Artificial Intelligence
For Hedspi Project
Lecture 1 – Course presentation

Lecturer:
Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
General information

- Course name
  - Artificial Intelligence

- Volume
  - 15 x 2h

- Lecturer
  - Trần Đức Khánh
  - Department of Information Systems, SOICT, HUT
  - khanhtd@soict.hut.edu.vn
Goals

- Total image of AI
- Basic concepts of AI
- Major techniques of AI
- Important applications of AI
Content

- Introduction
- Agents
- Problem Solving
  - Search Algorithms
  - Constraint Satisfaction Problems
- Logic and Inference
- Planning
- Machine Learning
Reference

- Book
Evaluation

- Final examination
  - 70%
  - Achievement test

- Continuous assessment
  - 30%
  - Class attendance, midterm test, short tests or quizzes, project
Some educational recommendations

- Attend classes
- Turn off your cell phone
- Read the reference book
- Do not hesitate to
  - Ask questions
  - Give your opinions/feedbacks
  - Discuss with your lecturer
Artificial Intelligence

For HEDSPI Project

Lecture 1 - Introduction

Lecturer:

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Outline

- What is AI?
- Foundations of AI
- Short history of AI
- Philosophical discussions
What is AI?

Views of AI fall into four categories:

<table>
<thead>
<tr>
<th>Think like humans</th>
<th>Thinking rationally</th>
</tr>
</thead>
<tbody>
<tr>
<td>Act like humans</td>
<td>Acting rationally</td>
</tr>
</tbody>
</table>

The textbook advocates "acting rationally"
Think like humans

- 1960s "cognitive revolution": information-processing psychology
- Scientific theories of internal activities of the brain
  - What level of abstraction? “Knowledge" or “circuits”?
  - **Cognitive science**: Predicting and testing behavior of human subjects (top-down)
  - **Cognitive neuroscience**: Direct identification from neurological data (bottom-up)
- Both approaches now distinct from AI
- Both share with AI the following characteristic:
  - The available theories do not explain (or engender) anything resembling human-level general intelligence}
Act like humans

- Turing (1950) "Computing machinery and intelligence": "Can machines think?" → "Can machines behave intelligently?"
- Operational test for intelligent behavior: the Imitation Game

- Predicted that by 2000, a machine might have a 30% chance of fooling a lay person for 5 minutes
- Anticipated all major arguments against AI in following 50 years
- Suggested major components of AI: knowledge, reasoning, language understanding, learning
Thinking rationally

- The “Laws of Thought” approach
  - What does it mean to “think rationally”?  
  - Normative / prescriptive rather than descriptive
- Logicist tradition:
  - Logic: notation and rules of derivation for thoughts
  - Aristotle: what are correct arguments/thought processes?
  - Direct line through mathematics, philosophy, to modern AI
- Problems:
  - Not all intelligent behavior is mediated by logical deliberation
  - What is the purpose of thinking? What thoughts should I have?
  - Logical systems tend to do the wrong thing in the presence of uncertainty
Acting rationally

- Rational behavior: doing the “right thing”
  - The right thing: that which is expected to maximize goal achievement, given the available information
  - Doesn't necessarily involve thinking, e.g., blinking
  - Thinking can be in the service of rational action
  - Entirely dependent on goals!
  - Irrational ≠ insane, irrationality is sub-optimal action
  - Rational ≠ successful
- Our focus here: rational agents
  - Systems which make the best possible decisions given goals, evidences, and constraints
  - In the real world, usually lots of uncertainty… and lots of complexity
  - Usually, we’re just approximating rationality
- “Computational rationality” a better title for this course
Rational agents

- An **agent** is an entity that perceives and acts.
- An agent function maps from percept histories to actions:
  \[ P^* \rightarrow A \]

- For any given class of environments and tasks, we seek the agent (or class of agents) with the best performance.
- Computational limitations make perfect rationality unachievable.
- So we want the best program for given machine resources.
Foundations of AI

- **Philosophy** logic, methods of reasoning, mind as physical system foundations of learning, language, rationality
- **Mathematics** formal representation and proof algorithms, computation, (un)decidability, (in)tractability, probability
- **Economics** utility, decision theory
- **Neuroscience** physical substrate for mental activity
- **Psychology** phenomena of perception and motor control, experimental techniques
- **Computer engineering** building fast computers
- **Control theory** design systems that maximize an objective function over time
- **Linguistics** knowledge representation, grammar
Short history of AI

- 1940-1950: Early days
  - 1943: McCulloch & Pitts: Boolean circuit model of brain
  - 1950: Turing's "Computing Machinery and Intelligence"
- 1950—70: Excitement: Look, Ma, no hands!
  - 1950s: Early AI programs, including Samuel's checkers program, Newell & Simon's Logic Theorist, Gelernter's Geometry Engine
  - 1956: Dartmouth meeting: "Artificial Intelligence" adopted
  - 1964: ELIZA
  - 1965: Robinson's complete algorithm for logical reasoning
- 1970—88: Knowledge-based approaches
  - 1969—79: Early development of knowledge-based systems
  - 1980—88: Expert systems industry booms
- 1988—: Statistical approaches
  - Resurgence of probability, focus on uncertainty
  - General increase in technical depth
  - Agents, agents, everywhere… “AI Spring”?
- 2000—: Where are we now?
Expert system

Expert system = Human Expertise + Inference/Reasoning

Some examples: DENDRAL, MYCIN, PROSPECTOR, MOLGEN, ICAD/ICAM
State of the art

- May, '97: Deep Blue vs. Kasparov
  - First match won against world-champion
  - “Intelligent creative” play
  - 200 million board positions per second!
  - Humans understood 99.9 of Deep Blue's moves
  - Can do about the same now with a big PC cluster
- Proved a mathematical conjecture (Robbins conjecture) unsolved for decades
- No hands across America (driving autonomously 98% of the time from Pittsburgh to San Diego)
- During the 1991 Gulf War, US forces deployed an AI logistics planning and scheduling program that involved up to 50,000 vehicles, cargo, and people
- NASA's on-board autonomous planning program controlled the scheduling of operations for a spacecraft
- Proverb solves crossword puzzles better than most humans
Philosophical discussions

What Can AI Do?

- Play a decent game of table tennis?
- Drive safely along a curving mountain road?
- Buy a week's worth of groceries on the web?
- Discover and prove a new mathematical theorem?
- Converse successfully with another person for an hour?
- Perform a complex surgical operation?
- Unload a dishwasher and put everything away?
- Translate spoken English into spoken Vietnamese in real time?
- Write an intentionally funny story?

Can machine think?
Some problems with AI

- People might lose their jobs to automation.
- People might have too much (or too little) leisure time.
- People might lose their sense of being unique.
- People might lose some of their privacy rights.
- The use of AI systems might result in a loss of accountability.
- The success of AI might mean the end of the human race.
Artificial Intelligence

For HEDSPI Project

Lecture 2 - Agent

Lecturer:
Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Outline

1. Agents and environments
2. PEAS (Performance measure, Environment, Actuators, Sensors)
3. Environment types
4. Agent types
Agents and environments

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

- Example 1: human agent
  - Sensors: eyes, ears, …
  - Actuators: hands, legs, mouth, …

- Example 2: robotic agent (e.g., Aishimo)
  - Sensors: camera, infrared range finders
  - Actuators: various motors
Agents and environments (con’t)

• The agent function maps from percept histories to actions:
  \[ f: P^* \rightarrow A \]

The agent program runs on the physical architecture to produce the agent function
agent = architecture + program
**Agent function based on conditional table**

**Function** TABLE-DRIVEN-AGENT(percept) **returns** an action

**static:**  
- **percepts**, a sequence, initially empty
- **table**, a table of actions, indexed by percept sequences, initially fully specified

Append **percept** to the end of **percepts**

**action** ← LOOKUP(**percepts**, **table**)

**Return** **action**

**Drawback:** huge table!
Vacuum-cleaner world

- Percepts: location (A or B), state (clean or dirty)
- Actions: Left, Right, Suck, NoOp

<table>
<thead>
<tr>
<th>Percept sequence</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>[A, clean]</td>
<td>Right</td>
</tr>
<tr>
<td>[A, dirty]</td>
<td>Suck</td>
</tr>
<tr>
<td>[B, clean]</td>
<td>Left</td>
</tr>
<tr>
<td>[B, dirty]</td>
<td>Suck</td>
</tr>
<tr>
<td>[A, clean][A, clean]</td>
<td>Right</td>
</tr>
<tr>
<td>[A, clean][A, dirty]</td>
<td>Suck</td>
</tr>
</tbody>
</table>
Vacuum-cleaner world

Function Reflex-Vacuum-Agent([position, state]) returns action
   If state = Dirty then return Suck
   Else if position = A then return Right
   Else if position = B then return Left
End Function

- Does the agent act reasonably?
A rational agent is one that does the right thing - the one that will cause the agent to be most successful.

Performance measure embodies the criterion for success of an agent's behavior.

- E.g., performance measure of a vacuum-cleaner agent:
  - amount of dirt cleaned up
  - amount of time taken
  - amount of electricity consumed
  - amount of noise generated
  - ...
Rational agent

For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

An agent is autonomous if its behavior is determined by its own experience (with ability to learn and adapt)
4 factors should be considered when designing an automated agent:

- Performance measure
- Environment
- Actuators
- Sensors
PEAS - automated taxi driver

- **Performance measure**: Safe, fast, legal, comfortable trip, maximize profits, …
- **Environment**: Roads, other traffic, pedestrians, weather, …
- **Actuators**: Steering wheel, accelerator, brake, signal, horn, …
- **Sensors**: Cameras, sonar, speedometer, GPS, odometer, engine sensors, keyboard, …
PEAS - Medical diagnosis system

- **Performance measure**: Healthy patient, minimize costs, lawsuits, …

- **Environment**: Patient, hospital, staff

- **Actuators**: Screen display (questions, tests, diagnoses, treatments, referrals)

- **Sensors**: Keyboard (entry of symptoms, findings, patient's answers)
PEAS - Spam Filtering Agent

- **Performance measure:** spam block, false positives, false negatives
- **Environment:** email client or server
- **Actuators:** mark as spam, transfer messages
- **Sensors:** emails (possibly across users), traffic, etc.
Environment types

- **Fully observable** (vs. partially observable): An agent's sensors give it access to the complete state of the environment at each point in time.

- **Deterministic** (vs. stochastic): The next state of the environment is completely determined by the current state and the action executed by the agent.

- **Episodic** (vs. sequential): The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action).
Environment types

- **Static** (vs. dynamic): The environment is unchanged while an agent is deliberating.

- **Discrete** (vs. continuous): A limited number of distinct, clearly defined percepts and actions.

- **Single agent** (vs. multiagent): An agent operating by itself in an environment.
Agent types

- Four basic agent types:
  - Simple reflex agents
  - Model-based reflex agents
  - Goal-based agents
  - Utility-based agents
Simple reflex agent

- These agents select actions on the basis of the current percept, ignoring the rest of the percept history.

**Function** SIMPLE-REFLEX-AGENT(percept) **returns** an action

```plaintext
static: rules, a set of condition-action rules
state ← INTERPRET-INPUT(percept)
rule ← RULE-MATCH(state, rules)
action ← RULE-ACTION[rule]
return action
```
Model-based reflex agents

- These agents maintain **internal states** that depend on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

```
function REFLEX-AGENT-WITH-STATE(percept) returns an action
static: state, a description of the current world state
        rules, a set of condition-action rules
        action, the most recent action, initially none
state ← UPDATE-STATE(state, action, percept)
rule ← RULE-MATCH(state, rules)
action ← RULE-ACTION[rule]
return action
```
Goal-based agents

- Agents that take actions in the pursuit of a goal or goals.

- Goals introduce the need to reason about the future or other hypothetical states. It may be the case that none of the actions an agent can currently perform will lead to a goal state.
Utility-based agents

Agents that take actions that make them the most happy in the long run.

More formally agents that prefer actions that lead to states with higher utility.

Utility-based agents can reason about multiple goals, conflicting goals, and uncertain situations.
Learning allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow.

The most important question: “What kind of performance element will my agent need to do this once it has learned how?”
Knowledge bases

- Knowledge base is a set of sentences in a formal language, telling an agent what it needs to know.

- Agent can ASK itself what to do, the answer should follow from the KB.

- Agents can be viewed at:
  - the knowledge level: what they know, what its goals are
  - the implementation level: data structures in KB and algorithms that manipulate them

- The agent must be able to:
  - Incorporate new percepts
  - Update internal representations of the world
  - Deduce hidden properties of the world
  - Deduce appropriate actions
Knowledge-based agents

function KB-AGENT(percept) returns an action
    static: KB, a knowledge base
        t, a counter, initially 0, indicating time
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept,t))
    action ← ASK(KB, MAKE-ACTION- QUERY(^))
    TELL(KB, MAKE-ACTION-SENTENCE(action,t) )
    t ← t+1
    return action
Multi-agent planning

- Environment: **cooperative** or **competitive**
- Issue: the environment is not **static → synchronization**
- Require a model of the other agent's plans

- **Cooperation:** joint goals and plans, e.g., team planning in doubles tennis.
  - Joint goal: returning the ball that has been hit to them and ensuring that at least one of them is covering the net
  - Joint plan: multibody planning
  - Coordination mechanisms: decompose and distribute tasks

- **Competition:** e.g., chess-playing
  - An agent in a competitive environment must
    - recognize that there are other agents
    - compute some of the other agent's possible plans
    - compute how the other agent's plans interact with its own plans
    - decide on the best action in view of these interactions.
Artificial Intelligence

For HEDSPI Project

Lecturer 3 - Search

Lecturer:

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
  - breadth-first search
  - depth-first search
  - depth-limited search
  - iterative deepening depth-first search
Problem-solving agents

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action

static: seq, an action sequence, initially empty

state, some description of the current world state

goal, a goal, initially null

problem, a problem formulation

state ← UPDATE-STATE(state, percept)

if seq is empty then do

goal ← FORMULATE-GOAL(state)

problem ← FORMULATE-PROBLEM(state, goal)

seq ← SEARCH(problem)

action ← FIRST(seq)

seq ← REST(seq)

return action
Example: Route Planning

- Performance: Get from Arad to Bucharest as quickly as possible
- Environment: The map, with cities, roads, and guaranteed travel times
- Actions: Travel a road between adjacent cities
Problem types

- **Deterministic, fully observable** → single-state problem
  - Agent knows exactly which state it will be in; solution is a sequence

- **Non-observable** → sensorless problem (conformant problem)
  - Agent may have no idea where it is; solution is a sequence

- **Nondeterministic and/or partially observable** → contingency problem
  - Percepts provide new information about current state
  - Often interleave → search, execution

- **Unknown state space** → exploration problem
A problem is defined by four items:
1. **initial state**: e.g., Arad
2. **actions or successor function** \( S(x) = \text{set of action-state pairs} \)
   - e.g., \( S(\text{Arad}) = \{<\text{Arad} \rightarrow \text{Zerind}, \text{Zerind}>, \ldots \} \)
3. **goal test**, can be
   - **explicit**, e.g., \( x = \text{Bucharest} \)
   - **implicit**, e.g., \( \text{Checkmate}(x) \)
4. **path cost** (additive)
   - e.g., sum of distances, number of actions executed, etc.
   - \( c(x,a,y) \) is the **step cost**, assumed to be \( \geq 0 \)

- **A solution** is a sequence of actions leading from the initial state to a goal state
Example: The 8-puzzle

- **states?** locations of tiles
- **actions?** move blank left, right, up, down
- **goal test?** = goal state (given)
- **path cost?** 1 per move
Search trees:
- Represent the branching paths through a state graph.
- Usually **much** larger than the state graph.
- Can a finite state graph give an infinite search tree?
Search space of the game Tic-Tac-Toe
Tree and graph

B is parent of C
C is child of B
A is ancestor of C
C is descendant of A
We can turn graph search problems into tree search problems by:

- replacing undirected links by 2 directed links
- avoiding loops in path (or keeping track of visited nodes globally)
Tree search algorithms

- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states

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

function TREE-SEARCH( problem, fringe) returns a solution, or failure
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node ← REMOVE-FRONT(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
  end loop

function EXPAND( node, problem) returns a set of nodes
  successors ← the empty set
  for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
    s ← a new NODE
    PARENT-NODE[s] ← node, ACTION[s] ← action, STATE[s] ← result
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  end for
  return successors
Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost \( g(x) \), depth

The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.
Search strategies

- A search strategy is defined by picking the order of node expansion.

- Strategies are evaluated along the following dimensions:
  - **completeness**: does it always find a solution if one exists?
  - **time complexity**: number of nodes generated
  - **space complexity**: maximum number of nodes in memory
  - **optimality**: does it always find a least-cost solution?

- Time and space complexity are measured in terms of
  - **$b$**: maximum branching factor of the search tree
  - **$d$**: depth of the least-cost solution
  - **$m$**: maximum depth of the state space (may be $\infty$)
Uninformed search strategies

- **Uninformed** search strategies use only the information available in the problem definition.

- **Breadth-first search**
  - Expand shallowest unexpanded node
  - fringe = queue (FIFO)

- **Uniform-cost search**
  - Expand cheapest unexpanded node
  - fringe = queue ordered by path cost

- **Depth-first search**
  - Expand deepest unexpanded node
  - fringe = stack (LIFO)

- **Depth-limited search**: depth-first search with depth limit

- **Iterative deepening search**
Breadth-first search

- Expand shallowest unexpanded node
Breadth-first search (con’t)

- **Complete?** Yes (if $b$ is finite)
- **Time?** $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
- **Space?** $O(b^{d+1})$ (keeps every node in memory)
- **Optimal?** Yes (if cost = 1 per step)
Uniform-cost search

- Expand cheapest unexpanded node \( n \) (cost so far to reach \( n \))
- \( fringe = \) queue ordered by path cost
- Equivalent to breadth-first if step costs all equal

- **Complete?** Yes, if step cost \( \geq \varepsilon \)
- **Time?** \# of nodes with \( g \leq \) cost of optimal solution, \( O(b^{\text{ceiling}(C^*/\varepsilon)}) \) where \( C^* \) is the cost of the optimal solution

- **Space?** \# of nodes with \( g \leq \) cost of optimal solution, \( O(b^{\text{ceiling}(C^*/\varepsilon)}) \)
- **Optimal?** Yes – nodes expanded in increasing order of \( g(n) \)
A priority queue is a data structure in which you can insert and retrieve (key, value) pairs with the following operations:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pq.setPriority(key, value)</td>
<td>inserts <em>(key, value)</em> into the queue.</td>
</tr>
<tr>
<td>pq.dequeue()</td>
<td>returns the key with the lowest value and removes it</td>
</tr>
</tbody>
</table>

- You can promote or demote keys by resetting their priorities.
- Unlike a regular queue, insertions into a priority queue are not constant time, usually $O(\log n)$.
- We’ll need priority queues for most cost-sensitive search methods.
Depth-first search

- Expand deepest unexpanded node
Depth-first search (con’t)

- **Complete?** No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path → complete in finite spaces
- **Time?** $O(b^m)$: terrible if $m$ is much larger than $d$
  - but if solutions are dense, may be much faster than breadth-first
- **Space?** $O(bm)$, i.e., linear space!
- **Optimal?** No
Depth-limited search

- Depth-first search can get stuck on infinite path when a different choice would lead to a solution
  ⇒ Depth-limited search = depth-first search with depth limit $l$, i.e., nodes at depth $l$ have no successors

```python
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    Recursive-DLS(Make-Node(INITIAL-STATE[problem]), problem, limit)

function Recursive-DLS(node, problem, limit) returns soln/fail/cutoff
cutoff-occurred? ← false
if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
else if DEPTH[node] = limit then return cutoff
else for each successor in EXPAND(node, problem) do
    result ← Recursive-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
if cutoff-occurred? then return cutoff else return failure
```
8-puzzle game with depth limit $l = 5$
Iterative deepening search

Problem with depth-limited search: if the shallowest goal is beyond the depth limit, no solution is found.

⇒ Iterative deepening search:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.
4. ....and so on.

```python
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
  inputs: problem, a problem
  for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
```
Iterative deepening search (con’t)

Limit = 0

Limit = 1

Limit = 2
Iterative deepening search (con’t)

Limit = 3
Iterative deepening search (con’t)

- Number of nodes generated in a depth-limited search to depth $d$ with branching factor $b$:
  \[ N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d \]

- Number of nodes generated in an iterative deepening search to depth $d$ with branching factor $b$:
  \[ N_{IDS} = (d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d \]

- For $b = 10$, $d = 5$,
  - $N_{DLS} = 1 + 10 + 100 + 1,000 + 10,000 + 100,000 = 111,111$
  - $N_{IDS} = 6 + 50 + 400 + 3,000 + 20,000 + 100,000 = 123,456$

- Overhead = \( (123,456 - 111,111)/111,111 = 11\% \)
Properties of iterative deepening search

- **Complete?** Yes
- **Time?** \((d+1)b^0 + d b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)\)
- **Space?** \(O(bd)\)
- **Optimal?** Yes, if step cost = 1
## Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^*/\epsilon})$</td>
<td>$O(b^m)$</td>
<td>$O(b^l)$</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^{d+1})$</td>
<td>$O(b^{C^*/\epsilon})$</td>
<td>$O(bm)$</td>
<td>$O(bl)$</td>
<td>$O(bd)$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Artificial Intelligence

For HEDSPI Project

Lecturer 4 - Search

Lecturer:

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Outline

- Graph search
- Best-first search
- A* search
Graph search

Get from Arad to Bucharest as quickly as possible
Graph search

- Failure to detect repeated states can turn a linear problem into an exponential one!

- Very simple fix: never expand a node twice
Graph search

function Graph-Search(problem, fringe) returns a solution, or failure
fringe ← Insert(Make-Node(Initial-State(problem)), fringe);
closed ← an empty set
while (fringe not empty)
    node ← RemoveFirst(fringe);
    if (Goal-Test(problem, State(node))) then return Solution(node);
    if (State(node) is not in closed then
        add State(node) to closed
        fringe ← InsertAll(Expand(node, problem), fringe);
    end if
end if
return failure;

Never expand a node twice!
Review uniform-cost search

- Task: Find the shortest/cheapest path from the start node to the goal.
- Pick cheapest unexpanded node (node with minimum cost from the start node to it)

- Problem
  - Uniform-cost search concerns only with expanding short paths; it pays no attention to the goal
Straight Line Distances
Best-first search

- Idea: use an evaluation function $f(n)$ for each node
  - estimate of "desirability"
  - Expand most desirable unexpanded node

- Order the nodes in fringe in decreasing order of desirability

- Special cases:
  - greedy best-first search
  - $A^*$ search
Greedy Best-First Search

- Evaluation function $f(n) = h(n)$ (heuristic)
  = estimate of cost from $n$ to goal
- e.g., $h_{SLD}(n) = \text{straight-line distance from } n \text{ to Bucharest}$
- Greedy best-first search expands the node that appears to be closest to goal
Greedy best-first search example

Arad

366
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Greedy Best-First Search

- **Complete?** No – can get stuck in loops, e.g., Iasi → Neamt → Iasi → Neamt → ...
- **Time?** $O(b^m)$, but a good heuristic can give dramatic improvement
- **Space?** $O(b^m)$ -- keeps all nodes in memory
- **Optimal?** No

- What do we need to do to make it complete?
  - $\Rightarrow$ A* search
- Can we make it optimal? $\Rightarrow$ No
A* search

- Idea: Expand unexpanded node with lowest evaluation value

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

- Nodes are ordered according to $f(n)$. 
A* search example

Arad

366 = 0 + 366
A* search example

![A* Search Example Diagram]

- **Arad**
  - **Sibiu**: $393 = 40 + 253$
  - **Timisoara**: $447 = 118 + 329$
  - **Zerind**: $449 = 75 + 374$

366 = 0 + 366
A* search example
A* search example

A* search example
A* search example
A* search example

Graph showing the A* search algorithm with cities and their distances. The algorithm starts from Arad and explores the tree, considering the sum of the path cost and the heuristic cost to each city. The graph shows the cities connected with edges, with distances labeled on each edge. The goal is to find the shortest path to Bucharest.
Can we Prove Anything?

- If the state space is finite and we avoid repeated states, the search is complete, but in general is not optimal.

- If the state space is finite and we do not avoid repeated states, the search is in general not complete.

- If the state space is infinite, the search is in general not complete.
Admissible heuristic

- Let $h^*(N)$ be the true cost of the optimal path from N to a goal node
- Heuristic $h(N)$ is admissible if:
  \[ 0 \leq h(N) \leq h^*(N) \]
- An admissible heuristic is always optimistic
Admissible heuristics

The 8-puzzle:

- $h_1(n) = \text{number of misplaced tiles}$
- $h_2(n) = \text{total Manhattan distance}$

(i.e., no. of squares from desired location of each tile)

- $h_1(S) = ?$ 7
- $h_2(S) = ?$ 2+3+3+2+4+2+0+2 = 18
Heuristic quality

- Effective branching factor $b^*$
  - Is the branching factor that a uniform tree of depth $d$ would have in order to contain $N+1$ nodes.

\[ N + 1 = 1 + b^* + (b^*)^2 + ... + (b^*)^d \]

- Measure is fairly constant for sufficiently hard problems.
  - Can thus provide a good guide to the heuristic’s overall usefulness.
  - A good value of $b^*$ is 1.
Heuristic quality and dominance

- 1200 random problems with solution lengths from 2 to 24.
- If \( h_2(n) \geq h_1(n) \) for all \( n \) (both admissible)
  then \( h_2 \) dominates \( h_1 \) and is better for search

<table>
<thead>
<tr>
<th>( d )</th>
<th>IDS</th>
<th>( A^*(h_1) )</th>
<th>( A^*(h_2) )</th>
<th>IDS</th>
<th>( A^*(h_1) )</th>
<th>( A^*(h_2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td></td>
<td>2.45</td>
<td>1.79</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
<td>12</td>
<td></td>
<td>2.87</td>
<td>1.48</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
<td></td>
<td>2.73</td>
<td>1.34</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
<td></td>
<td>2.80</td>
<td>1.33</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>39</td>
<td></td>
<td>2.79</td>
<td>1.38</td>
</tr>
<tr>
<td>12</td>
<td>3644035</td>
<td>227</td>
<td>73</td>
<td></td>
<td>2.78</td>
<td>1.42</td>
</tr>
<tr>
<td>14</td>
<td>–</td>
<td>539</td>
<td>113</td>
<td></td>
<td>–</td>
<td>1.44</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>1301</td>
<td>211</td>
<td></td>
<td>–</td>
<td>1.45</td>
</tr>
<tr>
<td>18</td>
<td>–</td>
<td>3056</td>
<td>363</td>
<td></td>
<td>–</td>
<td>1.46</td>
</tr>
<tr>
<td>20</td>
<td>–</td>
<td>7276</td>
<td>676</td>
<td></td>
<td>–</td>
<td>1.47</td>
</tr>
<tr>
<td>22</td>
<td>–</td>
<td>18094</td>
<td>1219</td>
<td></td>
<td>–</td>
<td>1.48</td>
</tr>
<tr>
<td>24</td>
<td>–</td>
<td>39135</td>
<td>1641</td>
<td></td>
<td>–</td>
<td>1.48</td>
</tr>
</tbody>
</table>
Inventing admissible heuristics

- Admissible heuristics can be derived from the exact solution cost of a relaxed version of the problem:
  - Relaxed 8-puzzle for $h_1$: a tile can move anywhere. As a result, $h_1(n)$ gives the shortest solution.
  - Relaxed 8-puzzle for $h_2$: a tile can move to any adjacent square. As a result, $h_2(n)$ gives the shortest solution.

The optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem.
Optimality of A*(standard proof)

- Suppose suboptimal goal $G_2$ in the queue.
- Let $n$ be an unexpanded node on a shortest to optimal goal $G$.
  
  $f(G_2) = g(G_2)$ since $h(G_2) = 0$
  > $g(G)$ since $G_2$ is suboptimal
  >= $f(n)$ since $h$ is admissible

Since $f(G_2) > f(n)$, A* will never select $G_2$ for expansion
Optimality for graphs?

- Admissibility is not sufficient for graph search
  - In graph search, the optimal path to a repeated state could be discarded if it is not the first one generated
  - Can fix problem by requiring consistency property for $h(n)$

- A heuristic is **consistent** if for every successor $n'$ of a node $n$ generated by any action $a$,

  $$h(n) \leq c(n, a, n') + h(n')$$

  *(aka “monotonic”)*

- Admissible heuristics are generally consistent
A* is optimal with consistent heuristics

- If $h$ is consistent, we have

\[
\begin{align*}
    f(n') &= g(n') + h(n') \\
          &= g(n) + c(n,a,n') + h(n') \\
          &\geq g(n) + h(n) \\
          &= f(n)
\end{align*}
\]

i.e., $f(n)$ is non-decreasing along any path.

Thus, first goal-state selected for expansion must be optimal

- Theorem:
  - If $h(n)$ is consistent, A* using GRAPH-SEARCH is optimal
Contours of A* Search

- A* expands nodes in order of increasing $f$ value
- Gradually adds "$f$-contours" of nodes
- Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$
Contours of $A^*$ Search

- With uniform-cost ($h(n) = 0$, contours will be circular
- With good heuristics, contours will be focused around optimal path
- $A^*$ will expand all nodes with cost $f(n) < C^*$
A* search, evaluation

- Completeness: YES
  - Since bands of increasing $f$ are added
  - Unless there are infinitely many nodes with $f < f(G)$
A* search, evaluation

- Completeness: YES
- Time complexity:
  - Number of nodes expanded is still exponential in the length of the solution.
A* search, evaluation

- Completeness: YES
- Time complexity: (exponential with path length)
- Space complexity:
  - It keeps all generated nodes in memory
  - Hence space is the major problem not time
A* search, evaluation

- Completeness: YES
- Time complexity: (exponential with path length)
- Space complexity: (all nodes are stored)
- Optimality: YES
  - Cannot expand $f_{i+1}$ until $f_i$ is finished.
  - A* expands all nodes with $f(n) < C^*$
  - A* expands some nodes with $f(n) = C^*$
  - A* expands no nodes with $f(n) > C^*$

Also *optimally efficient* (not including ties)
Compare Uniform Cost and A*

- Uniform-cost expanded in all directions
- A* expands mainly toward the goal, but does hedge its bets to ensure optimality
Artificial Intelligence

For HEDSPI Project

Lecturer 5 - Advanced search methods

Lecturers:
  Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Outline

- Memory-bounded heuristic search
- Hill-climbing search
- Simulated annealing search
- Local beam search
Memory-bounded heuristic search

- Some solutions to A* space problems (maintain completeness and optimality)
  - Iterative-deepening A* (IDA*)
    - Here cutoff information is the $f$-cost ($g+h$) instead of depth
  - Recursive best-first search (RBFS)
    - Recursive algorithm that attempts to mimic standard best-first search with linear space.
  - (simple) Memory-bounded A* ((S)MA*)
    - Drop the worst-leaf node when memory is full
Iterative Deeping A*

- Iterative Deeping version of A*
  - use threshold as depth bound
    - To find solution under the threshold of $f(.)$
  - increase threshold as minimum of $f(.)$ of previous cycle
- Still admissible
- same order of node expansion
- Storage Efficient – practical
  - but suffers for the real-valued $f(.)$
  - large number of iterations
Iterative Deepening A* Search Algorithm (for tree search)

1. **Start**
   - Set threshold as \( h(s) \)

2. **Put** \( s \) in OPEN, compute \( f(s) \)

3. **Check** if OPEN is empty?
   - Yes: \( \text{threshold} = \min( f(.) | f(.) > \text{threshold} ) \)

4. **Remove** the node of OPEN whose \( f(.) \) value is smallest and put it in CLOSE (call it \( n \))

5. **Check** if \( n = \text{goal} \)?
   - Yes: Success

6. **Expand** \( n \). Calculate \( f(.) \) of successor if \( f(\text{suc}) < \text{threshold} \) then
   - Put successors to OPEN if pointers back to \( n \)
Recursive best-first search

- A variation of Depth-first search
- Keep track of $f$-value of the best alternative path
- Unwind if $f$-value of all children exceed its best alternative
- When unwind, store $f$-value of best child as its $f$-value
- When needed, the parent regenerate its children again.
Recursive best-first search

function RECURSIVE-BEST-FIRST-SEARCH(problem) return a solution or failure
return RFBS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RFBS( problem, node, f_limit) return a solution or failure and a new f-cost limit
if GOAL-TEST[problem](STATE[node]) then return node
successors ← EXPAND(node, problem)
if successors is empty then return failure, ∞
for each s in successors do
    f[s] ← max(g(s) + h(s), f[node])
repeat
    best ← the lowest f-value node in successors
    if f[best] > f_limit then return failure, f[best]
alternative ← the second lowest f-value among successors
result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
if result ≠ failure then return result
Recursive best-first search

- Keeps track of the f-value of the best-alternative path available.
  - If current f-values exceeds this alternative f-value than backtrack to alternative path.
  - Upon backtracking change f-value to best f-value of its children.
  - Re-expansion of this result is thus still possible.
RBFS evaluation

- RBFS is a bit more efficient than IDA*
  - Still excessive node generation (mind changes)
- Like A*, optimal if \( h(n) \) is admissible
- Space complexity is \( O(bd) \).
  - IDA* retains only one single number (the current f-cost limit)
- Time complexity difficult to characterize
  - Depends on accuracy if \( h(n) \) and how often best path changes.
- IDA* and RBFS suffer from too little memory.
(simplified) memory-bounded A*

- Use all available memory.
  - I.e. expand best leaves until available memory is full
  - When full, SMA* drops worst leaf node (highest $f$-value)
  - Like RBFS, we remember the best descendant in the branch we delete

- What if all leaves have the same $f$-value?
  - Same node could be selected for expansion and deletion.
  - SMA* solves this by expanding newest best leaf and deleting oldest worst leaf.

- The deleted node is regenerated when all other candidates look worse than the node.

- SMA* is complete if solution is reachable, optimal if optimal solution is reachable.

- Time can still be exponential.
Local search algorithms

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution.
- State space = set of "complete" configurations.
- Find configuration satisfying constraints, e.g., n-queens.
- In such cases, we can use local search algorithms.

- Local search = use single current state and move to neighboring states.
- Advantages:
  - Use very little memory.
  - Find often reasonable solutions in large or infinite state spaces.
- Are also useful for pure optimization problems.
  - Find best state according to some objective function.
Local search and optimization
Example: $n$-queens

- Put $n$ queens on an $n \times n$ board with no two queens on the same row, column, or diagonal.
Hill-climbing search

- Simple, general idea:
  - Start wherever
  - Always choose the best neighbor
  - If no neighbors have better scores than current, quit

- Hill climbing does not look ahead of the immediate neighbors of the current state.

- Hill-climbing chooses randomly among the set of best successors, if there is more than one.

- Some problem spaces are great for hill climbing and others are terrible.
function HILL-CLIMBING(problem) return a state that is a local maximum
input: problem, a problem
local variables: current, a node.
            neighbor, a node.

current ← MAKE-NODE(INITIAL-STATE[problem])
loop do
    neighbor ← a highest valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
    current ← neighbor
Drawbacks of hill climbing

- **Problems:**
  - **Local Maxima:** depending on initial state, can get stuck in local maxima
  - **Plateaus:** the space has a broad flat region that gives the search algorithm no direction (random walk)
  - **Ridges:** flat like a plateau, but with dropoffs to the sides; steps to the North, East, South and West may go down, but a combination of two steps (e.g. N, W) may go up

- Introduce randomness
Hill-climbing variations

- **Stochastic hill-climbing**
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.

- **First-choice hill-climbing**
  - Stochastic hill climbing by generating successors randomly until a better one is found.

- **Random-restart hill-climbing**
  - Tries to avoid getting stuck in local maxima.
  - If at first you don’t succeed, try, try again…
Simulated Annealing

- Simulates slow cooling of annealing process
- Applied for combinatorial optimization problem by S. Kirkpatrick (‘83)

**What is annealing?**
- Process of slowly cooling down a compound or a substance
- Slow cooling let the substance flow around \( \rightarrow \) thermodynamic equilibrium
- Molecules get optimum conformation
Simulated annealing

gradually decrease shaking to make sure the ball escape from local minima and fall into the global minimum
Simulated annealing

- Escape local maxima by allowing “bad” moves.
  - Idea: but gradually decrease their size and frequency.
- Origin; metallurgical annealing
- Implement:
  - Randomly select a move instead of selecting best move
  - Accept a bad move with probability less than 1 (p<1)
  - p decreases by time

- If T decreases slowly enough, best state is reached.
- Applied for VLSI layout, airline scheduling, etc.
Simulated annealing

**function** SIMULATED-ANNEALING( *problem, schedule*) **return** a solution state

**input:** *problem*, a problem
*schedule*, a mapping from time to temperature

**local variables:** *current*, a node; *next*, a node.
*T*, a “temperature” controlling the probability of downward steps

*current* ← MAKE-NODE(INITIAL-STATE[problem])

**for** *t* ← 1 to ∞ **do**

*T* ← *schedule*[t]

**if** *T* = 0 **then** **return** *current*

*next* ← a randomly selected successor of *current*

*ΔE* ← VALUE[*next*] - VALUE[*current*]

**if** *ΔE* > 0 **then** *current* ← *next*

**else** *current* ← *next* only with probability $e^{ΔE/T}$

What’s the probability when: $T \to \infty$?
What’s the probability when: $T \to 0$?
What’s the probability when: $Δ=0$?
What’s the probability when: $Δ \to -\infty$?

Similar to hill climbing, but a **random** move instead of best move

Case of improvement, make the move

Otherwise, choose the move with probability that decreases exponentially with the “badness” of the move.
Simulated Annealing parameters

- Temperature T
  - Used to determine the probability
  - High T: large changes
  - Low T: small changes

- Cooling Schedule
  - Determines rate at which the temperature T is lowered
  - Lowers T slowly enough, the algorithm will find a global optimum

- In the beginning, aggressive for searching alternatives, become conservative when time goes by.
Simulated Annealing Cooling Schedule

- if $T_i$ is reduced too fast, poor quality
- if $T_t \geq T(0) / \log(1+t)$ - Geman
  - System will converge to minimum configuration
- $T_t = k/1+t$ - Szu
- $T_t = a \cdot T(t-1)$ where $a$ is in between 0.8 and 0.99
Tips for Simulated Annealing

- To avoid of entrainment in local minima
  - Annealing schedule: by trial and error
    - Choice of initial temperature
    - How many iterations are performed at each temperature
    - How much the temperature is decremented at each step as cooling proceeds

- Difficulties
  - Determination of parameters
  - If cooling is too slow → Too much time to get solution
  - If cooling is too rapid → Solution may not be the global optimum
Properties of simulated annealing

- Theoretical guarantee:
  - Stationary distribution: \( p(x) \propto e^{\frac{E(x)}{kT}} \)
  - If \( T \) decreased slowly enough, will converge to optimal state!

- Is this an interesting guarantee?

- Sounds like magic, but:
  - The more downhill steps you need to escape, the less likely you are to ever make them all in a row
  - People think hard about *ridge operators* which let you jump around the space in better ways
Local beam search

- Like greedy search, but keep K states at all times:
  - Initially: $k$ random states
  - Next: determine all successors of $k$ states
  - If any of successors is goal → finished
  - Else select $k$ best from successors and repeat.

Greedy Search

Beam Search
Local beam search

- **Major difference with random-restart search**
  - Information is shared among k search threads: If one state generated good successor, but others did not → “come here, the grass is greener!”

- Can suffer from lack of diversity.
  - Stochastic variant: choose k successors at proportionally to state success.

- The best choice in MANY practical settings
Artificial Intelligence

For HEDSPI Project

Lecturer 6 - Advanced search methods

Lecturers:

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Outline

- Game and search
- Alpha-beta pruning
Games and search

Why study games?
Why is search a good idea?

Majors assumptions about games:
- Only an agent’s actions change the world
- World is deterministic and accessible
Why study games?

machines are better than humans in:
  othello
humans are better than machines in:
  go
here: perfect information zero-sum games

May 1997
Deep Blue - Garry Kasparov
3.5 - 2.5
Why study games?

- Games are a form of *multi-agent environment*
  - What do other agents do and how do they affect our success?
  - Cooperative vs. competitive multi-agent environments.
  - Competitive multi-agent environments give rise to adversarial search a.k.a. *games*.

- Why study games?
  - Fun; historically entertaining
  - Interesting subject of study because they are hard
  - Easy to represent and agents restricted to small number of actions
Relation of Games to Search

- **Search** – no adversary
  - Solution is (heuristic) method for finding goal
  - Heuristics and CSP techniques can find *optimal* solution
  - Evaluation function: estimate of cost from start to goal through given node
  - Examples: path planning, scheduling activities

- **Games** – adversary
  - Solution is strategy (strategy specifies move for every possible opponent reply).
  - Time limits force an *approximate* solution
  - Evaluation function: evaluate “goodness” of game position
  - Examples: chess, checkers, Othello, backgammon

- Ignoring computational complexity, games are a perfect application for a complete search.
- Of course, ignoring complexity is a bad idea, so games are a good place to study resource bounded searches.
# Types of Games

<table>
<thead>
<tr>
<th>Types of Information</th>
<th>Deterministic</th>
<th>Chance</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfect information</td>
<td>chess, checkers, go, othello</td>
<td>backgammon monopoly</td>
</tr>
<tr>
<td>imperfect information</td>
<td>battleships, blind tictactoe</td>
<td>bridge, poker, scrabble nuclear war</td>
</tr>
</tbody>
</table>
Minimax

- Two players: MAX and MIN
- MAX moves first and they take turns until the game is over. Winner gets award, looser gets penalty.
- Games as search:
  - Initial state: e.g. board configuration of chess
  - Successor function: list of (move, state) pairs specifying legal moves.
  - Terminal test: Is the game finished?
  - Utility function: Gives numerical value of terminal states.
    - E.g. win (+1), loose (-1) and draw (0) in tic-tac-toe
- MAX uses search tree to determine next move.
- Perfect play for deterministic games
Minimax

- From among the moves available to you, take the best one
- The best one is determined by a search using the MiniMax strategy
Optimal strategies

- Find the contingent strategy for MAX assuming an infallible MIN opponent.
- Assumption: Both players play optimally !!
- Given a game tree, the optimal strategy can be determined by using the minimax value of each node:

$$MINIMAX-VALUE(n) =$$
- $$UTILITY(n)$$ if $n$ is a terminal
- $$\max_{s \in \text{successors}(n)} MINIMAX-VALUE(s)$$ if $n$ is a max node
- $$\min_{s \in \text{successors}(n)} MINIMAX-VALUE(s)$$ if $n$ is a min node
Minimax
Minimax algorithm

function Minimax-Decision(state) returns an action

    $v \leftarrow \text{Max-Value}(state)$
    return the action in Successors(state) with value $v$

function Max-Value(state) returns a utility value

    if Terminal-Test(state) then return Utility(state)
    $v \leftarrow -\infty$
    for $a, s$ in Successors(state) do
        $v \leftarrow \text{Max}(v, \text{Min-Value}(s))$
    return $v$

function Min-Value(state) returns a utility value

    if Terminal-Test(state) then return Utility(state)
    $v \leftarrow \infty$
    for $a, s$ in Successors(state) do
        $v \leftarrow \text{Min}(v, \text{Max-Value}(s))$
    return $v$
Properties of minimax

- **Complete?** Yes (if tree is finite)
- **Optimal?** Yes (against an optimal opponent)
- **Time complexity?** $O(b^m)$
- **Space complexity?** $O(bm)$ (depth-first exploration)

- For chess, $b \approx 35$, $m \approx 100$ for "reasonable" games
  - exact solution completely infeasible
Problem of minimax search

- Number of games states is exponential to the number of moves.
  - Solution: Do not examine every node

⇒ Alpha-beta pruning:
  - Remove branches that do not influence final decision
  - Revisit example …
α-β pruning

- Alpha values: the best values achievable for MAX, hence the max value so far

- Beta values: the best values achievable for MIN, hence the min value so far

- At MIN level: compare result V of node to alpha value. If V>alpha, pass value to parent node and BREAK

- At MAX level: compare result V of node to beta value. If V<beta, pass value to parent node and BREAK
α-β pruning example
α-β pruning example
$\alpha-\beta$ pruning example
α-β pruning example
$\alpha$-$\beta$ pruning example
Properties of $\alpha$-\(\beta\)

- Pruning does not affect final result
- Entire sub-trees can be pruned.
- Good move ordering improves effectiveness of pruning. With "perfect ordering"
  - Time complexity = $O(b^{m/2})$
    - Doubles depth of search
  - Branching factor of $\sqrt{b}$ !!
  - Alpha-beta pruning can look twice as far as minimax in the same amount of time
- Repeated states are again possible.
  - Store them in memory = transposition table
- A simple example of the value of reasoning about which computations are relevant (a form of metareasoning)
Why is it called $\alpha$-$\beta$?

- $\alpha$ is the value of the best (i.e., highest-value) choice found so far at any choice point along the path for $\text{max}$
- If $v$ is worse than $\alpha$, $\text{max}$ will avoid it $\rightarrow$ prune that branch
- Define $\beta$ similarly for $\text{min}$
The $\alpha$-$\beta$ algorithm

function $\text{Alpha-Beta-Search}(state)$ returns an action
inputs: $state$, current state in game
$v \leftarrow \text{Max-Value}(state, -\infty, +\infty)$
return the action in $\text{Successors}(state)$ with value $v$

function $\text{Max-Value}(state, \alpha, \beta)$ returns a utility value
inputs: $state$, current state in game
$\alpha$, the value of the best alternative for $\text{Max}$ along the path to $state$
$\beta$, the value of the best alternative for $\text{Min}$ along the path to $state$
if $\text{Terminal-Test}(state)$ then return $\text{Utility}(state)$
$v \leftarrow -\infty$
for $a, s$ in $\text{Successors}(state)$ do
$v \leftarrow \text{Max}(v, \text{Min-Value}(s, \alpha, \beta))$
if $v \geq \beta$ then return $v$
$\alpha \leftarrow \text{Max}(\alpha, v)$
return $v$
The $\alpha$-$\beta$ algorithm

```
function Min-Value(state, $\alpha$, $\beta$) returns a utility value
  inputs: state, current state in game
          $\alpha$, the value of the best alternative for MAX along the path to state
          $\beta$, the value of the best alternative for MIN along the path to state
  if Terminal-Test(state) then return Utility(state)
  $v \leftarrow +\infty$
  for $a$, $s$ in Successors(state) do
    $v \leftarrow \min(v, \max-Value(s, \alpha, \beta))$
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow \min(\beta, v)$
  return $v$
```
Imperfect, real-time decisions

- Minimax and alpha-beta pruning require too much leafnode evaluations.

- May be impractical within a reasonable amount of time.

- Suppose we have 100 secs, explore $10^4$ nodes/sec $\Rightarrow 10^6$ nodes per move

- Standard approach (SHANNON, 1950):
  - Cut off search earlier (replace TERMINAL-TEST by CUTOFF-TEST)
  - Apply heuristic evaluation function EVAL (replacing utility function of alpha-beta)
Cut-off search

- Change:
  
  ```
  if TERMINAL-TEST(state) then return UTILITY(state)
  ```

  into:

  ```
  if CUTOFF-TEST(state, depth) then return EVAL(state)
  ```

- Introduces a fixed-depth limit `depth`
  - Is selected so that the amount of time will not exceed what the rules of the game allow.

- When cut-off occurs, the evaluation is performed.
Heuristic evaluation (EVAL)

- Idea: produce an estimate of the expected utility of the game from a given position.
- Performance depends on quality of EVAL.
- Requirements:
  - EVAL should order terminal-nodes in the same way as UTILITY.
  - Computation may not take too long.
  - For non-terminal states the EVAL should be strongly correlated with the actual chance of winning.
- Only useful for quiescent (no wild swings in value in near future) states
Evaluation function example

- For chess, typically **linear** weighted sum of **features**
  \[ Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]
- e.g., \( w_1 = 9 \) with
  \[ f_1(s) = (\text{number of white queens}) - (\text{number of black queens}), \text{etc.}\]
Chess complexity

- PC can search 200 millions nodes/3min.
- Branching factor: ~35
  - $35^5 \sim 50$ millions
  - if use minimax, could look ahead 5 plies, defeated by average player, planning 6-8 plies.
- Does it work in practice?
  - 4-ply $\approx$ human novice $\rightarrow$ hopeless chess player
  - 8-ply $\approx$ typical PC, human master
  - 12-ply $\approx$ Deep Blue, Kasparov
- To reach grandmaster level, needs a better extensively tuned evaluation and a large database of optimal opening and ending of the game
Deterministic games in practice

- Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.


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

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

- Chance introduces by dice, card-shuffling, coin-flipping...
- Example with coin-flipping:

```
MAX
- CHANCE
- MIN
```

change nodes
Backgammon

Possible moves: (5-10,5-11), (5-11,19-24),(5-10,10-16) and (5-11,11-16)
**Expected minimax value**

... 

if *state* is a MAX node then
    return the highest $\text{EXPECTED-MINIMAX-VALUE}$ of $\text{SUCCESSORS}(\text{state})$

if *state* is a MIN node then
    return the lowest $\text{EXPECTED-MINIMAX-VALUE}$ of $\text{SUCCESSORS}(\text{state})$

if *state* is a chance node then
    return average of $\text{EXPECTED-MINIMAX-VALUE}$ of $\text{SUCCESSORS}(\text{state})$

$\text{EXPECTED-MINIMAX-VALUE}(n) =$

$\text{UTILITY}(n)$  \hspace{1cm} If $n$ is a terminal

$max_{s \in \text{successors}(n)} \text{EXPECTED-MINIMAX}(s)$  \hspace{1cm} If $n$ is a max node

$min_{s \in \text{successors}(n)} \text{EXPECTED-MINIMAX}(s)$  \hspace{1cm} If $n$ is a max node

$\sum_{s \in \text{successors}(n)} P(s) \cdot \text{EXPECTED-MINIMAX}(s)$  \hspace{1cm} If $n$ is a chance node

$P(s)$ is probability of $s$ occurrence
Games of imperfect information

- E.g., card games, where opponent's initial cards are unknown
- Typically we can calculate a probability for each possible deal
- Seems just like having one big dice roll at the beginning of the game
- Idea: compute the minimax value of each action in each deal, then choose the action with highest expected value over all deals
- Special case: if an action is optimal for all deals, it's optimal.
- GIB, current best bridge program, approximates this idea by
  - generating 100 deals consistent with bidding information
  - picking the action that wins most tricks on average
Artificial Intelligence
For HEDSPI Project

Lecturer 8 – Constraint Satisfaction Problems

Lecturer:
Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Constraints Satisfaction Problems (CSPs)

- CSPs example
- Backtracking search
- Problem structure
- Local search for CSPs
CSP

- Standard search problems
  - State is a “black-box”
    - Any data structure that implements initial states, goal states, successor function

- CSPs
  - State is composed of variables $X_i$ with value in domain $D_i$
  - Goal test is a set of constraints over variables
Example: Map Coloring

- **Variables**
  - WA, NT, Q, NSW, V , SA

- **Domain**
  - $D_i = \{\text{red, green, blue}\}$

- **Constraint**
  - Neighbor regions must have different colors
    - WA /= NT
    - WA /= SA
    - NT /= SA
    - ...

![Diagram of Australia with states and territories labeled.

Western Australia, Northern Territory, South Australia, Queensland, New South Wales, Victoria.](image)
Example: Map Coloring

- Solution is an assignment of variables satisfying all constraints
  - WA=red, and
  - NT=green, and
  - Q=red, and
  - NSW=green, and
  - V=red, and
  - SA=blue
Constraint Graph

- **Binary CSPs**
  - Each constraint relates at most two variables
- **Constraint graph**
  - Node is variable
  - Edge is constraint

![Constraint Graph Diagram](image-url)
Varieties of CSPs

- **Discrete variables**
  - Finite domain, e.g., SAT Solving
  - Infinite domain, e.g., work scheduling
    - Variables is start/end of working day
    - Constraint language, e.g., \( \text{StartJob}_1 + 5 \leq \text{StartJob}_3 \)
    - Linear constraints are decidable, non-linear constraints are undecidable

- **Continuous variables**
  - e.g., start/end time of observing the universe using Hubble telescope
  - Linear constraints are solvable using Linear Programming
Varieties of Constraints

- **Single-variable constraints**
  - e.g., SA /= green

- **Binary constraints**
  - e.g., SA /= WA

- **Multi-variable constraints**
  - Relate at least 3 variables

- **Soft constraints**
  - Priority, e.g., red better than green
  - Cost function over variables
Example: Cryptarithmetic

- **Variables**
  - F, T, O, U, R, W, X_1, X_2, X_3

- **Domain**
  - \{0,1,2,3,4,5,6,7,8,9\}

- **Constraints**
  - AllDiff(F, T, O, U, R, W)
  - O + O = R + 10 \times X_1
  - X_1 + W + W = U + 10 \times X_2
  - X_2 + T + T = O + 10 \times X_3
  - X_3 = F

\[
\text{TWO} \quad + \quad \text{TWO} \\
\text{FOUR}
\]
Real World CSP

- Assignment
  - E.g., who teach which class
- Scheduling
  - E.g., when and where the class takes place
- Hardware design
- Spreadsheets
- Transport scheduling
- Manufacture scheduling
CSPs by Standard Search

- **State**
  - Defined by the values assigned so far
- **Initial state**
  - The empty assignment
- **Successor function**
  - Assign a value to a unassigned variable that does not conflict with current assignment
  - Fail if no legal assignment
- **Goal test**
  - All variables are assigned and no conflict
CSP by Standard Search

- Every solution appears at depth $n$ with $n$ variables
  - Use depth-first search
- Path is irrelevant
- Number of leaves
  - $n!d^n$
    - Two many
Backtracking Search

- Variable assignments are commutative, e.g.,
  - \{WA=red, NT =green\}
  - \{NT =green, WA=red\}

- Single-variable assignment
  - Only consider one variable at each node
  - \(d^n\) leaves

- Backtracking search
  - Depth-first search + Single-variable assignment

- Backtracking search is the basic uninformed algorithm for CSPs
  - Can solve n-Queen with \(n = 25\)
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING(\{\}, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add \{var = value\} to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failure then return result
        remove \{var = value\} from assignment
    return failure
Backtracking Search Algorithm
Improving Backtracking Search

- Which variable should be assigned next?
- In what order should its values be tried?
- Can we detect inevitable failure early?
- Can we take advantage of problem structure?
Choosing Variables

- Minimum remaining values (MRV)
  - Choose the variable with the fewest legal values

- Degree heuristic
  - Choose the variable with the most constraints on remaining variables
Choosing Values

- Least constraining value (LCV)
  - Choose the least constraining value
    - the one that rules out the fewest values in the remaining variables

- Keep track of remaining legal values for unassigned variables
  - Terminate search when any variable has no legal values
Forward Checking

- Constraint propagation

- NT and SA cannot both be blue

- Simplest form of propagation makes each arc consistent
  - X -> Y is consistent iff for each value x of X there is some allowed value y for Y
Problem Structure

- Assume we have a new region $T$ in addition
- $T$ and the rest are two independent problems
  - Each problem is a connected component in the constraint graph
Problem Structure

- Tree-structured problem

- Theorem
  - If the constraint graph has no loop then CSPs can be solved in $O(nd^2)$ time
Problem Structure

- Algorithm for tree-structured problems
Iterative Algorithms for CSPs

- Hill-climbing, Simulated Annealing can be used for CSPs
  - Complete state, e.g., all variables are assigned at each node
- Allow states with unsatisfiable constraints
- Operators reassign variables
- Variable selection
  - Random
- Value selection by min-conflicts heuristic
  - Choose value that violates the fewest constraints
    - i.e., hill climbing with \( h(n) = \text{total number of violated constraints} \)
Example: 4-Queens

- State: 4 queens in four columns \((4 \times 4 = 256\) states\)
- Operators: move queen in column
- Goal test: no attacks
- Evaluation: \(h(n) = \text{number of attacks}\)
CSPs are a special kind of problem:
- states defined by values of a fixed set of variables
- goal test defined by constraints on variable values

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

The CSPs representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice
Exercice

- Solve the following cryptarithmetic problem by combining the heuristics
  - Constraint Propagation
  - Minimum Remaining Values
  - Least Constraining Values

\[ \begin{array}{c}
T \times \circ \times \circ \\
\text{FOUR}
\end{array} \]

\[ \begin{array}{c}
\text{TWO} \\
+ \text{TWO}
\end{array} \]

\[ \begin{array}{c}
\text{FOUR}
\end{array} \]
Exercice

1. Choose $X_3$: domain $\{0,1\}$
2. Choose $X_3=1$: use constraint propagation $F/=0$
3. $F = 1$
4. Choose $X_2$: $X_1$ and $X_2$ có have the same remaining values
5. Choose $X_2=0$
6. Choose $X_1$: $X_1$ has MRV
7. Choose $X_1=0$
8. Choose $O$: $O$ must be even, less than 5 and therefore has MRV
   $(T+T=O$ due $1$ và $O+O=R+10*0)$
9. Choose $O=4$
10. $R=8$
11. $T=7$
12. Choose $U$: $U$ must be even, less than 9
13. $U=6$: constraint propagation
14. $W=3$
Artificial Intelligence

For HEDSPI Project

Lecturer 9 – Propositional Logic

Lecturers:

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Outline

- What is Logic
- Propositional Logic
  - Syntax
  - Semantic
- Inference in Propositional Logic
  - Forward Chaining
  - Backward Chaining
Knowledge-based Agents

- **Know about the world**
  - They maintain a collection of facts (sentences) about the world, their Knowledge Base, expressed in some formal language.

- **Reason about the world**
  - They are able to derive new facts from those in the KB using some inference mechanism.

- **Act upon the world**
  - They map percepts to actions by querying and updating the KB.
What is Logic?

- **A logic** is a triplet $<L, S, R>$
  - $L$, the **language** of the logic, is a class of sentences described by a precise syntax, usually a formal grammar
  - $S$, the logic’s **semantic**, describes the meaning of elements in $L$
  - $R$, the logic’s **inference system**, consisting of derivation rules over $L$

- **Examples of logics**:
  - Propositional, First Order, Higher Order, Temporal, Fuzzy, Modal, Linear, …
Propositional Logic

- Propositional Logic is about **facts** in the world that are either true or false, nothing else.
- Propositional variables stand for **basic facts**.
- Sentences are made of
  - propositional variables (A, B, ...),
  - logical constants (TRUE, FALSE), and
  - logical connectives (not, and, or, ...)
- The meaning of sentences ranges over the Boolean values \{True, False\}
- Examples: It’s sunny, John is married
Language of Propositional Logic

Symbols
- Propositional variables: A, B, ..., P, Q, ...
- Logical constants: TRUE, FALSE
- Logical connectives:
  \[ \neg, \land, \lor, \Rightarrow, \Leftrightarrow \]

Sentences
- Each propositional variable is a sentence
- Each logical constant is a sentence
- If $\alpha$ and $\beta$ are sentences then the following are sentences
  
  \[
  (\alpha), \neg\alpha, \alpha \land \beta, \alpha \lor \beta, \alpha \Rightarrow \beta, \alpha \Leftrightarrow \beta
  \]
Formal Language of Propositional Logic

- **Symbols**
  - Propositional variables: A, B, ..., P, Q, ...
  - Logical constants: T, F
  - Logical connectives: ¬, ∧, ∨, →, ↔

- **Formal Grammar**
  - Sentence -> Asentence | Csentence
  - Asentence -> TRUE | FALSE | A | B | ...  
  - Csentence -> (Sentence) | ¬ Sentence | Sentence Connective Sentence
  - Connective -> ¬ | ∧ | ∨ | → | ↔
Semantic of Propositional Logic

- The meaning of TRUE is always True, the meaning of FALSE is always False
- The meaning of a propositional variable is either True or False
  - depends on the interpretation
    - assignment of Boolean values to propositional variables
- The meaning of a sentence is either True or False
  - depends on the interpretation
## Semantic of Propositional Logic

- **True table**

<table>
<thead>
<tr>
<th>P</th>
<th>Q</th>
<th>Not P</th>
<th>P and Q</th>
<th>P or Q</th>
<th>P implies Q</th>
<th>P equiv Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
Semantic of Propositional Logic

- Satisfiability
  - A sentence is **satisfiable** if it is true under some interpretation
  - Ex: \( P \) or \( H \) is satisfiable
    \[ P \land \neg P \] is unsatisfiable (not satisfiable)

- Validity
  - A sentence is **valid** if it is true in every interpretation
  - Ex: \((P \lor H) \land \neg A\) \(\Rightarrow P\) is valid
    \[ P \lor H \] is not valid
Semantic of Propositional Logic

- **Entailment**
  - Given
    - A set of sentences \( \Gamma \)
    - A sentence \( \psi \)
  - We write
    \[
    \Gamma \models \psi
    \]
    if and only if every interpretation that makes all sentences in \( \Gamma \) true also makes \( \psi \) true
  - We said that \( \Gamma \) entails \( \psi \)
Semantic of Propositional Logic

- Satisfiability vs. Validity vs. Entailment
  - \( \psi \) is valid iff \( \text{True} \models \psi \) (also written \( \models \psi \))
  - \( \psi \) is unsatisfiable iff \( \psi \notmodels \text{False} \)
  - \( \Gamma \models \psi \) iff \( \Gamma \cup \{\neg \psi\} \) is unsatisfiable
Inference in Propositional Logic

- Forward Chaining
- Backward Chaining
Forward Chaining

- Given a set of rules, i.e. formulae of the form
  \[ p_1 \land p_2 \land \ldots \land p_n \Rightarrow q \]
  and a set of known facts, i.e., formulae of the form
  \[ q, r, \ldots \]

- A new fact \( p \) is added

- Find all rules that have \( p \) as a premise

- If the other premises are already known to hold then
  - add the consequent to the set of known facts, and
  - trigger further inferences
Forward Chaining

- Example

\[ P \Rightarrow Q \]
\[ L \land M \Rightarrow P \]
\[ B \land L \Rightarrow M \]
\[ A \land P \Rightarrow L \]
\[ A \land B \Rightarrow L \]
\[ A \]
\[ B \]
Forward Chaining
Forward Chaining
Forward Chaining
Forward Chaining
Forward Chaining
Forward Chaining
Forward Chaining

- Soundness
  - Yes
- Completeness
  - Yes
Backward Chaining

- Given a set of rules, and a set of known facts
- We ask whether a fact $P$ is a consequence of the set of rules and the set of known facts
- The procedure check whether $P$ is in the set of known facts
- Otherwise find all rules that have $P$ as a consequent
  - If the premise is a conjunction, then process the conjunction conjunct by conjunct
Backward Chaining

Example

\[ P \Rightarrow Q \]
\[ L \land M \Rightarrow P \]
\[ B \land L \Rightarrow M \]
\[ A \land P \Rightarrow L \]
\[ A \land B \Rightarrow L \]
\[ A \]
\[ B \]
Backward Chaining
Backward Chaining
Backward Chaining
Backward Chaining
Backward Chaining

- Soundness
  - Yes
- Completeness
  - Yes
Lecturer 10 – First Order Logic

Lecturer:

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
First Order Logic

- Syntax
- Semantic
- Inference
  - Resolution
First Order Logic (FOL)

- First Order Logic is about
  - Objects
  - Relations
  - Facts

- The world is made of objects
  - *Objects* are things with individual identities and properties to distinguish them
  - Various *relations* hold among objects. Some of these relations are functional
  - Every fact involving objects and their relations are either *true* or *false*
FOL

- Syntax
- Semantic
- Inference
  - Resolution
FOL Syntax

- **Symbols**
  - Variables: $x, y, z, \ldots$
  - Constants: $a, b, c, \ldots$
  - Function symbols (with arities): $f, g, h, \ldots$
  - Relation symbols (with arities): $p, r, r$
  - Logical connectives: $\neg, \land, \lor, \Rightarrow, \Leftrightarrow$
  - Quantifiers: $\exists, \forall$
FOL Syntax

- Variables, constants and function symbols are used to build terms
  - X, Bill, FatherOf(X), …

- Relations and terms are used to build predicates
  - Tall(FatherOf(Bill)), Odd(X), Married(Tom,Marry), Loves(Y,MotherOf(Y)), …

- Predicates and logical connective are used to build sentences
  - Even(4), ∀ X. Even(X) ⇒ Odd(X+1), ∃ X. X > 0
FOL Syntax

- **Terms**
  - Variables are terms
  - Constants are terms
  - If $t_1, \ldots, t_n$ are terms and $f$ is a function symbol with arity $n$ then $f(t_1, \ldots, t_n)$ is a term
Predicates

- If $t_1, \ldots, t_n$ are terms and $p$ is a relation symbol with arity $n$ then $p(t_1, \ldots, t_n)$ is a predicate
FOL Syntax

- Sentences
  - True, False are sentences
  - Predicates are sentences
  - If $\alpha, \beta$ are sentences then the followings are sentences

\[
\exists x.\alpha, \forall x.\alpha, (\alpha), \neg \alpha, \alpha \land \beta, \alpha \lor \beta, \alpha \Rightarrow \beta, \alpha \Leftrightarrow \beta
\]
FOL Formal grammar

Sentence ::= AtomicS | ComplexS

AtomicS ::= True | False | RelationSymb(Term, . . .) | Term = Term

ComplexS ::= (Sentence) | Sentence Connective Sentence | ¬Sentence |

Quantifier Sentence

Term ::= FunctionSymb(Term, . . .) | ConstantSymb | Variable

Connective ::= ∧ | ∨ | → | ↔

Quantifier ::= ∀ Variable | ∃ Variable

Variable ::= a | b | . . . | x | y | . . .

ConstantSymb ::= A | B | . . . | John | 0 | 1 | . . . | π | . . .

FunctionSymb ::= F | G | . . . | Cosine | Height | FatherOf | + | . . .

RelationSymb ::= P | Q | . . . | Red | Brother | Apple | > | . . .
FOL

- Syntax
- Semantic
- Inference
  - Resolution
FOL Semantic

- Variables
  - Objects
- Constants
  - Entities
- Function symbol
  - Function from objects to objects
- Relation symbol
  - Relation between objects
- Quantifiers
  - $\exists x. P$ true if $P$ is true under some value of $x$
  - $\forall x. P$ true if $P$ is true under every value of $x$
- Logical connectives
  - Similar to Propositional Logic
FOL Semantic

- **Interpretation** \((D, \sigma)\)
  - \(D\) is a set of objects, called *domain* or *universe*
  - \(\sigma\) is a mapping from variables to \(D\)
  - \(C^D\) is a member of \(D\) for each constant \(C\)
  - \(F^D\) is a mapping from \(D^n\) to \(D\) for each function symbol \(F\) with arity \(n\)
  - \(R^D\) is a relation over \(D^n\) for each relation symbol \(R\) with arity \(n\)
Given an interpretation \((D, \sigma)\), semantic of a term/sentence \(\alpha\) is denoted

\[
[\alpha]_a^D
\]

Interpretation of terms

\[
\begin{align*}
[x]_\sigma^K & := \sigma(x) \\
[C]_\sigma^K & := C^K \\
[F(t_1, \ldots, t_n)]_\sigma^K & := F^K([t_1]_\sigma^K, \ldots, [t_n]_\sigma^K)
\end{align*}
\]
FOL Semantic

- Interpretation of sentence

\[
\begin{align*}
[R(t_1, \ldots, t_n)]^D_{\sigma} & := \text{True} \quad \text{iff} \quad \langle [t_1]^D_{\sigma}, \ldots, [t_n]^D_{\sigma} \rangle \in R^D \\
[\neg \varphi]^D_{\sigma} & := \text{True}/\text{False} \quad \text{iff} \quad [\varphi]^D_{\sigma} = \text{False}/\text{True} \\
[\varphi_1 \lor \varphi_2]^D_{\sigma} & := \text{True} \quad \text{iff} \quad [\varphi_1]^D_{\sigma} = \text{True} \text{ or } [\varphi_2]^D_{\sigma} = \text{True} \\
[\exists x \varphi]^D_{\sigma} & := \text{True} \quad \text{iff} \quad [\varphi]^D_{\sigma'} = \text{True} \text{ for some } \sigma' \text{ the} \\
[\varphi_1 \land \varphi_2]^D_{\sigma} & := [\neg (\neg \varphi_1 \lor \neg \varphi_2)]^D_{\sigma} \\
[\varphi_1 \rightarrow \varphi_2]^D_{\sigma} & := [\neg \varphi_1 \lor \varphi_2]^D_{\sigma} \\
[\varphi_1 \leftrightarrow \varphi_2]^D_{\sigma} & := [(\varphi_1 \rightarrow \varphi_2) \land (\varphi_2 \rightarrow \varphi_1)]^D_{\sigma} \\
[\forall x \varphi]^D_{\sigma} & := [\neg \exists x \neg \varphi]^D_{\sigma}
\end{align*}
\]
Example

Symbols
- Variables: x, y, z, …
- Constants: 0, 1, 2, …
- Function symbols: +, *
- Relation symbols: >, =

Semantic
- Universe: N (natural numbers)
- The meaning of symbols
  - Constants: the meaning of 0 is the number zero, …
  - Function symbols: the meaning of + is the natural number addition, …
  - Relation symbols: the meaning of > is the relation greater than, …
FOL Semantic

- **Satisfiability**
  - A sentence $\alpha$ is satisfiable if it is true under some interpretation $(D, \sigma)$

- **Model**
  - An interpretation $(D, \sigma)$ is a model of a sentence $\alpha$ if $\alpha$ is true under $(D, \sigma)$
  - Then we write $(D, \sigma) \models \alpha$

- A sentence is valid if every interpretation is its mode
- A sentence $\alpha$ is valid in $D$ if $(D, \sigma) \models \alpha$ for all $\sigma$
- A sentence is unsatisfiable if it has no model
Example

Consider the universe $N$ of natural numbers

- $\exists x. x + 1 > 5$ is satisfiable
- $\forall x. x + 1 > 0$ is valid in $N$
- $\exists x. 2x + 1 = 6$ is unsatisfiable
Artificial Intelligence
For HEDSPI Project

Lecturer 11 – Inference in First Order Logic

Lecturers:

Le Thanh Huong
Tran Duc Khanh
Dept of Information Systems
Faculty of Information Technology - HUT
First Order Logic

- Syntax
- Semantic
- Inference
  - Resolution
Inference in FOL

- **Difficulties**
  - Quantifiers
  - Infinite sets of terms
  - Infinite sets of sentences

- **Examples:** \( \forall x. King(x) \land Greedy(x) \implies Evil(x) \)
  - Infinite set of instances

\[
\begin{align*}
  King(Bill) \land Greedy(Bill) & \implies Evil(Bill) \\
  King(FatherOf(Bill)) \land Greedy(FatherOf(Bill)) & \implies Evil(FatherOf(Bill)) \\
  & \ldots
\end{align*}
\]
Robinson’s Resolution

- Herbrand’s Theorem (~1930)
  - A set of sentences $S$ is unsatisfiable if and only if there exists a finite subset $S_g$ of the set of all ground instances $Gr(S)$, which is unsatisfiable.

- Herbrand showed that there is a procedure to demonstrate the unsatisfiability of a unsatisfiable set of sentences.

- Robinson propose the Resolution procedure (~1950)
Idea of Resolution

- **Refutation-based procedure**
  - $S \models A$ if and only if $S \cup \{\neg A\}$ is unsatisfiable

- **Resolution procedure**
  - Transform $S \cup \{\neg A\}$ into a set of clauses
  - Apply Resolution rule to find a the empty calause (contradiction)
    - If the empty clause is found
      - Conclude $S \models A$
    - Otherwise
      - No conclusion
A clause is a disjunction of literals, i.e., has the form

\[ P_1 \lor P_2 \lor \ldots \lor P_n \quad P_i \equiv [\neg]R_i \]

Example

\[ P(x) \lor Q(x, a) \lor R(b) \]
\[ P(y) \lor \neg Q(b, y) \lor R(y) \]

The empty clause corresponds to a contradiction

Any sentence can be transformed to an equi-satisfiable set of clauses
Elements of Resolution

- Resolution rule
- Unification
- Transform a sentence to a set of clauses
Elements of Resolution

- Resolution rule
- Unification
- Transform a sentence to a set of clauses
Resolution rule

- Resolution rule

\[
\frac{A \lor B \quad \neg C \lor D}{\theta(A \lor D)} \quad \theta = \text{mgu}(B, C)
\]

- mgu: most general unifier
  - The most general assignment of variables to terms in such a way that two terms are equal
  - Syntactical unification algorithm
Example of Resolution rule

- \(x, y\) are variables
- \(a, b\) are constants

\[
\frac{P(x) \lor Q(x, a) \quad \neg Q(b, y) \lor R(y)}{P(b) \lor R(a)}
\]

\[\theta = \{x = b, y = a\}\]

\[A \equiv P(x)\]
\[B \equiv Q(x, a)\]
\[C \equiv Q(b, y)\]
\[D \equiv R(y)\]
Example of Resolution rule

\[
\frac{\neg \text{Pet}(\text{Joe}) \lor \text{Cat}(\text{Joe}) \lor \text{Bird}(\text{Joe}) \quad \text{Parrot}(x) \lor \neg \text{Bird}(x)}{\neg \text{Pet}(\text{Joe}) \lor \text{Cat}(\text{Joe}) \lor \text{Parrot}(\text{Joe})}
\]

(1) \( \text{mgu}(\text{Bird}(x), \text{Bird}(\text{Joe})) = \{x/\text{Joe}\} \)

\[
\frac{\neg \text{On}(x, y) \lor \text{Above}(x, y) \quad \text{On}(B, A) \lor \text{On}(A, B)}{\text{Above}(A, B) \lor \text{On}(B, A)}
\]

(2) \( \text{mgu}(\text{On}(x, y), \text{On}(A, B)) = \{x/A, y/B\} \)

\[
\frac{\neg \text{Bird}(x) \lor \text{Feathers}(x) \quad \neg \text{Feathers}(y) \lor \text{Flies}(y)}{\neg \text{Bird}(x) \lor \text{Flies}(x)}
\]

(3) \( \text{mgu}(\text{Feathers}(x), \text{Feathers}(y)) = \{y/x\} \)
Elements of Resolution

- Resolution rule
- Unification
- Transform a sentence to a set of clauses
Unification

- **Input**
  - Set of equalities between two terms

- **Output**
  - Most general assignment of variables that satisfies all equalities
  - Fail if no such assignment exists
Unification algorithm

Decompose
\[ U \cup \{ f(t_1, \ldots, t_n) =? f(s_1, \ldots, s_n) \} \longrightarrow U \cup \{ t_1 =? s_1, \ldots, t_n =? s_n \} \]

Orient.
\[ U \cup \{ t =? v \} \longrightarrow U \cup \{ v =? t \} \]

Delete.
\[ U \cup \{ v =? v \} \longrightarrow U \]

Eliminate.
\[ U \cup \{ v =? t \}, \ v \in \mathcal{V}ars(U) \setminus \mathcal{V}ars(t) \longrightarrow U[v/t] \cup \{ v =? t \} \]

Mismatch.
\[ U \cup \{ f(t_1, \ldots, t_m) =? g(s_1, \ldots, s_n) \}, \ f, g \text{ distinct or } m \neq n \longrightarrow \text{FAIL} \]

Occurs.
\[ U \cup \{ v =? t \}, \ v \neq t \text{ but } v \in \mathcal{V}ars(t) \longrightarrow \text{FAIL} \]

- \( \mathcal{V}ars(U), \mathcal{V}ars(t) \) are sets of variables in \( U \) and \( t \)
- \( v \) is a variable
- \( s \) and \( t \) are terms
- \( f \) and \( g \) are function symbols
Example of Unification

\[
\{ F(G(H(y)), H(A)) = \rightleftharpoons F(G(x), x) \} \\
\{ G(H(y)) = \rightleftharpoons G(x), H(A) = \rightleftharpoons x \} \\
\{ H(y) = \rightleftharpoons x, H(A) = \rightleftharpoons x \} \\
\{ x = \rightleftharpoons H(y), H(A) = \rightleftharpoons H(y) \} \\
\{ x = \rightleftharpoons H(y), A = \rightleftharpoons y \} \\
\{ x = \rightleftharpoons H(y), y = \rightleftharpoons A \} \\
\{ x = \rightleftharpoons H(A), y = \rightleftharpoons A \}
\]
Elements of Resolution

- Resolution rule
- Unification
- Transform a sentence to a set of clauses
Transform a sentence to a set of clauses

1. Eliminate implication
2. Move negation inward
3. Standardize variable scope
4. Move quantifiers outward
5. Skolemize existential quantifiers
6. Eliminate universal quantifiers
7. Distribute and, or
8. Flatten and, or
9. Eliminate and
Eliminate implication

\[ \{ \forall x (\forall y P(x, y)) \rightarrow \neg(\forall y Q(x, y) \rightarrow R(x, y)) \} \]

\[
\begin{array}{c}
\alpha \rightarrow \beta & \rightarrow & \neg \alpha \lor \beta \\
\alpha \leftrightarrow \beta & \rightarrow & (\neg \alpha \lor \beta) \land (\neg \beta \lor \alpha)
\end{array}
\]

\[ \{ \forall x \neg(\forall y P(x, y)) \lor \neg(\forall y \neg Q(x, y) \lor R(x, y)) \} \]
Move negation inward

\[ \{ \forall x \neg (\forall y P(x, y)) \lor \neg (\forall y \neg Q(x, y) \lor R(x, y)) \} \]

\[ \begin{array}{ccc}
\neg \neg \alpha & \rightarrow & \alpha \\
\neg (\alpha \lor \beta) & \rightarrow & \neg \alpha \land \neg \beta \\
\neg (\alpha \land \beta) & \rightarrow & \neg \alpha \lor \neg \beta \\
\end{array} \]

\[ \neg \forall v \alpha \rightarrow \exists v \neg \alpha \]

\[ \neg \exists v \alpha \rightarrow \forall v \neg \alpha \]

\[ \{ \forall x (\exists y \neg P(x, y)) \lor (\exists y Q(x, y) \land \neg R(x, y)) \} \]
Standardize variable scope

\[ \{ \forall x (\exists y \neg P(x, y)) \lor (\exists y Q(x, y) \land \neg R(x, y)) \} \]

Each variable for each quantifier

\[ \{ \forall x (\exists y \neg P(x, y)) \lor (\exists z Q(x, z) \land \neg R(x, z)) \} \]
Move quantifiers outward

\[ \{ \forall x (\exists y \neg P(x, y)) \lor (\exists z Q(x, z) \land \neg R(x, z)) \} \]

\[
\begin{array}{c}
(Q x \alpha) \land \beta & \rightarrow & Q x (\alpha \land \beta) & \alpha \land (Q x \beta) & \rightarrow & Q x (\alpha \land \beta) \\
(Q x \alpha) \lor \beta & \rightarrow & Q x (\alpha \lor \beta) & \alpha \lor (Q x \beta) & \rightarrow & Q x (\alpha \lor \beta)
\end{array}
\]

\[ \{ \forall x \exists y \exists z \neg P(x, y) \lor (Q(x, z) \land \neg R(x, z)) \} \]
Skolemize existential quantifiers

\[ \forall x \exists y \exists z \neg P(x, y) \lor (Q(x, z) \land \neg R(x, z)) \]

\[ \exists v \alpha \longrightarrow \alpha[v/\pi(v_1, \ldots, v_n)] \]
with \( \pi \) new and \( v_1, \ldots, v_n \) universally quantified outside \( \exists v \alpha \)

\[ \forall x \neg P(x, F_1(x)) \lor (Q(x, F_2(x)) \land \neg R(x, F_2(x))) \]
Eliminate universal quantifiers

\[ \{ \forall x \neg P(x, F_1(x)) \lor (Q(x, F_2(x)) \land \neg R(x, F_2(x))) \} \]

\[ \forall v \alpha \quad \rightarrow \quad \alpha \]

\[ \{ \neg P(x, F_1(x)) \lor (Q(x, F_2(x)) \land \neg R(x, F_2(x))) \} \]
Distribute and, or

\[ \{ \neg P(x, F_1(x)) \lor (Q(x, F_2(x)) \land \neg R(x, F_2(x))) \} \]

| \(\alpha \lor (\beta \land \gamma)\) | \(\rightarrow\) | \((\alpha \lor \beta) \land (\alpha \lor \gamma)\) |
| \((\beta \land \gamma) \lor \alpha\) | \(\rightarrow\) | \((\beta \lor \alpha) \land (\gamma \lor \alpha)\) |

\[ \{ (\neg P(x, F_1(x)) \lor Q(x, F_2(x))) \land (\neg P(x, F_1(x)) \lor \neg R(x, F_2(x))) \} \]
Flatten and, or

\[\{(\neg P(x, F_1(x)) \lor Q(x, F_2(x))) \land (\neg P(x, F_1(x)) \lor \neg R(x, F_2(x)))\}\]

\[
\begin{align*}
(\alpha \land (\beta \land \gamma)) & \rightarrow (\alpha \land \beta \land \gamma) \\
(\alpha \lor (\beta \lor \gamma)) & \rightarrow (\alpha \lor \beta \lor \gamma) \\
((\alpha \land \beta) \land \gamma) & \rightarrow (\alpha \land \beta \land \gamma) \\
((\alpha \lor \beta) \lor \gamma) & \rightarrow (\alpha \lor \beta \lor \gamma)
\end{align*}
\]

\[\{(\neg P(x, F_1(x)) \lor Q(x, F_2(x))) \land (\neg P(x, F_1(x)) \lor \neg R(x, F_2(x)))\}\]
Eliminate and

\[\{(\neg P(x, F_1(x)) \lor Q(x, F_2(x))) \land (\neg P(x, F_1(x)) \lor \neg R(x, F_2(x)))\}\]

\[\{\alpha \land \beta\} \quad \rightarrow \quad \{\alpha, \beta\}\]

\[\{\neg P(x, F_1(x)) \lor Q(x, F_2(x)), \neg P(x, F_1(x)) \lor \neg R(x, F_2(x))\}\]
Example of proof by Resolution

Prove Criminal(West)

Success
Summary of Resolution

- Refutation-based procedure
  - $S \models A$ if and only if $S \cup \{\neg A\}$ is unsatisfiable

- Resolution procedure
  - Transform $S \cup \{\neg A\}$ into a set of clauses
  - Apply Resolution rule to find a the empty clause (contradiction)
    - If the empty clause is found
      - Conclude $S \models A$
    - Otherwise
      - No conclusion
Summary of Resolution

Theorem

- A set of clauses $S$ is unsatisfiable if and only if upon the input $S$, Resolution procedure finds the empty clause (after a finite time).
Exercice

■ The law says that it is a crime for an American to sell weapons to hostile nations.
■ The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.
■ Is West a criminal?
"... it is a crime for an American to sell weapons to hostile nations":
\[ \forall x, y, z \ \text{American}(x) \land \text{Weapon}(y) \land \text{Nation}(z) \land \text{Hostile}(z) \land \text{Sells}(x, z, y) \Rightarrow \text{Criminal}(x) \]

"Nono ... has some missiles":
\[ \exists x \ \text{Owns}(\text{Nono}, x) \land \text{Missile}(x) \]

"All of its missiles were sold to it by Colonel West":
\[ \forall x \ \text{Owns}(\text{Nono}, x) \land \text{Missile}(x) \Rightarrow \text{Sells}(\text{West}, \text{Nono}, x) \]

We will also need to know that missiles are weapons:
\[ \forall x \ \text{Missile}(x) \Rightarrow \text{Weapon}(x) \]

and that an enemy of America counts as "hostile":
\[ \forall x \ \text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x) \]

"West, who is American ...":
\[ \text{American}(\text{West}) \]

"The country Nono ...":
\[ \text{Nation}(\text{Nono}) \]

"Nono, an enemy of America ...":
\[ \text{Enemy}(\text{Nono}, \text{America}) \]
\[ \text{Nation}(\text{America}) \]
Transform the problem to set of clauses and Resolution

\[ \neg \text{American}(x) \lor \neg \text{Weapon}(y) \lor \neg \text{Sells}(x,y,z) \lor \neg \text{Hostile}(z) \lor \text{Criminal}(x) \]

Prove \text{Criminal}(\text{West})
Exercice

- Jack owns a dog
- Every dog owner is an animal lover
- No animal lover kills an animal
- Either Jack or Curiosity killed the cat, who is named Tuna
- Did Curiosity kill the cat?
Modeling

\[\exists x. \text{Dog}(x) \land \text{Owns}(Jack, x)\]
\[\forall x \forall y. (\text{Dog}(y) \land \text{Owns}(x, y)) \Rightarrow \text{AnimalLover}(x)\]
\[\forall x \forall y. (\text{AnimalLover}(x) \land \text{Animal}(y) \Rightarrow \neg \text{Kills}(x, y))\]
\[\text{Kills}(Jack, Tuna) \lor \text{Kill}(Curiosity, Tuna)\]
\[\text{Cat}(Tuna)\]
\[\forall x. \text{Cat}(x) \Rightarrow \text{Animal}(x)\]
Transform the problem to set of clauses

\[ \text{Dog}(D) \]
\[ \text{Owns}(Jack, D) \]
\[ \lnot \text{Dog}(y) \lor \lnot \text{Owns}(x, y) \lor \text{AnimalLover}(x) \]
\[ \lnot \text{AnimalLover}(x) \land \lnot \text{Animal}(y) \lor \lnot \text{Kills}(x, y) \]
\[ \text{Kills}(Jack, Tuna) \lor \text{Kill}(Curiosity, Tuna) \]
\[ \text{Cat}(Tuna) \]
\[ \lnot \text{Cat}(x) \lor \text{Animal}(x) \]
Artificial Intelligence

For HEDSPI Project

Lecturer 12 - Planning

Lecturer:

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Outline

- Planning problem
- State-space search
- Partial-order planning
- Planning graphs
- Planning with propositional logic
Search vs. planning

- Consider the task *get milk, bananas, and a cordless drill*
- Standard search algorithms seem to fail miserably:

  - After-the-fact heuristic/goal test inadequate
Planning problem

- Planning is the task of determining a sequence of actions that will achieve a goal.

- Domain independent heuristics and strategies must be based on a domain independent representation
  - General planning algorithms require a way to represent states, actions and goals
  - STRIPS, ADL, PDDL are languages based on propositional or first-order logic

- Classical planning environment: fully observable, deterministic, finite, static and discrete.
Additional complexities

- Because the world is …
  - Dynamic
  - Stochastic
  - Partially observable
- And because actions
  - take time
  - have continuous effects
Focus on classical planning; assume none of the above.

Deterministic, static, fully observable:
- "Basic"
- Most of the recent progress
- Ideas often also useful for more complex problems
Problem Representation

- State
  - What is true about the (hypothesized) world?
- Goal
  - What must be true in the final state of the world?
- Actions
  - What can be done to change the world?
  - Preconditions and effects

- We’ll represent all these as logical predicates
STRIPS operators

- Tidily arranged actions descriptions, restricted language

- Action: $\textit{Buy}(x)$
  - Precondition: $\textit{At}(p); \textit{Sells}(p; x)$
  - Effect: $\textit{Have}(x)$

- [Note: this abstracts away many important details!]

- Restricted language $\Rightarrow$ efficient algorithm
  - Precondition: conjunction of positive literals
  - Effect: conjunction of literals

- A complete set of STRIPS operators can be translated into a set of successor-state axioms
Example: blocks world

- On(b, x): block b is on x, x is another block or Table.
- MoveToTable(b, x): move block b from the top of x to Table
- Move(b, x, y): move block b from the top of x to the top of y
- Clear(x): nothing is on x

- Action (Move (b, x, y) ,
  - PRECOND: On(b, x) \land Clear(b) \land Clear(y),
  - EFFECT: On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y)) .
- Action(MoveToTable(b, x) ,
  - PRECOND: On(b, x) \land Clear(b),
  - EFFECT: On(b, Table) \land Clear(x) \land \neg On(b, x )) .

- Initial state: On(A,Table), On(C,A), On(B,Table), Clear(B), Clear(C)
- Goal: On(A,B), On(B,C)
Planning with state-space search

- Both forward and backward search possible
- Progression planners
  - forward state-space search
  - consider the effect of all possible actions in a given state
- Regression planners
  - backward state-space search
  - Determine what must have been true in the previous state in order to achieve the current state
Progression and regression

(a) initial state

At(P₁, A)  
At(P₂, A)

Fly(P₁,A,B)  
Fly(P₂,A,B)

At(P₁, B)  
At(P₂, A)

(b) goal

At(P₁, A)  
At(P₂, B)

Fly(P₁,A,B)  
Fly(P₂,A,B)

At(P₁, B)  
At(P₂, B)
Progression algorithm

- Formulation as state-space search problem:
  - Initial state and goal test: obvious
  - Successor function: generate from applicable actions
  - Step cost = each action costs 1
- Any complete graph search algorithm is a complete planning algorithm.
  - E.g. A*
- Inherently inefficient:
  - (1) irrelevant actions lead to very broad search tree
  - (2) good heuristic required for efficient search
Forward Search Methods: can use A* with some h and g
Regression algorithm

How to determine predecessors?
- What are the states from which applying a given action leads to the goal?
  Goal state = $At(C1, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$
  Relevant action for first conjunct: $Unload(C1,p,B)$
  Works only if pre-conditions are satisfied.
  Previous state = $In(C1, p) \wedge At(p, B) \wedge At(C2, B) \wedge \ldots \wedge At(C20, B)$
  Subgoal $At(C1,B)$ should not be present in this state.

- Actions must not undo desired literals (consistent)
- Main advantage: only relevant actions are considered.
  - Often much lower branching factor than forward search.
Regression algorithm

- General process for predecessor construction
  - Give a goal description G
  - Let A be an action that is relevant and consistent
  - The predecessors are as follows:
    - Any positive effects of A that appear in G are deleted.
    - Each precondition literal of A is added, unless it already appears.
- Any standard search algorithm can be added to perform the search.
- Termination when predecessor satisfied by initial state.
  - In FO case, satisfaction might require a substitution.
Backward search methods

Regressing a ground operator

Continue until a subgoal is produced that is satisfied by current world state
Regressing an ungrounded operator

Because $A$ cannot be on itself

Because $\text{On}(B, C)$ and $\text{On}(A, C)$ cannot both be true

Because we are moving $A$ from somewhere else to $B$
Example of Backward Search

Instantiate rules:
F1/x, F1/y

This goal is satisfied by current state description
Heuristics for state-space search

- Use relax problem idea to get lower bounds on least number of actions to the goal.
  - Remove all or some preconditions

- **Subgoal independence**: the cost of solving a set of subgoals equal the sum cost of solving each one independently.
  - Can be pessimistic (interacting subplans)
  - Can be optimistic (negative effects)

- Simple: number of unsatisfied subgoals.

- Various ideas related to removing negative effects or positive effects.
Partial order planning

- Least commitment planning
- Nonlinear planning
- Search in the space of partial plans
- A state is a partial incomplete partially ordered plan
- Operators transform plans to other plans by:
  - Adding steps
  - Reordering
  - Grounding variables
- SNLP: Systematic Nonlinear Planning (McAllester and Rosenblitt 1991)
- NONLIN (Tate 1977)
A partial order plan for putting shoes and socks

**Figure 11.6** A partial-order plan for putting on shoes and socks, and the six corresponding linearizations into total-order plans.
Partial-order planning

- Partially ordered collection of steps with
  - Start step has the initial state description as its effect
  - Finish step has the goal description as its precondition
  - Causal links from outcome of one step to precondition of another temporal ordering between pairs of steps

- Open condition = precondition of a step not yet causally linked

- A plan is complete iff every precondition is achieved

- A precondition is achieved iff it is the effect of an earlier step and no possibly intervening step undoes it
Example

Start

At(Home) Sells(HWS,Drill) Sells(SM,Milk) Sells(SM,Ban.)

Have(Milk) At(Home) Have(Ban.) Have(Drill)

Finish
Example

Start

At(Home)  Sells(HWS,Drill)  Sells(SM,Milk)  Sells(SM,Ban.)

At(HWS)  Sells(HWS,Drill)

Buy(Drill)

At(x)

Go(SM)

At(SM)  Sells(SM,Milk)

Buy(Milk)

Have(Milk)  At(Home)  Have(Ban.)  Have(Drill)

Finish
Example
Planning process

- Operators on partial plans:
  - add a link from an existing action to an open condition
  - add a step to fulfill an open condition
  - order one step wrt another to remove possible conflicts

- Gradually move from incomplete/vague plans to complete, correct plans

- Backtrack if an open condition is unachievable or if a conflict is unresolvable
POP algorithm sketch

function POP(initial, goal, operators) returns plan

plan ← Make-Minimal-Plan(initial, goal)

loop do
    if Solution?(plan) then return plan
    S_{need}, c ← Select-Subgoal(plan)
    Choose-Operator(plan, operators, S_{need}, c)
    Resolve-Threats(plan)
end

function Select-Subgoal(plan) returns S_{need}, c

pick a plan step S_{need} from Steps(plan)

    with a precondition c that has not been achieved

return S_{need}, c
procedure Choose-Operator\(\text{plan, operators, } S_{\text{need}}, \ c\) 

choose a step \(S_{\text{add}}\) from \(\text{operators}\) or \(\text{Steps( plan)}\) that has \(c\) as an effect

if there is no such step then fail

add the causal link \(S_{\text{add}} \rightarrow c\) \(S_{\text{need}}\) to \(\text{Links( plan)}\)

add the ordering constraint \(S_{\text{add}} < S_{\text{need}}\) to \(\text{Orderings( plan)}\)

if \(S_{\text{add}}\) is a newly added step from \(\text{operators}\) then

add \(S_{\text{add}}\) to \(\text{Steps( plan)}\)

add \(\text{Start} < S_{\text{add}} < \text{Finish}\) to \(\text{Orderings( plan)}\)

procedure Resolve-Threats\(\text{plan}\)

for each \(S_{\text{threat}}\) that threatens a link \(S_i \rightarrow c\) \(S_j\) in \(\text{Links( plan)}\) do

choose either

\textit{Demotion:} Add \(S_{\text{threat}} < S_i\) to \(\text{Orderings( plan)}\)

\textit{Promotion:} Add \(S_j < S_{\text{threat}}\) to \(\text{Orderings( plan)}\)

if not \(\text{Consistent( plan)}\) then fail

end
Clobbering and promotion/demotion

- A clobberer is a potentially intervening step that destroys the condition achieved by a causal link. E.g., \textit{Go(Home)} clobbers \textit{At(Supermarket)}:

Demotion: put before \textit{Go(Supermarket)}

Promotion: put after \textit{Buy(Milk)}
Properties of POP

- Nondeterministic algorithm: backtracks at choice points on failure
  - choice of $S_{\text{add}}$ to achieve $S_{\text{need}}$
  - choice of demotion or promotion for clobberer
  - selection of $S_{\text{need}}$ is irrevocable

- POP is sound, complete, and systematic (no repetition)

- Extensions for disjunction, universals, negation, conditionals

- Can be made efficient with good heuristics derived from problem description

- Particularly good for problems with many loosely related subgoals
Example: Blocks world

"Sussman anomaly" problem

Start State

\[
\begin{align*}
\text{Clear}(x) & \quad \text{On}(x,z) \quad \text{Clear}(y) \\
\text{PutOn}(x,y) & \\
\sim\text{On}(x,z) & \quad \sim\text{Clear}(y) \\
\sim\text{Clear}(z) & \quad \text{On}(x,y)
\end{align*}
\]

Goal State

\[
\begin{align*}
\text{Clear}(x) & \quad \text{On}(x,z) \\
\text{PutOnTable}(x) & \\
\sim\text{On}(x,z) & \quad \text{Clear}(z) \quad \text{On}(x,\text{Table})
\end{align*}
\]

+ several inequality constraints
Example: Blocks world

On(C,A) On(A,Table) Cl(B) On(B,Table) Cl(C)

On(A,B) On(B,C)

FINISH
Example: Blocks world
Example: Blocks world
Example: Blocks world
Planning Graphs

- A planning graph consists of a sequence of levels that correspond to time-steps in the plan.
- Level 0 is the initial state.
- Each level contains a set of literals and a set of actions.
- Literals are those that could be true at the time step.
- Actions are those that their preconditions could be satisfied at the time step.
- Works only for propositional planning.
Example: Have cake and eat it too

\begin{verbatim}
Init(Have(Cake))
Goal(Have(Cake) \& Eaten(Cake))
Action(Eat(Cake))
    PRECOND: Have(Cake)
    EFFECT: \neg Have(Cake) \& Eaten(Cake)
Action(Bake(Cake))
    PRECOND: \neg Have(Cake)
    EFFECT: Have(Cake)
\end{verbatim}

**Figure 11.11** The "have cake and eat cake too" problem.
The Planning graphs for “have cake”,

- Persistence actions: Represent “inactions” by boxes: frame axiom
- Mutual exclusions (mutex) are represented between literals and actions.
- S1 represents multiple states
- Continue until two levels are identical. The graph levels off.
- The graph records the impossibility of certain choices using mutex links.
- Complexity of graph generation: polynomial in number of literals.

![Planning Graph Diagram]

**Figure 11.12** The planning graph for the “have cake and eat cake too” problem up to level $S_2$. Rectangles indicate actions (small squares indicate persistence actions) and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines.
Defining Mutex relations

- A mutex relation holds between two actions on the same level iff any of the following holds:
  - **Inconsistency effect:** one action negates the effect of another. Example “eat cake and persistence of have cake”
  - **Interference:** One of the effect of one action is the negation of the precondition of the other. Example: eat cake and persistence of Have cake
  - **Competing needs:** one of the preconditions of one action is mutually exclusive with a precondition of another. Example: Bake(cake) and Eat(Cake).
  - A mutex relation holds between 2 literals at the same level iff one is the negation of the other or if each possible pair of actions that can achieve the 2 literals is mutually exclusive.
Planning graphs for heuristic estimation

- Estimate the cost of achieving a goal by the level in the planning graph where it appears.
- To estimate the cost of a conjunction of goals use one of the following:
  - Max-level: take the maximum level of any goal (admissible)
  - Sum-cost: Take the sum of levels (inadmissible)
  - Set-level: find the level where they all appear without Mutex
- Graph plans are relaxation of the problem.
- Representing more than pair-wise mutex is not cost-effective
The graphplan algorithm

function graphplan(problem) returns solution or failure

\[
\begin{align*}
\text{graph} & \leftarrow \text{INITIAL-PLANNING-GRAPH}(\text{problem}) \\
\text{goals} & \leftarrow \text{GOALS}[\text{problem}] \\
\text{loop do} & \\
\text{if goals all non-mutex in last level of graph then do} & \\
\quad \text{solution} & \leftarrow \text{EXTRACT-SOLUTION}(\text{graph}, \text{goals}, \text{LENGTH}(\text{graph})) \\
\quad \text{if solution} \neq \text{failure then return solution} & \\
\quad \text{else if \text{NO-SOLUTION-POSSIBLE}(\text{graph}) then return failure} & \\
\quad \text{graph} & \leftarrow \text{EXPAND-GRAPH}(\text{graph}, \text{problem})
\end{align*}
\]

**Figure 11.13** The graphplan algorithm. graphplan alternates between a solution extraction step and a graph expansion step. extract-solution looks for whether a plan can be found, starting at the end and searching backwards. expand-graph adds the actions for the current level and the state literals for the next level.
Planning graph for spare tire a S2

goal: at(spare, axle)

- S2 has all goals and no mutex so we can try to extract solutions
- Use either CSP algorithm with actions as variables
- Or search backwards

**Figure 11.14** The planning graph for the spare tire problem after expansion to level $S_2$. Mutex links are shown as gray lines. Only some representative mutexes are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.
Search planning-graph backwards with heuristics

- How to choose an action during backwards search:
- Use greedy algorithm based on the level cost of the literals.
- For any set of goals:
  1. Pick first the literal with the highest level cost.
  2. To achieve the literal, choose the action with the easiest preconditions first (based on sum or max level of precond literals).
Properties of planning graphs; termination

- Literals increase monotonically
  - Once a literal is in a level it will persist to the next level

- Actions increase monotonically
  - Since the precondition of an action was satisfied at a level and literals persist the action’s precond will be satisfied from now on

- Mutexes decrease monotonically:
  - If two actions are mutex at level Si, they will be mutex at all previous levels at which they both appear

- Because literals increase and mutex decrease it is guaranteed that we will have a level where all goals are non-mutex
Planning with propositional logic

- Express propositional planning as a set of propositions.
- Index propositions with time steps:
  - $\text{On}(A,B)_0$, $\text{ON}(B,C)_0$
- Goal conditions: the goal conjuncts at time $T$, $T$ is determined arbitrarily.
- Unknown propositions are not stated.
- Propositions known not to be true are stated negatively.
- Actions: a proposition for each action for each time slot.
- Successor state axioms need to be expressed for each action (like in the situation calculus but it is propositional)
We write the formula:
- Initial state and successor state axioms and goal

We search for a model to the formula. Those actions that are assigned true constitute a plan.

To have a single plan we may have a mutual exclusion for all actions in the same time slot.

We can also choose to allow partial order plans and only write exclusions between actions that interfere with each other.

Planning: iteratively try to find longer and longer plans.
**SATplan algorithm**

```
function SATPLAN(problem, T_max) returns solution or failure
  inputs: problem, a planning problem
         T_max, an upper limit for plan length

  for T = 0 to T_max do
    cnf, mapping ← TRANSLATE-TO-SAT(problem, T)
    assignment ← SAT-SOLVER(cnf)
    if assignment is not null then
      return EXTRACT-SOLUTION(assignment, mapping)
  return failure
```

**Figure 11.15** The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step T and axioms are included for each time step up to T. (Details of the translation are given in the text.) If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned true in the model. If no model exists, then the process is repeated with the goal moved one step later.
Complexity of satplan

- The total number of action symbols is:
  - \(|T| \times |Act| \times |O|^p\)
  - \(O\) = number of objects, \(p\) is scope of atoms.
- Number of clauses is higher.
- Example: 10 time steps, 12 planes, 30 airports, the complete action exclusion axiom has 583 million clauses.
Artificial Intelligence

For HEDSPI Project

Lecturer 13 – Machine Learning

Lecturer:
Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Introduction of Machine learning

- Definitions of Machine learning…
  - A process by which a system improves its performance [Simon, 1983]
  - Any computer program that improves its performance at some task through experience [Mitchell, 1997]
  - Programming computers to optimize a performance criterion using example data or past experience [Alpaydin, 2004]

- Representation of the learning problem [Mitchell, 1997]
  Learning = Improving with experience at some task
  - Improve over task \( T \)
  - With respect to performance measure \( P \)
  - Based on experience \( E \)
Application examples of ML (1)

Web pages filtering problem
- **T**: to predict which Web pages a given user is interested in
- **P**: % of Web pages correctly predicted
- **E**: a set of Web pages identified as interested/uninterested for the user

Web pages categorization problem
- **T**: to categorize Web pages in predefined categories
- **P**: % of Web pages correctly categorized
- **E**: a set of Web pages with specified categories

Which cat.?
Application examples of ML (2)

Handwriting recognition problem
- **T**: to recognize and classify handwritten words within images
- **P**: % of words correctly classified
- **E**: a database of handwritten words with given classifications (i.e., labels)

Robot driving problem
- **T**: to drive on public highways using vision sensors
- **P**: average distance traveled before an error (as judged by human overseer)
- **E**: a sequence of images and steering commands recorded while observing a human driver

Which word?
- we
- do
- in
- the
- right
- way

Which steering command?
- Go straight
- Move left
- Move right
- Slow down
- Speed up
Key elements of a ML problem (1)

- **Selection of the training examples**
  - Direct or indirect training feedback
  - With teacher (i.e., with labels) or without
  - The training examples set should be representative of the future test examples

- **Choosing the target function (a.k.a. hypothesis, concept, etc.)**
  - $F: X \rightarrow \{0,1\}$
  - $F: X \rightarrow$ a set of labels
  - $F: X \rightarrow \mathbb{R}^+$ (i.e., the positive real numbers domain)
  - ...
  - ...
Key elements of a ML problem (2)

- Choosing a representation of the target function
  - A polynomial function
  - A set of rules
  - A decision tree
  - A neural network
  - ...

- Choosing a learning algorithm that learns (approximately) the target function
  - Regression-based
  - Rule induction
  - ID3 or C4.5
  - Back-propagation
  - ...
Issues in Machine Learning (1)

- **Learning algorithm**
  - What algorithms can approximate the target function?
  - Under which conditions does a selected algorithm converge (approximately) to the target function?
  - For a certain problem domain and given a representation of examples which algorithm performs best?

- **Training examples**
  - How many training examples are sufficient?
  - How does the size of the training set influence the accuracy of the learned target function?
  - How does noise and/or missing-value data influence the accuracy?
Issues in Machine Learning (2)

- **Learning process**
  - What is the best strategy for selecting a next training example? How do selection strategies alter the complexity of the learning problem?
  - How can prior knowledge (held by the system) help?

- **Learning capability**
  - What target function should the system learn?
    - Representation of the target function: expressiveness vs. complexity
  - What are the theoretical limits of learnability?
  - How can the system generalize from the training examples?
    - To avoid the overfitting problem
  - How can the system automatically alter its representation?
    - To improve its ability to represent and learn the target function
Artificial Intelligence

For HEDSPI Project

Lecturer 13 – Decision Tree Learning

Lecturer:

Tran Duc Khanh

Dept of Information Systems
School of Information and Communication Technology
HUST
Decision tree – Introduction

- Decision tree (DT) learning
  - To approximate a \textit{discrete-valued target function}
  - The target function is represented by a decision tree

- A DT can be represented (interpreted) as a set of IF-THEN rules (i.e., easy to read and understand)

- Capable of learning disjunctive expressions

- DT learning is robust to noisy data

- One of the most widely used methods for inductive inference

- Successfully applied to a range of real-world applications
Example of a DT: Which documents are of my interest?

- (…,"sport",…,"player",…) → Interested
- (…,"goal",…) → Interested
- (…,"sport",…) → Uninterested
Example of a DT: Does a person play tennis?

- (Outlook=Overcast, Temperature=Hot, Humidity=High, Wind=Weak) → Yes
- (Outlook=Rain, Temperature=Mild, Humidity=High, Wind=Strong) → No
- (Outlook=Sunny, Temperature=Hot, Humidity=High, Wind=Strong) → No
Each *internal node* represents an *attribute to be tested* by instances

Each *branch* from a node corresponds to a *possible value of the attribute* associated with that node

Each *leaf node* represents a *classification* (e.g., a class label)

A learned DT classifies an instance by sorting it down the tree, from the root to some leaf node

→ The classification associated with the leaf node is used for the instance
Decision tree – Representation (2)

- A DT represents a disjunction of conjunctions of constraints on the attribute values of instances
- Each path from the root to a leaf corresponds to a conjunction of attribute tests
- The tree itself is a disjunction of these conjunctions
- Examples
  - Let’s consider the two previous example DTs…
Which documents are of my interest?

\[
\begin{align*}
&\text{is present} \quad \text{is absent} \\
\text{“sport”?} & \quad \text{“player”?} \\
& \quad \text{“football”?} \\
& \quad \text{“goal”?} \\
& \text{Interested} \quad \text{Uninterested} \\
& \text{Interested} \\
& \text{Interested} \quad \text{Uninterested} \\
& \text{Interested} \\
& \quad \text{Uninterested}
\end{align*}
\]

\[
\begin{align*}
&\left[\text{“sport” is present}\right] \land \left[\text{“player” is present}\right] \lor \\
&\left[\text{“sport” is absent}\right] \land \left[\text{“football” is present}\right] \lor \\
&\left[\text{“sport” is absent}\right] \land \left[\text{“football” is absent}\right] \land \left[\text{“goal” is present}\right]
\end{align*}
\]
Does a person play tennis?

\[(\text{Outlook}=\text{Sunny}) \land (\text{Humidity}=\text{Normal})\] \lor 
(\text{Outlook}=\text{Overcast}) \lor 
[(\text{Outlook}=\text{Rain}) \land (\text{Wind}=\text{Weak})]\]
Decision tree learning – ID3 algorithm

ID3_alg(Training_Set, Class_Labels, Attributes)

Create a node Root for the tree
If all instances in Training_Set have the same class label \( c \), Return the tree of the single-node Root associated with class label \( c \)
If the set Attributes is empty, Return the tree of the single-node Root associated with class label \( \equiv \text{Majority_Class_Label}(\text{Training_Set}) \)

\( A \leftarrow \text{The attribute in Attributes that “best” classifies Training_Set} \)

The test attribute for node Root \( \leftarrow A \)

For each possible value \( v \) of attribute \( A \)

Add a new tree branch under Root, corresponding to the test: “value of attribute \( A \) is \( v \)”
Compute \( \text{Training_Set}_v = \{ \text{instance } x \mid x \subseteq \text{Training_Set}, x_A = v \} \)
If (Training_Set_v is empty) Then

Create a leaf node with class label \( \equiv \text{Majority_Class_Label}(\text{Training_Set}) \)
Attach the leaf node to the new branch
Else Attach to the new branch the sub-tree ID3_alg(Training_Set_v, Class_Labels, \{Attributes \( \setminus A\})

Return Root
ID3 algorithm – Intuitive idea

- Perform a greedy search through the space of possible DTs
- Construct (i.e., learn) a DT in a top-down fashion, starting from its root node
- At each node, the test attribute is the one (of the candidate attributes) that best classifies the training instances associated with the node
- A descendant (sub-tree) of the node is created for each possible value of the test attribute, and the training instances are sorted to the appropriate descendant node
- Every attribute can appear at most once along any path of the tree
- The tree growing process continues
  - Until the (learned) DT perfectly classifies the training instances, or
  - Until all the attributes have been used
Selection of the test attribute

- A very important task in DT learning: at each node, how to choose the test attribute?
- To select the attribute that is most useful for classifying the training instances associated with the node
- How to measure an attribute’s capability of separating the training instances according to their target classification
  → Use a statistical measure – Information Gain
- Example: A two-class \((c_1, c_2)\) classification problem
  → Which attribute, \(A_1\) or \(A_2\), should be chosen to be the test attribute?
Entropy

- A commonly used measure in the Information Theory field
- To measure the impurity (inhomogeneity) of a set of instances
- The entropy of a set $S$ relative to a $c$-class classification

$$\text{Entropy}(S) = \sum_{i=1}^{c} - p_i \cdot \log_2 p_i$$

where $p_i$ is the proportion of instances in $S$ belonging to class $i$, and $0 \cdot \log_2 0 = 0$

- The entropy of a set $S$ relative to a two-class classification

$$\text{Entropy}(S) = -p_1 \cdot \log_2 p_1 - p_2 \cdot \log_2 p_2$$

- Interpretation of entropy (in the Information Theory field)
  - The entropy of $S$ specifies the expected number of bits needed to encode class of a member randomly drawn out of $S$
    - Optical length code assigns $-\log_2 p$ bits to message having probability $p$
    - The expected number of bits needed to encode a class: $p \cdot \log_2 p$
Entropy – Two-class example

- $S$ contains 14 instances, where 9 belongs to class $c_1$ and 5 to class $c_2$
- The entropy of $S$ relative to the two-class classification:
  \[
  \text{Entropy}(S) = -(9/14) \cdot \log_2(9/14) - (5/14) \cdot \log_2(5/14) \approx 0.94
  \]
- Entropy = 0, if all the instances belong to the same class (either $c_1$ or $c_2$)
  → Need 0 bit for encoding (no message need be sent)
- Entropy = 1, if the set contains equal numbers of $c_1$ and $c_2$ instances
  → Need 1 bit per message for encoding (whether $c_1$ or $c_2$)
- Entropy = some value in (0,1), if the set contains unequal numbers of $c_1$ and $c_2$ instances
  → Need on average <1 bit per message for encoding
Information gain

- Information gain of an attribute relative to a set of instances is
  - the expected reduction in entropy
  - caused by partitioning the instances according to the attribute

- Information gain of attribute $A$ relative to set $S$

$$Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$

where $Values(A)$ is the set of possible values of attribute $A$, and

$S_v = \{x \mid x \in S, x_A = v\}$

- In the above formula, the second term is the expected value of the entropy after $S$ is partitioned by the values of attribute $A$

- Interpretation of $Gain(S, A)$: The number of bits saved (reduced) for encoding class of a randomly drawn member of $S$, by knowing the value of attribute $A$
Let’s consider the following dataset (of a person) $S$:

<table>
<thead>
<tr>
<th>Day</th>
<th>Outlook</th>
<th>Temperature</th>
<th>Humidity</th>
<th>Wind</th>
<th>Play Tennis</th>
</tr>
</thead>
<tbody>
<tr>
<td>D1</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>D2</td>
<td>Sunny</td>
<td>Hot</td>
<td>High</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>D3</td>
<td>Overcast</td>
<td>Hot</td>
<td>High</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D4</td>
<td>Rain</td>
<td>Mild</td>
<td>High</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D5</td>
<td>Rain</td>
<td>Cool</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D6</td>
<td>Rain</td>
<td>Cool</td>
<td>Normal</td>
<td>Strong</td>
<td>No</td>
</tr>
<tr>
<td>D7</td>
<td>Overcast</td>
<td>Cool</td>
<td>Normal</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D8</td>
<td>Sunny</td>
<td>Mild</td>
<td>High</td>
<td>Weak</td>
<td>No</td>
</tr>
<tr>
<td>D9</td>
<td>Sunny</td>
<td>Cool</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D10</td>
<td>Rain</td>
<td>Mild</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D11</td>
<td>Sunny</td>
<td>Mild</td>
<td>Normal</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D12</td>
<td>Overcast</td>
<td>Mild</td>
<td>High</td>
<td>Strong</td>
<td>Yes</td>
</tr>
<tr>
<td>D13</td>
<td>Overcast</td>
<td>Hot</td>
<td>Normal</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>D14</td>
<td>Rain</td>
<td>Mild</td>
<td>High</td>
<td>Strong</td>
<td>No</td>
</tr>
</tbody>
</table>

[Mitchell, 1997]
Information gain – Example

- What is the information gain of attribute Wind relative to the training set $S$ – $\text{Gain}(S, \text{Wind})$?
- Attribute Wind have two possible values: Weak and Strong
- $S = \{9$ positive and $5$ negative instances$\}$
- $S_{\text{weak}} = \{6$ pos. and $2$ neg. instances having $\text{Wind}=\text{Weak}\}$
- $S_{\text{strong}} = \{3$ pos. and $3$ neg. instances having $\text{Wind}=\text{Strong}\}$

\[
\text{Gain}(S, \text{Wind}) = \text{Entropy}(S) - \sum_{v \in \{\text{Weak}, \text{Strong}\}} \frac{|S_v|}{|S|} \text{Entropy}(S_v)
\]

\[
= \text{Entropy}(S) - \left(\frac{8}{14}\right) \text{Entropy}(S_{\text{weak}}) - \left(\frac{6}{14}\right) \text{Entropy}(S_{\text{strong}})
\]

\[
= 0.94 - \left(\frac{8}{14}\right) \times (0.81) - \left(\frac{6}{14}\right) \times (1) = 0.048
\]
Decision tree learning – Example (1)

- At the root node, which attribute of \{Outlook, Temperature, Humidity, Wind\} should be the test attribute?
  
  - \( \text{Gain}(S, \text{Outlook}) = \ldots = 0.246 \)
  
  - \( \text{Gain}(S, \text{Temperature}) = \ldots = 0.029 \)
  
  - \( \text{Gain}(S, \text{Humidity}) = \ldots = 0.151 \)
  
  - \( \text{Gain}(S, \text{Wind}) = \ldots = 0.048 \)

→ So, **Outlook** is chosen as the test attribute for the root node!

\[
\text{Outlook}=?, \quad S = \{9+, 5-\}
\]

- Sunny: \( S_{\text{Sunny}} = \{2+, 3-\} \)
- Overcast: \( S_{\text{Overcast}} = \{4+, 0-\} \)
- Rain: \( S_{\text{Rain}} = \{3+, 2-\} \)
Decision tree learning – Example (2)

- At Node1, which attribute of \{Temperature, Humidity, Wind\} should be the test attribute?

  **Note!** Attribute Outlook is excluded, since it has been used by Node1’s parent (i.e., the root node)

  - Gain(S\text{Sunny}, Temperature) = \ldots = 0.57
  - Gain(S\text{Sunny}, Humidity) = \ldots = 0.97
  - Gain(S\text{Sunny}, Wind) = \ldots = 0.019

  So, Humidity is chosen as the test attribute for Node1!
DT learning – Hypothesis space search (1)

- ID3 searches in a space of hypotheses (i.e., of possible DTs) for one that fits the training instances.

- ID3 performs a simple-to-complex, hill-climbing search, beginning with the empty tree.

- The hill-climbing search is guided by an evaluation metric – the information gain measure.

- ID3 searches only one (rather than all possible) DT consistent with the training instances.
DT learning – Hypothesis space search (2)

- ID3 does not perform backtracking in its search
  - Guaranteed to converge to a locally (but not the globally) optimal solution
  - Once an attribute is selected as the test for a node, ID3 never backtracks to reconsider this choice

- At each step in the search, ID3 uses a statistical measure of all the instances (i.e., information gain) to refine its current hypothesis
  - The resulting search is much less sensitive to errors in individual training instances
Inductive bias in DT learning (1)

- Both the two DTs below are consistent with the given training dataset
- So, which one is preferred (i.e., selected) by the ID3 algorithm?
Inductive bias in DT learning (2)

- Given a set of training instances, there may be many DTs consistent with these training instances.
- So, which of these candidate DTs should be chosen?
- ID3 chooses the first acceptable DT it encounters in its simple-to-complex, hill-climbing search.
  - Recall that ID3 searches incompletely through the hypothesis space (i.e., without backtracking).
- ID3’s search strategy:
  - Select in favor of shorter trees over longer ones.
  - Select trees that place the attributes with highest information gain closest to the root node.
Issues in DT learning

- Over-fitting the training data
- Handling continuous-valued (i.e., real-valued) attributes
- Choosing appropriate measures for attribute selection
- Handling training data with missing attribute values
- Handling attributes with differing costs

→ An extension of the ID3 algorithm with the above mentioned issues resolved results in the C4.5 algorithm
Artificial Intelligence

For HEDSPI Project

Lecturer 14 – Reinforcement Learning

Lecturer:

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Reinforcement Learning (RL)

- RL is ML method that optimize the reward
  - A class of tasks
  - A process of trial-and-error learning
    - Good actions are “rewarded”
    - Bad actions are “punished”
Features of RL

- Learning from numerical rewards
- Interaction with the task; sequences of states, actions and rewards
- Uncertainty and non-deterministic worlds
- Delayed consequences
- The explore/exploit dilemma
- The whole problem of goal-directed learning
Points of view

- From the point of view of agents
  - RL is a process of trial-and-error learning
  - How much reward will I get if I do this action?

- From the point of view of trainers
  - RL is training by rewards and punishments
  - Train computers like we train animals
Applications of RL

- Robot
- Animal training
- Scheduling
- Games
- Control systems
- ...
Supervised Learning vs. Reinforcement Learning

- **Supervised learning**
  - Teacher: Is this an AI course or a Math course?
  - Learner: Math
  - Teacher: No, AI
  - …
  - Teacher: Is this an AI course or a Math course?
  - Learner: AI
  - Teacher: Yes

- **Reinforcement learning**
  - World: You are in state 9. Choose action A or B
  - Learner: A
  - World: Your reward is 100
  - …
  - World: You are in state 15. Choose action C or D
  - Learner: D
  - World: Your reward is 50
Examples

- Chess
  - Win +1, loose -1

- Elevator dispatching
  - reward based on mean squared time for elevator to arrive (optimization problem)

- Channel allocation for cellular phones
  - Lower rewards the more calls are blocked
Policy, Reward and Goal

- **Policy**
  - defines the agent’s behaviour at a given time
  - maps from perceptions to actions
  - can be defined by: look-up table, neural net, search algorithm...
  - may be stochastic

- **Reward Function**
  - defines the goal(s) in an RL problem
  - maps from states, state-action pairs, or state-action-successor state, triplets to a numerical reward
  - goal of the agent is to maximise the total reward in the long run
  - the policy is altered to achieve this goal
Reward and Return

- The reward function indicates how good things are right now
- But the agent wants to maximize reward in the long-term i.e. over many time steps
- We refer to long-term (multi-step) reward as return

\[ R_t = r_{t+1} + r_{t+2} + \ldots + r_T \]

where
- T is the last time step of the world
Discounted Return

- The geometrically discounted model of return

\[ R_t = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^T r_T \]

\[ 0 \leq \gamma \leq 1 \]

- \( \gamma \) is called discount rate, used to
  - Bound the infinite sum
  - Favor earlier rewards, in other words to give preference to shorter paths
Optimal Policies

- An RL agent adapts its policy in order to increase return
- A policy $p_1$ is at least as good as a policy $p_2$ if its expected return is at least as great in each possible initial state
- An optimal policy $p$ is at least as good as any other policy
Policy Adaptation Methods

- Value function-based methods
  - Learn a value function for the policy
  - Generate a new policy from the value function
  - Q-learning, Dynamic Programming
Value Functions

- A value function maps each state to an estimate of return under a policy.
- An action-value function maps from state-action pairs to estimates of return.
- Learning a value function is referred to as the “prediction” problem or ‘policy evaluation’ in the Dynamic Programming literature.
Q-learning

- Learns action-values $Q(s,a)$ rather than state-values $V(s)$
- Action-values learning

$$Q(s,a) = R(s,a) + \gamma \max_{a'} Q(T(s,a), a')$$

- Q-learning improves action-values iteratively until it converges
Q-learning Algorithm

1. Algorithm Q {
2. For each \((s,a)\) initialize \(Q'(s,a)\) at zero
3. Choose current action \(s\)
4. Iterate infinitely{
5. Choose and execute action \(a\)
6. Get immediate reward \(r\)
7. Choose new state \(s'\)
8. Update \(Q'(s,a)\) as follows: \(Q'(s,a) \leftarrow r + \gamma \max_{a'} Q'(s',a')\)
9. \(s \leftarrow s'\)
10. }
11. }
Example

- Initially

- Initialization
Example

- S₁

- Assume $\gamma = 0.9$

- Go right: S₂
  - Reward: 0
Example

- **Go right**
  - Reward: 100

- **Update s_2**
  - Reward: 100
Example

- **Update** $s_1$
  - Reward: 90

- **$s_2$**
Example: result of Q-learning
Exercice

- Agent is in room C of the building
- The goal is to get out of the building
Modeling the problem

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100</td>
</tr>
<tr>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>100</td>
</tr>
</tbody>
</table>
$\gamma = 0.8$

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>400</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>320</td>
<td></td>
<td>500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>320</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>400</td>
<td>255</td>
<td>400</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>320</td>
<td></td>
<td>320</td>
<td>500</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>400</td>
<td></td>
<td></td>
<td>400</td>
<td>500</td>
<td></td>
</tr>
</tbody>
</table>

Divide all rewards by 5

Result: C => D => B => F
       C => D => E => F
Artificial Intelligence

For HEDSPI Project

Lecturer 15 – Artificial Neuron Networks

Lecturer :

Tran Duc Khanh
Dept of Information Systems
School of Information and Communication Technology
HUST
Artificial neural networks

- Artificial neural network (ANN)
  - Inspired by biological neural systems, i.e., human brains
  - ANN is a network composed of a number of artificial neurons

- Neuron
  - Has an input/output (I/O) characteristic
  - Implements a local computation

- The output of a unit is determined by
  - Its I/O characteristic
  - Its interconnections to other units
  - Possibly external inputs
Artificial neural networks

- ANN can be seen as a parallel distributed information processing structure
- ANN has the ability to learn, recall, and generalize from training data by assigning and adjusting the interconnection weights
- The overall function is determined by
  - The network topology
  - The individual neuron characteristic
  - The learning/training strategy
  - The training data
Applications of ANNs

- **Image processing and computer vision**
  - E.g., image matching, preprocessing, segmentation and analysis, computer vision, image compression, stereo vision, and processing and understanding of time-varying images

- **Signal processing**
  - E.g., seismic signal analysis and morphology

- **Pattern recognition**
  - E.g., feature extraction, radar signal classification and analysis, speech recognition and understanding, fingerprint identification, character recognition, face recognition, and handwriting analysis

- **Medicine**
  - E.g., electrocardiographic signal analysis and understanding, diagnosis of various diseases, and medical image processing
Applications of ANNs

■ Military systems
  □ E.g., undersea mine detection, radar clutter classification, and tactical speaker recognition

■ Financial systems
  □ E.g., stock market analysis, real estate appraisal, credit card authorization, and securities trading

■ Planning, control, and search
  □ E.g., parallel implementation of constraint satisfaction problems, solutions to Traveling Salesman, and control and robotics

■ Power systems
  □ E.g., system state estimation, transient detection and classification, fault detection and recovery, load forecasting, and security assessment

■ ...
Structure and operation of a neuron

- The **input signals** to the neuron \( (x_i, i = 1..m) \)
  - Each input \( x_i \) associates with a weight \( w_i \)
- The **bias** \( w_0 \) (with the input \( x_0=1 \))
- **Net input** is an integration function of the inputs – \( \text{Net} (w, x) \)
- **Activation (transfer) function** computes the output of the neuron – \( f(\text{Net} (w, x)) \)
- **Output** of the neuron:
  \[ \text{Out} = f(\text{Net} (w, x)) \]
Net input and The bias

■ The net input is typically computed using a linear function

\[ Net = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_m x_m = w_0 + \sum_{i=1}^{m} w_i x_i = \sum_{i=0}^{m} w_i x_i \]

■ The importance of the bias \((w_0)\)
  - The family of separation functions \(Net = w_1 x_1\) cannot separate the instances into two classes
  - The family of functions \(Net = w_1 x_1 + w_0\) can
Activation function – Hard-limiter

- Also called the threshold function
- The output of the hard-limiter is either of the two values
- $\theta$ is the threshold value
- **Disadvantage**: neither continuous nor continuously differentiable

Mathematical expressions:

- **Binary hard-limiter**
  
  \[
  \text{Out}(\text{Net}) = h1(\text{Net}, \theta) = \begin{cases} 
  1, & \text{if } \text{Net} \geq \theta \\
  0, & \text{otherwise} 
  \end{cases}
  \]

- **Bipolar hard-limiter**
  
  \[
  \text{Out}(\text{Net}) = h2(\text{Net}, \theta) = \text{sign}(\text{Net}, \theta)
  \]
Activation function – Threshold logic

\[
Out(Net) = tl(Net, \alpha, \theta) = \begin{cases} 
0, & \text{if } \ Net < -\theta \\
\alpha(Net + \theta), & \text{if } -\theta \leq Net \leq \frac{1}{\alpha} - \theta \\
1, & \text{if } \ Net > \frac{1}{\alpha} - \theta 
\end{cases}
\]

\[= \max(0, \min(1, \alpha(Net + \theta)))\]

- It is called also saturating linear function
- A combination of linear and hard-limiter activation functions
- \(\alpha\) decides the slope in the linear range
- **Disadvantage**: continuous – but not continuously differentiable
**Activation function – Sigmoidal**

\[
Out(\text{Net}) = sf(\text{Net}, \alpha, \theta) = \frac{1}{1 + e^{-\alpha(\text{Net}+\theta)}}
\]

- Most often used in ANNs
- The slope parameter \( \alpha \) is important
- The output value is always in (0,1)

**Advantage**

- Both continuous and continuously differentiable
- The derivative of a sigmoidal function can be expressed in terms of the function itself
Activation function – Hyperbolic tangent

\[ \text{Out}(\text{Net}) = \tanh(\text{Net}, \alpha, \theta) = \frac{1-e^{-\alpha(\text{Net}+\theta)}}{1+e^{-\alpha(\text{Net}+\theta)}} = \frac{2}{1+e^{-\alpha(\text{Net}+\theta)}} - 1 \]

- Also often used in ANNs
- The slope parameter \( \alpha \) is important
- The output value is always in \((-1,1)\)

**Advantage**
- Both continuous and continuously differentiable
- The derivative of a \( \tanh \) function can be expressed in terms of the function itself
Network structure

- Topology of an ANN is composed by:
  - The number of input signals and output signals
  - The number of layers
  - The number of neurons in each layer
  - The number of weights in each neuron
  - The way the weights are linked together within or between the layer(s)
  - Which neurons receive the (error) correction signals

- Every ANN must have
  - exactly one input layer
  - exactly one output layer
  - zero, one, or more than one hidden layer(s)

- An ANN with one hidden layer
- Input space: 3-dimensional
- Output space: 2-dimensional
- In total, there are 6 neurons
  - 4 in the hidden layer
  - 2 in the output layer
Network structure

- A layer is a group of neurons
- A hidden layer is any layer between the input and the output layers
- Hidden nodes do not directly interact with the external environment
- An ANN is said to be **fully connected** if every output from one layer is connected to every node in the next layer
- An ANN is called **feed-forward network** if no node output is an input to a node in the same layer or in a preceding layer
- When node outputs can be directed back as inputs to a node in the same (or a preceding) layer, it is a **feedback network**
  - If the feedback is directed back as input to the nodes in the same layer, then it is called **lateral feedback**
- Feedback networks that have closed loops are called **recurrent networks**
Network structure – Example

- Single layer feed-forward network
- Multilayer feed-forward network
- Single node with feedback to itself
- Single layer recurrent network
- Multilayer recurrent network
Learning rules

- Two kinds of learning in neural networks
  - Parameter learning
    - Focus on the update of the connecting weights in an ANN
  - Structure learning
    - Focus on the change of the network structure, including the number of processing elements and their connection types

- These two kinds of learning can be performed simultaneously or separately

- Most of the existing learning rules are the type of parameter learning

- We focus the parameter learning
General weight learning rule

- At a learning step \((t)\) the adjustment of the weight vector \(w\) is proportional to the product of the learning signal \(r(t)\) and the input \(x(t)\)
  \[
  \Delta w(t) \sim r(t) \cdot x(t)
  \]
  \[
  \Delta w(t) = \eta \cdot r(t) \cdot x(t)
  \]
  where \(\eta (>0)\) is the learning rate

- The learning signal \(r\) is a function of \(w, x, \) and the desired output \(d\)
  \[
  r = g(w, x, d)
  \]

- The general weight learning rule
  \[
  \Delta w(t) = \eta \cdot g(w(t), x(t), d(t)) \cdot x(t)
  \]

Note that \(x_j\) can be either:
- an (external) input signal, or
- an output from another neuron
Perceptron

- A perceptron is the simplest type of ANNs
- Use the hard-limit activation function

\[ Out = \text{sign}(\text{Net}(w, x)) = \text{sign}\left(\sum_{j=0}^{m} w_j x_j\right) \]

- For an instance \( x \), the perceptron output is
  - 1, if \( \text{Net}(w, x) > 0 \)
  - -1, otherwise
Perceptron – Illustration

The decision hyperplane

\[ w_0 + w_1 x_1 + w_2 x_2 = 0 \]
Perceptron – Learning

- Given a training set \( D = \{(x,d)\} \)
  - \( x \) is the input vector
  - \( d \) is the desired output value (i.e., -1 or 1)

- The perceptron learning is to determine a weight vector that makes the perceptron produce the correct output (-1 or 1) for every training instance

- If a training instance \( x \) is correctly classified, then no update is needed

- If \( d=1 \) but the perceptron outputs -1, then the weight \( w \) should be updated so that \( \text{Net}(w,x) \) is increased

- If \( d=-1 \) but the perceptron outputs 1, then the weight \( w \) should be updated so that \( \text{Net}(w,x) \) is decreased
**Perceptron_incremental**($D$, $\eta$)

Initialize $w$ ($w_i \leftarrow$ an initial (small) random value)

do

    for each training instance $(x, d) \in D$

        Compute the real output value $Out$

        if $(Out \neq d)$

            $w \leftarrow w + \eta (d - Out) x$

        end if

    end for

until all the training instances in $D$ are correctly classified

return $w$
**Perceptron\_batch**(\(D, \eta\))

Initialize \( \mathbf{w} \) \((w_i \leftarrow \text{an initial (small) random value}) \)

do

\( \Delta \mathbf{w} \leftarrow 0 \)

for each training instance \((x, d) \in D\)

Compute the real output value \( \text{Out} \)

if \((\text{Out} \neq d)\)

\( \Delta \mathbf{w} \leftarrow \Delta \mathbf{w} + \eta (d - \text{Out}) \mathbf{x} \)

end for

\( \mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w} \)

until all the training instances in \(D\) are correctly classified

return \( \mathbf{w} \)
The perceptron learning procedure is proven to converge if
- The training instances are linearly separable
- With a sufficiently small $\eta$ used

The perceptron may not converge if the training instances are not linearly separable

We need to use the **delta rule**
- Converges toward a best-fit approximation of the target function
- The delta rule uses **gradient descent** to search the hypothesis space (of possible weight vectors) to find the weight vector that best fits the training instances

A perceptron cannot correctly classify this training set!
Let’s consider an ANN that has $n$ output neurons

Given a training instance $(x,d)$, the **training error** made by the currently estimated weights vector $w$:

$$E_x(w) = \frac{1}{2} \sum_{i=1}^{n} (d_i - Out_i)^2$$

The **training error** made by the currently estimated weights vector $w$ over the entire training set $D$:

$$E_D(w) = \frac{1}{|D|} \sum_{x \in D} E_x(w)$$
Gradient descent

- **Gradient** of $E$ (denoted as $\nabla E$) is a vector
  - The direction points most uphill
  - The length is proportional to steepness of hill

- The gradient of $\nabla E$ specifies the direction that produces the **steepest increase** in $E$

$$\nabla E(w) = \left( \frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_N} \right)$$

where $N$ is the number of the weights in the network (i.e., $N$ is the length of $w$)

- Hence, the direction that produces the **steepest decrease** is the negative of the gradient of $E$

$$\Delta w = -\eta \cdot \nabla E(w)$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}, \; \forall i = 1..N$$

- Requirement: The activation functions used in the network must be continuous functions of the weights, differentiable everywhere
Gradient descent – Illustration

One-dimensional
$E(w)$

Two-dimensional
$E(w_1,w_2)$
Gradient\_descent\_incremental (\(D, \eta\))

Initialize \(w\) (\(w_i \leftarrow\) an initial (small) random value)

\[
\begin{align*}
\text{do} \\
\quad &\text{for each training instance } (x, d) \in D \\
\quad &\quad \text{Compute the network output} \\
\quad &\quad \text{for each weight component } w_i \\
\quad &\quad \quad w_i \leftarrow w_i - \eta \left( \frac{\partial E_x}{\partial w_i} \right) \\
\quad &\text{end for} \\
\quad &\text{end for} \\
\text{until (stopping criterion satisfied)} \\
\text{return } w
\end{align*}
\]

Stopping criterion: \# of iterations (epochs), threshold error, etc.
Multi-layer NNs and Back-propagation alg.

- As we have seen, a perceptron can only express a linear decision surface

- A multi-layer NN learned by the back-propagation (BP) algorithm can represent *highly non-linear decision surfaces*

- The BP learning algorithm is used to learn the weights of a multi-layer NN
  - *Fixed structure* (i.e., fixed set of neurons and interconnections)
  - For every neuron the activation function must be *continuously differentiable*

- The BP algorithm *employs gradient descent* in the weight update rule
  - To minimize the error between the actual output values and the desired output ones, given the training instances
Back-propagation algorithm (1)

- Back-propagation algorithm searches for the weights vector that **minimizes the total error** made over the training set

- Back-propagation consists of the two phases
  - **Signal forward** phase. The input signals (i.e., the input vector) are propagated (forwards) from the input layer to the output layer (through the hidden layers)
  
  - **Error backward** phase
    - Since the desired output value for the current input vector is known, the error is computed
    
    - Starting at the output layer, the error is propagated backwards through the network, layer by layer, to the input layer
    
    - The error back-propagation is performed by recursively computing the local gradient of each neuron
Back-propagation algorithm (2)

Signal forward phase
• Network activation

Error backward phase
• Output error computation
• Error propagation
Let’s use this 3-layer NN to illustrate the details of the BP learning algorithm:

- $m$ input signals $x_j$ ($j=1..m$)
- $l$ hidden neurons $z_q$ ($q=1..l$)
- $n$ output neurons $y_i$ ($i=1..n$)
- $w_{qj}$ is the weight of the interconnection from input signal $x_j$ to hidden neuron $z_q$
- $w_{iq}$ is the weight of the interconnection from hidden neuron $z_q$ to output neuron $y_i$
- $Out_q$ is the (local) output value of hidden neuron $z_q$
- $Out_i$ is the network output w.r.t. the output neuron $y_i$
BP algorithm – Forward phase (1)

For each training instance $\mathbf{x}$
- The input vector $\mathbf{x}$ is propagated from the input layer to the output layer
- The network produces an actual output $\mathbf{Out}$ (i.e., a vector of $Out_i$, $i=1..n$)

Given an input vector $\mathbf{x}$, a neuron $z_q$ in the hidden layer receives a net input of

$$Net_q = \sum_{j=1}^{m} w_{qj} x_j$$

...and produces a (local) output of

$$Out_q = f(Net_q) = f\left(\sum_{j=1}^{m} w_{qj} x_j\right)$$

where $f(.)$ is the activation (transfer) function of neuron $z_q$
BP algorithm – Forward phase (2)

- The net input for a neuron \( y_i \) in the output layer is
  
  \[
  Net_i = \sum_{q=1}^{l} w_{iq} Out_q = \sum_{q=1}^{l} w_{iq} f \left( \sum_{j=1}^{m} w_{qj} x_j \right)
  \]

- Neuron \( y_i \) produces the output value (i.e., an output of the network)
  
  \[
  Out_i = f(Net_i) = f \left( \sum_{q=1}^{l} w_{iq} Out_q \right) = f \left( \sum_{q=1}^{l} w_{iq} f \left( \sum_{j=1}^{m} w_{qj} x_j \right) \right)
  \]

- The vector of output values \( Out_i (i=1..n) \) is the actual network output, given the input vector \( x \)
BP algorithm – Backward phase (1)

- For each training instance $x$
  - The error signals resulting from the difference between the desired output $d$ and the actual output $Out$ are computed
  - The error signals are back-propagated from the output layer to the previous layers to update the weights

- Before discussing the error signals and their back propagation, we first define an error (cost) function

$$E(w) = \frac{1}{2} \sum_{i=1}^{n} (d_i - Out_i)^2 = \frac{1}{2} \sum_{i=1}^{n} [d_i - f(Net_i)]^2$$

$$= \frac{1}{2} \sum_{i=1}^{n} \left[ d_i - f\left(\sum_{q=1}^{l} w_{iq} Out_q \right) \right]^2$$
According to the gradient-descent method, the weights in the hidden-to-output connections are updated by

\[ \Delta w_{iq} = -\eta \frac{\partial E}{\partial w_{iq}} \]

Using the derivative chain rule for \( \partial E / \partial w_{iq} \), we have

\[ \Delta w_{iq} = -\eta \left[ \frac{\partial E}{\partial \text{Out}_i} \right] \left[ \frac{\partial \text{Out}_i}{\partial \text{Net}_i} \right] \left[ \frac{\partial \text{Net}_i}{\partial w_{iq}} \right] = \eta \left[ d_i - \text{Out}_i \right] f'(\text{Net}_i) \left[ \text{Out}_q \right] = \eta \delta_i \text{Out}_q \]

(note that the negative sign is incorporated in \( \partial E / \partial \text{Out}_i \))

\( \delta_i \) is the error signal of neuron \( y_i \) in the output layer

\[ \delta_i = -\frac{\partial E}{\partial \text{Net}_i} = -\left[ \frac{\partial E}{\partial \text{Out}_i} \right] \left[ \frac{\partial \text{Out}_i}{\partial \text{Net}_i} \right] = \left[ d_i - \text{Out}_i \right] f'(\text{Net}_i) \]

where \( \text{Net}_i \) is the net input to neuron \( y_i \) in the output layer, and \( f'(\text{Net}_i) = \partial f(\text{Net}_i) / \partial \text{Net}_i \)
To update the weights of the input-to-hidden connections, we also follow gradient-descent method and the derivative chain rule

\[ \Delta w_{qj} = -\eta \frac{\partial E}{\partial w_{qj}} = -\eta \left[ \frac{\partial E}{\partial Out_q} \left( \frac{\partial Out_q}{\partial Net_q} \right) \left( \frac{\partial Net_q}{\partial w_{qj}} \right) \right] \]

From the equation of the error function \( E(w) \), it is clear that each error term \((d_i-y_i) \) \((i=1..n)\) is a function of \( Out_q \)

\[ E(w) = \frac{1}{2} \sum_{i=1}^{n} \left[ d_i - f \left( \sum_{q=1}^{l} w_{iq} Out_q \right) \right]^2 \]
Evaluating the derivative chain rule, we have

\[ \Delta w_{qj} = \eta \sum_{i=1}^{n} \left[ (d_i - Out_i) f'(Net_i) w_{iq} \right] f'(Net_q) x_j \]

\[ = \eta \sum_{i=1}^{n} \left[ \delta_i w_{iq} \right] f'(Net_q) x_j = \eta \delta_q x_j \]

- \( \delta_q \) is the **error signal** of neuron \( z_q \) in the **hidden layer**

\[ \delta_q = - \frac{\partial E}{\partial Net_q} = - \left[ \frac{\partial E}{\partial Out_q} \right] \left[ \frac{\partial Out_q}{\partial Net_q} \right] = f'(Net_q) \sum_{i=1}^{n} \delta_i w_{iq} \]

where \( Net_q \) is the net input to neuron \( z_q \) in the hidden layer, and \( f'(Net_q) = \frac{\partial f(Net_q)}{\partial Net_q} \)
BP algorithm – Backward phase (5)

- According to the error equations $\delta_i$ and $\delta_q$ above, the error signal of a neuron in a hidden layer is different from the error signal of a neuron in the output layer.
- Because of this difference, the derived weight update procedure is called the generalized delta learning rule.
- The error signal $\delta_q$ of a hidden neuron $z_q$ can be determined
  - in terms of the error signals $\delta_i$ of the neurons $y_i$ (i.e., that $z_q$ connects to) in the output layer
  - with the coefficients are just the weights $w_{iq}$
- The important feature of the BP algorithm: the weights update rule is local.
  - To compute the weight change for a given connection, we need only the quantities available at both ends of that connection!
BP algorithm – Backward phase (6)

- The discussed derivation can be easily extended to the network with more than one hidden layer by using the chain rule continuously.

- The general form of the BP update rule is

\[ \Delta w_{ab} = \eta \delta_a x_b \]

  - \( b \) and \( a \) refer to the two ends of the \((b \to a)\) connection (i.e., from neuron (or input signal) \( b \) to neuron \( a \)).
  - \( x_b \) is the output of the hidden neuron (or the input signal) \( b \).
  - \( \delta_a \) is the error signal of neuron \( a \).
**Back_propagation_incremental**($D, \eta$)  

A network with $Q$ feed-forward layers, $q = 1, 2, ..., Q$

$qNet_i$ and $qOut_i$ are the net input and output of the $i^{th}$ neuron in the $q^{th}$ layer

The network has $m$ input signals and $n$ output neurons

$qw_{ij}$ is the weight of the connection from the $j^{th}$ neuron in the $(q-1)^{th}$ layer to the $i^{th}$ neuron in the $q^{th}$ layer

**Step 0 (Initialization)**

Choose $E_{threshold}$ (a tolerable error)

Initialize the weights to small random values

Set $E=0$

**Step 1 (Training loop)**

Apply the input vector of the $k^{th}$ training instance to the input layer ($q=1$)

$qOut_i = 1Out_i = x_i^{(k)}, \forall i$

**Step 2 (Forward propagation)**

Propagate the signal forward through the network, until the network outputs (in the output layer) $QOut_i$ have all been obtained

$$qOut_i = f(qNet_i) = f\left(\sum_j qw_{ij}^{q-1}Out_j\right)$$
**Step 3 (Output error measure)**
Compute the error and error signals $Q\delta_i$ for every neuron in the output layer

$$E = E + \frac{1}{2} \sum_{i=1}^{n} (d_i^{(k)} - QOut_i)^2$$

$$Q\delta_i = (d_i^{(k)} - QOut_i)f'(QNet_i)$$

**Step 4 (Error back-propagation)**
Propagate the error backward to update the weights and compute the error signals $q^{-1}\delta_i$ for the preceding layers

$$\Delta^q_{wij} = \eta.(q\delta_i).(q^{-1}Out_j); \quad q_{wij} = q_{wij} + \Delta^q_{wij}$$

$$q^{-1}\delta_i = f'(q^{-1}Net_i)\sum_{j} q^{wij} q\delta_j; \quad \text{for all } q = Q, Q-1, \ldots, 2$$

**Step 5 (One epoch check)**
Check whether the entire training set has been exploited (i.e., one epoch)
If the entire training set has been exploited, then go to step 6; otherwise, go to step 1

**Step 6 (Total error check)**
If the current total error is acceptable ($E<E_{\text{threshold}}$) then the training process terminates and output the final weights;
Otherwise, reset $E=0$, and initiate the new training epoch by going to step 1
BP illustration – Forward phase (1)
BP illustration – Forward phase (2)

\[ \text{Out}_1 = f(w_{1x_1}x_1 + w_{1x_2}x_2) \]
BP illustration – Forward phase (3)

\[ \text{Out}_2 = f \left( w_{2x_1} x_1 + w_{2x_2} x_2 \right) \]
BP illustration – Forward phase (4)

\[ \text{Out}_3 = f(w_{3x_1}x_1 + w_{3x_2}x_2) \]
BP illustration – Forward phase (5)

\[ Out_4 = f(w_{41}Out_1 + w_{42}Out_2 + w_{43}Out_3) \]
BP illustration – Forward phase (6)

\[ \text{Out}_5 = f(w_{51} \text{Out}_1 + w_{52} \text{Out}_2 + w_{53} \text{Out}_3) \]
BP illustration – Forward phase (7)

\[ \text{Out}_6 = f \left( w_{64} \text{Out}_4 + w_{65} \text{Out}_5 \right) \]
BP illustration – Compute the error

\[ \delta_6 = -\frac{\partial E}{\partial \text{Net}_6} = -\left[ \frac{\partial E}{\partial \text{Out}_6} \right] \left[ \frac{\partial \text{Out}_6}{\partial \text{Net}_6} \right] = [d - \text{Out}_6][f'(\text{Net}_6)] \]

d is the desired output value
BP illustration – Backward phase (1)

\[ \delta_4 = f'(Net_4)(w_{64}\delta_6) \]
BP illustration – Backward phase (2)

\[ \delta_5 = f'(Net_5)(w_{65}\delta_6) \]
BP illustration – Backward phase (3)

\[ \delta_1 = f'(Net_1)(w_{41}\delta_4 + w_{51}\delta_5) \]
BP illustration – Backward phase (4)

\[ \delta_2 = f'(Net_2)(w_{42}\delta_4 + w_{52}\delta_5) \]
BP illustration – Backward phase (5)

\[ \delta_3 = f'(Net_3)(w_{43}\delta_4 + w_{53}\delta_5) \]
BP illustration – Weight update (1)

\[
\begin{align*}
\delta_1 &= f(Net_1) \\
w_{1x_1} &= w_{1x_1} + \eta \delta_1 x_1 \\
w_{1x_2} &= w_{1x_2} + \eta \delta_1 x_2
\end{align*}
\]
BP illustration – Weight update (2)

\[ w_{2x_1} = w_{2x_1} + \eta \delta_2 x_1 \]

\[ w_{2x_2} = w_{2x_2} + \eta \delta_2 x_2 \]
BP illustration – Weight update (3)

\[ w_{3x1} = w_{3x1} + \eta \delta_3 x_1 \]
\[ w_{3x2} = w_{3x2} + \eta \delta_3 x_2 \]
BP illustration – Weight update (4)

\[ w_{41} = w_{41} + \eta \delta_4 \text{Out}_1 \]
\[ w_{42} = w_{42} + \eta \delta_4 \text{Out}_2 \]
\[ w_{43} = w_{43} + \eta \delta_4 \text{Out}_3 \]
BP illustration – Weight update (5)

\[
\begin{align*}
    w_{51} &= w_{51} + \eta \delta_5 \text{Out}_1 \\
    w_{52} &= w_{52} + \eta \delta_5 \text{Out}_2 \\
    w_{53} &= w_{53} + \eta \delta_5 \text{Out}_3
\end{align*}
\]
BP illustration – Weight update (6)

\[ w_{64} = w_{64} + \eta \delta_6 Out_4 \]
\[ w_{65} = w_{65} + \eta \delta_6 Out_5 \]
Advantages vs. Disadvantages

- **Advantages**
  - Massively parallel in nature
  - Fault (noise) tolerant because of parallelism
  - Can be designed to be adaptive

- **Disadvantages**
  - No clear rules or design guidelines for arbitrary applications
  - No general way to assess the internal operation of the network (therefore, an ANN system is seen as a “black-box”)
  - Difficult to predict future network performance (generalization)
When using ANNs?

- Input is high-dimensional discrete or real-valued
- The target function is real-valued, discrete-valued or vector-valued
- Possibly noisy data
- The form of the target function is unknown
- Human readability of result is not (very) important
- Long training time is accepted
- Short classification/prediction time is required