UiO : Department of Informatics
University of Oslo

INF5390

Curriculum

Joakim Myrvoll

Exam: 04.06.14 14:30-18:30

Contents

1 Intelligent agents

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators
- A rational agent is an agent that for each situation selects the action that maximizes its performance based on its perception and built - in knowledge
- The task of AI is to build problem solving computer programs as rational agents

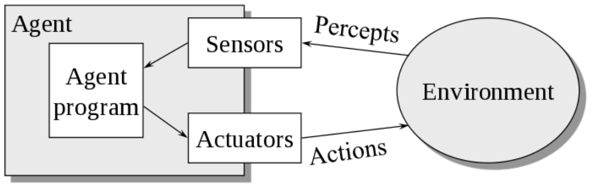


Figure 1: Generic agent architecture

1.1 Rational agent

"The best performance under the circumstances. . . "

- Definition:
  For each possible percept sequence, a rational agent selects an action that maximizes its expected performance measure, given its current knowledge
- Definition depends on:
  - Performance measure (success criterion)
  - Agent's percept sequence to date
  - Actions that the agent can perform
  - Agent's knowledge of the environment

This means that an agent can be rational under some asumptions, but may not be it under other asumptions.

1.2 Tasks and environment

1.2.1 PEAS - task environment characterization

P - Performance measure
E - Environment
A - Actuators
S - Sensors

Example (Agent type: Medical diagnosis system):
P: Healthy patient, minimize cost
E: Patient, hospital, staff
A: Questions, tests, treatments
S: Symptoms, findings, patient answers

1.2.2 Properties of environments

- Fully observable vs. partially observable
  Sensors detect all aspects relevant for selecting action?
- Single agent vs. multi agent
  Does the environment include other agents?
- Deterministic vs. stochastic
  Next state determined by current state and action? (yes$\rightarrow$stochastic)
- Episodic vs. sequential
  Agent's experience divided into independent episodes?
  In episodic environments, the choice of action in each episode depends only on the episode itself.
- Static vs. dynamic
  Dynamic: environment can change while an agent is deliberating
  Semidynamic: environment itself does not change with the passage of time but the agent's performance score does
  Static: No change under deliberation
- Discrete vs. continuous
  Limited number of distinct percepts and action? (yes$\rightarrow$discrete)
- Known vs. unknown
  Outcomes of all actions (or probabilities) known to agent?

1.3 Types

1.3.1 Table driven agent

The table driven agent program is invoked for each new percept and returns an action each time using a table that contains the appropriate actions for every possible percept sequence. Normally it's doomed to failure because of the huge number percepts it needs to store in its table. The lookup table gets huge with increasing number of percepts.

```
function TABLE—DRIVEN—AGENT(percept) returns action
persistent : percepts , a sequence, initially empty table , a table , indexed by
percept sequences, initially fully specified
append percept to the end of percepts
action <= LOOKUP(percepts , table)
return action
```

1.3.2 Simple reflex agent

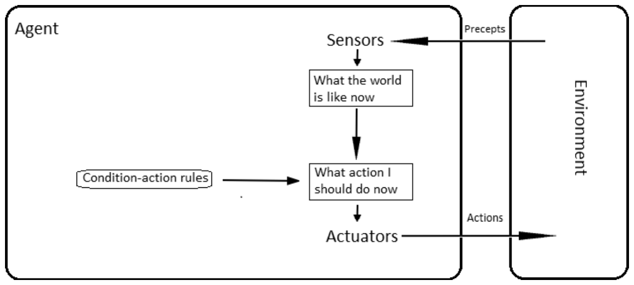


Figure 2: Simple reflex agent

Simple reflex agents act only on the basis of the current percept, ignoring the rest of the percept history. The agent function is based on the condition-action rule: if condition then action.

This agent function only succeeds when the environment is fully observable. Some reflex agents can also contain information on their current state which allows them to disregard conditions whose actuators are already triggered.

Infinite loops are often unavoidable for simple reflex agents operating in partially observable environments. Note: If the agent can randomize its actions, it may be possible to escape from infinite loops.

```
function SIMPLE–REFLEX–AGENT(percept) returns an action
persistent : rules , a set of condition–action rules
state <= INTERPRET–INPUT(percept)
rule <= RULE–MATCH(state , rules)
action <= rule.ACTION
return action
```

Example:

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

1.3.3 Model-based reflex agent

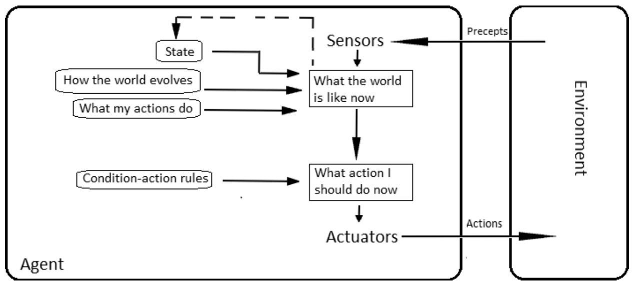


Figure 3: Model-based reflex agent

A model-based agent can handle a partially observable environment. Its current state is stored inside the agent maintaining some kind of structure which describes the part of the world which cannot be seen. This knowledge about "how the world works" is called a model of the world, hence the name "model-based agent".

A model-based reflex agent should maintain some sort of internal model that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. It then chooses an action in the same way as the reflex agent. The reasoning may involve *searching* and *planning*.

```
function MODEL—BASED—REFLEX—AGENT(percept) returns an action
persistent: state, description of the current world state
model, how next state depends on current state and action
rules, a set of condition—action rules
action, most recent action, initially none
state <= UPDATE—STATE(state, action, percept, model)
rule <= RULE—MATCH(state, rules)
action <= rule.ACTION
return action
```

1.3.4 Goal-based agent

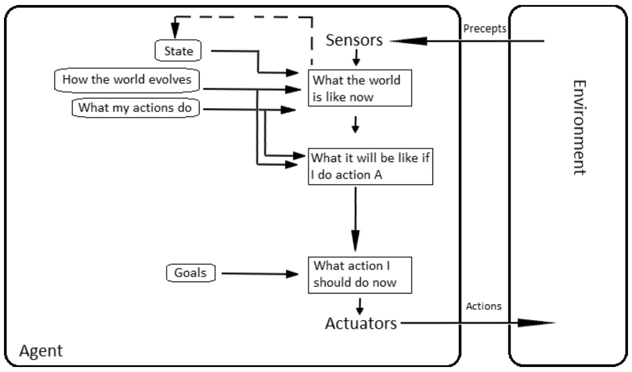


Figure 4: Goal-based agent

Goal-based agents further expand on the capabilities of the model-based agents, by using "goal" information. Goal information describes situations that are desirable. This allows the agent a way to choose among multiple possibilities, selecting the one which reaches a goal state. Search and planning are the subfields of artificial intelligence devoted to finding action sequences that achieve the agent's goals.

In some instances the goal-based agent appears to be less efficient; it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.

1.3.5 Utility-based agent

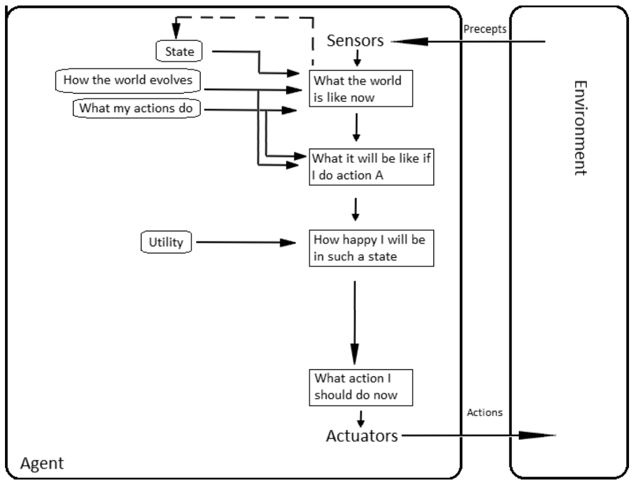

State
How the world evolves
What my actions do
Utility
Sensors
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Actuators
Agent
Precepts
Actions
Environment

Figure 5: Utility-based agent

Goal-based agents only distinguish between goal states and non-goal states. It is possible to define a measure of how desirable a particular state is. This measure can be obtained through the use of a utility function which maps a state to a measure of the utility of the state. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. The term utility, can be used to describe how "happy" the agent is.

A rational utility-based agent chooses the action that maximizes the expected utility of the action outcomes- that is, the agent expects to derive, on average, given the probabilities and utilities of each outcome. A utility-based agent has to model and keep track of its environment, tasks that have involved a great deal of research on perception, representation, reasoning, and learning.

1.3.6 Learning agent

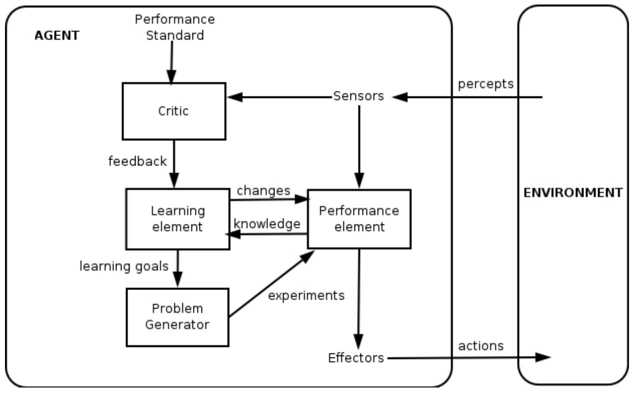


Figure 6: Learning agent

Learning has an advantage that it allows the agents to initially operate in unknown environments and to become more competent than its initial knowledge alone might allow. The most important distinction is between the "learning element", which is responsible for making improvements, and the "performance element", which is responsible for selecting external actions.

The learning element uses feedback from the "critic" on how the agent is doing and determines how the performance element should be modified to do better in the future. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.

The last component of the learning agent is the "problem generator". It is responsible for suggesting actions that will lead to new and informative experiences.

1.3.7 Summary

- **Table lookup agents** only useful for tiny problems
- **Simple reflex agents** respond immediately to percepts
- **Model-based reflex agents** remember the state
- **Goal-based agents** act to achieve goals
- **Utility-based agents** maximize utility
- **Learning agents** improve their performance over time
- Agents need environment models of increasing complexity

2 Searching

- The search starts in an initial state
- Then, it iteratively explores the state space by selecting a state node and applying operators to generate successor nodes until it finds the goal state node or has to give up
- The choice of which node to expand at each level is determined by the search strategy
- The part of the state space (defined by inital state and actions) that is explored is called the search tree

2.1 Formulation of a search problem

- Initial state
  Initial state of environment
- Actions
  Set of actions available to agent
- Path
  Sequence of actions leading from one state to another
- Goal test
  Test to check if a state is a goal state
- Path cost
  Function that assigns cost to a path
- Solution
  Path from initial state to a state that satisfies goal test

2.2 Problem-solving agents

Problem-solving agents are goal-based agents that use search to find action sequences

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
persistent: 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
goal <= FORMULATE-GOAL(state)
problem <= FORMULATE-PROBLEM(state, goal)
seq <= SEARCH(problem)
if seq = failure then
return a null action
action <= FIRST(seq)
seq <= REST(seq)
return action
```

2.3 Example environment

- Properties
  - Fully observable: Agent has full knowledge
  - Deterministic: No surprises
  - Static: No changes under deliberation
  - Discrete: Discrete alternative actions
- Formulation of problem
  - States: Location of each tile
  - Operators: Blank moves left, right, up, down
  - Goal test: State matches goal configuration
  - Path cost: Number of moves

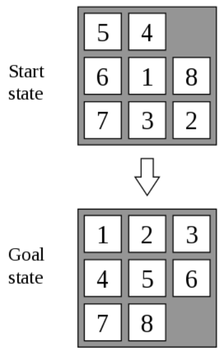


Figure 7: Toy Problem

2.3.1 Search tree

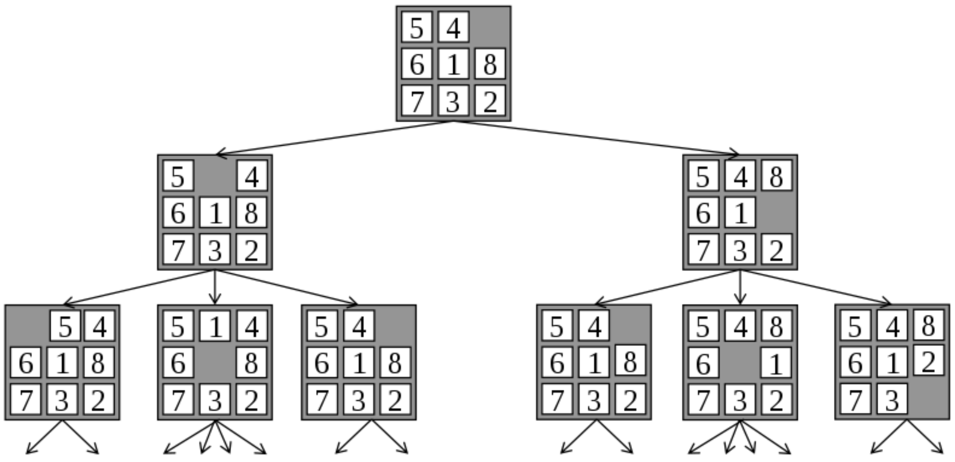


Figure 8: Search tree

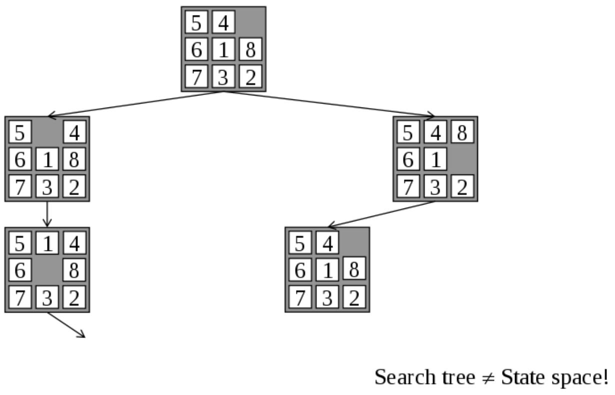


Figure 9: Partially/Different order search tree

2.4 Searching for solutions

- The search starts in an initial state
- Thereafter, it iteratively explores the state space by selecting a state node and applying operators to generate successor nodes
- The choice of which node to expand at each level is determined by the search strategy
- The part of the state space that is explored is called the search tree

2.4.1 Tree search vs. graph search

The state space may contain loops (path back to earlier state) or redundant paths (more than one path between two states). Simple tree expansion will run infinitely or "explode" in such search spaces. To avoid the problem, tree search can be replaced by generalized graph search. In graph search, the algorithm keeps track and avoids expanding already visited nodes.

2.4.1.1 Tree search

Start in initial state. Expand possible nodes. Keep a frontier of unexpanded nodes. Select next node to expand according to strategy. Continue until goal (or give up)

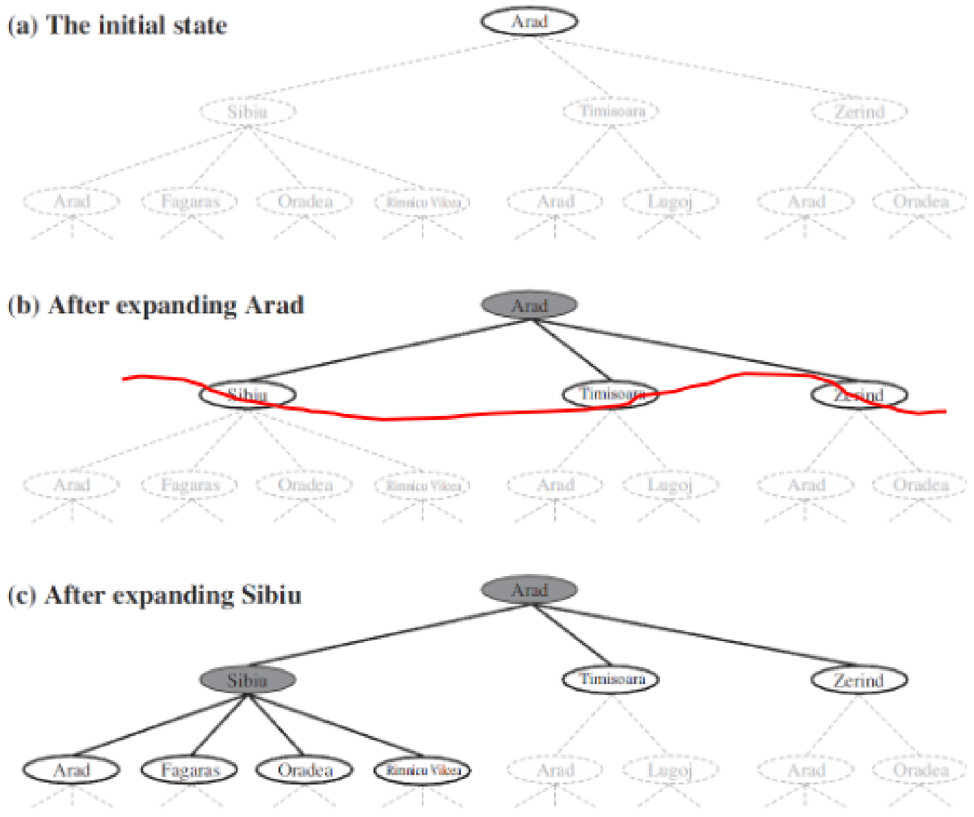


Figure 10: Tree search

2.5 Uninformed search strategies

Uninformed - No information on path cost from current to goal states

Six uninformed strategies:

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

Differ by order in which nodes are expanded.

2.6 Evaluation of search strategies

- Completeness:
  Guaranteed to find a solution when there is one?
- Optimality:
  Finds the best solution when there are several different possible solutions?
- Time complexity:
  How long does it take to find a solution?
- Space complexity:
  How much memory is needed?

2.7 Search trees

2.7.1 Data structures

- Datatype node with components
  - STATE - search space state corresponding to the node
  - PARENT-NODE - node that generated this node
  - ACTION - action that was applied to generate this node
  - PATH-COST - cost of path from initial node (called g)
  - DEPTH - number of nodes on path from initial node
- Search tree nodes kept in a queue with operators
  - MAKE-QUEUE(Elements) - create queue with given elements
  - EMPTY?(Queue) - true if no more elements in queue
  - FIRST(Queue) - returns first element of the queue
  - REMOVE-FIRST(Queue) - removes and returns first element
  - INSERT(Element, Queue) - inserts an element into queue
  - INSERT-ALL(Elements, Queue) - inserts set of elements into queue

2.7.2 General tree-search algorithm

```
function TREE−SEARCH(problem, frontier) returns a solution, or failure
frontier <= INSERT(MAKE−NODE (problem.INITIAL−STATE), frontier)
loop do
if EMPTY?(frontier) then return failure
node <= REMOVE−FIRST(frontier)
if problem.GOAL−TEST applied to node.STATE succeeds then return SOLUTION(node)
frontier <= INSERT−ALL(EXPAND(node,problem), frontier)

function EXPAND(node, problem) returns a set of nodes
```

- frontier is an initially empty queue of a certain type (FIFO, etc.)
- SOLUTION returns sequence of actions back to root
- EXPAND generates all successors of a node

2.7.3 Breadth-first search

```
function BREADTH−FIRST−SEARCH(problem) returns a solution or failure
  return TREE−SEARCH(problem, FIFO−QUEUE())
```

- FIFO - First In First Out (add nodes as last)
- Expands all nodes at a certain depth of search tree before expanding any node at next depth
- Exhaustive method - if there is a solution, breadth - first will find it (completeness)
- Will find the shortest solution first (optimal)
- All nodes on one level are explored before moving to next level
- Complexity: $O(b^d)$ (exponential)
  - b = branching factor
  - d = depth (root node, d = 0)

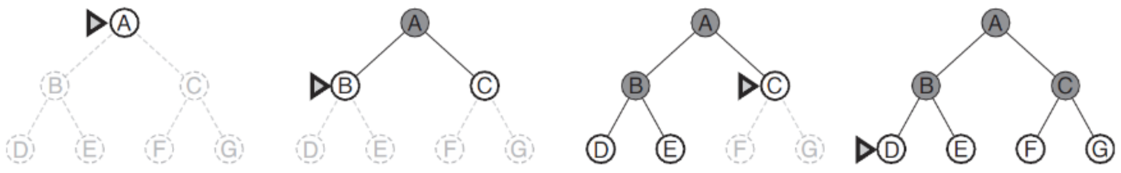


Figure 11: Breadth-first search illustrated

2.7.4 Uniform-cost search

The search begins at the root node. The search continues by visiting the next node which has the least total cost from the root. Nodes are visited in this manner until a goal state is reached.

Uniform-cost search is optimal since it always xpands the node with the lowest cost so far. Completeness is guaranteed if all path costs > 0.

2.7.5 Depth-first search (DFS)

```
function DEPTH—FIRST—SEARCH(problem) returns a solution or failure
  return TREE—SEARCH(problem, LIFO—QUEUE())
```

Start at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

LIFO-Last In First Out (add nodes as first). Always expands a node at deepest level of the tree, backtracks if it finds node with no successor. May never terminate if it goes down an infinite branch, even if there is a solution (not complete). May return an early found solution even if a better one exists (not optimal).

Worst case time complexity is $O(b^m)$ for branching factor b and depth m. But depth-first may find solution much quicker if there are many solutions (m may be much larger than d - the depth of the shallowest solution).

2.7.6 Depth-limited search

Modifies depth-first search by imposing a cutoff on the maximum depth of a path. Avoids risk of non-terminating search down an infinite path. Finds a solution if it exists within cutoff limit (not generally complete). Not guaranteed to find shortest solution (not optimal). Time and space complexity as for depth-first

2.7.7 Iterative deepening search

```
function ITERATIVE—DEEPENING—SEARCH(problem) returns a solution or failure
  for depth <= 0 to infinity do
    result <= DEPTH—LIMITED—SEARCH(problem, depth)
    if result not equal to cutoff then return result
```

Modifies depth-limited search by iteratively trying all possible depths as the cutoff limit. Combines benefits of depth-first and breadth-first. Time complexity is $O(b^d)$, space complexity $O(bd)$. Iterative deepening is the preferred (uninformed) search strategy when there is a large search space and the solution depth is unknown.

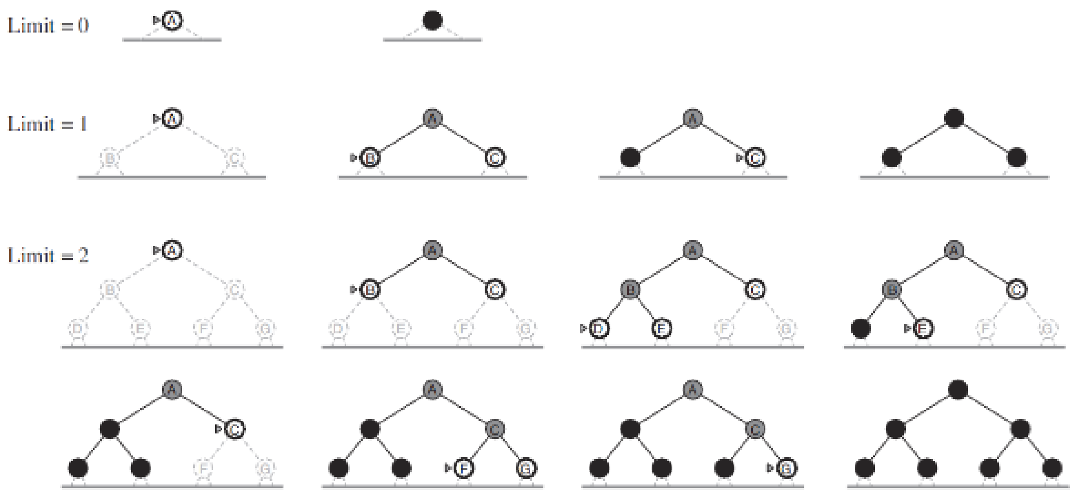


Figure 12: Iterative deepening search illustrated

2.7.8 Bidirectional search

Searches simultaneously both forward from initial state and backward from goal state. Time complexity reduced from $O(b^d)$ to $O(b^{d/2})$. But... Does the node predecessor function exist? What if there are many possible goals? Must check a new node if it exists in other tree. Must keep at least one tree, space complexity $O(b^{d/2})$.

2.8 Comparing uninformed search strategies

Algorithm	Complete	Time	Space	Optimal
Breadth-first	Yes	b^d	b^d	Yes
Uniform-cost	Yes	$b^{1+c/e}$	$b^{1+c/e}$	Yes
Depth-first	No	b^m	bm	No
Depth-limited	No	b^l	bl	No
Iterative deepening	Yes	b^d	bd	Yes
Bidirectional	Yes	$b^{d/2}$	$b^{d/2}$	Yes

b - branching factor	**l** - depth limit
m - maximum depth of tree	**c** - cost of solution
d - depth of solution	**e** - cost of action

2.9 Informed search

Search can be improved by applying knowledge to better select which node to expand (best-first). An function to estimate the cost to reach a solution is called a search heuristic (h). Greedy search: Minimizes h(n) - the estimated cost of the cheapest path from n to the goal. Greedy search reduces search time compared to uninformed search, but is neither optimal nor complete.

2.9.1 A* search

A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals). As A* traverses the graph, it follows a path of the lowest expected total cost or distance, keeping a sorted priority queue of alternate path segments along the way.

It uses a knowledge-plus-heuristic cost function of node n (usually denoted f(n)) to determine the order in which the search visits nodes in the tree. The cost function is a sum of two functions:

- the past path-cost function, which is the known distance from the starting node to the current node n (usually denoted g(n))
- a future path-cost function, which is an admissible "heuristic estimate" of the distance from n to the goal (usually denoted h(n)).
- f(n) = g(n) + h(n)

The h(x) part of the f(x) function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, h(x) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.

A* is the most widely known informed search method. Identical to Uniform-Cost except that it minimizes f(n) instead of g(n).

Properties of A*:

- Optimal (and optimally efficient)

- Complete
- Time/space exponential (space most severe problem)

2.9.2 Heuristic functions

Some admissible h for 8-puzzle

- h1 - number of misplaced tiles
- h2 - sum of distances of tiles from their goal positions
- Neither overestimate true cost

Branching factor b of 8 - puzzle approx. 3. Effective branching factor b* using A* depends on chosen heuristic function h:

- h1 - effective b* 1.79 - 1.48 (depending on d)
- h2 - effective b* 1.79 - 1.26 (always better than h1)

Dramatic reduction of search time/space compared to uninformed search

3 Logical agents

3.1 Knowledge-based agents

Knowledge-based agents are able to:

- Maintain a description of the environment
- Deduce a course of action that will achieve goals

Knowledge-based agents have:

- A knowledge base
- Logical reasoning abilities

The performance of a knowledge-based agent is determined by its knowledge base.

A knowledge base is a set of representations of facts about the world, called sentences, expressed in a *knowledge representation language*. Knowledge base (KB) interface:

- TELL(KB,fact) - Add a new fact
- fact ← ASK(KB,query) - Retrieves a fact
- RETRACT(KB,fact) - Removes a fact

A knowledge - base agent can be built by **TELL**ing it what it needs to know (declarative approach). The agent can be used to solve problems by **ASK**ing questions.

```
 function KB_AGENT(percept) returns an action
 persistent: KB, the agent's knowledge base
             t, a counter, initially 0, indicating time
TELL(KB, MAKE_PERCEPT_SENTENCE(percept,t))
 action <= ASK(KB, MAKE_ACTION_QUERY(t))
TELL(KB, MAKE_ACTION_SENTENCE(action,t))
 t <= t + 1
 return action
```

3.2 Knowledge representation

Knowledge representation languages should be:

- Expressive and concise
- Unambiguous and independent of context
- Able to express incomplete knowledge
- Effective inference of new knowledge

Existing languages not suited:

- Programming languages - precise descriptions and recipes for machines
- Natural languages - flexible communication between humans

In AI, logic is used for knowledge representation.

3.2.1 Knowledge representation languages

- Syntax
  - How are legal sentences in the language composed
- Semantics
  - What do the sentences mean
  - What is the truth of every sentence with respect to each possible world (also called a model)
- Entailment
  - The fact that sentences logically follow from other sentences
- Inference
  - How to derive new sentences that are entailed by known ones

3.3 Logical reasoning

3.3.1 Logical entailment

Logical entailment between two sentences:
$\alpha \mid = \beta$
means that β *follows logically* from α: in every model (possible world) in which α is true, β is also true. We can also say that an entire KB (all sentences in the knowledge base) entails a sentence:
$KB \mid = \beta$
β follows logically from KB: in every model (possible world) in which KB is true, β is also true. Model checking: Can check entailment by reviewing all possible models.

3.3.2 Logical inference

Logical inference between two sentences
$\alpha \mid - \ \beta$
means that β can be derived from a by following an inference algorithm (can also say $KB \mid - \ \beta$). Model checking is an example of an inference algorithm.

3.3.3 Inference and entailment

"Entailment is like the needle being in the haystack; inference is like finding it"

Sound inference: The inference algorithm only derives entailed sentences

- Required property
- Model checking is sound

Complete inference: The inference algorithm can derive any entailed sentence

- Desirable property
- Not always possible

3.4 Propositional logic and first-order logic

Propositional logic

- Symbols represent true or false facts
- More complex sentences can be constructed with Boolean connectives (and, or, . . .)

First-order logic

- Symbols represent objects and predicates on objects
- More complex sentences can be constructed with connectives and quantifiers (for-all, there-is, . . .)

In AI, both propositional and and first-order logic are heavily used

3.4.1 Propositional logic

Syntax

Sentence	$\rightarrow$	AtomicSentence | ComplexSentence
AtomicSentence	$\rightarrow$	True | False
	|	P|Q|R
ComplexSentence	$\rightarrow$	(Sentence)
	|	Sentence Connective Sentence
	|	$\neg$ Sentence Connective
Connective	$\rightarrow$	$\wedge$| $\vee$ | $\Leftrightarrow$ | $\Rightarrow$

3.4.1.1 Logical connectives

$\wedge$ = and (Conjunction P $\wedge$ Q)
$\vee$ = or (Disjunction P $\vee$ Q)
$\neg$ = not (Negation $\neg$ P)
$\Leftrightarrow$ = equivalent (Equivalence (P $\wedge$ Q) $\Leftrightarrow$ (Q $\wedge$ P))
$\Rightarrow$ = implies (Implication (P $\wedge$ Q) $\Rightarrow$ R)

3.4.1.2 Semantic

Semantics defines the rules for determining the truth of a sentence with respect to a certain model. A model in propositional logic fixes the truth (true or false) of every propositional symbol. The truth of a complex sentence is given by the truth value of its parts and the connectives used

Truth table for logical connectives

P	**Q**	**$\neg$P**	**P $\wedge$ Q**	**P $\neg$ Q**	**P $\Rightarrow$ Q**	**P $\Leftrightarrow$ Q**
False	False	True	False	False	True	True
False	True	True	False	True	True	False
True	False	False	False	True	False	False
True	True	False	True	True	True	True

3.4.1.3 Complexity of propositional inference

The checking algorithm is sound and complete, but

- Time complexity is $O(2^n)$
- Space complexity is $O(n)$

All known inference algorithms for propositional logic have worst-case complexity that is exponential in the size of inputs

3.4.2 Equivalence, validity and satisfiability

Two sentences are *equivalent* if they are true in the same models. A sentence is *valid* (necessarily true, tautological) if it is true in all possible models. A sentence is *satisfiable* if it true in some model. A sentence that is not satisfiable is *unsatisfiable* (contradictory)

3.4.3 Inference by applying rules

An inference rule is a standard pattern of inference that can be applied to drive chains of conclusions leading to goal. A sequence of applications of inference rules is called a proof. Searching for proofs is similar (or in some cases identical) to problem-solving by search. Using rules for inference is an alternative to inference by model checking

3.4.3.1 Inference rules $\frac{\alpha}{\beta}$ for propositional logic

Modus ponens ("the way that affirms by affirming"):

$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$$

Modus ponens can be stated formally as:

$$\frac{P \to Q,\ P}{\therefore Q}$$

where the rule is that whenever an instance of "P $\to$ Q" and "P" appear by themselves on lines of a logical proof, Q can validly be placed on a subsequent line; furthermore, the premise P and the implication "dissolves", their only trace being the symbol Q that is retained for use later e.g. in a more complex deduction.

And-Elimination:

$$\frac{\alpha_1 \wedge \alpha_2 \wedge K \wedge \alpha_2}{\alpha_1}$$

Unit Resolution:

$$\frac{\alpha \vee \beta, \neg\beta}{\alpha}$$

3.4.3.2 The resolution rule

Takes two sentences with complementary parts and produces and new sentence with the part removed. Example:

$$\frac{P1,1 \vee P3,1 \neg P1,1 \vee \neg P2,2}{P3,1 \vee \neg P2,2}$$

It can be shown that resolution is sound and complete. Any complete search algorithm, using only resolution as inference rule, can derive any conclusion entailed by any knowledge base in propositional logic

3.4.4 Searching - forward and backward chaining

Forward chaining:

- Start with sentences in KB and generate consequences
- Uses inference rules in forward direction
- Also called *data-driven procedure*

Backward chaining:

- Start with goal to be proven and look for premises
- Uses inference rules in backward direction
- Also called *goal-directed procedure*

Specialized search algorithms for propositional logic can be very efficient

3.4.5 Keeping track of state of the world

Knowledge base contains general knowledge of how the world works plus specific percept history. All percept need to be indexed by time t to capture changes in percepts: $\neg Stench^3$, $Stench^4$, . . . The term *fluent* is used for any state variable that changes over time. *Effect axioms* are used to describe outcome of an action at the next time step. *Frame problem*: Effect axioms fail to state what does not change as a result of an action. Solved by writing *successor-state axioms* for fluents, e.g. $HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t)$. Propositional agent requires very large number of rules.

3.5 Wumpus agent

```
function PLAN–ROUTE(current, goals, allowed)
  returns an action sequence using A* search

function HYBRID–WUMPUS–AGENT(percept) returns an action
inputs: percept, a list, [stench, breeze, glitter, bump, scream]
persistent : KB, knowledge base, initially containing Wumpus "physics"
             t , a counter, initially 0, indicating time;
             plan , an action sequence, initially empty
```

```
TELL(KB, new percept and temporal "physics" sentences for time t)
if glitter then // grab the gold, plan safe retreat and climb out
  plan <= [ Grab ]+PLAN – ROUTE(current,[1,1],safe)+[Climb]
if plan is empty then // plan safe route to safe unvisited square
  plan <= PLAN–ROUTE(current,unvisited & safe, safe)
if plan is empty and HaveArrow t then // plan move to shoot at Wumpus
  plan <= PLAN–SHOT(current, possible–wumpus,safe)
if plan is empty then // go to a square that is not provably unsafe
  plan <= PLAN–ROUTE(current, unvisited & not–unsafe, safe)
if plan is empty then // mission impossible, plan retreat and climb out
  plan <= PLAN–ROUTE(current,[1,1],safe)+[Climb]
action <= POP(plan)
TELL(KB, MAKE–ACTION–SENTENCE(action,t))
t <= t + 1
return action
```

4 First-order Logic (FOL)

4.1 From propositonal to first-order logic

Features of propositional logic:
+ Declarative
+ Compositional
+ "Possible-world" semantics
- Lacks expressiveness

First-order logic:

- Extends propositional logic
- Keeps good features
- Adds expressiveness

4.2 First-order logic

First-order logic is based on "common sense" or linguistic concepts:

- Objects: people, houses, numbers, ...
- Properties: tall, red, ...
- Relations: brother, bigger than, ...
- Functions : father of, roof of, ...
- Variables : x, y, ... (takes objects as values)

First-order logic is the most important and best understood logic in philosophy, mathematics, and AI

4.3 Logical commitments

- Ontological commitment
  - Ontology - in philosophy, the study of "what is"

- What are the underlying assumptions of the logic with respect to the nature of reality

- Epistemological commitment
  - Epistemology - in philosophy, the study of "what can be known"
  - What are the underlying assumptions of the logic with respect to the nature of what the agent can know

Language	Ontological Commitment	Epistemological Commitment
Propositional logic	Facts	True/false/unknown
First-order logic	Facts, objects,relations	True/false/unknown
Temporal logic	Facts, objects,relations,times	True/false/unknown
Probability theory	Facts	Degree of belief 0 . . . 1
Fuzzy logic	Facts w/degree of truth	Known interval value

4.4 Syntax

Sentence	$\rightarrow$	AtomicSentence
	|	Sentence Connective Sentence
	|	Quantifier Variable, Λ Sentence
	|	$\neg$ Sentence
	|	(Sentence)
AtomicSentence	$\rightarrow$	Predicate(Term,K) | Term = Term
Term	$\rightarrow$	Function(Term, $\wedge$) | Constant t | Variable
Connective	$\rightarrow$	$\wedge$ | $\vee$ | $\Leftrightarrow$ | $\Rightarrow$
Quantifier	$\rightarrow$	$\forall$ | $\exists$

4.4.1 Constants, predicates, functions and terms

Constant symbols refer to specific objects in the world: John, 3, . . .
Predicate symbols refer to particular relations in the model: Brother, LargerThan, i.e. sets of objects that satisfy the relation
Function symbols refer to many-t-one object mappings: FatherOf
Term, a logical expression refers to an object in the model. Can be a Constant, a Variable, or a Function of other terms.

4.4.2 Atomic and complex sentences

Atomic sentences state basic facts.

Examples:
Brother(Richard,John)
Married(FatherOf(Richard),MotherOf(John))

Complex sentences is composed of atomic sentences by using logical connectives (same as for propositional logic)

Examples:
Brother(Richard, John) $\wedge$ Brother (John, Richard)
Older(John,30) $\Rightarrow$ $\neg$Younger(John, 30).

4.4.3 Quantifiers and equality

Quantifiers are used to express properties of classes of objects.

Universal quantification $\forall$
"For all . . . " quantification
Example: $\forall x\ Cat(x) \Rightarrow Mammal(x)$

Existential quantification $\exists$
"There exists . . . " quantification
Example: $\exists\ Sister(x, Spot) \wedge Cat(x)$

Equality =
Two terms refer to same object
Example: $Father(John) = Henry$

4.4.4 Kinship domain

The domain of family relationship

- Objects: Persons
- Unary predicates: Male, Female
- Binary predicates: Parent, Sibling, Brother, Sister, . . .
- Functions: Mother , Father

Example sentences:
$\forall m, c\ Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m, c)$
$\forall w, h\ Husband(h, w) \Leftrightarrow Male(h) \wedge Spouse(h, w)$
$\forall g, c\ Grandparent(c) \Leftrightarrow \exists p\ Parent(g, p) \wedge Parent(p, c)$
$\forall x, y\ Sibling(x, y) \Leftrightarrow x \neq y \wedge \exists p\ Parent(p, x) \wedge Parent(p, c)$

4.4.5 TELLing and ASKing sentences

TELL an agent about kinship using assertions
$TELL(KB, (\forall m, c\ Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(mc)))$
$TELL(KB, (Female(Maxi) \wedge Parent(Maxi, Spot) \wedge Parent(Spot, Boots)))$

ASK an agent by posing queries (or goals)
$ASK(KB, (Grandparent(Maxi, Boots)))$ Answer: Yes
$ASK(KB, (\exists x\ Child(x, Spot)))$ Answer: x = Boots

4.5 First-order inference

- Reduce to propositional inference
- Generalized Modus Ponens
- Forward chaining
- Backward chaining

- Resolution

- Require *substitution* and/or *unification* of terms

4.5.1 Substitution and unification

Substitution is the replacement of variable(s) in a sentence with expressions
$SUBST(x/Richard, y/John, Brother(x,y)) = Brother(Richard, John)$

Unification is finding substitutions (a unifier) that make different sentences look identical
$UNIFY(p,q) = \Theta \; where \; SUBST(\theta, p) = SUBST(\theta, p)$
E.g.
$UNIFY(Knows(John, x), Knows(John, Jane)) = \{x/Jane\}$
$UNIFY(Knows(John, x), Knows(y, Bill)) = \{x/Bill, y/John\}$
$UNIFY(Knows(John, x), Knows(y, Mother(y))) = \{y/John, x/Mother(John)\}$
$UNIFY(Knows(John, x), Knows(x, Elizabeth)) = fail$

4.6 Example knowledge base

Known facts
It is a crime for an American to sell weapons to hostile nations. The country Nono is an enemy of America and has some missiles, all sold to it by Colonel West, an American.

Knowledge base:
$American(x) \wedge Weapon(y) \wedge (Sells(x,y,z) \wedge Hostile(z) \Rightarrow Criminal(x)$
$Owns(Nono, M1), Missile(M1), Missile(x) \Rightarrow Weapon(x)$
$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono)$
$Enemy(x, America) \Rightarrow Hostile(x)$
$American(West), Enemy(Nono, America)$
This proves that West is a criminal.

4.7 Reduce to propositional inference

Can reduce first-order sentence to propositional sentence by instantiation rules

- UI - Universal instantiation - Substitutes a constant in KB for a universally quantified variable

- EI - Existential instantiation - Substitutes a new constant for an existentially quantified variable

Apply UI and EI systematically to replace all sentences with quantifiers with variable-free sentences. Can use propositional inference rules to derive proofs, but it is an inefficient procedure.

4.8 Generalized modus ponens - GMP

Assumes knowledge base containing facts and rules like
$p_1 \wedge p_2 \wedge \Lambda \wedge p_n \Rightarrow q$

The generalized modus ponens rule
$\frac{p_2', p_2', \Lambda, p_n', (p_1 \wedge p_2 \wedge \Lambda \wedge p_n \Rightarrow q}{SUBST(\Theta, q)}$

$SUBST(\Theta, p_i') = SUBST(\Theta, p_i) forall i$

Example
$Missile(M1) \quad SUBST\{x/M1\}$
$Owns(Nono, M1)$
$\frac{\forall x \ Missile(x) \wedge Owns(Nono,x) \Rightarrow Sells(West,Nono,x)}{Sells(West,Nono,M1)}$

4.9 Completeness

First-order logic is complete. If a sentence is entailed by KB, then this can be proved (Gödel's completeness theorem, 1930). But generalized modus ponens is not complete. However, first-order logic is semi-decidable. If the sentence is not entailed by the KB, this can not always be shown. Extension of first-order logic with mathematical induction is incomplete. There are true statements that cannot be proved (Gödel's incompleteness theorem, 1931).

4.10 Forward chaining - FC

Start with sentences in KB, apply inference rules in forward direction, adding new sentences until goal found or no further inference can be made.

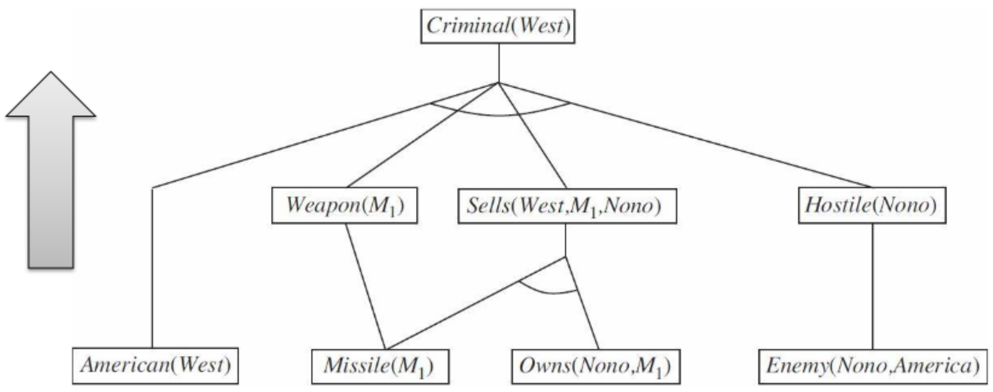


Figure 13: Forward chaining

4.10.1 Production systems using FC

Main features

- Consists of rule base and working memory
- Uses rule matching and forward chaining to add facts to the working memory until a solution is found
- *Rete networks* are used to speed up matching

Applications

- Used in many expert systems, especially early ones
- Design and configuration systems
- Real-time monitoring and alarming
- "Cognitive architectures" (SOAR)

4.11 Backward chaining - BC

Start with goal sentence, search for rules that support goal, adding new sub-goals until match with KB facts or no further inference can be made.

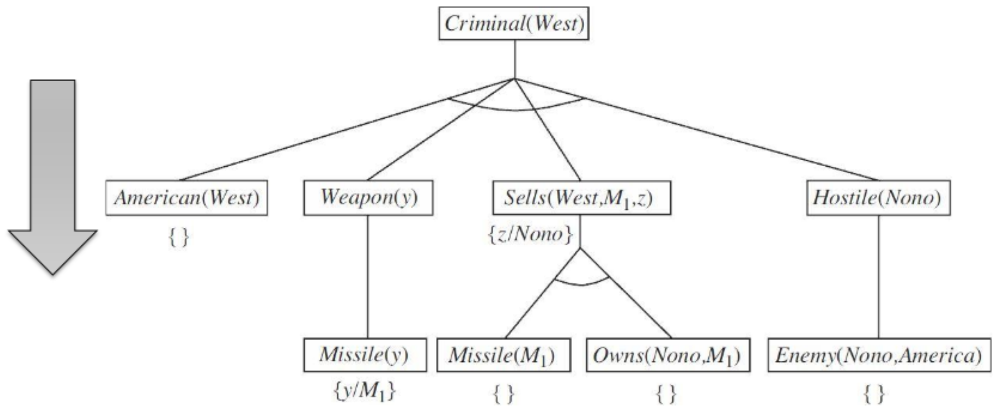


Figure 14: Backward chaining

4.11.1 Logic programming using BC

Main features

- Restricted form of first-order logic
- Mixes control information with declarative sentences
- Backward chaining search: Prove a goal

Prolog is the dominant logic programming language, and has been used for

- Expert systems
- Natural language systems
- Compilers
- Many others

4.12 Resolution - A complete inference procedure

Extends generalized modus ponens by allowing rules with disjunctive conclusions
$p_1 \wedge p_2 \wedge \Lambda \wedge p_n \Rightarrow q_1 \vee \Lambda \vee q_m$

Resolution rule:
$\frac{p \vee q, \neg p \vee r}{q \vee r}$

Assumes that all sentences are written in a normal form, for which there exists an efficient algorithm. E.g. eliminate implications, replace $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$. A complete resolution inference procedure works by refutation, i.e. include $\neg Goal$ in the KB. Then apply resolution rule until a contradiction is found.

5 Knowledge Engineering in FOL

5.1 Knowledge engineering

Knowledge engineering is the process of building a knowledge base (KB) for a domain. Carried out by *knowledge engineers* (KE) doing *knowledge acquisition*, often by interviewing domain experts. The KE investigates the domain, learns important concepts, and creates a formal representation of domain objects and relations. Most KBs are *special purpose*, covering a specific domain in detail. Other KBs are *general purpose*, valid across many domains.

5.1.1 Knowledge engineering vs. programming

Knowledge engineering	Programming
• Choosing a logic for knowledge representation	• Choosing a programming language
• Building a knowledge base of facts and rules	• Writing a program of data and control structures
• Implementing the inference procedure	• Choosing a compiler for the language
• Inferring new facts	• Running the program

5.1.2 Declarative approach

Main point is that knowledge engineering is *declarative*

- The knowledge engineer tells the system what is true
- The system knows how to infer new facts and solutions

Some advantages:

- More efficient/high-level development, less debugging
- The knowledge base can be (re-)used for other tasks
- The inference engine can be (re-)used for other knowledge bases

5.1.3 Knowledge engineering process (FOL)

1. Identify the task
2. Assemble relevant knowledge
3. Decide on a vocabulary of predicates, functions and concepts (ontology)
4. Encode general domain knowledge (axioms)
5. Encode specific problem instance
6. Pose queries to inference engine - get answers
7. Debug knowledge base

5.1.4 Electronics circuits domain

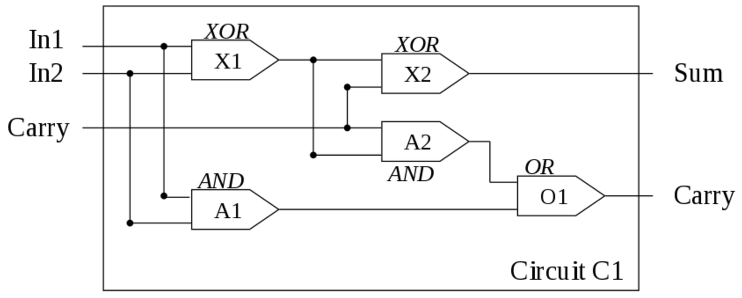


Figure 15: One-bit full adder circuit

Desired behavior:

In1	In2	CarryIn	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

1. Identify the task
   - We are only interested in circuit topology and behavior, not physical properties (which would be needed for design)
   - The purpose is to check circuit functional behavior, e.g.:
     Does the circuit shown actually add properly?
   - Also interested in circuit structure, e.g.:
     Does the circuit contain feedback loops?
2. Assemble relevant knowledge
   - Digital circuits are composed of wires and gates
   - A gate has input and output terminals
   - Signals flow on wires to input terminals of gates
   - A gate produces an output signal on the output wire
   - There are four gate types : AND, OR, XOR, NOT
3. Decide on a vocabulary
   - Constants for naming gates: $X_1,\ X_2,\ etc.$
   - Function for type of gate: $Type(X_1) = XOR$
   - Similar for circuit and terminal: $Circuit(C_1)\ Terminal(x)$
   - Terminal arity: $Arity(c, ins, outs)$

- Functions to select terminal: $Out(1, X_1)$, $In(2, X_2)$
- Predicate for connectivity between gates: $Connected(Out(1, X_1),\ In(2, X_2))$
- Function for signal at terminal: $Signal(terminal) = 1/0$

4. Encode general domain knowledge

- Two connected terminals have same signal
  $\forall t_1, t_2\ Terminal(t_1) \vee Terminal(t_2) \vee Connected(t_1.t_2) \Rightarrow Signal(t_1) = Signal(t_2)$
- Signal at a terminal is either on or off
  $\forall t\ Terminal(t) \Rightarrow (Signal(t) = 1) \vee (Signal(t) = 0)$
- Connected is commutative
  $\forall t_1, t_2\ Connected(t_1, t_2) \Leftrightarrow Connected(t_2, t_1)$
- OR gate behavior
  $\forall g\ Gate(g) \vee Type(g) = OR \Rightarrow (Signal(Out(1, g)) = 1 \Leftrightarrow \exists n\ Signal(In(n, g)) = 1)$

5. Encode specific problem instance

- Circuit
  $Circuit(C_1) \wedge Arity(C_1, 3, 2)$
- Gates
  $Gate(X_1) \wedge Type(X_1) = XOR$, $Gate(X_2) \wedge Type(X_2) = XOR$,
  $Gate(A_1) \wedge Type(A_1) = AND$, $Gate(A_2) \wedge Type(A_2) = AND$,
  $Gate(O_1) \wedge Type(O_1) = OR$
- Connections
  $Connected(Out(1, X_1), In(1, X_2))$, $Connected(In(1, C_1), In(1, X_1))$,
  $Connected(Out(1, X_1), In(2, A_2))$, $Connected(In(1, C_1), In(1, A_1))$,
  Etc.

6. Pose queries - get answers

- What combinations of inputs would cause first output of C1 (sum) to be 0 and the second output (carry) to be 1?
  $\exists i_1, i_2, i_3\ Signal(In(1, C_1) = i_1 \wedge Signal(In(2, C_1)) = 1_2 \wedge Signal(In(3, C_1)) = i_3 \wedge Signal(Out(1, C_1)) = 0 \wedge Signal(Out(2, c_1) = 1$
- The answer
  $(i_1 = 1 \wedge i_2 = 1 \wedge i_3 = 0) \vee (i_1 = 1 \wedge i_2 = 0 \wedge i_3 = 1) \vee (i_1 = 0 \wedge i_2 = 1 \wedge i_3 = 1)$

- What are the possible sets of values of all the terminals of the one-bit adder circuit?
  $\exists i_1, i_2, i_3\ Signal(In(1, C_1) = i_1 \wedge Signal(In(2, C_1)) = 1_2 \wedge Signal(In(3, C_1)) = i_3 \wedge Signal(Out(1, C_1)) = 0_1 \wedge Signal(Out(2, c_1) = o_2$
- The answer is the full table of circuit behavior
- Compare with specified behavior to **verify** that circuit behaves as desired

6 Knowledge Representation

6.1 General ontology

An ontology is a "vocabulary" and a "theory" of a certain "part of reality". *Special-purpose* ontologies apply to restricted domains (e.g. electronic circuits). *General-purpose* ontologies have wider applicability across domains, i.e.:

- Must include concepts that cover many subdomains
- Cannot use special "short-cuts" (such as ignoring time)
- Must allow unification of different types of knowledge

Genetic programming (GP) ontologies are useful in widening applicability of reasoning systems, e.g. by including time

6.1.1 Ontological engineering

Representing a general-purpose ontology is a difficult task called ontology engineering. Existing GP ontologies have been created in different ways:

- By team of trained ontologists
- By importing concepts from database(s)
- By extracting information from text documents
- By inviting anybody to enter commonsense knowledge

Ontological engineering has only been partially successful, and few large AI systems are based on GP ontologies (use special-purpose ontologies)

6.1.2 Elements of a general ontology

- Categories of objects
- Measures of quantities
- Composite objects
- Time, space, and change
- Events and processes
- Physical objects
- Substances
- Mental objects and beliefs

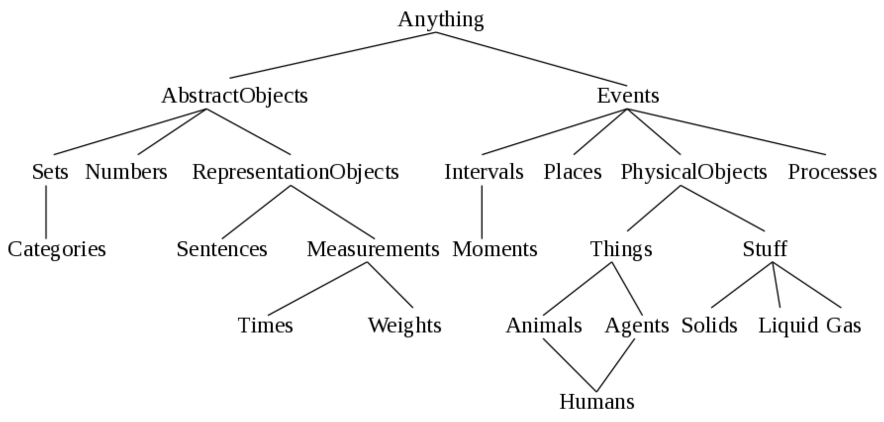


Figure 16: Top-level ontology of the world

6.2 Categories and objects

Categories are used to classify objects according to common properties or definitions
$\forall x \; x \in Tomatoes \Rightarrow Red(x) \wedge Round(x)$

Categories can be represented by:

- Predicates: $Tomato(x)$
- Sets: The constant Tomatoes represents set of tomatoes (reification)

Roles of category representations

- Instance relations (is-a): $x_1 \in Tomatoes$
- Taxonomical hierarchies (Subset): $Tomatoes \subset Fruit$
- Inheritance of properties
- (Exhaustive) decompositions

6.2.1 Objects and substance

Need to distinguish between substance and discrete objects.

Substance ("stuff")

- Mass nouns - not countable
- Intrinsic properties
- Part of a substance is (still) the same substance

Discrete objects ("things")

- Count nouns - countable
- Extrinsic properties
- Parts are (generally) not of same category

6.2.2 Composite objects

A *composite object* is an object that has other objects as parts. The *PartOf* relation defines the object containment, and is transitive and reflexive:
$PartOf(x, y) \wedge PartOf(y, z) \Rightarrow PartOf(x, z)$
$PartOf(x, x)$

Objects can be grouped in *PartOf* hierarchies, similar to *Subset* hierarchies. The structure of the composite object describes how the parts are related.

6.2.3 Measurements

Need to be able to represent properties like height, mass, cost, etc. Values for such properties are measures. *Unit functions* represent and convert measures:
$Length(L_1) = Inches(1.5) = Centimeters(3.81)$
$\forall l Centimeters(2.53xl) = Inches(l)$

Measures can be used to describe objects:
$Mass(Tomato_1) = Kilograms(0.16)$
$\forall d \; d \in Days \Rightarrow Duration(d) = Hours(24)$

Non-numerical measures can also be represented, but normally there is an order (e.g. $>$). Used in *qualitative physics*.

6.3 Events and processes

6.3.1 Event calculus

Event calculus: How to deal with change based on representing points of time. Reifies fluents (Norwegian: "forløp") and events:

- A fluent: $At(Shankar, Berkeley)$
- The fluent is true at time t: $T(At(Shankar, Berkeley), t)$

Events are instances of event categories:
$E_1 \in Flyings \wedge Flyer(E_1, Shankar) \wedge Origin(E_1, SF) \wedge Destination(E_1, LA)$

Event E_1 took place over interval i:
$Happens(E_1, i)$

Time intervals represented by (start, end) pairs:
$i = (t_1, t_2)$

6.3.1.1 Predicates

$T(f, t)$	Fluent f is true at time t
$Happens(e, i)$	Event e happens over interval i
$Initiates(e, f, t)$	Event e causes fluent f to start at t
$Terminates(e, f, t)$	Event e causes f to cease at t
$Clipped(f, i)$	Fluent f ceases to be true in int. i
$Restored(f, i)$	Fluent f becomes true in interval i

6.3.1.2 Using event calculus

Can extend predicate to cover intervals:
$T(f, (t_1, t_2)) \Leftrightarrow [\forall t(t_1 \leq t < t) \Rightarrow T(f, t)]$

Fluents and actions are defined with domain-specific axioms, e.g. in the wumpus world:
$Initiates(e, HaveArrow(a), t) \Leftrightarrow e = Start$
$Terminates(e, HaveArrow(a), t) \Leftrightarrow e \in Shootings(a)$

Can extend event calculus to represent simultaneous events, continuous events, etc.

6.3.1.3 Time intervals

Time intervals are partitioned into *moments* (zero duration) and *extended intervals*
$Partition(\{Moments, ExtendedIntervals\}, Intervals)$
$\forall i\ i \in Intervals \Rightarrow (i \in Moments \Leftrightarrow Duration(i) = 0)$

Functions *Start* and *End* delimit intervals:
$\forall i\ Intervals \Rightarrow Duration(i) = (Time(End(i)) - Time(Start(i)))$

May use e.g. January 1, 1900 as arbitrary time 0
$Time(Start(AD1900)) = Seconds(0)$

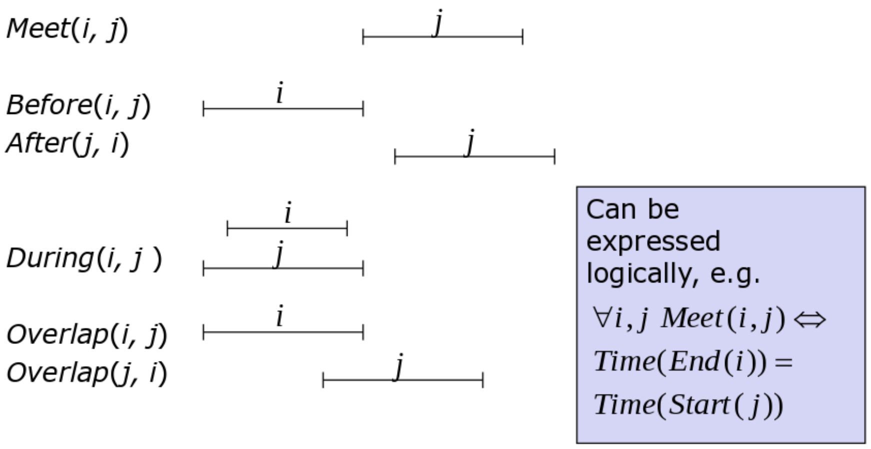

Meet(i, j)
i
j
Before(i, j)
After(j, i)
i
j
During(i, j)
i
j
Overlap(i, j)
Overlap(j, i)
i
j
Can be expressed logically, e.g.
∀i, j Meet(i, j) ⇔ Time(End(i)) = Time(Start(j))

Figure 17: Relations between time intervals

6.3.2 Mental events and mental objects

Need to represent *beliefs* in self and other agents, e.g. for controlling reasoning, or for planning actions that involve others. How are beliefs represented?

- Beliefs are reified as *mental objects*
- Mental objects are represented as strings in a language
- Inference rules for this language can be defined

Rules for reasoning about logical agents' use their beliefs:
$\forall a, p, q\ LogicalAgent(a) \land Believes(a, p) \land Believes(a, "p \Rightarrow q") \Rightarrow Believes(a, q)$
$\forall a, p\ LogicalAgent(a) \land Believes(a, p) \Rightarrow Believes(a, "Believes(Name(a), p)")$

6.3.3 Semantic networks

Graph representation of categories, objects, relations, etc. (i.e. essentially FOL). Natural representation of inheritance and default values. E.g. a Person has normally 2 legs, but the default is overridden for John with 1 leg

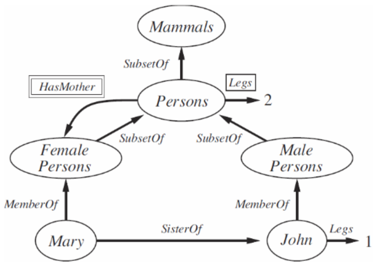


Figure 18: Example

6.4 Description logic (DL)

FOL enables ascribing properties to objects, while DL allows formal specification of and reasoning about *definitions* and *categories*. DL inference tasks:

- Subsumption - Check if a category is a subset of another
- Classification - Check if object belongs to a category
- Consistency - Check if category definition is satisfiable

DL evolved from semantic networks as a more formalized approach, still based on taxonomies. DL in different versions is the logical foundation for the Semantic Web.

6.4.1 CLASSIC

CLASSIC is an early example of DL, in which definitions can be stated and reasoned about. Simple category definitions:
$Single = And(Unmarried, Adult)$
$Bachelor = And(Unmarried, Adult, Male)$

CLASSIC can answer questions like:

- Is category Bachelor subsumed by category Single?
- Is the individual Adam of category Bachelor?

CLASSIC definitions can be translated to FOL, but inference in DL is more efficient

6.5 Default and non-monotonic logic

Classical logic is *monotonic*: true statements remain true after new facts are added to KB:
If $KB \models \alpha$, then $KB \wedge \beta \models \alpha$

In the closed-world assumption (facts not mentioned assumed false), monotonicity is violated:
If α is not mentioned in KB, then $KB \models \neg\alpha$, but $KB \wedge \alpha \models \alpha$

Non-monotonic reasoning is widespread in common-sense reasoning. We assume default in absence of other input, and are able to retract assumption if new evidence occurs. Non-monotonic logics support such reasoning

6.5.1 Circumscription

Circumscription (Norwegian: "begrensning") is a more powerful version of the closed-world assumption. The idea is to specify particular predicates "as false as possible", i.e. false for every object except for those for which they are known to be true. E.g. for the default that birds can fly:
$Bird(x) \land \neg Abnormal(x) \Rightarrow Flies(x)$

If Abnormal is *circumscribed*, a circumscriptive reasoner can:

- Assume $\neg Abnormal(x)$ unless the opposite is known
- Infer $Flies(Tweety)$ from $Bird(Tweety)$
- Retract the conclusion if $Abnormal(Tweety)$ is asserted

6.5.2 Truth maintenance systems (TMS)

Many inferences in the KB may have default status, and may need to be retracted in a process called *belief revision*.

- E.g. KB contains statement $P(adefault)$
- New evidence that P is not true: $TELL(KB, \neg P)$
- To avoid contradiction: $RETRACT(KB, P)$
- Other statements may have been added by P, e.g. Q if the KB contains $P \Rightarrow Q$, and Q may also have to go
- However, Q may also be true if the KB contains $R \Rightarrow Q$, in which case Q need not be retracted after all.

Systems to handle such "book keeping" are called *Truth Maintenance Systems (TMS)*

6.6 Example: Internet shopping world

An agent that understands and acts in an internet shopping environment. The task is to shop for a product on the Web, given the user's product description. The product description may be precise, in which case the agent should find the best price. In other cases the description is only partial, and the agent has to compare products. The shopping agent depends on having product knowledge, incl. category hierarchies

6.6.1 PEAS specification of shopping agent

Performance goal:	Recommend product(s) to match user's description
Environment:	All of the Web
Actions:	Follow links, retrieve page contents
Sensors:	Web pages: HTML, XML

6.6.2 Outline of shopping agent behavior

Start at home page of known web store(s). Must have knowledge of relevant web addresses, such as www.amazon.com etc. Spread out from home page, following links to relevant pages containing product offers. Must be able to identify page relevance, using product category ontologies, as well as parse page contents to detect product offers. Having located one or more product offers, agent must compare and recommend product. Comparison range from simple price ranking to complex tradeoffs in several dimensions.

7 Classical Planning

7.1 What is planning?

Planning is a type of problem solving in which the agent uses beliefs about actions and their consequences to find a solution plan, where a plan is a sequence of actions that leads from an initial state to a goal state.

7.1.1 Previously described approaches

Planning by search (section 2)

- Atomic representations of states
- Very large number of possible actions
- Needs good domain heuristics to bound search space

Planning by logical reasoning (section 3)

- Hybrid agent can use domain-independent heuristics
- But relies on propositional inference (no variables)
- Model size rises sharply with problem complexity

Neither of these approaches scale directly to industrially significant problems.

7.2 Plan representation

7.2.1 Factored plan representation

Factored representation of:

- Initial state
- Available actions in a state
- Results of applying actions
- Goal tests

Representation language PDDL

- Planning Domain Definition Language
- Developed from early AI planners, e.g. STRIPS, pioneering robot work at Stanford in early 1970's

Used for *classical* planning: Environment is observable, deterministic, finite, static, and discrete

7.2.2 Representation of states and goals

States are represented by conjunctions of function-free ground literals in first-order logic. Example: $At(Plane_1, Melbourne) \wedge At(Plane_2, Sydney)$. Closed-world assumption: Any condition not mentioned in a state is assumed to be false.
Goal state - a partially specified state, satisfied by any state that contains the goal conditions. Example goal: $At(Plane_2, Tahiti)$.

7.2.3 Representation of actions

An action schema has three components:

- *Action* description: Name and parameters (universally quantified variables)
- *Precondition*: Conjunction of positive literals stating what must be true before action application
- *Effect*: Conjunction of positive or negative literals stating how situation changes with operator application

Example:
$Action(Fly(p, from, to),$
$\quad PRECOND : At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to),$
$\quad EFFECT : \neg At(p, from) \wedge At(p, to))$

7.2.4 How are planning actions applied?

Actions are applicable in states that satisfy its preconditions (by binding variables)

- State: $At(P_1, JFK) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(JFK) \wedge Airport(SFO)$
- Precondition: $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$
- Binding: $p/P_1, from/JFK, to/SFO$

State after executing action is same as before, except positive effects added (add list) and negative deleted (delete list).
New state: $At(P_1, SFO) \wedge At(P_2, SFO) \wedge Plane(P_1) \wedge Plane(P_2) \wedge Airport(JFK) \wedge Airport(SFO)$

7.2.5 Planning solution

The planned actions that will take the agent from the initial state to the goal state. Simple version: An action sequence, such that when executed from the initial state, results in a final state that satisfies the goal. More complex cases: Partially ordered set of actions, such that every action sequence that respects the partial order is a solution

7.2.6 Example - Air cargo planning in PDDL

- $Init(At(C1, SFO) \wedge At(C2, JFK) \wedge At(P1, SFO) \wedge At(P2, JFK) \wedge Cargo(C1) \wedge Cargo(C2) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO))$
- $Goal(At(C1, JFK) \wedge At(C2, SFO))$
- $Action(Load(c, p, a),$
  $\quad PRECOND : At(c, a) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a),$
  $\quad EFFECT : \neg At(c, a) \wedge In(c, p))$
- $Action(Unload(c, p, a),$
  $\quad PRECOND : In(c, p) \wedge At(p, a) \wedge Cargo(c) \wedge Plane(p) \wedge Airport(a),$
  $\quad EFFECT : At(c, a) \wedge \neg In(c, p))$
- $Action(Fly(p, from, to),$
  $\quad PRECOND : At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to),$
  $\quad EFFECT : \neg At(p, from) \wedge At(p, to))$

- From initial state:
  $Init(At(C1, SFO) \wedge At(C2, JFK) \wedge At(P1, SFO) \wedge At(P2, JFK) \wedge Cargo(C1) \wedge Cargo(C2) \wedge Plane(P1) \wedge Plane(P2) \wedge Airport(JFK) \wedge Airport(SFO))$
- To goal state:
  $Goal(At(C1, JFK) \wedge At(C2, SFO))$
- Solution - a sequence of actions:
  $[Load(C1, P1, SFO), Fly(P1, SFO, JFK), Unload(C1, P1, JFK),$
  $Load(C2, P2, JFK), Fly(P2, JFK, SFO), Unload(C2, P2, SFO)]$
- How can the planner generate the plan?

7.2.7 Current popular planning approaches

- Forward state-space search with strong heuristics
- Planning graphs and GRAPHPLAN algorithm
- Partial order planning in plan space
- Planning as Boolean satisfiability (SAT)
- Planning as first-order deduction
- Planning as constraint-satisfaction

7.3 State-space search

7.3.1 Forward and backward state search

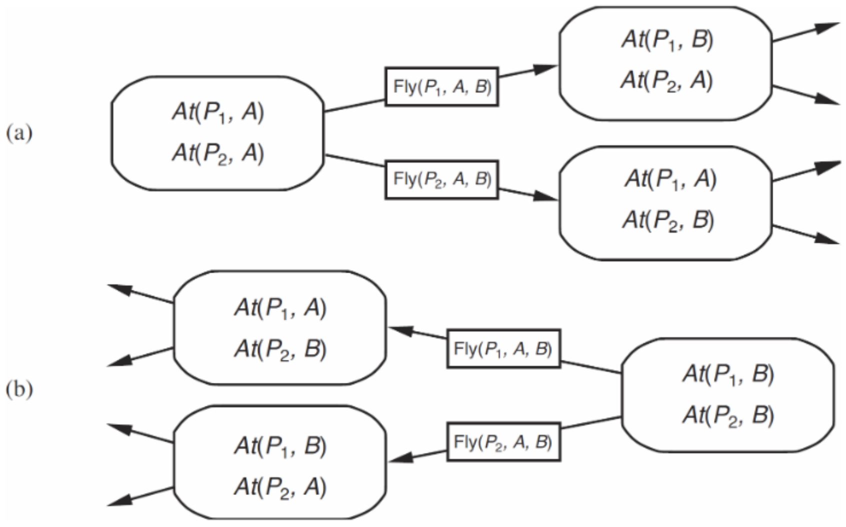


Figure 19: Forward and backward state search

7.3.2 Forward state-space search

Progression planning:

- Start in initial state
- Apply actions whose preconditions are satisfied
- Generate successor states by adding/deleting literals
- Check if successor state satisfies goal test

Can be highly inefficient

- All actions are applied, even when irrelevant
- Large branching factor (many possible actions)

Heuristics to guide search are required!

7.3.3 Backward state-space search

Regression planning:

- Start in goal state
- Apply actions that are relevant and consistent
  - Relevant: The action can lead to the goal (adds goal literal)
  - Consistent: The action does not undo (delete) a goal literal
- Create predecessor states
- Continue until initial state is satisfied

More efficient, but still requires heuristics. State-space searches can only produce linear plans.

7.3.4 Heuristics for planning

Neither forward nor backward search is efficient without a good heuristic, which has to be admissible (i.e. optimistic). Possible heuristics include:

- Adding more edges to the search graph, thereby making it easier to find a solution path, e.g. ignore pre-conditions or ignore delete lists
- Create state abstractions, many-to-one mapping from ground states to abstract ones, solve problem in the abstract space, and map down to ground again

Heuristics generate estimates h(s) for remaining cost of a state that can be used by e.g. A*.

7.4 Planning graphs

A planning graph is a special data structure that can be used as a heuristic in search algorithms or directly in an algorithm that generates a solution plan. Directed graph organized into one level for each time step of plan, where a level contains all literals that may be true at that step. Literals may be mutually exclusive (mutex links). Works only for propositional planning problems (no variables), but action schemas with variables may be converted to this form.

7.4.1 Example planning problem

Goal: "Have cake and eat cake too"

$Init(Have(Cake))$
$Goal(Have(Cake) \wedge Eaten(Cake))$
$Action(Eat(Cake)$
$\quad PRECOND : Have(Cake)$
$\quad EFFECT : \neg Have(Cake) \wedge Eaten(Cake))$
$Action(Bake(Cake)$
$\quad PRECOND : \neg Have(Cake)$
$\quad EFFECT : Have(Cake))$

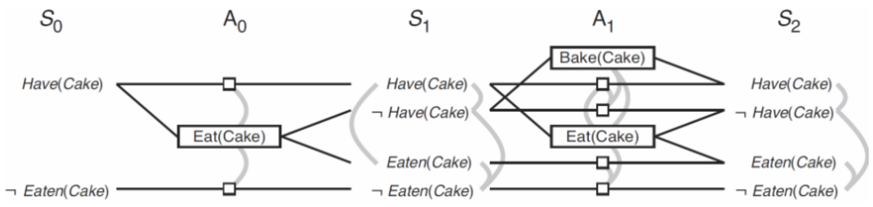


Figure 20: Planning graph for the example

Alternating state and action layers. Real and "persistence" actions (small rectangles). *Mutex* links (grey arcs) btw. incompatible states. Graph levels off at S_2 (states repeat themselves).

7.4.2 Mutex links (mutual exclusion)

Between two actions:

- Inconsistent effects - one action negates an effect of the other (e.g. $Eat(Cake)$ and persistent $Have(Cake)$)
- Interference - an effect of one action negates a pre-condition of the other (e.g. $Eat(Cake)$ and $Have(Cake)$)
- Competing needs - a pre-condition of one action negates a pre-condition of the other (e.g. $Eat(Cake)$ and $Bake(Cake)$)

Between two states (literals):

- One literal is the negation of the other
- Each possible pair of actions that could achieve the two literals is mutually exclusive

7.5 GRAPHPLAN algorithm

Uses a planning graph to extract a solution to a planning problem. Repeatedly:

- Extend planning graph by one level
- If all goal literals are included non-mutex in level
  - Try to extract solution that does not violate any mutex links, by following links backward in graph
  - Return solution if successful extraction
- If the graph has leveled off then report failure

Creating planning graph is only of polynomial complexity, but plan extraction is exponential.

7.5.1 Extracting a solution

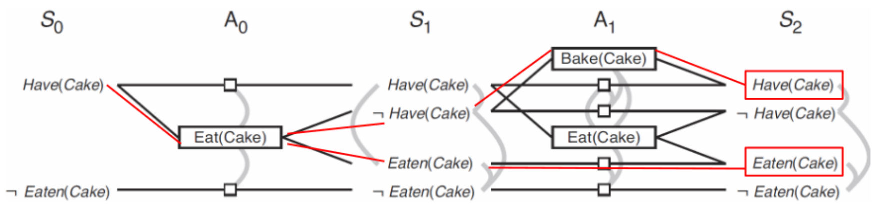


Figure 21: Extracting a solution

The goal is $Have(Cake) \wedge Eaten(Cake)$
Both goal literals non-mutex in S_2
$Bake(Cake)$ and $Eaten(Cake)$ non-mutex in A_1
$\neg Have(Cake)$ and $Eaten(Cake)$ non-mutex in S_1
$Eat(Cake)$ non-mutex in A_0
$Have(Cake)$ in S_0 is initial state

7.6 Partial-order planning

7.6.1 Partial order planning in plan space

Each node in the search space corresponds to a (partial) plan. Search starts with empty plan that is expanded progressively until complete plan is found. Search operators work in plan space, e.g. *add step*, *add ordering*, etc. The solution is the final plan, the path to it is irrelevant. Can create *partially ordered* plans.

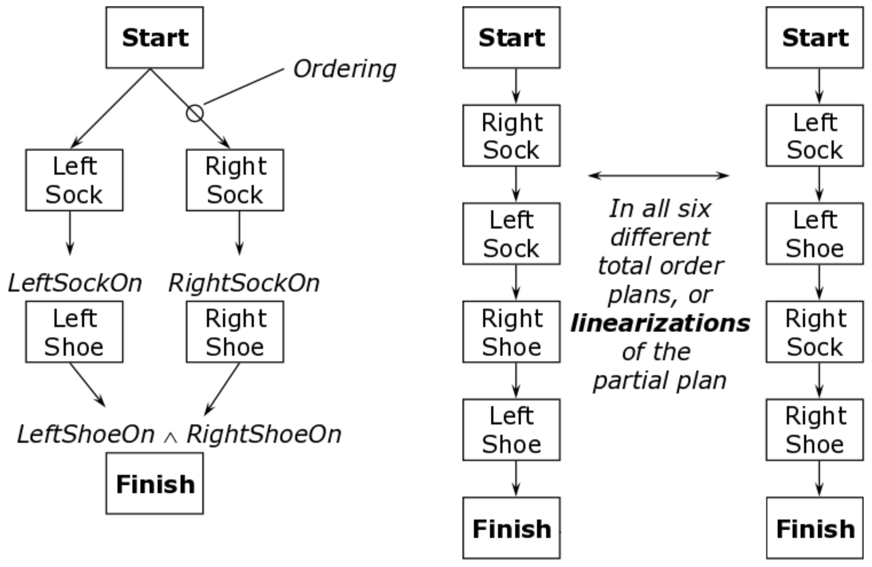


Figure 22: Example - Partial and total order plans

7.6.2 Partial-order plan representation

A set of steps, where each step is an action (taken from action set of planning problem). Initial empty plan contains just Start (no precondition, initial state as effect) and Finish (goal as precondition, no effects). A set of step ordering constraints of the form $A < B$ ("A before B"): A must be executed before B. A set of *causal links* $A \xrightarrow{C} B$, "A achieves c for B": the purpose of A is to achieve precondition c for B; no action is allowed between A and B that negates c. Set of open preconditions, not achieved by any action yet. The planner must reduce this set to empty set.

7.6.3 Protected causal links

Causal links in a partial plan are protected by ensuring that threats (steps that might delete the protected condition) are ordered to come before or after the protected link.

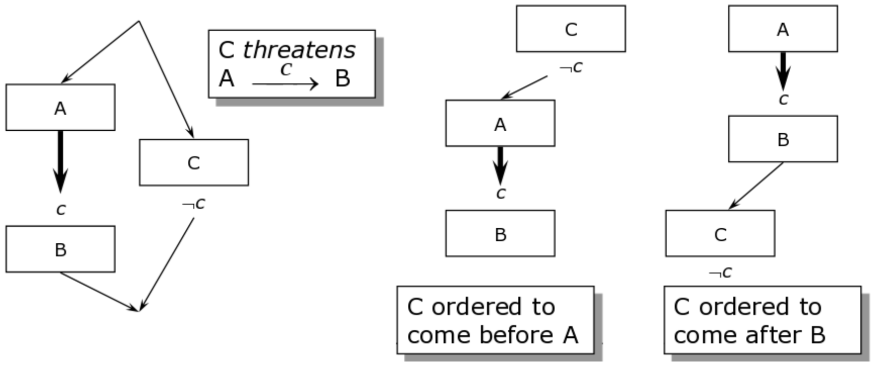

C threatens A $\xrightarrow{c}$ B
A
B
C
c
¬c
C ordered to come before A
C ordered to come after B

Figure 23: Protected causal links

7.6.4 POP - Partial Order Planning

Start with initial plan

- Contains Start and Finish steps
- All preconditions of Finish (goals) as open preconditions
- The ordering constraint $Start < Finish$, no causal links

Repeatedly

- Pick arbitrarily one open precondition c on an action B
- Generate a successor plan for every consistent way of choosing an action A that achieves c
- Stop when a solution has been found, i.e. when there are no open preconditions for any action

Successful solution plan

- Complete and consistent plan the agent can execute
- May be partial, agent may choose arbitrary linearization

7.6.4.1 Example

$Init(At(Flat, Axle) \wedge At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(Spare, Trunk),$
$\quad PRECOND : At(Spare, Trunk),$
$\quad EFFECT : \neg At(Spare, Trunk) \wedge At(Spare, Ground))$

$Action(Remove(Flat, Axle),$
$\quad PRECOND : At(Flat, Axle), \qquad EFFECT : \neg At(Flat, Axle) \wedge At(Flat, Ground))$
$Action(PutOn(Spare, Axle),$
$\quad PRECOND : At(Spare, Ground) \wedge \neg At(Flat, Axle),$
$\quad EFFECT : \neg At(Spare, Ground) \wedge At(Spare, Axle))$

Initial Plan

For each planning iteration, one step will be added. If this leads to an inconsistent state, the planner will backtrack. The planner will only consider steps that serve to achieve a precondition that has not yet been achieved

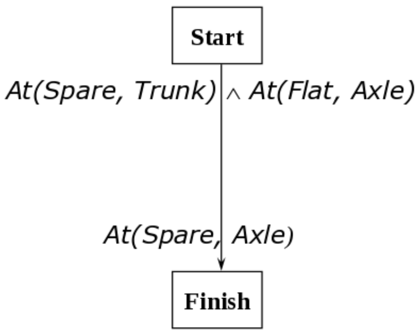


Figure 24: Initial plan

Achieving open preconditions

Start by selecting $PutOn$ action that achieves $Finish$. Select $At(Spare, Ground)$ precondition of $PutOn$, and choose $Remove(Spare, Trunk)$ action. The planner will protect the causal links by not inserting new steps that violate achievements.

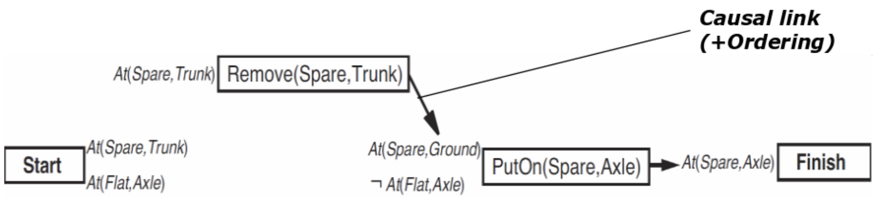


Figure 25: Achieving open preconditions

Finishing the plan

Planner selects to achieve $\neg At(Flat, Axle)$ precondition of $PutOn$ by $Remove(Flat, Axle)$. Final two preconditions are satisfied by $Start$.

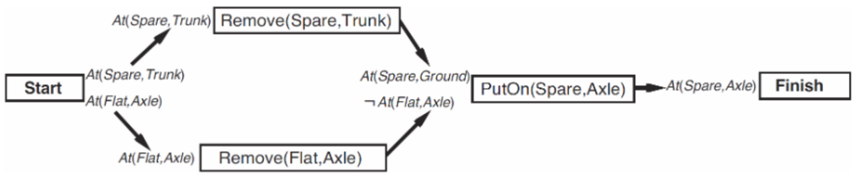


Figure 26: Finishing the plan

8 Planning and Acting

Classical planners assume:

- Fully observable, static and deterministic domains
- Correct and complete action descriptions
- . . . allowing a "plan-first-then-act" planning approach

. . . but in the real world:

- The world is dynamic, and time cannot be ignored
- Information on the world is incomplete and incorrect
- . . . the agent must be prepared for unexpected events

Plus - scaling up to real-world problem size!

8.1 Planning and scheduling

8.1.1 Time, schedules, and resources

The PDDL (Planning Domain Definition Language) language allows events (actions) and ordering of events, but not time duratio. In real-life planning, we must take duration, delays, etc. into account (not just ordering). *Job shop scheduling*:

- The problem is to complete a set of jobs
- Each job consists of a set of actions, with given duration and resource requirements
- Determine a schedule that minimizes total time (*makespan*) needed while respecting resource constraints

Must extend representation language to express duration and resource constraints

8.1.1.1 Example - Assembling two cars

- Jobs($\{AddEngine1 < AddWheels1 < Inspect1\}, \{AddEngine2 < AddWheels2 < Inspect2\}$)
- Resources($EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500)$)
- Action($AddEngine1, DURATION : 30, USE : EngineHoists(1)$)
- Action($AddEngine2, DURATION : 60, USE : EngineHoists(1)$)
- Action($AddWheels1, DURATION : 30, CONSUME : LugNuts(20), USE : WheelStations(1)$)
- Action($AddWheels2, DURATION : 15, CONSUME : LugNuts(20), USE : WheelStations(1)$)
- Action($InspectI, DURATION : 10, USE : Inspectors(1)$)

8.1.2 Scheduling - No resource constraints

- Partial order plan produced by e.g. POP (Partial Order Planning)
- To create a schedule, we must place actions on a timeline
- Can use critical path method (CPM): the longest path, no slack - determines total duration
- Shortest duration schedule, given partial-order plan:
  85 minutes

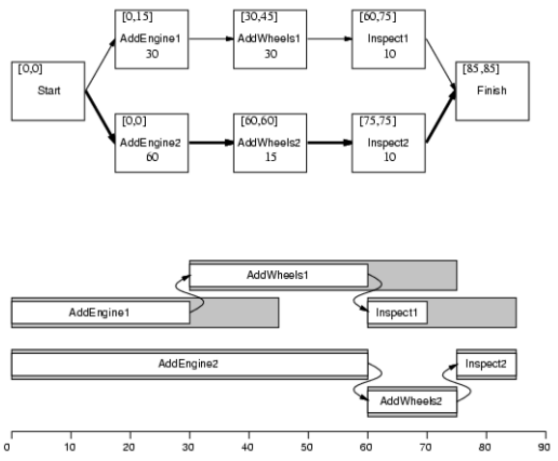


Figure 27: Scheduling

8.1.3 Scheduling with resource constraints

Actions typically require resources:

- *Consumable* resources - e.g. *LugNuts*
- *Reusable* resources - e.g. *EngineHoists*

Resource constraints make scheduling more complex because of interaction between actions. AI and OR (Operations Research) methods can be used to solve scheduling problems with resources. Shortest duration gone up from 85 to 115 minutes.

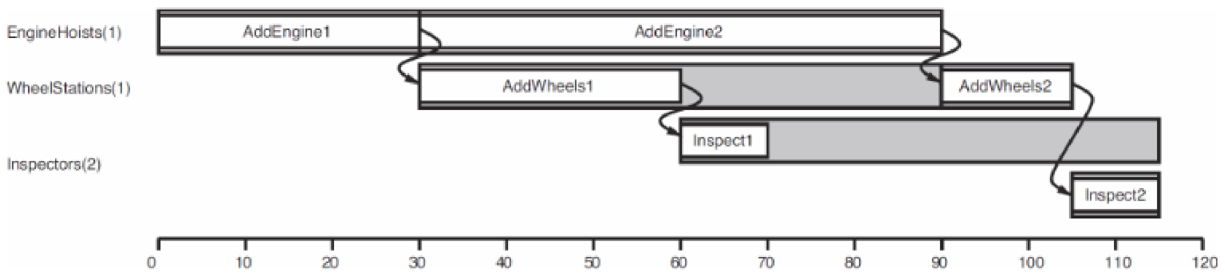


Figure 28: Scheduling with resource constraints

8.1.4 Usage

The approach shown here is common in real-world AI applications for manufacturing scheduling, airline scheduling, etc.:

- First generate partial-order plan without timing information (*planning*)
- Then use separate algorithm to find optimal (or satisfactory) time behavior (*scheduling*)

In some cases it may be better to interleave planning and scheduling, e.g. to consider temporal constraints already at the planning stage

8.2 Hierarchical task networks (HTN)

8.2.1 Reduce complexity by decomposition

Often possible to reduce problem complexity by decompose to subproblems, solve independently, and assemble solution. Hierarchical Task Networks (HTN):

- Planner keeps library of subplans
- Extend planning algorithm to use subplans
- Can reduce time&space requirements considerably

Most real-world planners use HTN variants

8.3 Planning in nondeterministic domains

Nondeterministic worlds:

- *Bounded* nondeterminism: Effects can be enumerated, but agent cannot know in advance which one will occur
- *Unbounded* nondeteminism: The set of possible effects is unbounded or too large to enumerate

Planning for bounded nondeterminism:

- Sensorless planning
- Contingent planning

Planning for unbounded nondeterminism:

- Online replanning
- Continuous planning

8.3.1 Sensorless planning

Agent has no sensors to tell which state it is in, therefore each action might lead to one of several possible outcomes. Must reason about sets of states (belief states), and make sure it arrives in a goal state regardless of where it comes from and results of actions. Nondeterminism of the environment does not matter - the agent cannot detect the difference anyway. The required reasoning is often not feasible, and sensorless planning is therefore often not applicable

8.3.2 Contingent planning

Constructs conditional plans with branches for each (enumerable) possible situation. Decides which action to choose based on special sensing actions that become parts of the plan. Can also tackle partially observable domains by including reasoning about belief states (as in sensorless planning). Planning algorithms have been extended to produce conditional branching plans

8.3.3 Online replanning

Monitors situation as plan unfold, detects when things go wrong. Performs replanning to find new ways to reach goals, if possible by repairing current plan

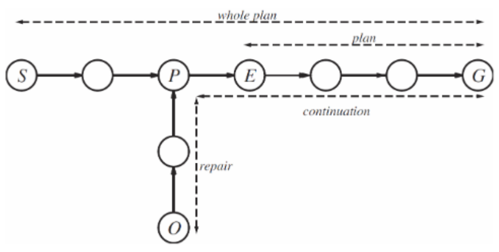


Figure 29: Online replanning

- Agent proceeds from S, and next expects E following original *whole-plan*
- Detects that it's actually in O
- Creates a repair plan that takes it from O to a state P in original plan
- New plan to reach G becomes *repair* + *continuation*

8.3.4 Contingent planning vs. replanning

Contingent planning:

- All actions in the real world have additional outcomes
- Number of possible outcomes grows exponentially with plan size, most of them are highly improbable
- Only one outcome will actually occur

Replanning:

- Basically assumes that no failure occurs
- Tries to fix problems as they occur
- May produce fragile plans, hard to fix if things go wrong

8.3.5 Continuous spectrum of planners

Contingent planning and replanning are extremes of a spectrum, where intermediate solutions exist

- Disjunctive outcomes for actions where more than one outcome is likely
- Agent can insert sensing action to detect what happened and construct corresponding conditional plan
- Other contingencies dealt with by replanning

More generally, agents in complex domains and with incomplete/incorrect information should:

- Assess likelihood and costs of various outcomes
- Construct plan that maximizes probability of success and minimizes cost
- Ignore contingencies that are unlikely or easy to deal with

8.4 Continuous planning

The planner persists over time - never stops, and interleaves planning, sensing and execution. The continuously planning agent must:

- Execute steps of current plan (even if not complete)
- Refine plan if not applicable or in conflict
- Modify plan in light of new information
- Formulate new goals when required

Planners, e.g. partial-order planning (POP) can be extended to provide required functionality

8.5 Multi-agent planning

Single-agent planning works against "nature", but in many cases the environment includes other agents with their own goals. Multi-agent environments can be:

- *Cooperative*: Agents work together to achieve some common goal
- *Competitive*: Agents have conflicting goals

Multi-agent architectures and applications, incl. planning, represent very active AI research area

8.5.1 Coordination of multi-agent planning

Cooperative planning can produce *joint* plans. For each agent, the joint plan tells what to do. If each follows its plan, overall goal will be achieved. Problems arise if several joint plans are possible. Each agent must know which plan to follow. Requires some form of coordination. Coordination can be by:

- Convention or *social law*
- Inter-agent *communication*

9 Quantifying Uncertainty

9.1 Agents and uncertainty

In all real domains, the agent must be able to handle uncertainty, due to:

- *Limited resources*: Cannot exhaustively enumerate all possible situations and consequences of actions
- *Theoretical ignorance*: No theory for the domain exists
- *Practical ignorance*: All necessary data is not available

The agent still has to act, and needs to make decisions where uncertainty is explicitly recognized. Probability can be used to summarize the state of the agent's beliefs (or ignorance)

10 Probabilistic Reasoning

10.1 Status of probability sentences

Statement: "The patient has cavity with probability 0.8"

In *logic*, a sentence is true or false, depending on the interpretation and the world. In *probability theory*, the probability assigned to a sentence depends on the evidence so far.

- *Prior* (unconditional) probability: Before any evidence
- *Posterior* (conditional) probability: After some evidence

Probability is more like entailment than truth!

10.2 Probability, utility and decisions

The agent can use *probability theory* to reason about uncertainty. The agent can use *utility theory* for rational selection of actions based on preferences. *Decision theory* is a general theory for combining probability with rational decisions.

$Decisiontheory = Probabilitytheory + Utilitytheory$

10.2.1 Decision theoretic agent

```
function DT—AGENT(percept) returns an action
persistent : belief—state, probabilistic beliefs about the state of the world
             action, the agent's action
update belief—state based on action and percept calculate outcome
             probabilities for actions, given action descriptions and current
             belief—state
select action with highest expected utility given probabilities of outcomes
             and utility information
return action
```

10.2.2 Basic probability notation

A probability model is a set of propositions, expressed in terms of random variables with domains:

- Boolean - E.g. Cavity : $< true, false >$
- Discrete - E.g. Weather : $< sunny, rainy, cloudy, snow >$
- Continuous - E.g. Index : $[0, 1]$

An atomic event is an assignment of particular values to all variables of the domain

- E.g. $Cavity = false \land Toothache = true$
- Mutually exclusive (only one event can be true at a time)
- Exhaustive (at least one must be true)

Prior (unconditional) probability of a proposition: P(A)
$P(Cavity) = 0.1$ . . . i.e. $P(Cavity = true) = 0.1$

Probability distribution of variable P(v)
$P(Weather) = (0.7, 0.2, 0.08, 0.02)$

Joint probability distribution

- Table of probabilities for all combinations: $P(v_1, v_2)$
- $P(Weather, Cavity)$ is a 4 x 2 table of probabilities (must sum to 1)
- *Full joint distribution*: all domain variables included

Conditional (posterior) probability: P(A|B)
$P(Cavity|Toothache) = 0.8$

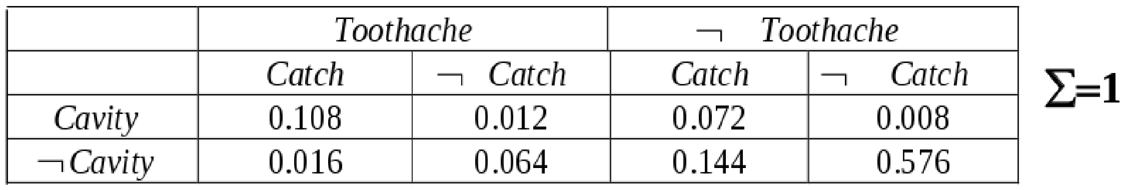
Product rule:
$P(A \wedge B) = P(A|B)P(B)$
$P(A \wedge B) = P(B|A)P(A)$
$P(A|B) = P(A \wedge B)/P(B)$

10.2.3 Axioms of probability

Basic axioms:
$0 \leq P(A) \leq 1$
$P(True) = 1 \quad P(False) = 0$
$P(A \vee B) = P(A) + P(B) - P(A \wedge B)$

All other properties can be derived, e.g.
$P(A \vee \neg A) = P(A) + P(\neg A) - P(A \wedge \neg A)$
$P(True) = P(A) + P(\neg A) - P(False)$
$1 = P(A) + P(\neg A)$
$P(\neg A) = 1 - P(A)$

10.2.4 Inference using full joint distribution



	Toothache		$\neg$ *Toothache*	
	Catch	$\neg$ *Catch*	*Catch*	$\neg$ *Catch*
Cavity	0.108	0.012	0.072	0.008
$\neg$*Cavity*	0.016	0.064	0.144	0.576

Σ=1

Figure 30: Inference using full joint distribution

Probability of combination of events:
$P(cavity \vee toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28$

Unconditional probability of a variable (marginalization)
$P(cavity) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2$

Conditional probability (using product rule)
$P(cavity|toothache) = P(cavity \wedge toothache)/P(toothache) = (0.108+0.012)/(0.108+0.012+0.16+0.064) = 0.6$

Problem: This approach does not scale up! Table size and calculation time is $O(2^n)$ for n Boolean variables.

10.3 Bayes' rule

Bayes' rule:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)}$$

Easily derived from product rule. Underlies most modern AI systems for probabilistic inference. Main application:

- How to use *prior* and *causal* knowledge ($cause \Rightarrow effect$) to derive *diagnosis* ($effect \Rightarrow cause$)

$$P(cause|effect) = \frac{P(effect|cause)P(cause)}{P(effect)}$$

Use of causal knowledge is crucial in making probabilistic reasoning sufficiently robust in applications

10.3.1 Simple example of using Bayes' rule

Diagnosing meningitis:

- Prior probability of meningitis: $P(M) = 1/50000$
- Prior probability of stiff neck: $P(S) = 1/20$
- Meningitis causes stiff neck: $P(S|M) = 1/2$

What is probability that a patient with stiff neck has meningitis?

$$P(M|S) = \frac{P(S|M)P(M)}{P(S)} = \frac{0.5x1/50000}{1/20} = 1/5000$$

Very low probability of meningitis (because prior prob. of stiff neck » prior prob. of meningitis)

10.3.2 Combining evidence

We can use Bayes' rule to find probability of a state given several pieces of evidence
$P(Cavity|Toothache \wedge Catch) = P(Toothache \wedge Catch|Cavity)P(Cavity)/P(Toothache \wedge Catch)$

For this to work we need conditional probability for all combinations of evidence variables. In general case there is an exponential number of conditional probabilities. For n Boolean evidence variables, we need 2^n conditional probabilities. This led AI researchers away from probability theory and towards more *ad hoc* systems

10.3.3 Conditional independence

Using Bayes' rule is simplified in situations of conditional independence between variables.

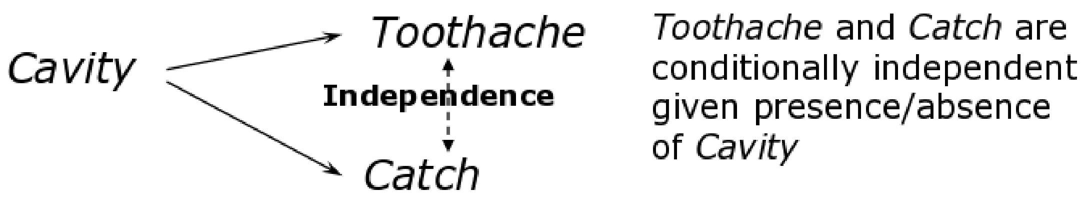

Cavity
Toothache
Catch
Independence
Toothache and Catch are conditionally independent given presence/absence of Cavity

Figure 31: Conditional independence

Expressed as
$P(Toothache \wedge Catch|Cavity) = P(Toothache|Cavity)P(Catch|Cavity)$

Simplifies evidence combination with Bayes' rule
$P(Cavity|Toothache \wedge Catch) =$
$P(Toothache \wedge Catch|Cavity)P(Cavity)/P(Toothache \wedge Catch) =$
$P(Toothache|Cavity)P(Catch|Cavity)P(Cavity)/P(Toothache \wedge Catch)$

Combining evidence does not need information for all combinations of evidence variables, only separate conditionals

10.4 Bayesian networks

Bayesian networks represent dependencies between variables and give concise specification of joint probability distribution. A Bayesian network is a graph where:

- A set of random variables are the nodes of the graph
- A set of directed links connects pairs of nodes
- A link from X to Y means that X has direct influence on Y, or is the parent of Y
- Each node has a conditional probability table (CPT) that quantifies effects parent nodes have on the node
- The graph has no directed cycles (it is a directed, acyclic graph, or DAG)

It can be shown that a Bayesian network is a representation of the *joint probability distribution*.

10.4.1 Incremental construction of Bayesian network

General procedure

- Choose set of relevant domain variables X_i
- Choose an ordering of the variables, preferably using causal domain knowledge ("root causes first", etc.)
- While there are variables left
  - Pick the next variable X i and add node for i
  - Set $Parents(X_i)$ to minimal set of nodes already in net such that X_i depends directly only on these nodes
  - Define conditional probability table for X_i
- Guarantees that
  - Network is acyclic
  - Axioms of probability are satisfied

10.4.2 Example - Bayesian network

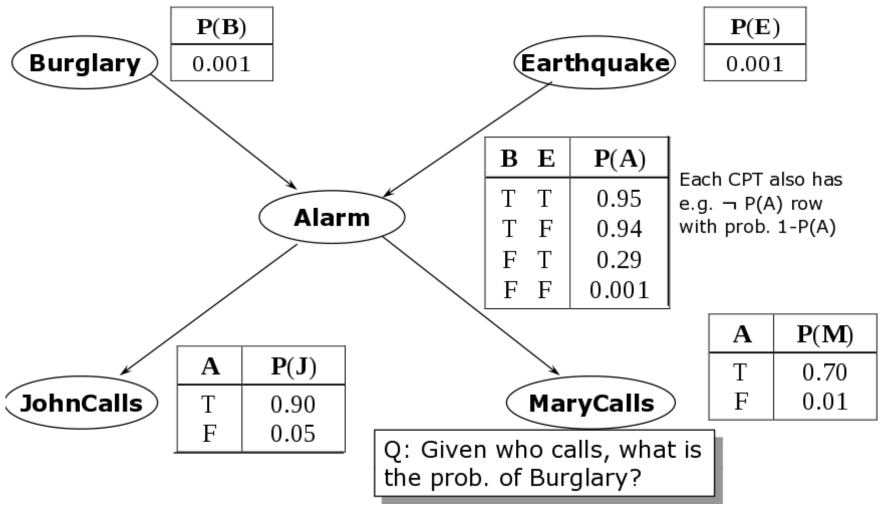


Figure 32: Example

10.5 Bayesian inference

Probabilistic inference procedure computes posterior probability distribution for a set of *query variables*, given exact values for some *evidence variables*
$P(Query|Evidence)$

An agent gets values for evidence variables from its percepts and queries for other variables in order to decide on action, using two functions.*Exact* and *approximate* (Monte Carlo) methods have been developed for Bayesian inference.

10.5.1 Example of Bayesian inference

What is $P(Burglary|JohnCalls)$? ("Diagnosis"). Incorrect reasoning:

- Since $P(JohnCalls|Alarm) = 0.9$, $P(Burglary|JohnCalls)$ "should be" 0.8 - 0.9
- Incorrect due to false alarms: $P(JohnCalls|\neg Alarm) = 0.05$

Correct reasoning: Using Bayes' rule $P(Burglary|JohnCalls) = 0.016$

10.5.2 Types of Bayesian inference

Diagnostic inference

- From effects to causes
- E.g. given that $JohnCalls$, infer that $P(Burglary|JohnCalls) = 0.016$

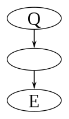


Figure 33: Diagnostic inference

Causal inference

- From causes to effects
- E.g. from Burglary, infer that $P(JohnCalls|Burglary) = 0.86$ and $P(MaryCalls|Burglary) = 0.67$

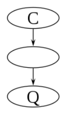


Figure 34: Causal inference

10.5.3 Other uses of Bayesian networks

Make decisions based on probabilities in the network and agent utilities. Decide which additional evidence variables should be observed in order to gain useful information. Perform sensitivity analysis to understand which aspects of model have greatest impact on probabilities of query variables. Explain results of probabilistic inference to users.

10.6 Other approaches to uncertainty

10.6.1 Default reasoning

Human judgmental reasoning is "qualitative" and does not rely on numerical probabilities. Default reasoning:

- Assume default state until new evidence presents itself
- If new evidence, conclusion may be retracted

This kind of reasoning is called *nonmonotonic*, because set of beliefs may both grow and shrink. Problems:

- Unclear semantic status of default rules
- What if two matching default rules have contradictory conclusions
- Managing retraction of beliefs (truth maintenance systems)
- How to use default rules for making decisions

No default reasoning system has solved all issues, and most systems are formally undecidable, and very slow in practice

10.6.2 Rule-based methods

Logical and rule-based reasoning systems have useful properties that probabilistic systems lack.

- *Locality*: Can use $A \Rightarrow B$ independent of other rules
- *Detachment*: Can use belief independent of its justification
- *Truth-functionality*: Truth of complex sentence follows from truth of components

Attempts have been made to modify rule systems by attaching degrees of belief to propositions/rules. Best known is the certainty factor model (Mycin, ca. 1980). The problem is that above properties are not appropriate for uncertain reasoning, and rule-based approaches to uncertainty have fallen out of use.

10.6.3 Dempster-Shafer theory

Dempster-Shafer theory deals with the distinction between uncertainty and ignorance. Instead of computing probability $P(X)$, it computes probability $Bel(X)$ that evidence supports proposition. Example:

- For unknown possibly non-fair coin, $Bel(Heads) = Bel(\neg Heads) = 0$
- If 90% certain that coin is fair ($P(Heads) = 0.5$),
  $Bel(Heads) = 0.5x0.9 = 0.45$, $Bel(\neg Heads) = 0.45$

One interpretation of Dempster-Shafer is that it calculates a probability interval.

- Heads interval for unknown possibly non-fair coin: $[0, 1]$
- Interval for 90% certain that coin is fair: $[0.45, 0.55]$

Width of the interval helps decide when more evidence is required

10.6.4 Fuzzy sets and fuzzy logic

Fuzzy set theory is a means of specifying how well an object satisfies a vague description. E.g. Is "Nate is tall" true if Nate is 5' 10"? In fuzzy set theory, $TallPerson$ is a fuzzy predicate and the truth value of $TallPerson(X)$ is in $[0, 1]$. The fuzzy predicate defines a fuzzy set that does not have sharp boundaries, i.e. not uncertainty about the world, but uncertainty about the meaning of "tall", and many claim that fuzzy set theory is not for uncertain reasoning at all. Fuzzy logic can be used to determine truth value of complex sentence from truth of its components. However, fuzzy logic is inconsistent with first-order logic. Despite this, fuzzy logic has been commercially successful, e.g. in automatic transmissions, trains, and video cameras.

11 Making Simple Decisions

11.1 Uncertainty and utility

11.1.1 Agents and decision theory

Agents need to make decisions in situations of uncertainty and conflicting goals. Basic principle of decision theory: Maximization of expected utility. *Decision-theoretic agents* are based decision theory, and need knowledge of probability and utility. Here, we are concerned with "simple" (one-shot) decisions, can be extended to sequential decisions.

11.2 Maximum expected utility (MEU)

11.2.1 Principle

Let

- $U(s)$ - Utility of state s
- $RESULT(a)$ - Random variable whose values are possible outcome states of action a in current state

- $P(RESULT(a) = s'|a, e)$ - Probability of outcome s', as a result of doing action a in current state, and given agent's available evidence e of the world

Then the *expected utility* EU of a, given e is

$$EU(a|e) = \sum_{s'} P(RESULT(a) = s'|a, e)U(s')$$

MEU: Agent should select a that maximizes EU

11.2.2 Problems with applying MEU

Often difficult to formulate problem completely, and required computation can be prohibitive. Knowing state of the world requires perception, learning, representation and inference. Computing $P(RESULT(a)|a, e)$ requires complete causal model and NP-complete belief net updating. Computing utility $U(s')$ may require search or planning since agent needs to know how to get to a state before its utility can be assessed.

11.2.3 Preference and utility

MEU appears to be a rational basis for decision making, but is not the only possible

- Why maximize average utility, instead of e.g. minimize losses?
- Can preferences between states really be compared by comparing two numbers?
- Etc.

We can state constraints on preference structures for a rational agent, and show that MEU is compatible with the constraints

11.3 Preference structures

11.3.1 Example - Lotteries and preferences

Lottery

- Scenario with different outcomes with different probabilities
- The agent have preferences regarding the outcomes

Example $L = [p, A; 1 - p, B]$

- Lottery L with two outcomes, A with probability p , B with probability 1-p

Preferences

- $A > B$ - A is preferred over B
- $A \approx B$ - Agent is indifferent between A and B
- $A \geq B$ - Prefers A over B or is indifferent

Constraints on preferences include orderability, transitivity, etc.

11.3.2 Utility follows from preferences

The constraints on preferences are the axioms of utility, from which utility principles follow. Utility principle:

If the agent's preferences obey axioms of utility, there exists a real-valued utility function U such that:
$U(A) > U(B) \Leftrightarrow A > B$
$U(A) = U(B) \Leftrightarrow A \approx B$

MEU principle: Utility of a lottery can be derived from outcome utilities
$U([p_1, S_1; \ldots; p_n, S_n]) = \sum_i p_i U(S_i)$

11.3.3 Utility of money

Utility theory comes from economics, and money is a common basis for utility functions.

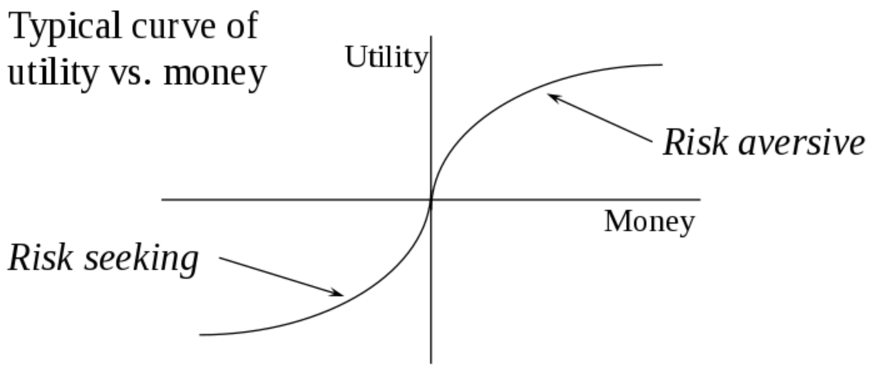


Figure 35: Utility of money

11.3.4 Human decision making

Decision theory is *normative*, but not *descriptive*: People violate axioms of utility in practice. Example:
A: 80% chance of \$4000
B: 100% chance of \$3000
C: 20% chance of \$4000
D: 25% chance of \$3000
Most people choose B over A, and C over D. Since only the scale is different, there does not seem to be a utility function that is consistent with the choices.

Possible descriptive theory. People are risk-aversive with high-probability events (A-B). People take more risks with unlikely payoffs (C-D).

11.4 Decision networks

Decision networks (also called *influence diagrams*) are a general mechanism for making rational decisions. Decision networks combine belief networks with nodes for actions and utilities, and can represent:

- Information about agent's current state
- Agent's possible actions

- States that will follow from actions
- Utilities of these states

Therefore, decision networks provide a substrate for implementing rational, utility-based agents

11.4.1 Example - Decision network for airport location

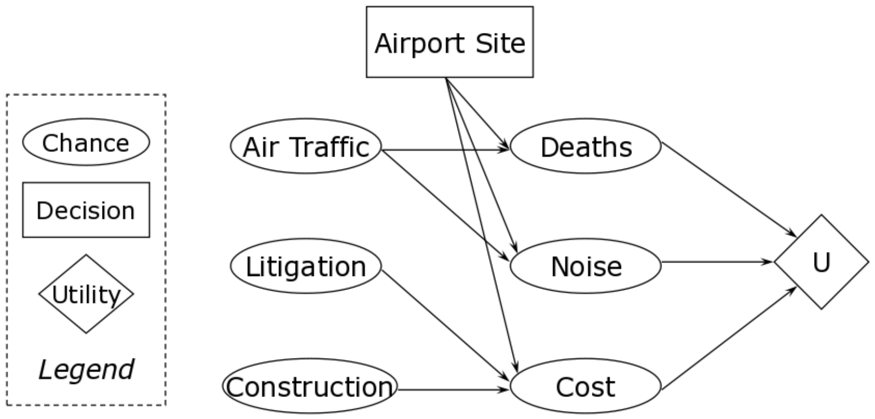


Figure 36: Decision network for airport location

11.4.2 Node types in decision networks

- *Chance* nodes (ovals)
  Represent random variables (as in belief networks), with associated conditional probability table (CPT) indexed by states of parent nodes (decisions or other chance nodes)
- *Decision* nodes (rectangles)
  Represent points where the decision maker has choice of actions to make
- *Utility* nodes (diamonds)
  Represent the agent's utility function, with parents all nodes that directly influence utility

11.4.3 Evaluating decision networks

Set the evidence variables (chance nodes with known values) for the current state. For each possible value of the decision node:

- Set decision node to that value (from now on, it behaves like a chance node that has been set as an evidence variable)
- Calculate posterior probabilities for parent nodes of the utility node, using standard probabilistic inference methods
- Calculate resulting utility for the action

Return the action with the highest utility

11.5 Value of information

The agent will normally not have all required information available before making a decision. Important to know which information to seek, by performing tests that may be expensive and/or hazardous. The importance of tests depend on:

- Will different outcomes make significant difference to the optimal action
- What is the probability of different outcomes

Information value theory helps agents decide which information to seek, by using sensing actions

11.5.1 Motivating example

Oil company want to buy one of n indistinguishable blocks, exactly one block contains oil worth C, price for each block is C/n. A seismologist offers to investigate block 3, determining if it has oil or not. How much is this information worth?

- With probability $1/n$, block 3 has oil. Then the company will buy block 3 for C/n, and make profit $C-C/n = (n-1)C/n$
- With probability $(n-1)/n$, block 3 is empty. The company will buy another block. Probability of oil there is $1/(n-1)$, with profit $C/(n-1) - C/n = C/n(n-1)$

Expected profit given the survey information:

$$\frac{1}{n}x\frac{(n-1)C}{n} + \frac{n-1}{n}x\frac{C}{n(n-1)} = \frac{C}{n}$$

The information is as much worth as the block itself!

11.5.2 Considerations for information gathering

Information has value if it is likely to cause a change of plan, and if the new plan will be significantly better than the old. An information-gathering agent should:

- Ask questions in a reasonable sequence
- Avoid asking irrelevant questions
- Take into account importance of information vs. cost
- Stop asking questions when appropriate

Requirements met by using VPI(E) - Value of Perfect Information of evidence E. Properties:

- Always non-negative
- Depends on current state and is non-additive
- Order-independent (simplifies sensing actions)

11.5.3 An information gathering agent

```
function INFORMATION—GATHERING—AGENT(percept) returns an action
persistent: D , a decision network
integrate percept into D
j <= the value that maximizes VPI(Ej)/Cost(Ej)
if VPI(Ej) > Cost(Ej) then
  return REQUEST(Ej)
  else return the best action from D*
```

*non-information seeking action
Information-gathering agent is *myopic*, i.e. it just considers one evidence variable at a time. It may hastily select an action where a better decision would be based on two or more information gathering actions. "Greedy" search heuristic - often works well in practice. A perfectly rational agent would consider all possible sequences of sensing action that terminate in an external action. May disregard permutations due to order-independence.

11.6 Decision analysis vs. expert systems

Decision analysis (application of decision theory)

- Focus on making decisions
- Defines possible actions and outcomes with preferences
- Roles
  - Decision maker states preferences
  - Decision analyst specifies problem

Expert systems ("classical" rule-based systems)

- Focus on answering questions
- Defines heuristic associations between evidence & answers
- Roles
  - Domain expert provides heuristic knowledge
  - Knowledge engineer elicits & encodes knowledge in rules

11.7 Decision-theoretic expert systems

Decision-theoretic expert systems.

- Inclusion of decision networks in expert system frameworks

Advantages:

- Make expert preferences explicit
- Automate action selection in addition to inference
- Avoid confusing likelihood with importance
  Common pitfall in expert systems: Conclusions are ranked in terms of likelihood, disregarding rare, but dangerous conclusion
- Availability of utility information helps in knowledge engineering process

11.7.1 Knowledge engineering for decision-theoretic expert systems

- Create causal model
- Simplify to qualitative decision model
- Assign probabilities
- Assign utilities
- Verify and refine model
- Perform sensitivity analysis

12 Learning from Examples

12.1 General model

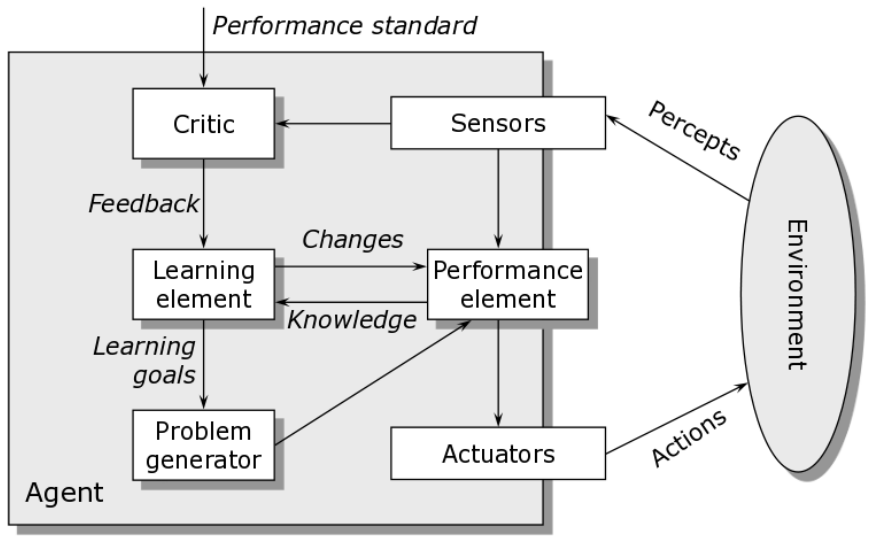


Figure 37: General model of learning agents

- Performance element:
  Carries out the task of the agent, i.e. processes percepts and decides on actions
- Learning element:
  Proposes improvements of the performance element, based on previous knowledge and feedback
- Critic:
  Evaluates performance element by comparing results of its actions with imposed performance standards
- Problem generator:
  Proposes exploratory actions to increase knowledge

12.1.1 Aspects of the learning element

Which *components* of the performance element are to be improved. Which parts of the agent's knowledge base is targeted? What *feedback* is available. Supervised, unsupervised or reinforcement learning differ in type of feedback agent receives. What *representation* is used for the components? E.g. logic sentences, belief networks, utility functions, etc. What *prior information (knowledge)* is available?

12.1.2 Performance element components

Possible components that can be improved:

- Direct mapping from states to actions
- Means to infer world properties from percept sequences
- Information about how the world evolves
- Information about the results of possible actions
- Utility information about the desirability of world states
- Desirability of specific actions in specific states
- Goals describing states that maximize utility

In each case, learning can be sees as learning an unknown function $y = f(x)$

12.1.3 Hypothesis space H

H: the set of hypothesis functions h to be considered in searching for $f(x)$.

Consistent hypothesis: Fits with all data
If several consistent hypotheses - choose simplest one! (Occam's razor)

Realizability of learning problem:

- Realizable if H contains the "true" function
- Unrealizable if not
- We do normally know what the true function is

Why not choose H as large as possible? May be very inefficient in learning and in applying.

12.2 Types of learning

12.2.1 Knowledge

Inductive learning:

- Given a collection of examples $(x, f(x))$
- Return a function h that approximates f
- Does not rely on prior knowledge ("just data")

Deductive (or analytical) learning:

- Going from known general f to a new f' that is logically entailed
- Based on prior knowledge ("data+knowledge")
- Resemble more human learning

12.2.2 Feedback

Unsupervised learning:
Agent learns patterns in data even though no feedback is given, e.g. via clustering

Reinforcement learning:
Agent gets reward or punishment at the end, but is not told which particular action led to the result

Supervised learning:
Agent receives learning examples and is explicitly told what the correct answer is for each case

Mixed modes, e.g. *semi-supervised* learning:
Correct answers for some but not all examples

12.3 Learning decision trees

A decision situation can be described by:

- A number of *attributes*, each with a set of possible values
- A decision which may be Boolean (yes/no) or multivalued

A *decision* tree is a tree structure where:

- Each internal node represents a test of the value of an attribute, with one branch for each possible attribute value
- Each leaf node represents the value of the decision if that node is reached

Decision tree learning is one of simplest and most successful forms of machine learning. An example of inductive and supervised learning

12.3.1 Example - Wait for restaurant table

Goal predicate: WillWait (for restaurant table)

Domain attributes:

- Alternate (other restaurants nearby)
- Bar (to wait in)
- Fri/Sat (day of week)
- Hungry (yes/no)
- Patrons (none, some, full)
- Price (range)

- Raining (outside)
- Reservation (made before)
- Type (French, Italian, ..)
- WaitEstimate(minutes)

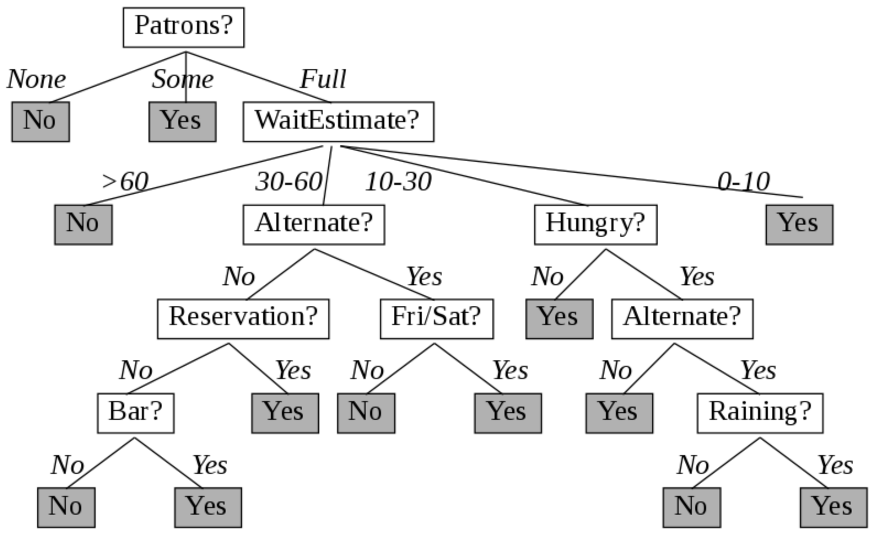


Figure 38: One decision tree for the example

12.3.2 Expressiveness of decision trees

The tree is equivalent to a conjunction of implications
$\forall r\ Patrons(r, FULL) \wedge WaitEstimate(r, 10-30) \wedge Hungry(r, No) \Rightarrow WillWait(r)$

Cannot represent tests on two or more objects, restricted to testing attributes of one object. Fully expressive as propositional language, e.g. any Boolean function can be written as a decision tree. For some functions, exponentially large decision trees are required. E.g. decision trees are good for some functions and bad for others.

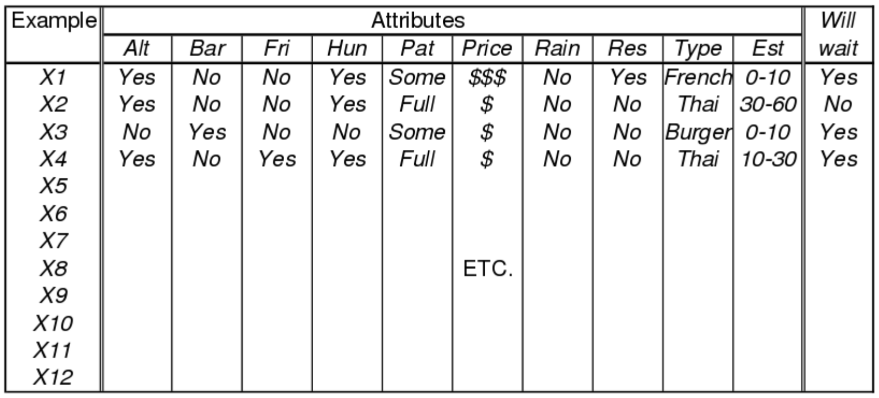
12.3.3 Inducing decision trees from examples

Terminology:

- Example - Specific values for all attributes, plus goal predicate
- Classification - Value of goal predicate of the example
- Positive/negative example - Goal predicate is true/false
- Training set - Complete set of examples

The task of inducing a decision tree from a training set is to find the *simplest tree that agrees with the examples*. The resulting tree should be more *compact* and *general* than the training set itself.

12.3.3.1 A training set for the restaurant example



Example	Attributes										Will
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*wait*
X1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0-10*	*Yes*
X2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30-60*	*No*
X3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0-10*	*Yes*
X4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*10-30*	*Yes*
X5											
X6											
X7											
X8						ETC.					
X9											
X10											
X11											
X12											

Figure 39: Training set

12.3.4 General idea of induction algorithm

Test the most important attribute first, i.e. the one that makes the most difference to the classification. *Patrons?* is a good choice for the first attribute, because it allows early decisions. Apply same principle recursively.

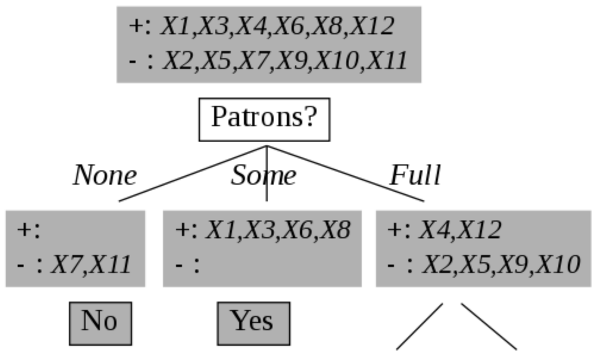


Figure 40: General idea of induction algorithm

12.3.5 Recursive step of induction algorithm

The attribute test splits the tree into smaller decision trees, with fewer examples and one attribute less. Four cases to consider for the smaller trees:

- If some positive and some negative examples, choose best attribute to split them
- If examples are all positive (negative), answer Yes (No)
- If no examples left, return a default value (no example observed for this case)
- If no attributes left, but both positive and negative examples: Problem! (same description, different classifications - *noise*)

12.3.6 Induced tree for the example set

The induced tree is simpler than the original "manual" tree. It captures some regularities that the original creator was unaware of.

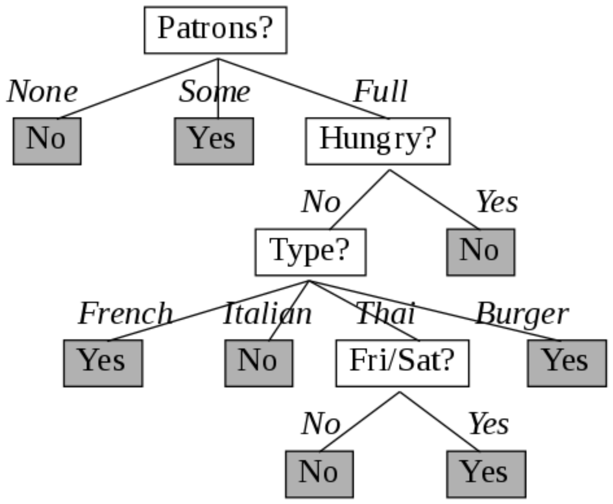


Figure 41: Induced tree

12.3.7 Broaden applicability of decision trees

Missing data: How to handle training samples with partially missing attribute values

Multi/many-valued attributes: How to treat attributes with many possible values

Continuous or integer-valued input attributes: How to branch the decision tree when attribute has a continuous value range

Continuous-valued output attributes: Requires regression tree rather than a decision tree, i.e. output value is a linear function of input variables rather than a point value

12.3.8 Assessing learning performance

Collect large set of examples. Divide into two disjoint sets, training set and test set. Use learning algorithm on training set to generate hypothesis h. Measure percentage of examples in test set that are correctly classified by h. Repeat steps above for differently sized training sets.

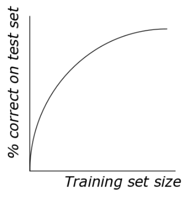


Figure 42: Performance

12.4 Neural networks

The human brain is a huge network of neurons. A neuron is a basic processing unit that collects, processes and disseminates electrical signals. Early AI tried to imitate the brain by building *artificial neural networks (ANN)*. Met with theoretical limits and "disappeared". In the 1980-90's, interest in ANNs resurfaced. Because of new theoretical development and massive industrial interest and applications.

12.4.1 The basic unit of neural networks

The network consists of units (nodes, "neurons") connected by links:

- Carries an activation a_i from unit i to unit j
- The link from unit i to unit j has a weight $W_{i,j}$
- Bias weight $W_{0,j}$ to fixed input $a_0 = 1$

Activation of a unit j

- Calculate input: $in_j = \sum W_{i,j} a_i (i = 0 \dots n)$
- Derive output: $a_j = g(in_j)$ where g is the activation function

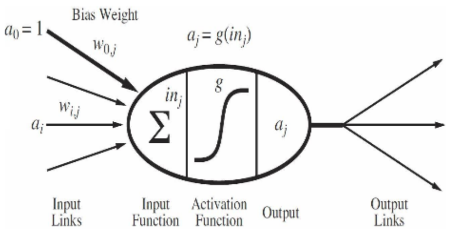


Figure 43: Neuron

12.4.2 Activation functions

Activation function should separate well:

- "Active" (near 1) for desired input
- "Inactive" (near 0) otherwise

It should be non-linear. Most used functions:
Threshold function and Sigmoid function.

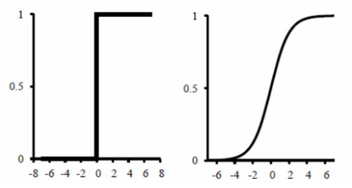


Figure 44: Activation

12.4.3 Neural networks as logical gates

With proper use of *bias weight* W_0 to set thresholds, neural networks can compute standard logical gate functions (here $a_0 = -1$)

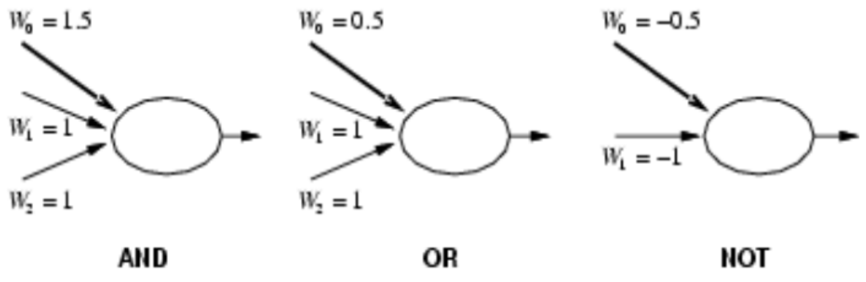


Figure 45: Logical gates

12.4.4 Neural network structures

Two main structures:

- *Feed-forward* (acyclic) networks
  - Represents a function of its inputs
  - No internal state
- *Recurrent* network:
  - Feeds outputs back to inputs
  - May be stable, oscillate or become chaotic
  - Output depends on initial state

Recurrent networks are the most interesting and "brain-like", but also most difficult to understand.

12.4.5 Feed-forward networks as functions

- A FF network calculates a function of its inputs
- The network may contain hidden units/layers

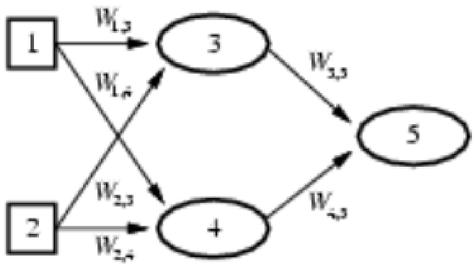


Figure 46: Feed-forward network

- By changing #layers/units and their weights, different functions can be realized
- FF networks are often used for classification

12.5 Perceptrons

Single-layer feed-forward neural networks are called *perceptrons*, and were the earliest networks to be studied. Perceptrons can only act as linear separators, a small subset of all interesting functions. This partly explains why neural network research was discontinued for a long time.

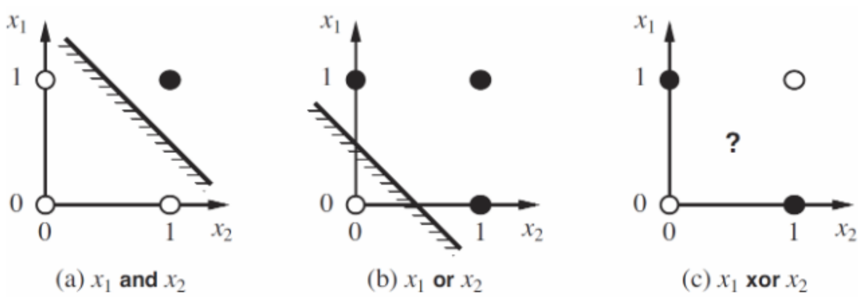


Figure 47: Linear seperators

12.5.1 Perceptron learning algorithm

How to train the network to do a certain function (e.g. classification) based on a training set of input/output pairs?

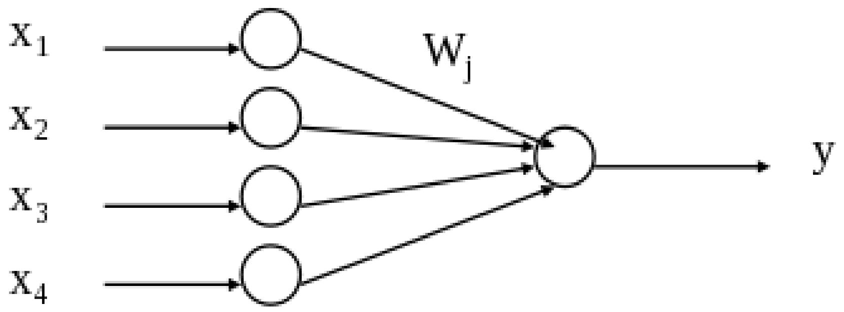


Figure 48: Perceptron learning

Basic idea:

- Adjust network link weights to minimize some measure of the error on the training set
- Adjust weights in direction that minimizes error

```
function PERCEPTRON-LEARNING(examples, network) returns a perceptron hypothesis
inputs : examples, a set of examples, each with inputs x1, x2 .. and output y
         network, a perceptron with weights Wj and act. function g
repeat
  for each e in examples do
    in = sum(Wj xj[e]]          j=0 .. n
    Err = y[e] - g(in)
    Wj = Wj + alfa Err xj[e]    alfa - the learning rate
until some stopping criterion is satisfied
return NEURAL-NETWORK-HYPOTHESIS(network)
```

12.5.2 Performance of perceptrons vs. decision trees

Perceptrons better at learning separable problem. Decision trees better at "restaurant problem".

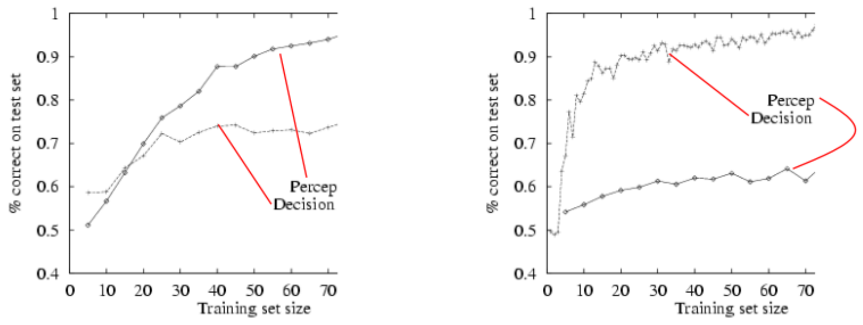


Figure 49: Performance of perceptrons vs. decision trees

12.6 Multi-layer feed-forward networks

Adds hidden layers:

- The most common is one extra layer
- The advantage is that more function can be realized, in effect by combining several perceptron functions

It can be shown that:

- A feed-forward network with a single sufficiently large hidden layer can represent any continuous function
- With two layers, even discontinuous functions can be represented

However:

- Cannot easily tell which functions a particular network is able to represent
- Not well understood how to choose structure/number of layers for a particular problem

12.6.1 Example network structure

Feed-forward network with 10 inputs, one output and one hidden layer - suitable for "restaurant problem".

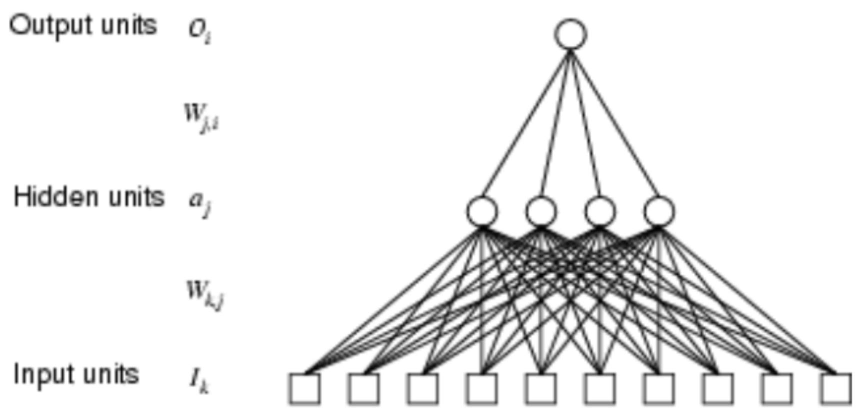


Figure 50: Example network structure

12.6.2 More complex activation functions

Multi-layer networks can combine simple (linear separation) perceptron activation functions into more complex functions.

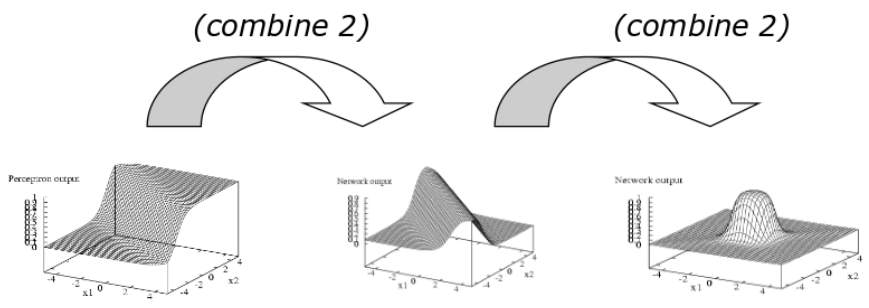


Figure 51: More complex activation functions

12.6.3 Learning in multi-layer networks

In principle as for perceptrons - adjusting weights to minimize error. The main difference is what "error" at internal nodes mean - nothing to compare to. Solution: *Propagate* error at output nodes back to hidden layers. Successively propagate backwards if the network has several hidden layers. The resulting *Back-propagation algorithm* is the standard learning method for neural networks.

12.6.4 Learning neural network structure

Need to learn network structure

- Learning algorithms have assumed fixed network structure
- However, we do not know in advance what structure will be necessary and sufficient

Solution approach:

- Try different configurations, keep the best
- Search space is very large (# layers and # nodes)
- "Optimal brain damage": Start with full network, remove nodes selectively (optimally)
- "Tiling": Start with minimal network that covers subset of training set, expand incrementally

13 Reinforcement Learning (RL)

Reinforcement learning (RL) is unsupervised learning: The agent receives no examples, and starts with no model or utility information. The agent must use trial-and-error, and receives rewards, or reinforcement, to guide learning.

Examples:

- Learning a game by making moves until lose or win: Reward only at the end
- Learning to ride a bicycle without any assistance: Rewards received more frequently

RL can be seen to encompass all of AI: An agent must learn to behave in an unknown environment

13.1 Variations of RL

Accessible environment (agent can use percepts) vs. *inaccessible* (must have some model). Agent may have some initial knowledge, or not have any domain model. Rewards can be received only in terminal states, or in any state. Rewards can be part of the actual utility, or just hint at the actual utility. The agent can be a passive (watching) or an active (exploring) learner. RL uses results from sequential decision processes.

13.2 Sequential decision processes

In a sequential decision process, the agent's utility depends on a sequence of decisions. Such problems involve utility (of states) and uncertainty (of action outcomes). The agent needs a policy that tells it what to do in any state it might reach: $a = \pi(s)$. An optimal policy $\pi * (s)$ is a policy that gives the highest expected utility.

13.2.1 Example

- 4x3 environment
- Agent starts in $(1, 1)$
- Probabilistic transition model
  Sequence $[Up, Up, Right, Right, Right]$ only has a probability of $0.8^5 = 0.33$ of reaching $+1(4, 3)$
- Terminal rewards are $+1(4, 3)$ and $-1(4, 2)$
- Other state rewards are -0.04

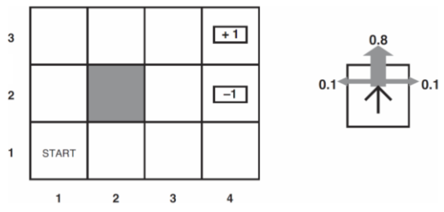


Figure 52: Example sequential decision process

13.3 Markov Decision Processes (MDP)

An MDP is a sequential decision process with the following characteristics:

- Fully observable environment (agent knows where it is at any time)
- State transitions are Markovian, i.e. $P(s'|s, a)$ - probability of reaching s' from s by action a depends only on s and a, not earlier state history
- Agent receives a reward $R(s)$ in each state s
- The total utility $U(s)$ of s is the sum of the rewards received, from s until a terminal state is reached

13.4 Optimal policy and utility of states

The utility of a state s depends on rewards received but also on the policy π followed:

$$U^{\pi}(s) = E[\sum_{t=0}^{\infty} R(S_t)]$$

Of all the possible policies the agent could follow, one gives the highest expected utility:

$$\pi_s^* = argmax_{\pi} U^{\pi}(s)$$

This is the optimal policy. Under certain assumptions, it is independent of starting state. For an MDP with known transition model, reward function and assuming the optimal policy, we can calculate the utility $U(s)$ of each state s. For the 4x3 example, the utilities are:

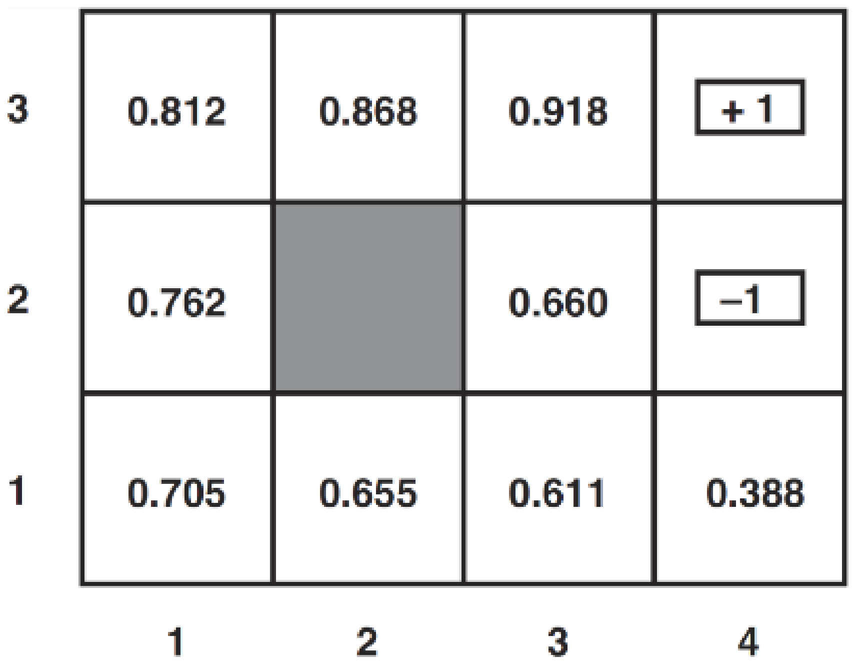


Figure 53: 4x3 example

13.4.1 Optimal policy

Knowing $U(s)$ allows the agent to select the optimal action:

$$\pi^*(s) = argmax_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

Optimal policy depends on non-final reward $R(s)$

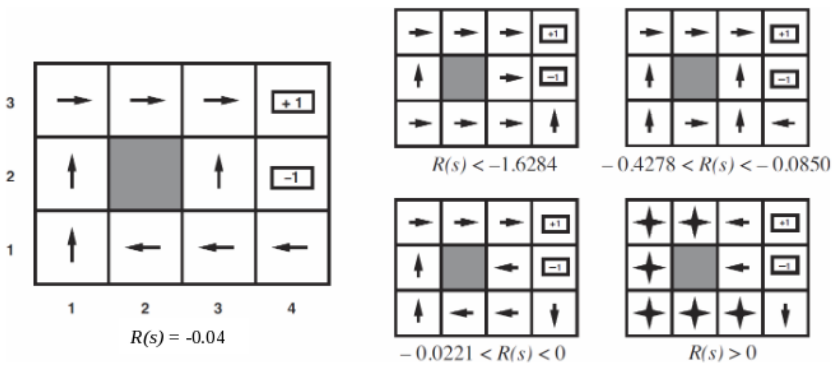


Figure 54: Optimal policy

13.5 Bellman equations

We need to be able to calculate utilities $U(s)$ in order to define optimal policy. Can exploit dependence between states: The utility of a state s is the reward $R(s)$ plus the maximum expected utility of the next state

$$U(S) = R(s) + max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')$$

This is the Bellman equation. There are n equations for n states, containing utilities $U(s)$ as n unknowns. Solving the equations yields the utilities $U(s)$.

13.5.1 Value iteration

The Bellman equations are nonlinear (due to the max opr.) and cannot be solved by linear algebra. Can use an iterative approach instead

- Start with arbitrary initial values
- Calculate right hand side
- Plug the value into left-hand sides U(s)
- Iterate until the values stabilize (within a margin)

This VALUE-ITERATION algorithm is guaranteed to converge and to produce unique solutions.

13.5.2 Policy iteration

Instead of iterating to find $U(s)$ and derive an optimal policy, we can iterate directly in policies. We can iterate the policy to get an optimal one:

- Policy evaluation: Given a policy π_i, calculate U_i, the utility of each state if π_i is followed
- Policy improvement: Calculate new policy π_{i+1} that selects the action that maximizes successor state value (MEU).
- Repeat until values no longer change

This POLICY-ITERATION algorithm is guaranteed to converge and to produce an optimal policy

13.6 Reinforcement learning of MDP

We could find an optimal policy for an MDP if we know the transition model $P(s'|s,a)$. However, an agent in an unknown environment does not know the transition model nor in advance what rewards it will get in new states. We want the agent to learn to behave rationally in an unsupervised process. The purpose of RL is to learn the optimal policy based only on received rewards.

13.6.1 Different RL agent designs

- *Utility-based* agents learn a utility function on states and uses it to select actions that maximize expected utility. Requires also a model of where actions lead
- *Q-learning* agents learn an action-utility function (Q-function), giving expected utility of taking a given action in a given state. Can select actions without knowing where they lead, at the expense not being able to look ahead.
- Reflex agents learn a policy that maps directly from states to actions, i.e. π^*

13.7 Passive learning

13.7.1 Direct utility estimation

In passive learning, the agent's policy π is fixed, it only needs to how good it is. Agent runs a number of trials, starting in $(1, 1)$ and continuing until it reaches a terminal state. The utility of a state is the expected total remaining reward (reward-to-go). Each trial provides a sample of the reward-to-go for each visited state. The agent keeps a running average for each state, which will converge to the true value. This is a direct utility estimation method.

13.7.1.1 Example

Training trials for (4,3) matrix

$(1, 1) - 0.04 \rightarrow (1, 2) - 0.04 \rightarrow (1, 3) - 0.04 \rightarrow (1, 2) - 0.04 \rightarrow (1, 3) - 0.04 \rightarrow (2, 3) - 0.04 \rightarrow (3, 3) - 0.04 \rightarrow (4, 3) + 1$
$(1, 1) - 0.04 \rightarrow (2, 1) - 0.04 \rightarrow (3, 1) - 0.04 \rightarrow (3, 2) - 0.04 \rightarrow (4, 2) + 1$

Sample $U(s)$ in first trial
$(1, 1)0.72$
$(1, 2)0.76$ and 0.84
$(1, 3)0.80$ and 0.88
Etc.

Direct utility estimation converges slowly

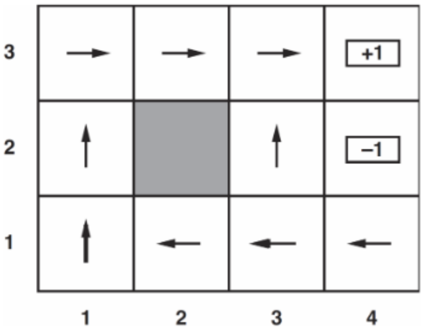


Figure 55: Example

13.7.2 Exploiting state dependencies

Direct utility fails to exploit the fact that states are dependent as shown by Bellman equations

$$U(S) = R(s) + max_{a \in A(s)} \sum_{s'} P(s'|s, a)U(s')$$

Learning can be speeded up by using these dependencies. Direct utility estimation can be seen to search a too large hypothesis space that contains many hypotheses violating Bellman equations

13.7.3 Adaptive Dynamic Programming (ADP)

An ADP agent uses dependencies between states to speed up value estimation. It follows a policy π and can use observed transitions to incrementally build the transition model $P(s'|s, \pi(s))$. It can then plug the learned transition model and observed rewards R(s) into the Bellman equations to get U(s). The equations are linear because there is no max operator, and therefore easier to solve. The result is U(s) for the given policy π.

13.7.4 Temporal Difference (TD) learning

TD is another passive utility value learning algorithm using Bellman equations. Instead of solving the equations, TD uses the observed transitions to adjust the utilities of the observed states to agree with Bellman. TD uses a l earning rate parameter α to select the rate of change of utility adjustment. TD does not need a transition model to perform its updates, only the observed transitions.

13.8 Active learning

While a passive RL agent executes a fixed policy π, an active RL agent has to decide which actions to take. An active RL agent is an extension of a passive one, e.g. the passive ADP agent, and adds:

- Needs to learn a complete transition model for all actions (not just π), using passive ADP learning
- Utilities need to reflect the optimal policy π^*, as expressed by the Bellman equations
- Equations can be solved by the VALUE-ITERATION or POLICY-ITERATION methods described before
- Action to be selected as the optimal/maximizing one

13.8.1 Exploration behavior

The active RL agent may select maximizing actions based on a faulty learned model, and fail to incorporate observations that might lead to a more correct model. To avoid this, the agent design could include selecting actions that lead to more correct models at the cost of reduced immediate rewards. This called exploitation vs. exploration tradeoff. The issue of optimal exploration policy is studied in a subfield of statistical decision theory dealing with so-called bandit problems.

13.8.2 Q-learning

An action-utility function Q assigns an expected utility to taking a given action in a given state: $Q(a, s)$ is the value of doing action a in state s. Q-values are related to utility values:
$U(s) = max_a Q(a, s)$

Q-values are sufficient for decision making without needing a transition model $P(s'|s, a)$. Can be learned directly from rewards using a TD-method based on an update equation $(s- > s')$
$Q(s, a) \leftarrow Q(a, s) + \alpha(R(s) + max_{a'} Q(s', a') - Q(s, a))$

13.9 RL applications

13.9.1 Generalization

In simple domains, U and Q can be represented by tables, indexed by state s. However, for large state spaces the tables will be too large to be feasible, e.g. chess 10^{40} states. Instead functional approximation can sometimes be used, e.g. $\breve{U}(s) = \sum parameter_i x feature_i(s)$. Instead of e.g . 10^{40} table entries, U can be estimated by e.g. 20 parameterized features. Parameters can be found by supervised learning. Problem: Such a function may not exist, and learning process may therefore fail to converge.

13.9.2 Policy search

A policy π maps states to actions: $a = \pi(s)$, and policy search tries to derive π directly. Normally interested in parameterized policy in order to get a compact representation. E.g., π can be represented by a collection of functional approximations $\hat{Q}(s, a)$, one per action a, and policy will be to maximize over a
$\pi(s) = max_a \hat{Q}(s, a)$

Policy search has been investigated in continuous/discrete and deterministic/stochastic domains.

13.9.3 Some examples of reinforcement learning

Game playing

- Famous program by Arthur Samuel (1959) to play checkers used linear evaluation function for board positions, updated by reinforcement learning
- Backgammon system (Tesauro, 1992) used TD-learning and self-play (200.000 games) to reach level comparable to top three human world masters

Robot control

- Inverted pendulum problem used as test case for several successful reinforcement learning programs, e.g. BOXES (Michie, 1968) learned to balance pole after 30 trials

14 Natural Language Communication

One definition of communication

- Communication is the intentional exchange of information brought about by the production and perception of signs drawn from a shared system of a limited number of conventional signs

Humans use language to communicate

- Language is a "shared system of a limited number of conventional signs"
- Its structure is sufficiently rich to allow an unbounded number of qualitatively different messages

14.1 Communication and action

To produce messages in a language is one of the actions available to an agent. This action is called a speech act (can be spoken, written, etc.). In a speech act, an utterance consisting of words is delivered from a speaker to a hearer. Different types of speech acts serve different purposes.

14.1.1 Some types of speech acts

Inform	Provide information to hearer
Query	Ask for information
Answer	Inform in response to query
Request	Ask hearer to perform action
Deny	Refuse to perform action
Command	Request with no option to deny
Promise	Commit to future action
Offer	Propose to do future action
Acknowledge	Confirm e.g. request or offer
...	

14.1.2 Planning and understanding speech acts

Deciding when a speech act is called for, and decide which one to use, is equivalent to planning. Understanding a speech act is similar to diagnosis or plan recognition. I.e., one can use methods from other parts of AI in implementing perception and action in communicating agents.

14.1.3 Natural and formal languages

Natural languages are a rich field of empirical and logical study, including in AI. *Formal languages* are invented ones, in contrast to natural languages, and include logic, etc. Formal language concepts are being used in analysis of natural languages.

14.2 Language structures

14.2.1 Formal language concepts

A formal language is a set of strings (sentences): "The wumpus is dead". A string is a sequence of symbols taken from a finite set called the terminal symbols (words): "dead", "is", "wumpus", "the". A phrase is a substring of a sentence. There are different categories (symbolized by nonterminal symbols) of phrases. NP (noun phrase): "the wumpus" and VP (verb phrase): "is dead".

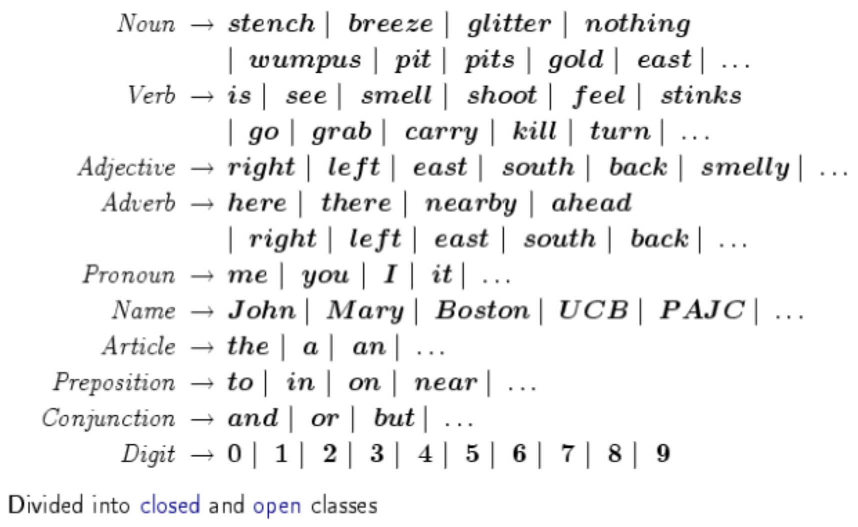
The structure (grammar) of a language can be defined using a phrase structure, i.e. combinations of terminal and nonterminal symbols $NP\ VP$. Rewrite rules define how a single nonterminal symbol (phrase) may be replaced by a structure $S \rightarrow NP\ VP$.

14.2.2 A grammar for a fragment of English

Lexicon:
- List of valid words
- Categories: Noun, verb, adjective, ...

Grammar:
- Rules for valid sentences
- Nonterminals: Sentence (S), noun phrase (NP) ...

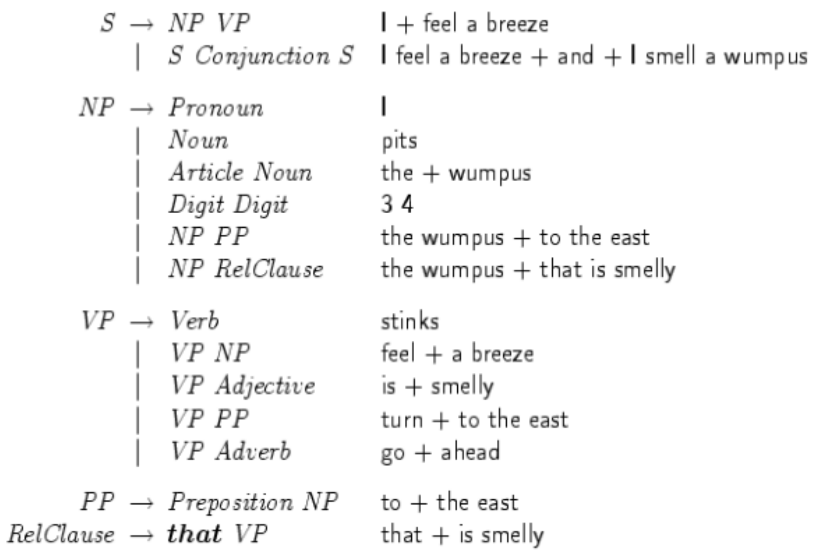
Parsing:
- Analyze a given sequence of lexicon words as a tree - structure allowed by grammar rules



$$
\begin{aligned}
\mathit{Noun} &\rightarrow \textbf{stench} \mid \textbf{breeze} \mid \textbf{glitter} \mid \textbf{nothing} \\
&\quad \mid \textbf{wumpus} \mid \textbf{pit} \mid \textbf{pits} \mid \textbf{gold} \mid \textbf{east} \mid \ldots \\
\mathit{Verb} &\rightarrow \textbf{is} \mid \textbf{see} \mid \textbf{smell} \mid \textbf{shoot} \mid \textbf{feel} \mid \textbf{stinks} \\
&\quad \mid \textbf{go} \mid \textbf{grab} \mid \textbf{carry} \mid \textbf{kill} \mid \textbf{turn} \mid \ldots \\
\mathit{Adjective} &\rightarrow \textbf{right} \mid \textbf{left} \mid \textbf{east} \mid \textbf{south} \mid \textbf{back} \mid \textbf{smelly} \mid \ldots \\
\mathit{Adverb} &\rightarrow \textbf{here} \mid \textbf{there} \mid \textbf{nearby} \mid \textbf{ahead} \\
&\quad \mid \textbf{right} \mid \textbf{left} \mid \textbf{east} \mid \textbf{south} \mid \textbf{back} \mid \ldots \\
\mathit{Pronoun} &\rightarrow \textbf{me} \mid \textbf{you} \mid \textbf{I} \mid \textbf{it} \mid \ldots \\
\mathit{Name} &\rightarrow \textbf{John} \mid \textbf{Mary} \mid \textbf{Boston} \mid \textbf{UCB} \mid \textbf{PAJC} \mid \ldots \\
\mathit{Article} &\rightarrow \textbf{the} \mid \textbf{a} \mid \textbf{an} \mid \ldots \\
\mathit{Preposition} &\rightarrow \textbf{to} \mid \textbf{in} \mid \textbf{on} \mid \textbf{near} \mid \ldots \\
\mathit{Conjunction} &\rightarrow \textbf{and} \mid \textbf{or} \mid \textbf{but} \mid \ldots \\
\mathit{Digit} &\rightarrow \textbf{0} \mid \textbf{1} \mid \textbf{2} \mid \textbf{3} \mid \textbf{4} \mid \textbf{5} \mid \textbf{6} \mid \textbf{7} \mid \textbf{8} \mid \textbf{9}
\end{aligned}
$$

Divided into closed and open classes

Figure 56: Lexicon of the fragment



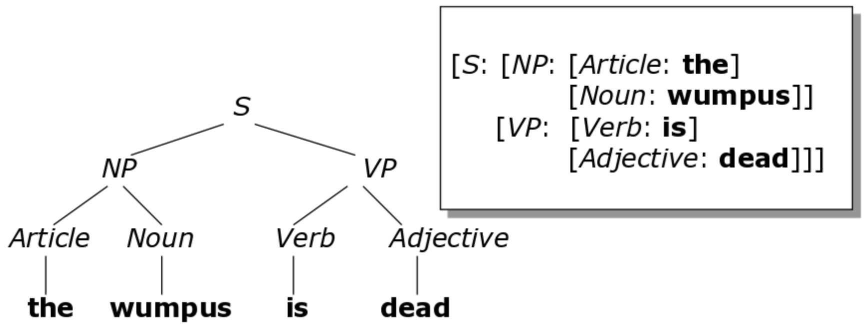
S	$\rightarrow$	$NP\ VP$	I + feel a breeze
	$\mid$	$S\ Conjunction\ S$	I feel a breeze + and + I smell a wumpus
NP	$\rightarrow$	$Pronoun$	I
	$\mid$	$Noun$	pits
	$\mid$	$Article\ Noun$	the + wumpus
	$\mid$	$Digit\ Digit$	3 4
	$\mid$	$NP\ PP$	the wumpus + to the east
	$\mid$	$NP\ RelClause$	the wumpus + that is smelly
VP	$\rightarrow$	$Verb$	stinks
	$\mid$	$VP\ NP$	feel + a breeze
	$\mid$	$VP\ Adjective$	is + smelly
	$\mid$	$VP\ PP$	turn + to the east
	$\mid$	$VP\ Adverb$	go + ahead
PP	$\rightarrow$	$Preposition\ NP$	to + the east
$RelClause$	$\rightarrow$	**that** VP	that + is smelly

Figure 57: Grammar of the fragment

14.3 Parsing and semantics

14.3.1 Parsing

Search for a parse tree for a given sentence, e.g.
$PARSE("thewumpusisdead", grammar, S)$



```
[S: [NP: [Article: the]
         [Noun: wumpus]]
    [VP: [Verb: is]
         [Adjective: dead]]]
```

Figure 58: Parsing

14.3.2 Top-down vs. bottom-up parsing

Top-down parsing:

- Initial parse tree is the root with unknown children $[S :?]$
- At each step, select leftmost node in the tree with unknown children and look for grammar rules with LHS that matches the node. Replace ? with RHS and repeat
- Stop when leaves of the tree exactly matches the string

Bottom-up parsing

- Initial list of words, seen as list of singleton parse trees
- At each step, replace each sequence of parse trees that matches an RHS of a grammar rule, with the corresponding LHS, and repeat
- Stop when the tree is the single node S

14.3.3 Semantic interpretation

Having analyzed the sentence, we need to interpret its meaning; i.e. decide its semantic content. We adopt first-order logic (FOL) as the representation language. E.g., "the wumpus is dead and John loves Mary" has the meaning:
$Dead(Wumpus) \land Loves(John, Mary)$

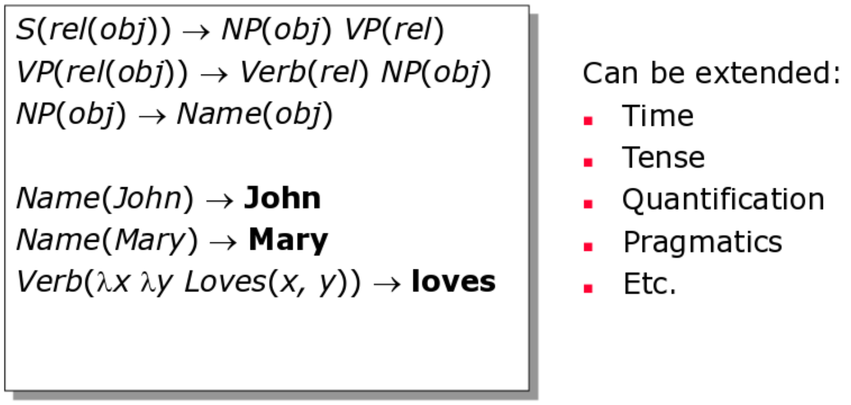
Compositional semantics: The meaning of the entire sentence is composed of the meanings of its constituents.

14.3.4 Augmenting grammar for semantics

Each category of the grammar is augmented with a single argument that represents the semantics:

- NP becomes $NP(obj)$ - where obj is the FOL term that represents the noun phrase
- VP becomes $VP(rel)$ - where rel is the FOL relation (predicate) that represents the verb
- Also needs λ - expressions for verbs:
  - $\lambda x Loves(x, Mary)$ - the predicate of variable x such that x loves Mary
  - $(\lambda x Loves(x, Mary))(John)$ - the predicate applied to the argument John, yielding $Loves(John, Mary)$

14.3.5 Semantically augmented grammar fragment



$S(rel(obj)) \rightarrow NP(obj)\ VP(rel)$
$VP(rel(obj)) \rightarrow Verb(rel)\ NP(obj)$
$NP(obj) \rightarrow Name(obj)$

$Name(John) \rightarrow$ **John**
$Name(Mary) \rightarrow$ **Mary**
$Verb(\lambda x\ \lambda y\ Loves(x, y)) \rightarrow$ **loves**

Can be extended:
- Time
- Tense
- Quantification
- Pragmatics
- Etc.

Figure 59: Semantically augmented grammar fragment

14.3.6 Deriving semantics during parsing

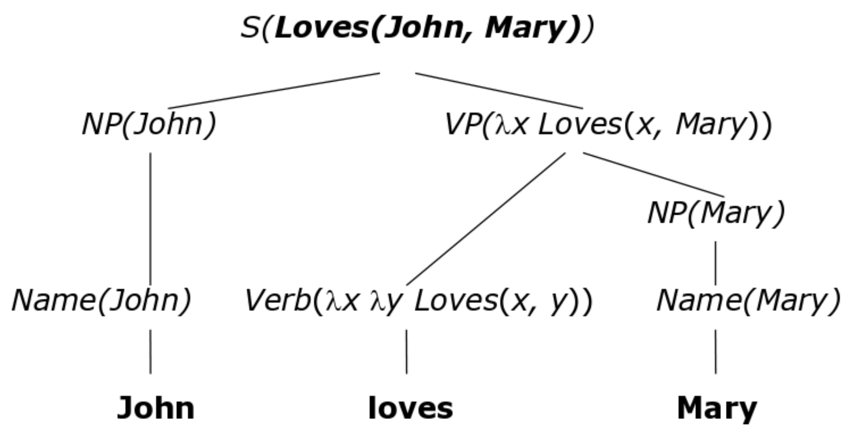


Figure 60: Deriving semantics during parsing

14.4 Steps of communication

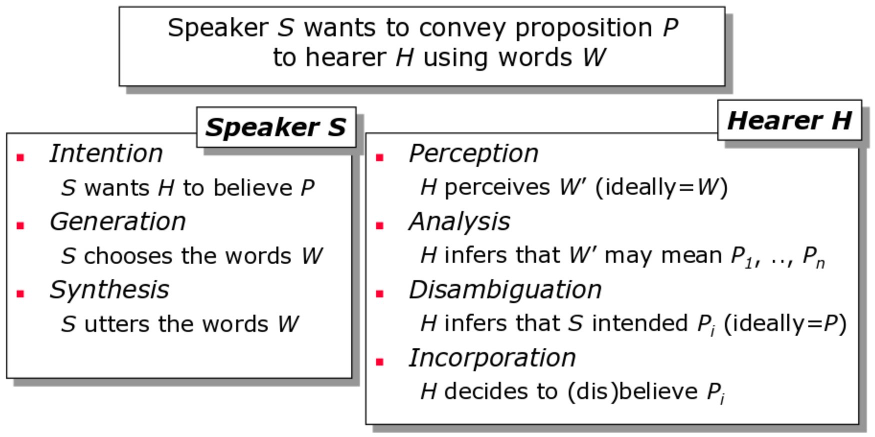

Speaker S wants to convey proposition P to hearer H using words W
Speaker S
Intention
S wants H to believe P
Generation
S chooses the words W
Synthesis
S utters the words W
Hearer H
Perception
H perceives W' (ideally=W)
Analysis
H infers that W' may mean P1, .., Pn
Disambiguation
H infers that S intended Pi (ideally=P)
Incorporation
H decides to (dis)believe Pi

Figure 61: Steps of communication

14.4.1 Speaker steps in more detail

Intention: Speaker decides that there is something to say, e.g. by reasoning about beliefs and goals of hearer, $Know(H, \neg Alive(Wumpus, S3))$. Generation: Speaker uses knowledge about language in deciding what to say, "The wumpus is dead". Synthesis: Finally, the sentence is uttered via the "speech act organ" (printer, screen, speech synthesizer, ...).

14.4.2 Hearer steps in more detail

Perception:
- The utterance is received, e.g. by speech recognition, scene analysis, ...

Analysis:
- Parsing : Recognizing constituent phrases (parse tree)
- Interpretation: Extract meaning as expression in e.g. logic

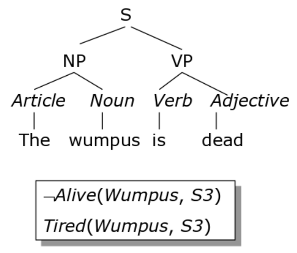


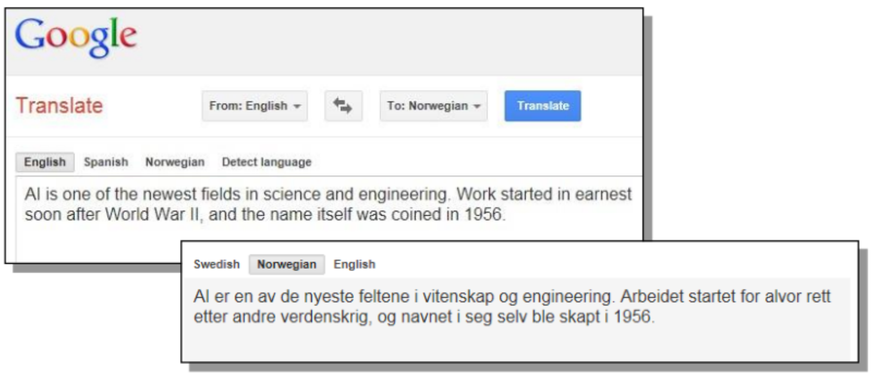
Figure 62: Hearer steps

Disambiguation:
- Analysis may yield different interpretations, and the agent must choose the most probable one, e.g. using probabilistic reasoning, $Alive(Wumpus, S3)$

Incorporation:
- Finally, the agent updates its knowledge base with the new information, $TELL(KB, \neg Alive(Wumpus, S3))$.

14.5 Machine translation

Machine translation is the automatic translation of one natural language (the source) to another language (the target)



AI is one of the newest fields in science and engineering. Work started in earnest soon after World War II, and the name itself was coined in 1956.

AI er en av de nyeste feltene i vitenskap og engineering. Arbeidet startet for alvor rett etter andre verdenskrig, og navnet i seg selv ble skapt i 1956.

Figure 63: Machine translation

14.5.1 MT by deep linguistic analysis

MT by three-step process
1. Analyze source text syntactically and semantically
2. Create deep knowledge representation of meaning of source text
3. Generate target text representing the same meaning in target syntax

Can use methods described earlier for natural language communication, but problematic. Requires rich semantic model (FOL not sufficient?) and strong parsing and generation capabilities.

14.5.2 MT by using transfer model

Large database of translation rules and examples on lexical, syntactic and semantic levels. Can match rules/examples on any level.

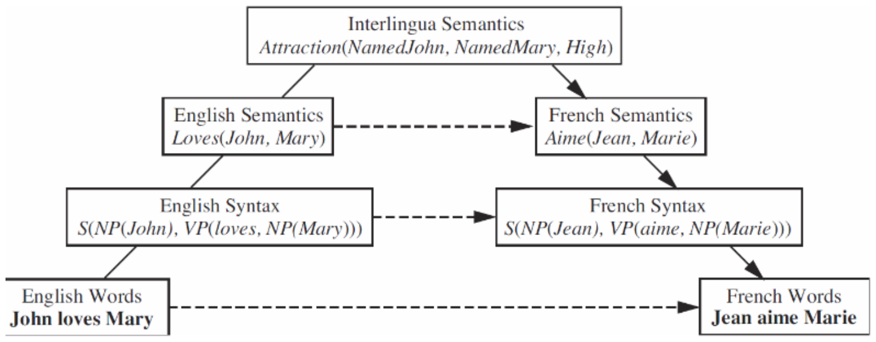


Figure 64: MT by using transfer model

14.5.3 Statistical machine translation

Successful MT systems (e.g. Google Translate) are built by training probabilistic models using statistics from large text collections. Does not need complex ontologies or grammars of source and target languages. Relies on large amounts of sample translations from which a transfer model can be learned.

15 Foundations and Prospects

15.1 The big questions

- What does it mean to think?
- Are machines able to think?
- What is intelligence?
- Can machines be intelligent?
- What does it mean to be conscious?
- Can machines be conscious?
- What is mind?
- Can machines have mind?

15.2 Weak vs. strong AI

Weak AI: Machines can be made to act as if they are intelligent

Strong AI: Machines can be made that are intelligent, have minds, and are conscious

15.3 The Turing Test

In an attempt to answer the question "Can machines think?", Alan Turing (1950) proposed the Turing test for intelligence: The computer shall have a conversation with an interrogator for 5 minutes and have a 30% chance of fooling the interrogator into believing it is human. Turing believed that by year 2000, a computer with a storage of 10^9 units will pass the Turing test. So far, no computer has passed the test. Such a machine will qualify as weak AI ("as if intelligent").

15.4 Objections to intelligent machines

Turing considered many objections to AI

- Argument from disability
- The mathematical objection
- The argument from informality

Disability: A machine can never do X

- X = to be kind, friendly, make mistakes, have sense of humor, fall in love, do something really new, ...
- Counter: Many such "impossibility claims" are unsupported, and some can be refuted

15.5 Mathematical objections to AI

An AI program is a *formal* system implemented on a computer, and subject to *theoretical limits*, e.g. the incompleteness theorem (*Gödel*): In any formal system powerful enough to do arithmetic, there are true statements that cannot be proved. Humans can overcome formal limits, e.g. by "meta-transfer" to other formalisms and are therefore inherently superior. Counter-arguments:

- Computers are finite machines, and are strictly not subject to Gödel's theorem
- Intelligent humans also suffer from inability to prove all true statements
- The brain is a deterministic physical device (some argue against this) and subject to the same formal limits as as computer

15.6 Informality objection to AI

Proposition (Dreyfus):

- Human behavior is too complex to be captured by a simple set of rules
- Since computers can only follow rules (can only do what the are told to), they cannot generate intelligent behavior on human level

This critique is directed towards simple first-order logic rule-based systems without learning, "GOFAI - Good Old Fashioned AI". Modern AI includes other reasoning and learning methods:

- Generalization from examples
- Supervised, unsupervised and reinforcement learning
- Learning with very large feature sets
- Directed sensing

Thus, AI makes progress to overcome the critique.

15.7 Strong AI - machine consciousness

Even if machines can be made to act as if they are intelligent (weak AI), "real" machine intelligence must have consciousness (strong AI). The machine must be aware of its own mental state and actions, be aware of its own beliefs, desires and intentions. Turing rejected this requirement, because we do not even know that other humans have consciousness, we can only observe their external behavior. Many will nevertheless require strong AI before they accept a machine as intelligent.

15.7.1 Can machines have mental states?

Functionalism answer: If the computer provides same answer to a problem as a human would (same function), it must have the same internal mental state.

Biological naturalism answer: Mental states are high-level and emergent features that are caused by neural activity in the brain that cannot be replicated by other means.

15.7.2 The mind-body problem

Ancient question: How is mind (soul, consciousness) related to body (brain)?

Dualist view: Mind and body are fundamentally different categories of existence

Materialist view: "Brains cause minds" (Searle). I.e. the brain is the "hardware" for the mind "software"

Accepting the materialist view, can a machine have consciousness?

15.7.3 The Chinese room (Searle)

Argument by Searle (1980):

- Human ("CPU") with no knowledge of Chinese operates in a closed room with a rulebook ("program") and a stack of paper ("memory").
- Human receives slips of paper with (for him non-intelligible) Chinese text, follows rules mechanically and returns sensible replies in Chinese.
- From the outside, it seems that the Chinese room behaves intelligently, yet the human has no idea of what he is responding to the inputs (just follows the rules).

This demonstrates that a system that passes Turing test need not be intelligent or conscious

15.7.4 The Systems reply (McCarthy)

The Chinese room argument relies on following claims:

- Certain kinds of objects are incapable of conscious understanding (in this case, Chinese)
- The human, paper, and rule book are objects of this kind
- If each of the objects is incapable of conscious understanding, then any system constructed from the objects is incapable of conscious understanding
- Therefore there is no conscious understanding in the Chinese room

In the "Systems reply" to Searle (McCarthy and others), the third claim is not accepted. If it was true, how could (conscious) humans be made of (unconscious) molecules?

15.7.5 Consciousness as emergent property

In more recent work, Searle claims that consciousness is an emergent property of properly arranged neurons, and only (biological) neurons. (Most) AI researchers agree that consciousness is an emergent property, but that the physical components underlying it can be neurons or electronic components or some other mechanism. Searle's argument is not more founded on "facts" than the opposite (AI) argument.

15.7.6 Can the strong AI question be settled?

Consciousness is not a well defined or well understood phenomena. We do not know what kind of experiment can be used to determine consciousness in a computer. Question could be settled if we discovered how consciousness can be reduced to other phenomena. As no such reduction is known, the strong AI question will remain open.

15.8 Tentative answers to some "big questions"

Weak AI (machines can be made that act as if they are intelligent). Many AI programs do in fact exhibit "intelligence". Arguments against weak AI are needlessly pessimistic. Strong AI (machines can be made that are intelligent and conscious). Difficult to prove either impossibility or possibility of this claim. The answer is not important for further progress for (weak) AI.

15.9 Recapitulation: AI as agent design

The AI "project" can be seen as the design of intelligent agents. Different agent designs are possible, from reflex agents to deliberative knowledge-based ones. Different paradigms are being used: logical, probabilistic, "neural". Do we have the necessary tools to build a complete, general-purpose agent?

15.10 Model- and utility-based agent

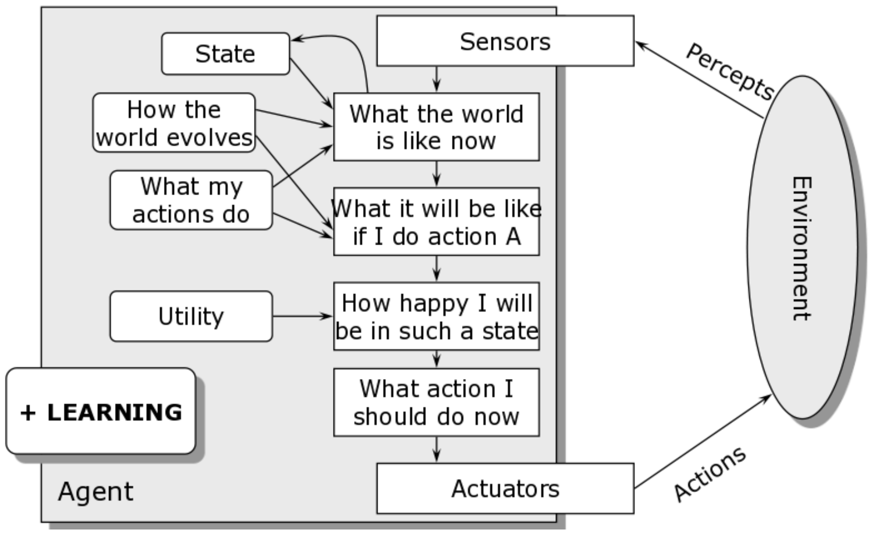

State
Sensors
Percepts
How the world evolves
What the world is like now
What my actions do
What it will be like if I do action A
Environment
Utility
How happy I will be in such a state
+ LEARNING
What action I should do now
Agent
Actuators
Actions

Figure 65: Model- and utility-based agent

15.11 State-of-the-art

Interaction with the environment:

- Improved greatly in recent years: cameras, MEMS, ...
- Dominant new environment: the Internet

Keeping track of environment's state:

- Perception and updating of internal representation
- Filtering methods for tracking uncertain environments
- Mostly low-level and propositional
- Need to improve ability to recognize higher-level objects, relations, scenes, etc.

Evaluate and select actions:

- Simple methods for planning and deciding exist
- Real-world complexity require strong abstraction ability (hierarchies)
- Great deal of development is needed

Utility as expression of preference:

- MEU is sound in principle, but depends on realistic utility functions
- Need to extract utility information from humans to guide agents

Learning capabilities

- Basic learning technology has progressed rapidly in recent years, sometimes with abilities that exceed human learning ability

However, little progress on how to learn higher level concepts from lower level (input) concepts

- Without such generalization ability, agents must be taught manually by humans

15.12 Status of AI

15.12.1 Uneven status of AI disciplines

Some parts of AI are mature, and agents can be built that outperform humans in these areas. E.g.: Game playing, logical inference, theorem proving, planning, diagnosis. Other parts of AI are evolving, where progress is being made. E.g.: Learning, vision, robotics, natural language understanding.

15.12.2 Hybrid agent architecture

Ability to incorporate different types of reasoning and decision making (from reflex to deliberation). Learning from experience (compiling).

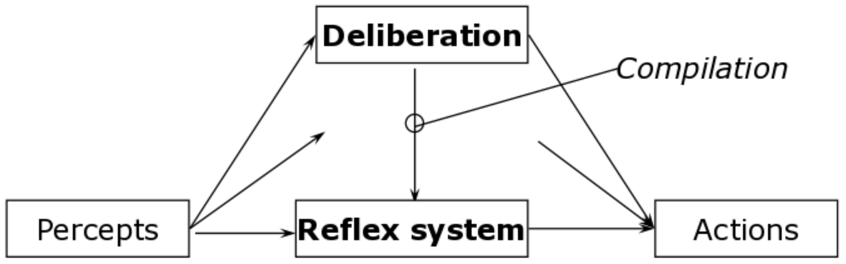


Figure 66: Hybrid agent architecture

15.12.3 Control of agent deliberation

Real-time AI: Agents in the real world must act in real-time

Anytime algorithms: Have an answer ready at all times, improve if more time available

Decision-theoretic metareasoning: Use value of information to reason about which computation to perform

Reflective architecture: Apply same kind of reasoning to internal decision-making as to external decision-making

15.12.4 AI as rational agents - right direction?

Perfect rationality

- Agent always does the right thing
- Not feasible in non-trivial domains

Calculative rationality

- Will eventually do the right ting, but must be "short-circuited"
- Underlies much of current AI

Bounded rationality

- Theory for how "real" agents solve problems
- Satisficing: Deliberate only until answer is "good enough"

Bounded optimality

- Agent does best possible given its computational resources
- Offers best promise for strong theoretical foundation for AI

16 If AI succeeds . . .

Intelligent agents, autonomous or working on behalf of humans: Who is responsible? AI impact on work and leisure, quality of life: Will it be positive or negative? AI impact on politics and power, governments and citizens: Who will gain and who will lose? If machines with high level intelligence develops, will they have rights? Relationship to humans? Will machines eventually supersede humans . . . ?

Index