to maintain a ranked list of nodes that are candidates for expansion

Key Issues

- Search process constructs a **search tree**, where root is the initial state and all of the leaf nodes are nodes that have not yet been expanded (i.e., they are in the list "nodes") or are nodes that have no successors (i.e., they're "deadends" because no operators were applicable and yet they are not goals)
- Search tree may be infinite because of loops even if state space is small

- Return a path or a node depending on problem. E.g., in cryptarithmetic return a node; in 8-puzzle return a path
- Changing definition of the QUEUEING-FUNCTION leads to different search strategies

Evaluating Search Strategies

- **Completeness**
  Guarantees finding a solution whenever one exists
- **Time Complexity**
  How long (worst or average case) does it take to find a solution? Usually measured in terms of the number of nodes expanded. For an overview of "Big-Oh" notation used for measuring time (and space) complexity, see CS 577 notes on Big-Oh notation.
- **Space Complexity**
  How much space is used by the algorithm? Usually measured in terms of the maximum size that the "nodes" list becomes during the search.
- **Optimality/Admissibility**
  If a solution is found, is it guaranteed to be an optimal one? That is, is it the one with minimum cost?

Uninformed Search Strategies

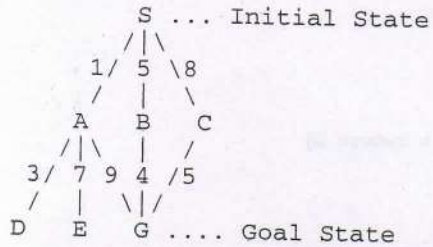
This set of strategies orders nodes without using any domain specific information.

- **Breadth-First** (BFS)
  - Enqueue nodes on `nodes` in FIFO (first-in, first-out) order. That is, `nodes` used as a *queue* data structure to order nodes.
  - Complete
  - Optimal (i.e., admissible) if all operators have the same cost. Otherwise, not optimal but finds solution with shortest path length.
  - Exponential time and space complexity, O(b^d), where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node
  - Will take a long time to find solutions with a large number of steps because must look at all shorter length possibilities first
  - A complete search tree of depth d where each non-leaf node has b children, has a total of 1 + b + b^2 + ... + b^d = (b^(d+1) - 1)/(b-1) nodes
  - For a complete search tree of depth 12, where every node at depths 0, ..., 11 has 10 children and every node at depth 12 has 0 children, there are 1 + 10 + 100 + 1000 + ... + 10^12 = (10^13 - 1)/9 = O(10^12) nodes in the complete search tree. If BFS expands 1000 nodes/sec and each node uses 100 bytes of storage, then BFS will take 35 years to run in the worst case, and it will use 111 terabytes of memory!
- **Depth-First** (DFS)
  - Enqueue nodes on `nodes` in LIFO (last-in, first-out) order. That is, `nodes` used as a *stack* data structure to order nodes.
  - May not terminate without a "depth bound," i.e., cutting off search below a fixed depth D
  - Not complete (with or without cycle detection, and with or without a cutoff (depth boound) depth)
  - Exponential time, O(b^d), but only linear space, O(bd), required

- Can find long solutions quickly if lucky
- When search hits a deadend, can only back up one level at a time even if the "problem" occurs because of a bad operator choice near the top of the tree. Hence, only does "*chronological backtracking*"

- **Uniform-Cost** (UCS)
  - Enqueue nodes by path cost. That is, let $g(n)$ = cost of the path from the start node to the current node n. Sort `nodes` by increasing value of g.
  - Called "Dijkstra's Algorithm" in the algorithms literature
  - Similar to "Branch and Bound Algorithm" in operations research literature
  - Complete
  - Optimal/Admissible
  - Admissibility depends on the goal test being applied when a node is removed from the nodes list, not when it's parent node is expanded and the node is first generated
  - Exponential time and space complexity, O(b^d)
- **Depth-First Iterative Deepening** (IDS)
  -
```
c=1
until solution found do
   DFS with depth bound (aka cutoff) c
   c = c+1
```
  - First do DFS to depth 1 (i.e., consider children of the start node to have no successors); then, if no solution found, do DFS to depth 2; etc.
  - Complete
  - Optimal/Admissible if all operators have the same cost. Otherwise, not optimal but does guarantee finding solution of shortest length (like BFS).
  - Time complexity is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, O(b^d)
  - Example: If branching factor is b and solution is at depth d, then all nodes at depth d are generated at most once, all nodes at depth d-1 are generated at most twice, etc. Hence b^d + 2b^(d-1) + ... + db <= b^d / (1 - 1/b)^2 = O(b^d). If b=4, then worst case is 1.78 * 4^d. In other words 78% more nodes searched than exist at depth d (in the worst case).
  - Linear space complexity, O(bd), like DFS
  - Has advantage of BFS (i.e., completeness) and also advantages of DFS (i.e., limited space and finds longer paths more quickly)

Example Illustrating Uninformed Search Strategies



Nodes expanded by:

- Depth-First Search: S A D E G
  Solution found: S A G
- Breadth-First Search: S A B C D E G

Solution found: S A G

- Uniform-Cost Search: S A D B C E G
  Solution found: S B G
  This is the only uninformed search that worries about costs.

- Iterative-Deepening Search: S A B C S A D E G
  Solution found: S A G

Depth-First Search

```
return GENERAL-SEARCH(problem, ENQUEUE-AT-FRONT)
```

```
expanded
node   nodes list
----   ----------
                { S }
  S             { A B C }
  A             { D E G B C}
  D             { E G B C }
  E             { G B C }
  G             { B C }

Solution path found is S A G  <-- this G has cost 10
Number of nodes expanded (including goal node) = 5
```

Breadth-First Search

```
return GENERAL-SEARCH(problem, ENQUEUE-AT-END)
```

```
expanded
node   nodes list
----   ----------
                { S }
  S             { A B C }
  A             { B C D E G }
  B             { C D E G G' }
  C             { D E G G' G" }
  D             { E G G' G" }
  E             { G G' G" }
  G             { G' G" }

Solution path found is S A G  <-- this G also has cost 10
Number of nodes expanded (including goal node) = 7
```

Uniform-Cost Search

```
return GENERAL-SEARCH(problem, ENQUEUE-BY-PATH-COST)
```

```
expanded
node   nodes list
----   ----------
                { S }
  S             { A(1) B(5) C(8) }
  A             { D(4) B(5) C(8) E(8) G(10) }   (note, we don't return G)
  D             { B(5) C(8) E(8) G(10) }
  B             { C(8) E(8) G(9) G(10) }
  C             { E(8) G(9) G(10) G(13) }
  E             { G(9) G(10) G(13) }
  G             { }

Solution path found is S B G  <-- this G has cost 9, not 10
Number of nodes expanded (including goal node) = 7
```

Last modified September 16, 1998

Informed Search (Chapter 4)

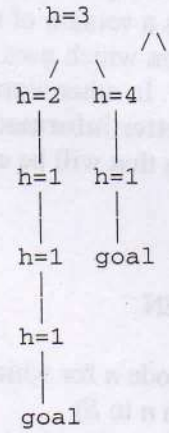
Informed Methods Add Domain-Specific Information

- Add domain-specific information to select what is the best path to continue searching along
- Define a **heuristic function**, $h(n)$, that estimates the "goodness" of a node n. Specifically, $h(n)$ = estimated cost (or distance) of minimal cost path from n to a goal state.
- The term *heuristic* means "serving to aid discovery" and is an estimate, based on domain-specific information that is computable from the current state description, of how close we are to a goal
- Example heuristics:
  - Missionaries and Cannibals: Number of people on the starting bank of the river
  - 8-puzzle: Number of tiles out of place
  - 8-puzzle: Sum of distances each tile is from its goal position
- $h(n) >= 0$ for all nodes n
- $h(n) = 0$ implies that n is a goal node
- $h(n)$ = infinity implies that n is a deadend from which a goal cannot be reached
- All domain knowledge used in the search is encoded in the heuristic function h. Consequently, this is an example of a "*weak method*" because of the limited way that domain-specific information is used to solve a problem.

Informed Methods

- **Best-First Search**
  Order nodes on the *nodes* list by increasing value of an evaluation function, f, that incorporates domain-specific information in some way. This is a generic way of referring to the class of informed methods.

- **Greedy Search**
  Use as an evaluation function $f(n) = h(n)$, sorting nodes by increasing values of f
  - Selects node to expand that is believed to be closest (hence it's "greedy") to a goal node (i.e., smallest f value)
  - Not complete
  - Not admissible, as shown in the following example:



Assuming all arc costs are 1, then Greedy search will find the left goal, which has a solution cost of 5, while the optimal solution is the path to the right goal, which has a cost of 3.

- **Beam Search**
  - Use an evaluation function $f(n) = h(n)$, but the maximum size of the *nodes* list is k, a fixed constant
  - Only keeps k best nodes as candidates for expansion, and throws the rest away
  - More space efficient than Greedy Search, but may throw away a node that is on a solution path
  - Not complete
  - Not admissible

- **Algorithm A**
  Use as an evaluation function $f(n) = g(n) + h(n)$, where $g(n)$ is as defined in Uniform-Cost search. That is, $g(n)$ = minimal cost path from the start state to the current state n.
  - Adds a "breadth-first" component to the evaluation function by including the g term
  - Ranks nodes on the search frontier by the estimated cost of a solution that goes from the start node through the given node to a goal node. That is, $g(n)$ is the cost from the start node to node n, and $h(n)$ is the estimated cost from node n to a goal.
  - Not complete since if a node n is on the solution path but $h(n)$ = infinity, then node n may never be expanded
  - Not admissible

- **Algorithm A***
  Use the same evaluation function as used by Algorithm A except add the constraint that for *all* nodes n in the search space, $h(n) <= h^*(n)$, where $h^*(n)$ = the *true cost* of the minimal cost path from n to a goal.
  - When the condition $h(n) <= h^*(n)$ holds, we say that h is **admissible**.
  - Using an admissible heuristic guarantees that a node on the optimal path can never look so bad that you bypass it forever
  - A* is **complete** whenever the branching factor is finite, and every operator has a fixed positive cost
  - A* is **admissible**
  - If $h(n) = h^*(n)$ for all n, then *only* the nodes on the optimal solution path will be expanded. So, no extra work will be performed.
  - If $h(n) = 0$ for all n, then this is an admissible heuristic and results in A* performing exactly as the Uniform-Cost Search does
  - If $h1(n) < h2(n) <= h^*(n)$ for all n that are not goal nodes, then $h2$ is a **better heuristic** than $h1$ in the sense that if A1* is a version of the A* algorithm which uses $h1$, and A2* is a version of the A* algorithm which uses $h2$, then every node expanded by A2* is also expanded by A1*. In other words, A1* expands at least as many nodes as A2*. We say that A2* is **better informed** than A1*.
  - The closer h is to h^*, the fewer extra nodes that will be expanded

A* Algorithm

1. Put the start node S on the *nodes* list, called OPEN
2. If OPEN is empty, exit with failure
3. Remove from OPEN and place on CLOSED a node n for which $f(n)$ is minimum
4. If n is a goal node, exit (trace back pointers from n to S)
5. Expand n, generating all its successors and attach to them pointers back to n. For each

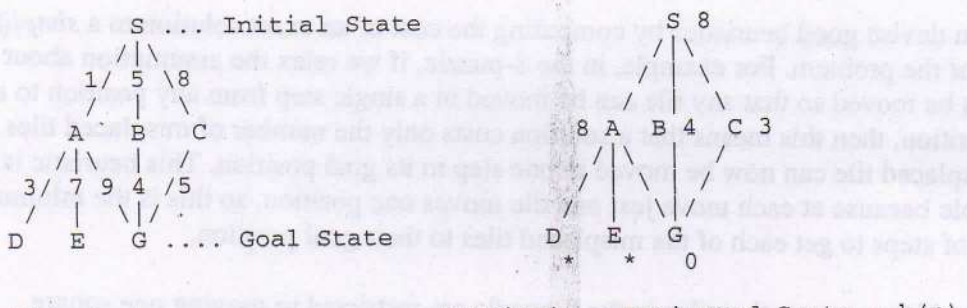
successor n' of n

1. If n' is not already on OPEN or CLOSED estimate $h(n'), g(n')=g(n)+c(n,n'), f(n')=g(n')+h(n')$, and place it on OPEN.
2. If n' is already on OPEN or CLOSED, then check if $g(n')$ is lower for the new version of n'. If so, then:
   (i) Redirect pointers backward from n' along path yielding lower $g(n')$.
   (ii) Put n' on OPEN.
   If $g(n')$ is not lower for the new version, do nothing.

6. Goto 2

Example

Consider the following search space where the start state is S and the goal state is G. The left figure shows the arcs labeled with the costs of the associated operators. The right figure shows the states labeled with the value of the heuristic function, h, if it is ever applied at that state.



Summary of **g(n)**, **h(n)**, **f(n) = g(n) + h(n)**, as well as **h*(n)**, the hypothetical perfect heuristic:

n	g(n)	h(n)	f(n)	h*(n)
S	0	8	8	9
A	1	8	9	9
B	5	4	9	4
C	8	3	11	5
D	4	inf	inf	inf
E	8	inf	inf	inf
G	10/9/13	0	10/9/13	0

```
Notice that since h(n) <= h*(n) for all n, h is admissible
Optimal path = S B G
Cost of the optimal path = 9
```

- **Greedy Search**: $f(n) = h(n)$

```
node
expanded    nodes list
----        ----------
            { S(8) }
  S         { C(3) B(4) A(8) }
  C             { G(0) B(4) A(8) }
  G             { B(4) A(8) }

Solution path found is S C G.  #nodes expanded = 3.
```

See how fast the search is!! But it is **NOT** optimal.

- **A* Search**: $f(n) = g(n) + h(n)$

```
node
expanded    nodes list
----        ------------------
            { S(8) }
  S         { A(9) B(9) C(11) }
  A              { B(9) G(10) C(11) D(inf) E(inf) }
  B              { G(9) G(10) C(11) D(inf) E(inf) }
  G              { C(11) D(inf) E(inf) }

Solution path found is S B G.  #nodes expanded = 4.
```

Still pretty fast. And optimal, too.

Devising Heuristics

- Good heuristics must be fast to compute, because if it takes so long to compute the value of a heuristic at a single node, it may have been preferable to have just expanded more nodes using a cheaper heuristic. For example, if the heuristic function is a breadth-first search to find a solution and its cost, then this is clearly too expensive to be useful.

- Can often devise good heuristics by computing the cost of an exact solution to a *simplified* version of the problem. For example, in the 8-puzzle, if we relax the assumption about how a tile can be moved so that any tile can be moved in a single step from any position to any other position, then this means that a solution costs only the number of misplaced tiles since each misplaced tile can now be moved in one step to its goal position. This heuristic is admissible because at each move just one tile moves one position, so this is the minimum number of steps to get each of the misplaced tiles to their goal position.

  Similarly, if we assume that tiles in the 8-puzzle are restricted to moving one square horizontally or vertically at a time, but we relax the assumption that only one tile can occur at a board position at a time, then each tile can be moved independently to its goal position, taking a number of steps equal to the Manhattan distance from its start position to its goal position. This leads to a heuristic which is the sum of the distances of the misplaced tiles to their goal positions. This heuristic is admissible.

- The heuristic function h is an indicator of "adventurousness" in that in Algorithms A and A* a good heuristic allows successive nodes on a single path to be expanded in succession even when several "good" steps are intermixed with a few "bad" steps.

- Unfortunately, A* often suffers because it cannot venture down a single path unless it is almost continuously having success (i.e., h is decreasing). Any failure to decrease h will almost immediately cause the search to switch to another path.

- In order to devise an admissible heuristic, h must frequently be very simple and therefore resorts to (almost) uniform-cost search through parts of the search space.

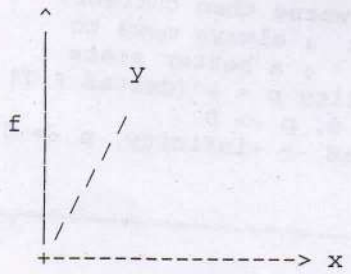
- If optimality is not required, i.e., a satisficing solution is enough, using a heuristic that occasionally overestimates the actual cost but is usually very close to the actual cost (over or under), will result in many fewer nodes being expanded to find a solution than using a provably admissible heuristic.

Iterative Improvement Algorithms

- Rather than searching for a solution path and then executing the steps associated with the solution path, iteratively pick a next best move, make that move, and then repeat. Hence, **irrevocably** make a decision about one step at each iteration.

- Best used in problems where all the information for a solution is contained in the node itself. For example, cryptarithmetic problems.
- Rather than trying to find a solution with minimum value of the evaluation function, *f*, for historical reasons, we instead will attempt to **maximize** the function. That is, the goal is to find a state, *n*, such that *f(n) >= f(i)* for all states *i* in the state space.

- **Hill-Climbing Search**
  - Look at all immediate successors of current state *m*
  - If there exists a successor *n* such that *f(n) > f(m)*, and *f(n) >= f(t)* for all the successors *t* of *m*, then move from *m* to *n*. Otherwise, halt at *m*. (Note: This definition differs from the textbook in that we require *f(n)* to be strictly greater than *f(m)*; the textbook's algorithm states that *f(n)* must only be greater than or equal to *f(m)*. Their definition allows the algorithm to move through states that are equal in their values of *f*.)
  - Looks one step ahead to determine if any successor is better than the current state; if there is, move to the best successor.
  - Similar to Greedy search but Hill-Climbing does not allow backtracking or jumping to an alternative path since there is no *nodes* list of other candidate frontier nodes from which the search could be continued. Corresponds to Beam search with a beam width of 1 (i.e., the maximum size of the *nodes* list is 1).
  - Not complete since the search will terminate at "local maxima," "plateaus," and "ridges."
  - Algorithm visualized in terms of a surface in 3D:



Consider the state space to be the set of points in the x,y plane, and for each such point the height f is the value of the evaluation function for that state. This height function, *f = f(x,y)*, defines a surface. The initial state corresponds to a point on this surface and the goal is to find a state where the height is a global maximum.

Hill-climbing (should be called Valley-Finding in this context where we are minimizing instead of maximizing a value) moves in the **direction of steepest ascent** since it moves to the successor (i.e., adjacent) node that increases *f* the most.

Notice that by considering the state space as a continuous space of points in the x,y plane, if the height surface is continuous (i.e., smooth so that derivatives are well-defined everywhere), then the direction of steepest ascent corresponds to the **gradient direction** = [df(x,y)/dx, df(x,y)/dy], and the search is called **gradient ascent**.

- **Simulated Annealing**
  - Named after a metal-casting technique called annealing where molten metal is heated and then gradually cooled resulting in an even distribution of the molecules and a desired crystalline structure
  - Attempts to fix the problem with hill-climbing methods where the search gets stuck in a local maximum.
  - Basic idea: Instead of picking the best move, pick a random move; if the successor

state obtained by this move is an improvement over the current state, then do it. Otherwise, make the move with some probability < 1. The probability decreases exponentially with the badness of the move.

- Define a temperature function that decreases over time. At each move, compute the current temperature T, and use T to determine the probability with which to allow a move to a worse state. In the limit, T goes to 0 at which point the method is doing hill-climbing. Hence the probability is proportional to T.
- If temperature is lowered slowly enough, simulated annealing is complete and admissible. Intuitively, this is the case because the temperature can be controlled so that it is large enough to move off a local maximum, but small enough to not move off a global maximum.
- Algorithm
  Assume we are trying to find a state where some evaluation function *f* is a global *maximum*:

```
current = Initial-State(problem)
for t = 1 to infinity do
   T = Schedule(t)    ; T is the current temperature, which
                      ; is monotonically decreasing with t
   if T=0 then return current  ; halt when temperature = 0
   next = Select-Random-Successor-State(current)
   deltaE = f(next) - f(current)  ; If positive, next is
                                  ; better than current.
                                  ; Otherwise, next is
                                  ; worse than current.
   if deltaE > 0 then current = next  ; always move to
                                      ; a better state
   else current = next with probability p = e^(deltaE / T)
                       ; as T -> 0, p -> 0
                       ; as deltaE -> -infinity, p -> 0
end
```

Last modified September 23, 1998

Game Playing (Chapter 5)

Formulating Game Playing as Search

- Consider 2-person, zero-sum, perfect information (i.e., both players have access to complete information about the state of the game, so no information is hidden from either player) games. Players alternate moves and there is no chance (e.g., using dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, and Othello
- **Iterative** methods apply here because search space is too large for interesting games to search for a "solution." Therefore, search will be done before EACH move in order to select the best next move to be made.
- **Adversary** methods needed because alternate moves are made by an opponent who is trying to win. Therefore must incorporate the idea that an adversary makes moves that are "not controllable" by you.
- **Evaluation function** is used to evaluate the "goodness" of a configuration of the game. Unlike in heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node, here the evaluation function, also called the *static evaluation function* estimates board quality in leading to a win for one player.
- Instead of modeling the two players separately, the zero-sum assumption and the fact that we don't have, in general, any information about how our opponent plays, means we'll use a single evaluation function to describe the goodness of a board with respect to BOTH players. That is, $f(n)$ = large positive value means the board associated with node n is good for me and bad for you. $f(n)$ = large negative value means the board is bad for me and good for you. $f(n)$ near 0 means the board is a neutral position. $f(n)$ = +infinity means a winning position for me. $f(n)$ = -infinity means a winning position for you.
- Example of an Evaluation Function for Tic-Tac-Toe:
  $f(n)$ = [number of 3-lengths open for me] - [number of 3-lengths open for you]
  where a 3-length is a complete row, column, or diagonal.
- Most evaluation functions are specified as a weighted sum of "features:" (w1 * feat1) + (w2 * feat2) + ... + (wn * featn). For example, in chess some features evaluate piece placement on the board and other features describe configurations of several pieces. Deep Blue has about 6000 features in its evaluation function.

Game Trees

- Root node represents the configuration of the board at which a decision must be made as to what is the best single move to make next. If it is my turn to move, then the root is labeled a "MAX" node indicating it is my turn; otherwise it is labeled a "MIN" node to indicate it is my opponent's turn.
- Arcs represent the possible legal moves for the player that the arcs emanate from
- Each level of the tree has nodes that are all MAX or all MIN; since moves alternate, the nodes at level i are of the opposite kind from those at level i+1

Searching Game Trees using the Minimax Algorithm

Steps used in picking the next move:

1. Create start node as a MAX node (since it's my turn to move) with current board configuration
2. Expand nodes down to some depth (i.e., ply) of lookahead in the game
3. Apply the evaluation function at each of the leaf nodes
4. "Back up" values for each of the non-leaf nodes until a value is computed for the root node. At MIN nodes, the backed up value is the minimum of the values associated with its children. At MAX nodes, the backed up value is the maximum of the values associated with its children.
5. Pick the operator associated with the child node whose backed up value determined the value at the root

Note: The above process of "backing up" values gives the optimal strategy that BOTH players would follow given that they both have the information computed at the leaf nodes by the evaluation function. This is implicitly assuming that your opponent is using the same static evaluation function you are, and that they are applying it at the same set of nodes in the search tree.

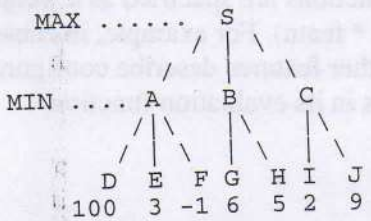
Minimax Algorithm in Java

```
public int minimax(s)
{
  int [] v = new int[#ofSuccessors];
  if (leaf(s))
       return(static-evaluation(s));
  else
  {
    // s1, s2, ..., sk are the successors of s
    for (int i = 1; i < #ofSuccessors; i++)
    {
      v[i] = minimax(si);
    }
    if (node-type(s) = max)
      return max(v1, ..., vk);
      else return min(v1, ..., vk);
   }
}
```

Example of Minimax Algorithm

For example, in a 2-ply search, the MAX player considers all (3) possible moves.

The opponent MIN also considers all possible moves. The evaluation function is applied to the leaf level only.



Once the static evaluation function is applied at the leaf nodes, backing up values can begin. First we compute the backed-up values at the parents of the leaves. Node A is a MIN node corresponding to the fact that it is a position where it's the opponent's turn to move. A's backed-up value is -1 (= min(100, 3, -1), meaning that if the opponent ever reaches the board associated with this node, then it will pick the move associated with the arc from A to F. Similarly, B's backed-up value is 5 (corresponding to child H) and C's backed-up value is 2 (corresponding to child I).

Next, we backup values to the next higher level, in this case to the MAX node S. Since it is our turn to move at this node, we select the move that looks best based on the backed-up values at

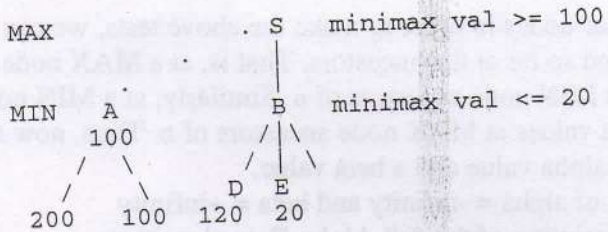
of S's children. In this case the best child is B since B's backed-up value is 5 (= max(-1, 5, So the minimax value for the root node S is 5, and the move selected based on this 2-ply arch is the move associated with the arc from S to B.

It is important to notice that the backed-up values are used at nodes A, B, and C to evaluate which is best for S; we do *not* apply the static evaluation function at any non-leaf node. Why? Because it is assumed that the values computed at nodes farther ahead in the game (and therefore lower in the tree) are more accurate evaluations of quality and therefore are preferred over the evaluation function values if applied at the higher levels of the tree.

Notice that, in general, the backed-up value of a node changes as we search more plies. For example, A's backed-up value is -1. But if we had searched one more ply, D, E and F will have their own backed-up values, which are almost certainly going to be different from 100, 3 and -1, respectively. And, in turn, A will likely not have -1 as its backed-up value. We are implicitly assuming that the deeper we search, the better the quality of the final outcome.

Alpha-Beta Pruning

- Minimax computes the optimal playing strategy but does so inefficiently because it first generates a complete tree and then computes and backs up static-evaluation-function values. For example, from an average chess position there are 38 possible moves. So, looking ahead 12 plies involves generating $1 + 38 + 38^2 + ... + 38^{12} = (38^{12}-1)/(38-1)$ nodes, and applying the static evaluation function at $38^{12} = 9$ billion billion positions, which is far beyond the capabilities of any computer in the foreseeable future. Can we devise another algorithm that is guaranteed to produce the same result (i.e., minimax value at the root) but does less work (i.e., generates fewer nodes)? Yes---Alpha-Beta.
- Basic idea: "If you have an idea that is surely bad, don't take the time to see how truly awful it is." -- Pat Winston
- Example of how to use this idea for pruning away useless work:

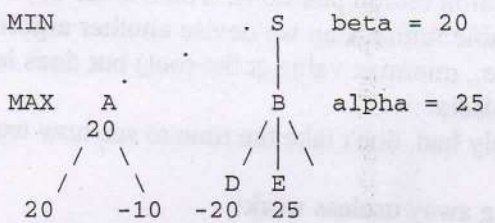


In the above example we are performing a depth-first search to depth (ply) 2, where children are generated and visited left-to-right. At this stage of the search we have just finished generating B's second child, E, and computed the static evaluation function at E (=20). Before generating B's third child notice the current situation: S is a MAX node and its left child A has a minimax value of 100, so S's minimax value **must** eventually be some number >= 100. Similarly, B has generated two children, D and E, with values 120 and 20, respectively, so B's final minimax value must be <= min(120, 20) = 20 since B is a MIN node.

The fact that S's minimax value must be at least 100 while B's minimax value must be no greater than 20 means that no matter what value is computed for B's third child, S's minimax value will be 100. In other words, S's minimax value does not depend on knowing the value of B's third child. Hence, we can cutoff the search below B, ignoring generating any other children after D and E.

Alpha-Beta Algorithm

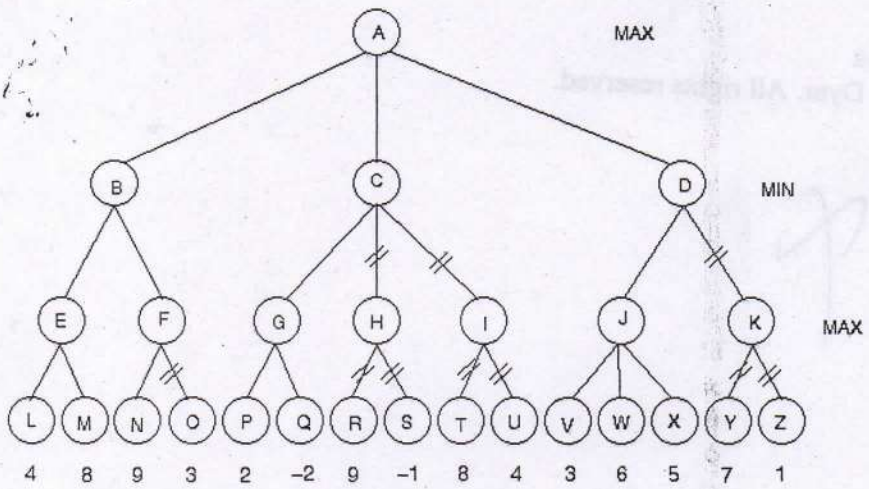
- Traverse the search tree in depth-first order
- Assuming we stop the search at ply *d*, then at each of these nodes we generate, we apply the static evaluation function and return this value to the node's parent
- At each non-leaf node, store a value indicating the best backed-up value found so far. At MAX nodes we'll call this **alpha**, and at MIN nodes we'll call the value **beta**. In other words, alpha = best (i.e., maximum) value found so far at a MAX node (based on its descendant's values). Beta = best (i.e., minimum) value found so far at a MIN node (based on its descendant's values).
- The alpha value (of a MAX node) is *monotonically non-decreasing*
- The beta value (of a MIN node) is *monotonically non-increasing*
- Given a node *n*, cutoff the search below *n* (i.e., don't generate any more of *n*'s children) if
  - *n* is a MAX node and alpha(n) >= beta(i) for some MIN node ancestor *i* of *n*. This is called a **beta cutoff**.
  - *n* is a MIN node and beta(n) <= alpha(i) for some MAX node ancestor *i* of *n*. This is called an **alpha cutoff**.
- In the example shown above an alpha cutoff occurs at node B because beta(B) = 20 < alpha (S) = 100
- An example of a beta cutoff at node B (because alpha(B) = 25 > beta(S) = 20) is shown below:



- To avoid searching for the ancestor nodes in order to make the above tests, we can carry *down* the tree the best values found so far at the ancestors. That is, at a MAX node *n*, beta = minimum of all the beta values at MIN node ancestors of *n*. Similarly, at a MIN node *n*, alpha = maximum of all the alpha values at MAX node ancestors of *n*. Thus, now at each non-leaf node we'll store both an alpha value and a beta value.
- Initially, assign to the root values of alpha = -infinity and beta = +infinity
- See the text for a pseudocode description of the full Alpha-Beta algorithm

Example of Alpha-Beta Algorithm on a 3-Ply Search Tree

Below is a search tree where a beta cutoff occurs at node F and alpha cutoffs occur at nodes C and D. In this case we've pruned 10 nodes (O,H,R,S,I,T,U,K,Y,Z) from the 26 that are generated by Minimax.



Effectiveness of Alpha-Beta

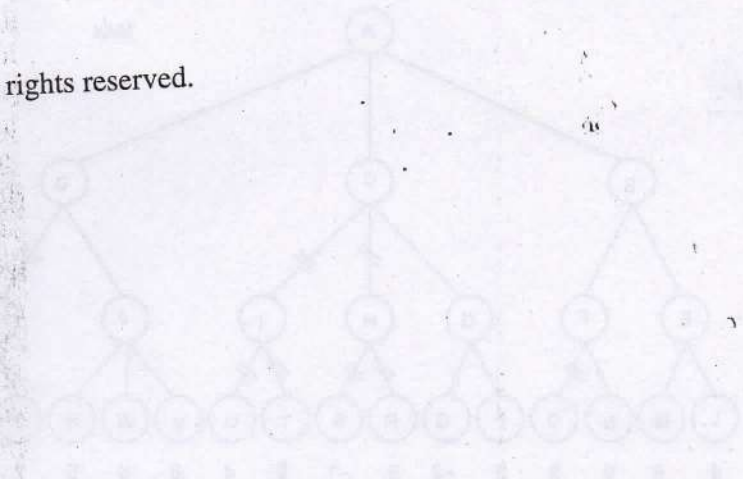
- Alpha-Beta is guaranteed to compute the same minimax value for the root node as computed by Minimax
- In the worst case Alpha-Beta does NO pruning, examining *b^d* leaf nodes, where each node has *b* children and a *d*-ply search is performed
- In the best case, Alpha-Beta will examine only *(2b)^(d/2)* leaf nodes. Hence if you hold fixed the number of leaf nodes (as a measure of the amount of time you have allotted before a decision must be made), then you can search twice as deep as Minimax!
- The best case occurs when each player's best move is the leftmost alternative (i.e., the first child generated). So, at MAX nodes the child with the largest value is generated first, and at MIN nodes the child with the smallest value is generated first.
- In the chess program Deep Blue, they found empirically that Alpha-Beta pruning meant that the average branching factor at each node was about 6 instead of about 35-40

Cutting off Search (or, when to stop and apply the evaluation function)

So far we have assumed a fixed depth *d* where the search is stopped and the static evaluation function is applied. But there are variations on this that are important to note:

- Don't stop at **non-quiescent nodes**. If a node represents a state in the middle of an exchange of pieces, then the node is not quiescent and therefore the evaluation function may not give a reliable estimate of board quality. Or, another definition for chess: "a state is non-quiescent if any piece is attacked by one of lower value, or by more pieces than defenses, or if any check exists on a square controlled by the opponent." In this case, expand more nodes and only apply the evaluation function at quiescent nodes.
- The identification of non-quiescent nodes partially deals with the **horizon effect**. A negative horizon is where the state seen by the evaluation function is evaluated as better than it really is because an undesirable effect is just beyond this node (i.e., the search horizon). A positive horizon is where the evaluation function wrongly underestimates the value of a state when positive actions just over the search horizon indicate otherwise.
- **Iterative Deepening** is frequently used with Alpha-Beta so that searches to successively deeper plies can be attempted if there is time, and the move selected is the one computed by the deepest search completed when the time limit is reached.

Last modified September 25, 1998



www-formal.stanford.edu/jmc/whatisai/whatisai.html

1-10

courtesy.

Next | Up | Previous

Next: Branches of AI **Up:** WHAT IS ARTIFICIAL **Previous:** WHAT IS ARTIFICIAL

Basic Questions

Q. What is artificial intelligence?

A. It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable.

Q. Yes, but what is intelligence?

A. Intelligence is the computational part of the ability to achieve goals in the world. Varying kinds and degrees of intelligence occur in people, many animals and some machines.

Q. Isn't there a solid definition of intelligence that doesn't depend on relating it to human intelligence?

A. Not yet. The problem is that we cannot yet characterize in general what kinds of computational procedures we want to call intelligent. We understand some of the mechanisms of intelligence and not others.

Q. Is intelligence a single thing so that one can ask a yes or no question ''Is this machine intelligent or not?''?

A. No. Intelligence involves mechanisms, and AI research has discovered how to make computers carry out some of them and not others. If doing a task requires only mechanisms that are well understood today, computer programs can give very impressive performances on these tasks. Such programs should be considered ''somewhat intelligent''.

Q. Isn't AI about simulating human intelligence?

A. Sometimes but not always or even usually. On the one hand, we can learn something about how to make machines solve problems by observing other people or just by observing our own methods. On the other hand, most work in AI involves studying the problems the world presents to intelligence rather than studying people or animals. AI researchers are free to use methods that are not observed in people or that involve much more computing than people can do.

Q. What about IQ? Do computer programs have IQs?

A. No. IQ is based on the rates at which intelligence develops in children. It is the ratio of the age at which a child normally makes a certain score to the child's age. The scale is extended to adults in a suitable way. IQ correlates well with various measures of success or failure in life, but making computers that can score high on IQ tests would be weakly correlated with their usefulness. For example, the ability of a child to repeat back a long sequence of digits correlates well with other intellectual abilities, perhaps because it measures how much information the child can compute with at once. However, ''digit span'' is trivial for even extremely limited computers.

However, some of the problems on IQ tests are useful challenges for AI.

Q. What about other comparisons between human and computer intelligence?

Arthur R. Jensen [Jen98], a leading researcher in human intelligence, suggests ''as a heuristic hypothesis'' that all normal humans have the same intellectual mechanisms and that differences in intelligence are related to ''quantitative biochemical and physiological conditions''. I see them as speed, short term memory, and the ability to form accurate and retrievable long term memories.

Whether or not Jensen is right about human intelligence, the situation in AI today is the reverse.

Computer programs have plenty of speed and memory but their abilities correspond to the intellectual mechanisms that program designers understand well enough to put in programs. Some abilities that children normally don't develop till they are teenagers may be in, and some abilities possessed by two year olds are still out. The matter is further complicated by the fact that the cognitive sciences still have not succeeded in determining exactly what the human abilities are. Very likely the organization of the intellectual mechanisms for AI can usefully be different from that in people.

Whenever people do better than computers on some task or computers use a lot of computation to do as well as people, this demonstrates that the program designers lack understanding of the intellectual mechanisms required to do the task efficiently.

Q. When did AI research start?

A. After WWII, a number of people independently started to work on intelligent machines. The English mathematician Alan Turing may have been the first. He gave a lecture on it in 1947. He also may have been the first to decide that AI was best researched by programming computers rather than by building machines. By the late 1950s, there were many researchers on AI, and most of them were basing their work on programming computers.

Q. Does AI aim to put the human mind into the computer?

A. Some researchers say they have that objective, but maybe they are using the phrase metaphorically. The human mind has a lot of peculiarities, and I'm not sure anyone is serious about imitating all of them.

Q. What is the Turing test?

A. Alan Turing's 1950 article *Computing Machinery and Intelligence* [Tur50] discussed conditions for considering a machine to be intelligent. He argued that if the machine could successfully pretend to be human to a knowledgeable observer then you certainly should consider it intelligent. This test would satisfy most people but not all philosophers. The observer could interact with the machine and a human by teletype (to avoid requiring that the machine imitate the appearance or voice of the person), and the human would try to persuade the observer that it was human and the machine would try to fool the observer.

The Turing test is a one-sided test. A machine that passes the test should certainly be considered intelligent, but a machine could still be considered intelligent without knowing enough about humans to imitate a human.

aniel Dennett's book *Brainchildren* [Den98] has an excellent discussion of the Turing test and the arious partial Turing tests that have been implemented, i.e. with restrictions on the observer's knowledge of AI and the subject matter of questioning. It turns out that some people are easily led into believing that a rather dumb program is intelligent.

Q. Does AI aim at human-level intelligence?

A. Yes. The ultimate effort is to make computer programs that can solve problems and achieve goals in the world as well as humans. However, many people involved in particular research areas are much less ambitious.

Q. How far is AI from reaching human-level intelligence? When will it happen?

A. A few people think that human-level intelligence can be achieved by writing large numbers of programs of the kind people are now writing and assembling vast knowledge basis of facts in the languages now used for expressing knowledge.

However, most AI researchers believe that new fundamental ideas are required, and therefore it cannot be predicted when human level intelligence will be achieved.

Q. Are computers the right kind of machine to be made intelligent?

A. Computers can be programmed to simulate any kind of machine.

Many researchers invented non-computer machines, hoping that they would be intelligent in different ways than the computer programs could be. However, they usually simulate their invented machines on a computer and come to doubt that the new machine is worth building. Because many billions of dollars that have been spent in making computers faster and faster, another kind of machine would have to be very fast to perform better than a program on a computer simulating the machine.

Q. Are computers fast enough to be intelligent?

A. Some people think much faster computers are required as well as new ideas. My own opinion is that the computers of 30 years ago were fast enough if only we knew how to program them. Of course, quite apart from the ambitions of AI researchers, computers will keep getting faster.

Q. What about parallel machines?

A. Machines with many processors are much faster than single processors can be. Parallelism itself presents no advantages, and parallel machines are somewhat awkward to program. When extreme speed is required, it is necessary to face this awkwardness.

Q. What about making a ''child machine'' that could improve by reading and by learning from experience?

A. This idea has been proposed many times, starting in the 1940s. Eventually, it will be made to work. However, AI programs haven't yet reached the level of being able to learn much of what a child learns from physical experience. Nor do present programs understand language well enough to learn much by reading.

Q. Might an AI system be able to bootstrap itself to higher and higher level intelligence by thinking about AI?

A. I think yes, but we aren't yet at a level of AI at which this process can begin.

Q. What about chess?

A. Alexander Kronrod, a Russian AI researcher, said "Chess is the *Drosophila* of AI." He was making an analogy with geneticists' use of that fruit fly to study inheritance. Playing chess requires certain intellectual mechanisms and not others. Chess programs now play at grandmaster level, but they do it with limited intellectual mechanisms compared to those used by a human chess player, substituting large amounts of computation for understanding. Once we understand these mechanisms better, we can build human-level chess programs that do far less computation than do present programs.

Unfortunately, the competitive and commercial aspects of making computers play chess have taken precedence over using chess as a scientific domain. It is as if the geneticists after 1910 had organized fruit fly races and concentrated their efforts on breeding fruit flies that could win these races.

Q. What about *Go*?

A. The Chinese and Japanese game of *Go* is also a board game in which the players take turns moving. *Go* exposes the weakness of our present understanding of the intellectual mechanisms involved in human game playing. *Go* programs are very bad players, in spite of considerable effort (not as much as for chess). The problem seems to be that a position in *Go* has to be divided mentally into a collection of subpositions which are first analyzed separately followed by an analysis of their interaction. Humans use this in chess also, but chess programs consider the position as a whole. Chess programs compensate for the lack of this intellectual mechanism by doing thousands or, in the case of Deep Blue, many millions of times as much computation.

Sooner or later, AI research will overcome this scandalous weakness.

Q. Don't some people say that AI is a bad idea?

A. The philosopher John Searle says that the idea of a non-biological machine being intelligent is incoherent. The philosopher Hubert Dreyfus says that AI is impossible. The computer scientist Joseph Weizenbaum says the idea is obscene, anti-human and immoral. Various people have said that since artificial intelligence hasn't reached human level by now, it must be impossible. Still other people are disappointed that companies they invested in went bankrupt.

Q. Aren't computability theory and computational complexity the keys to AI? [Note to the layman and beginners in computer science: These are quite technical branches of mathematical logic and computer science, and the answer to the question has to be somewhat technical.]

A. No. These theories are relevant but don't address the fundamental problems of AI.

In the 1930s mathematical logicians, especially Kurt Gödel and Alan Turing, established that there did not exist algorithms that were guaranteed to solve all problems in certain important mathematical domains. Whether a sentence of first order logic is a theorem is one example, and whether a polynomial equations in several variables has integer solutions is another. Humans solve

blems in these domains all the time, and this has been offered as an argument (usually with me decorations) that computers are intrinsically incapable of doing what people do. However, people can't guarantee to solve *arbitrary* problems in these domains either.

In the 1960s computer scientists, especially Steve Cook and Richard Karp developed the theory of NP-complete problem domains. Problems in these domains are solvable, but seem to take time exponential in the size of the problem. Which sentences of propositional calculus are satisfiable is a basic example of an NP-complete problem domain. Humans often solve problems in NP-complete domains in times much shorter than is guaranteed by the general algorithms, but can't solve them quickly in general.

What is important for AI is to have algorithms as capable as people at solving problems. The identification of subdomains for which good algorithms exist is important, but a lot of AI problem solvers are not associated with readily identified subdomains.

The theory of the difficulty of general classes of problems is called *computational complexity*. So far this theory hasn't interacted with AI as much as might have been hoped. Success in problem solving by humans and by AI programs seems to rely on properties of problems and problem solving methods that the neither the complexity researchers nor the AI community have been able to identify precisely.

Algorithmic complexity theory as developed by Solomonoff, Kolmogorov and Chaitin (independently of one another) is also relevant. It defines the complexity of a symbolic object as the length of the shortest program that will generate it. Proving that a candidate program is the shortest or close to the shortest is an unsolvable problem, but representing objects by short programs that generate them should be often illuminating even when you can't prove that the program is the shortest.

Next Up Previous

Next: Branches of AI **Up:** WHAT IS ARTIFICIAL **Previous:** WHAT IS ARTIFICIAL

John McCarthy
Tue Apr 4 16:09:39 PDT 2000

Branches of AI

Q. What are the branches of AI?

A. Here's a list, but some branches are surely missing, because no-one has identified them yet. Some of these may be regarded as concepts or topics rather than full branches.

logical AI

What a program knows about the world in general the facts of the specific situation in which it must act, and its goals are all represented by sentences of some mathematical logical language. The program decides what to do by inferring that certain actions are appropriate for achieving its goals. The first article proposing this was [McC59]. [McC89] is a more recent summary. [McC96] lists some of the concepts involved in logical aI. [Sha97] is an important text.

search

AI programs often examine large numbers of possibilities, e.g. moves in a chess game or inferences by a theorem proving program. Discoveries are continually made about how to do this more efficiently in various domains.

pattern recognition

When a program makes observations of some kind, it is often programmed to compare what it sees with a pattern. For example, a vision program may try to match a pattern of eyes and a nose in a scene in order to find a face. More complex patterns, e.g. in a natural language text, in a chess position, or in the history of some event are also studied. These more complex patterns require quite different methods than do the simple patterns that have been studied the most.

representation

Facts about the world have to be represented in some way. Usually languages of mathematical logic are used.

inference

From some facts, others can be inferred. Mathematical logical deduction is adequate for some purposes, but new methods of *non-monotonic* inference have been added to logic since the 1970s. The simplest kind of non-monotonic reasoning is default reasoning in which a conclusion is to be inferred by default, but the conclusion can be withdrawn if there is evidence to the contrary. For example, when we hear of a bird, we man infer that it can fly, but this conclusion can be reversed when we hear that it is a penguin. It is the possibility that a conclusion may have to be withdrawn that constitutes the non-monotonic character of the reasoning. Ordinary logical reasoning is monotonic in that the set of conclusions that can the drawn from a set of premises is a monotonic increasing function of the premises

common sense knowledge and reasoning

This is the area in which AI is farthest from human-level, in spite of the fact that it has been an active research area since the 1950s. While there has been considerable progress, e.g. in developing systems of *non-monotonic reasoning* and theories of action, yet more new ideas are needed. The Cyc system contains a large but spotty collection of common sense facts.

learning from experience

Programs do that. The approaches to AI based on *connectionism* and *neural nets* specialize in that. There is also learning of laws expressed in logic. [Mit97] is a comprehensive undergraduate text on machine learning. Programs can only learn what facts or behaviors their formalisms can represent, and unfortunately learning systems are almost all based on very limited abilities to represent information.

planning

Planning programs start with general facts about the world (especially facts about the effects of actions), facts about the particular situation and a statement of a goal. From these, they generate a strategy for achieving the goal. In the most common cases, the strategy is just a sequence of actions.

epistemology

This is a study of the kinds of knowledge that are required for solving problems in the world.

ontology

Ontology is the study of the kinds of things that exist. In AI, the programs and sentences deal with various kinds of objects, and we study what these kinds are and what their basic properties are. Emphasis on ontology begins in the 1990s.

heuristics

A heuristic is a way of trying to discover something or an idea imbedded in a program. The term is used variously in AI. *Heuristic functions* are used in some approaches to search to measure how far a node in a search tree seems to be from a goal. *Heuristic predicates* that compare two nodes in a search tree to see if one is better than the other, i.e. constitutes an advance toward the goal, may be more useful. [My opinion].

genetic programming

Genetic programming is a technique for getting programs to solve a task by mating random Lisp programs and selecting fittest in millions of generations. It is being developed by John Koza's group and here's a tutorial.

Next Up Previous

Next: Applications of AI **Up:** WHAT IS ARTIFICIAL **Previous:** Basic Questions

John McCarthy
Tue Apr 4 16:09:39 PDT 2000

Applications of AI

Q. What are the applications of AI?

A. Here are some.

game playing

You can buy machines that can play master level chess for a few hundred dollars. There is some AI in them, but they play well against people mainly through brute force computation--looking at hundreds of thousands of positions. To beat a world champion by brute force and known reliable heuristics requires being able to look at 200 million positions per second.

speech recognition

In the 1990s, computer speech recognition reached a practical level for limited purposes. Thus United Airlines has replaced its keyboard tree for flight information by a system using speech recognition of flight numbers and city names. It is quite convenient. On the the other hand, while it is possible to instruct some computers using speech, most users have gone back to the keyboard and the mouse as still more convenient.

understanding natural language

Just getting a sequence of words into a computer is not enough. Parsing sentences is not enough either. The computer has to be provided with an understanding of the domain the text is about, and this is presently possible only for very limited domains.

computer vision

The world is composed of three-dimensional objects, but the inputs to the human eye and computers' TV cameras are two dimensional. Some useful programs can work solely in two dimensions, but full computer vision requires partial three-dimensional information that is not just a set of two-dimensional views. At present there are only limited ways of representing three-dimensional information directly, and they are not as good as what humans evidently use.

expert systems

A ''knowledge engineer'' interviews experts in a certain domain and tries to embody their knowledge in a computer program for carrying out some task. How well this works depends on whether the intellectual mechanisms required for the task are within the present state of AI. When this turned out not to be so, there were many disappointing results. One of the first expert systems was MYCIN in 1974, which diagnosed bacterial infections of the blood and suggested treatments. It did better than medical students or practicing doctors, provided its limitations were observed. Namely, its ontology included bacteria, symptoms, and treatments and did not include patients, doctors, hospitals, death, recovery, and events occurring in time. Its interactions depended on a single patient being considered. Since the experts consulted by the knowledge engineers knew about patients, doctors, death, recovery, etc., it is clear that the

knowledge engineers forced what the experts told them into a predetermined framework. In the present state of AI, this has to be true. The usefulness of current expert systems depends on their users having common sense.

Heuristic classification

One of the most feasible kinds of expert system given the present knowledge of AI is to put some information in one of a fixed set of categories using several sources of information. An example is advising whether to accept a proposed credit card purchase. Information is available about the owner of the credit card, his record of payment and also about the item he is buying and about the establishment from which he is buying it (e.g., about whether there have been previous credit card frauds at this establishment).

Next Up Previous

Next: More questions **Up:** WHAT IS ARTIFICIAL **Previous:** Branches of AI

John McCarthy
Tue Apr 4 16:09:39 PDT 2000

Next Up Previous

Next: References **Up:** WHAT IS ARTIFICIAL **Previous:** Applications of AI

More questions

Q. How is AI research done?

A. AI research has both theoretical and experimental sides. The experimental side has both basic and applied aspects.

There are two main lines of research. One is biological, based on the idea that since humans are intelligent, AI should study humans and imitate their psychology or physiology. The other is phenomenal, based on studying and formalizing common sense facts about the world and the problems that the world presents to the achievement of goals. The two approaches interact to some extent, and both should eventually succeed. It is a race, but both racers seem to be walking.

Q. What should I study before or while learning AI?

A. Study mathematics, especially mathematical logic. The more you learn about science in general the better. For the biological approaches to AI, study psychology and the physiology of the nervous system. Learn some programming languages--at least C, Lisp and Prolog. It is also a good idea to learn one basic machine language. Jobs are likely to depend on knowing the languages currently in fashion. In the late 1990s, these include C++ and Java.

Q. What is a good textbook on AI?

A. *Artificial Intelligence* by Stuart Russell and Peter Norvig, Prentice Hall is the most commonly used textbbook in 1997. The general views expressed there do not exactly correspond to those of this essay. *Artificial Intelligence: A New Synthesis* by Nils Nilsson, Morgan Kaufman, may be easier to read.

Q. What organizations and publications are concerned with AI?

A. The American Association for Artificial Intelligence (AAAI), the European Coordinating Committee for Artificial Intelligence (ECCAI) and the Society for Artificial Intelligence and Simulation of Behavior (AISB) are scientific societies concerned with AI research. The Association for Computing Machinery (ACM) has a special interest group on artificial intelligence SIGART.

The International Joint Conference on AI (IJCAI) is the main international conference. The AAAI runs a US National Conference on AI. *Electronic Transactions on Artificial Intelligence*, *Artificial Intelligence*, and *Journal of Artificial Intelligence Research*, and IEEE Transactions on Pattern Analysis and Machine Intelligence are four of the main journals publishing AI research papers. I have not yet found everything that should be in this paragraph.

Page of Positive Reviews lists papers that experts have found important.

Funding a Revolution: Government Support for Computing Research by a committee of the National Research covers support for AI research in Chapter 9.