Intro AI compendium

Jacob Clements og Edgar Hellesegg

August 2024

1 Introduction

1.1 What is AI?

There are no crisp definitions. The reason for this is that there is no formal definition covering all aspects of intelligence. Historically, researchers have pursued several different versions of AI. Some have defined intelligence in terms of fidelity to human performance, while others prefer an abstract, formal definition of intelligence called rationality—loosely speaking, doing the "right thing." The subject matter itself also varies: some consider intelligence to be a property of internal thought processes and reasoning, while others focus on intelligent behavior, an external characterization. From these two dimensions—human vs. rational and thought vs. behavior—there are four possible combinations, which can be summarized in the following table:

thinking vs. acting \ humanly vs. rationally		
	Systems that think like humans	Systems that think rationally
	Systems that act like humans	Systems that act rationally Rational Agents

Acting humanly: The Turing test approach

Alan Turing designed an operational test for intelligence - 1950.
The basic setup is the following:

- Interrogator in one room, human in another, system in a third. Interrogator asks questions; human and system answer.

- After 5 minutes of discussion, the Interrogator tries to guess if he has seen the human's or the computer's answers.
- The system has passed the Turing Test if the Interrogator fails 30-percent of the time.

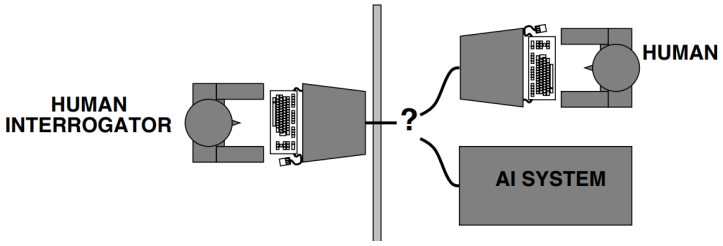


Figure 1: Turing test

Thinking humanly: The cognitive modelling approach

To say that a program thinks like a human, we must know how humans think. We can learn about human thought in three ways:

- introspection—trying to catch our own thoughts as they go by;
- psychological experiments—observing a person in action;
- brain imaging—observing the brain in action.

Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input–output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans.

Thinking rationally: The "laws of thought" approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking"— that is, irrefutable reasoning processes. His syllogisms provided patterns for argument structures that always yielded correct conclusions when given correct premises. The canonical example starts with "Socrates is a man and all men are mortal" and concludes that "Socrates is mortal". These laws of thought were supposed to govern the operation of the mind; their study initiated the field called logic. Logic as conventionally understood requires knowledge of the world that is certain—a condition that, in reality, is seldom achieved. We simply don't know the rules of, say, politics or warfare in the same way that we know the rules of chess or arithmetic. The theory of probability fills this gap, allowing rigorous reasoning with uncertain information. In principle, it allows the construction of a comprehensive model of rational thought, leading from raw perceptual information to an understanding of how the world works

to predictions about the future. This is, however, computationally intractable. Moreover, what it does not do, is generate intelligent behavior.

Acting rationally: The rational agent approach

An agent is just something that acts. Of course, all computer programs do something, but computer agents are expected to do more: operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. A *rational agent* is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome. A rational agent doesn't inherently have to think, e.g. blinking reflex, however, thinking should be in the service of rational action.

There are two advantages with rational agents:

- Not limited to "laws of thought" to achieve rationality
- Rationality is mathematically well-defined, hence, more operationable.

1.2 Foundations of AI

There are multiple subjects AI is built upon. Here is a summary of these, and why they are prominent in AI:

- Philosophy; logic, methods of reasoning, mind as physical system
- Mathematics; formal logic, computation, algorithms, probability, learning from data
- Psychology; behaviourism, cognitive psychology
- Economics; foraml theory of rational decisions, utility knowledge representation
- Linguistics; knowledge representation grammar, syntax, semantics
- Neuroscience; neurons as information processing units synapse as learning mechanism (Neural networks)
- Control theory; homeostatic systems, stability simple optimal agent designs, maximize objective function
- Computer science; engineering, hardware, computational complexity theory

1.3 Brief Histroy of AI

1969–79	Early development of knowledge-based systems
1980–88	Expert systems industry booms
1988–93	Expert systems industry busts: "AI Winter"
1985–95	Backpropagation learning returns neural networks to popularity
1988–	Resurgence of probability; general increase in technical depth "Nouvelle AI": ALife, GA
1995–	Agents, agents, everywhere...
2003–	Human-level AI back on the agenda
2005–2010	AI disappoints again, AI is not much appreciated
2012–	Deep learning and a very hot AI summer since then

1.4 Risks and Benefits of AI

- Lethal autonomous weapons; Don't require human supervision. Scalability
- Surveillance and persuasion; Security personnel vs AI, Scalability
- Biased decision making; related to hiring, evaluating bank loans and parole
- Impact on employment; machine vs human
- Safety critical applications; Difficult formal verification. Technical standards lack
- Cybersecurity; Cyberattack vs detection of cyberattack

2 Intelligent Agents

2.1 Agents and environment

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

- Human agent has the senses as sensors and bodily functions as actuators.
- Robotic agent can have cameras or radar as sensors and wheels or motors as actuators
- A software agent receives file contents, network packets, and human input (keyboard/mouse/touchscreen/voice) as sensory inputs and acts on the environment by writing files, sending network packets, and displaying information or generating sounds

We use the term percept to refer to the content an agent's sensors are perceiving. An agent's percept sequence is the complete history of everything the agent has ever perceived. Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any given percept sequence to an action.

An agent function is a math description of behaviour, maps from percept sequences to actions: $f : P- > A$. An agent program is a concrete implementation of agent function, runs on the physical architecture to produce f. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running within some physical system.

Before closing this section, we should emphasize that the notion of an agent is meant to be a tool for analyzing systems, not an absolute characterization that divides the world into agents and non-agents.

2.2 Good Behavior: The Concept of Rationality

A rational agent is one that does the right thing. Obviously, doing the right thing is better than doing the wrong thing, but what does it mean to do the right thing?

The right action should cause the agent to be most successful. Success needs to be evaluated with respect to an objective performance measure, which depends on what the agent is designed to achieve.

2.2.1 Performance measures

AI´s notion of the "right thing" is called consequentialism. Consequentialism can be described as evaluating an agent's behavior by its consequences. When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives. This sequence of actions causes the environment to go through a sequence of states. If the sequence is desirable, then the agent has performed well. This notion of desirability is captured by a performance measure that evaluates any given sequence of environment states. As a general rule, it is better to design performance measures according to what one actually wants to be achieved in the environment, rather than according to how one thinks the agent should behave.

2.2.2 Rationality

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

This leads to a definition of a rational agent. For each possible percept sequence, an ideal rational agent should do whatever is expected to maximise its performance measure, on the basis of the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

2.2.3 Omniscience, learning, and autonomy

We need to be careful to distinguish between rationality and omniscience. An omniscient agent knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality. A performance measure must therefor be realistic and rational, even though the omniscient agent knows the outcome of the actions is not optimal.

Rationality maximizes expected performance, while perfection maximizes actual performance. A rational agent is therefore not the same as a perfect agent. Our definition of rationality does not require omniscience, then, because the rational choice depends only on the percept sequence to date.

A learning agent modifies knowledge of the environment from experience. An agent is autonomous to the extent that its behaviour is determined by its own experience. Complete autonomy from the start is too difficult. Therefore the agent's designer must give guidance in terms of some initial knowledge and the ability to learn and/or reason as it operates in its environment. To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts and learning processes, we say that the agent lacks autonomy. A rational agent should be autonomous as it should learn what it can to compensate for partial or incorrect prior knowledge

2.3 The Nature of Environments

2.3.1 Specifying the task environment

Task environments are essentially the "problems" to which rational agents are the "solutions." The task environment is defined through "PEAS":(Performance, Environment, Actuators, Sensors) or:

- Performance measure
- The agent's prior knowledge of the environment
- The actions that the agent can perform
- The agent's percept sequence to date

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits, minimize impact on other road users	Roads, other traffic, police, pedestrians, customers, weather	Steering, accelerator, brake, signal, horn, display, speech	Cameras, radar, speedometer, GPS, engine sensors, accelerometer, microphones, touchscreen

Figure 2: PEAS example

2.3.2 Properties of task environments

Fully observable vs Partially observable

In fully observable environments relevant parts of the state of the environment can be sensed. In partially observable environments some parts of the environment cannot be sensed. In fully observable environments there is no need to maintain any internal state to keep track of the world, whereas in partially observable environments an agent must make informed guesses about world.

Single agent vs Multi agent

In multi agent environments the environment contains other agents whose performance measure depends on my actions and vice versa. In a single agent environment, there may exist other agents, however, these are part of the environment and do not depend on the actions of the single agent.

Deterministic vs. nondeterministic

In a deterministic environment any action has a single guaranteed effect, and no uncertainty/failure. In a nondeterministic environment There is some uncertainty about the outcome of an action. There exists multiple outcome alternatives. Nondeterministic environments are called "stochastic" if alternatives are quantified in terms of probabilities.

Episodic vs Sequential

In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action. Crucially, the next episode does not depend on the actions taken in previous episodes.In sequential environments, on the other hand, the current decision could affect all future decisions.

Static vs Dynamic vs semidynamic

If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is semidynamic.

Discrete vs Continuous

The discrete/continuous distinction applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agent. For example, the chess environment has a finite number of distinct states. Discrete has finite number of distinct states, percepts and actions.

Known vs Unknown

Strictly speaking, this distinction refers not to the environment itself but to the agent's (or designer's) state of knowledge about the "laws of physics" of the environment. In a known environment, the outcomes (or outcome probabilities if the environment is nondeterministic) for all actions are given. Obviously, if the environment is unknown, the agent will have to learn how it works in order to make good decisions.

2.4 The structure of agents

The job of AI is to design an agent program that implements the agent function—the mapping from percepts to actions. We assume this program will run on some sort of computing device with physical sensors and actuators,we call this the agent architecture. We then have $agent = arcitecture + program$

2.4.1 Agent programs

```
function TABLE-DRIVEN-AGENT(percept) returns an action
  persistent: percepts, a sequence, initially empty
              table, a table of actions, indexed by percept sequences, initially fully specified

  append percept to the end of percepts
  action ← LOOKUP(percepts, table)
  return action
```

Figure 3: Example of a trivial agent program

The Table driven agent has multiple fallacies, however it does what we want. The key challenge for AI is to find out how to write programs that, to the extent possible, produce rational behavior from a smallish program rather than from

a vast table. There are five basic kinds of agent programs that embody the principles underlying almost all intelligent systems:

- simple reflex agents
- model-based reflex agent
- goal-based agents
- utility-based agents
- learning agents

2.4.2 Simple reflex agents

These agents select actions on the basis of the current percept, ignoring the rest of the percept history(percept sequence). It is implemented through condition-action rules. An example of this rule is "The car in front is braking." Then, this triggers some established connection in the agent program to the action "initiate braking." In other words **if** car-in-front-is-braking **then** initiate-braking.

```
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
```

Figure 4: Example of a simple reflex agent

Simple reflex agents have the admirable property of being simple, but they are of limited intelligence.

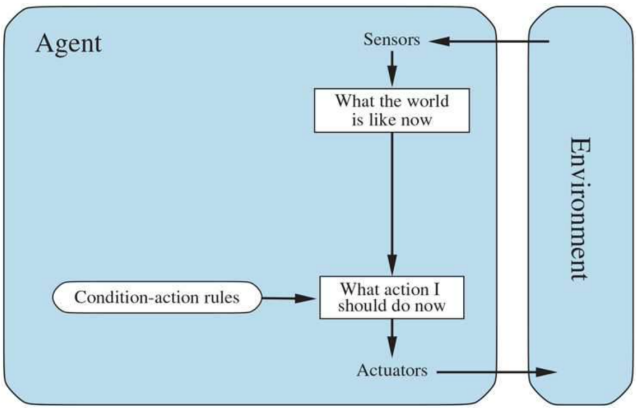


Figure 5: Simple reflex agent structure

2.4.3 Model-based reflex agents

The most effective way to handle partial observability is for the agent to keep track of the part of the world it can't see now. That is, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state. Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program in some form. First, we need some information about how the world changes over time. This knowledge about "how the world works", whether implemented in simple Boolean circuits or in complete scientific theories, is called a **transition model** of the world. Second, we need some information about how the state of the world is reflected in the agent's percepts. This kind of knowledge is called a **sensor model**.Together, the transition model and sensor model allow an agent to keep track of the state of the world, to the extent possible given the limitations of the agent's sensors. An agent that uses such models is called a **model-based agent**.

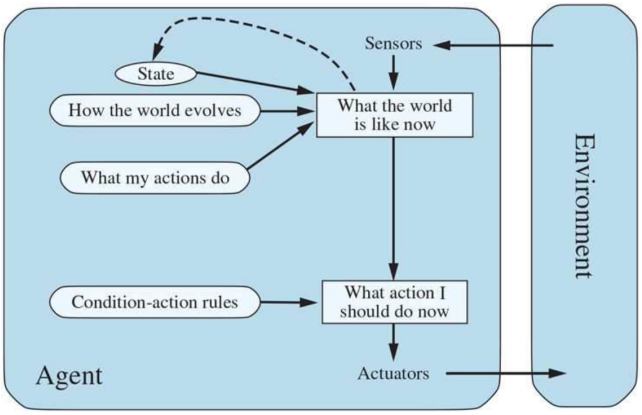


Figure 6: Model-based agent structure

2.4.4 Goal-based agents

Knowing something about the current state of the environment is not always enough to decide what to do. Sometimes, as well as a current state description, the agent needs some sort of goal information that describes situations that are desirable. The agent program can combine this with the model (the same information as was used in the model-based reflex agent) to choose actions that achieve the goal. Continuing the break light example mentioned earlier, a goal-based agent brakes when it sees brake lights because that's the only action that it predicts will achieve its goal of not hitting other cars. Although the goal-based agent appears less efficient, it is more flexible because the knowledge that supports its decisions is represented explicitly and can be modified.

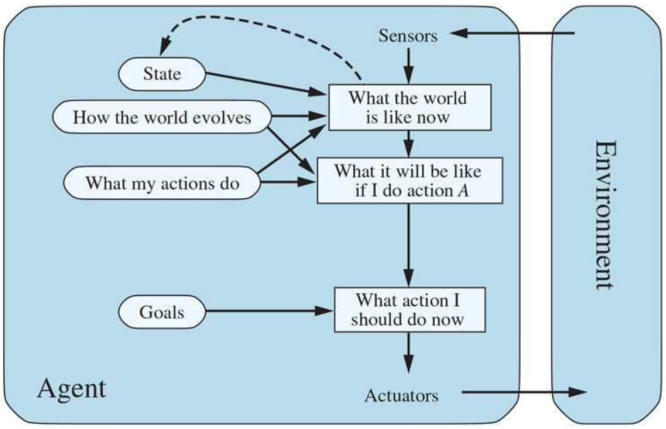


Figure 7: Goal-based agent structure

2.4.5 Utility-based agents

Goals alone are not enough to generate high-quality behavior in most environments. Goals just provide a crude binary distinction between "happy" and "unhappy" states. A more general performance measure should allow a comparison of different world states according to exactly how happy they would make the agent. Because "happy" does not sound very scientific, economists and computer scientists use the term utility instead. Utility functions provide a continuous scale rather than happy/unhappy. An agent's utility function is essentially an internalization of the performance measure. Provided that the internal utility function and the external performance measure are in agreement, an agent that chooses actions to maximize its utility will be rational according to the external performance measure. Technically speaking, a rational utility-based agent chooses the action that maximizes the **expected utility** of the action outcomes, that is, the utility the agent expects to derive, on average, given the probabilities and utilities of each outcome. An agent chooses actions that maximize the expected utility of the outcomes.

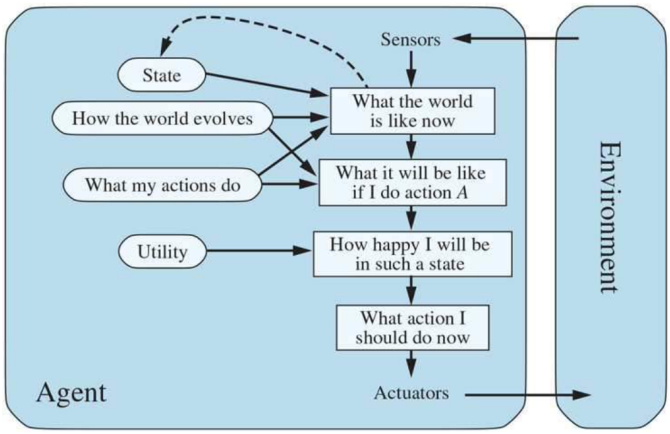


Figure 8: Utility-based agent structure

2.4.6 Learning agents

A learning agent can be divided into four conceptual components, as shown in the figure. 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 performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on 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. In shorter terms:

- Critic: evaluates current world state, determines how the performance should be modified
- Learning element: responsible for making improvements
- Problem generator: suggests explorations
- Performance element: responsible for selecting external actions

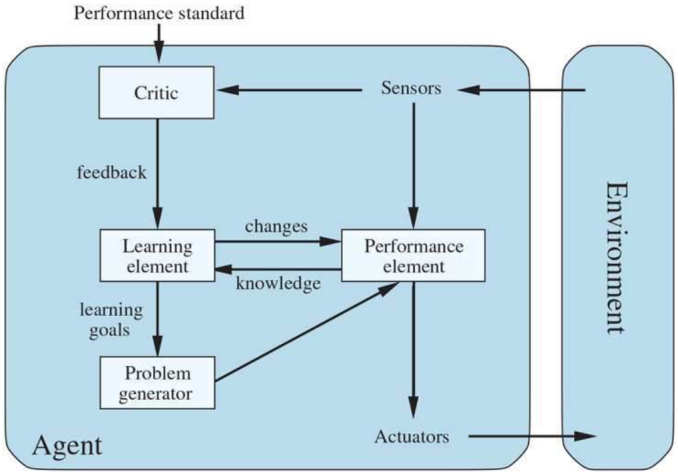
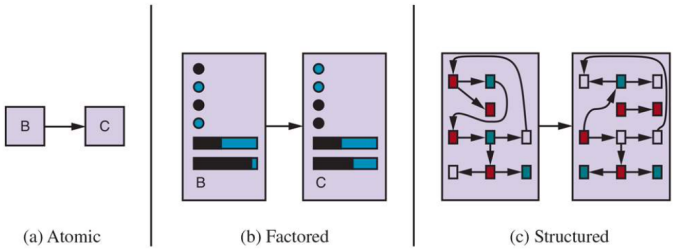


Figure 9: Learning agent structure

2.4.7 Representation of environment



Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols. (c) Structured representation: a state includes objects, each of which may have attributes of its own as well as relationships to other objects.

Figure 10: Representation of environment

In an **atomic representation** each state of the world is indivisible—it has no internal structure. A **factored representation** splits up each state into a fixed set of variables or attributes, each of which can have a value. A **structured representation**, in which objects and their various and varying relationships can be described explicitly.

3 Solving Problems by Searching

When the correct action to take is not immediately obvious, an agent may need to to plan ahead: to consider a sequence of actions that form a path to a goal state. Such an agent is called a **problem-solving agent**, and the computational process it undertakes is called **search**.

3.1 Problem-Solving Agents

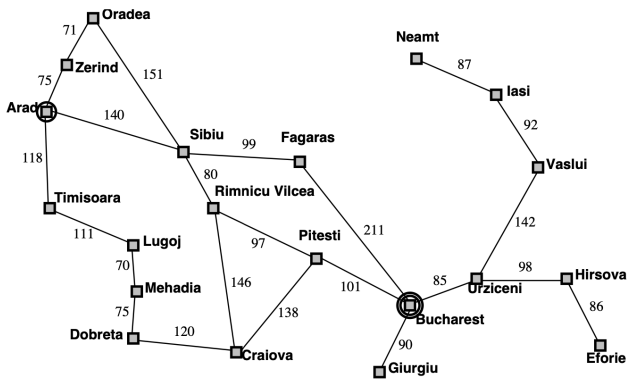


Figure 11:

We will begin with an example about search. We want to get from Arad to Bucharest. None of the roads out of Arad leades to Bucharest in one step. We must therefore have an agent that has access to information about the world, in other words, the enviorment is **known**. With this information the agent can follow a four-phase problem-solving process:

- **GOAL FORMULATION**: The agent adopts the goal of reaching Bucharest. Goals organize behavior by limiting the objectives and hence the actions to be considered.
- **PROBLEM FORMULATION**: The agent devises a description of the states and actions necessary to reach the goal.For our agent, one good model is to consider the actions of traveling from one city to an adjacent city, and therefore the only fact about the state of the world that will change due to an action is the current city.
- **SEARCH**: Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal. Such a sequence is called a **solution**.
- **EXECUTION**: The agent can now execute the actions in the solution, one at a time.

3.1.1 Search problems and solutions

Breaking the problem down further one can look at it in a different way than initially presented.

- State space: all locations in the Romania map
- Initial state: Arad
- Goal state: Bucharest
- Actions: Go from Arad to Sibiu, and to Timisoara, . . .

This can be translated into a transition model that can consider action costs.

Furthermore this can be translated to a mathematical representation:

The **state space** is a *set*:

$$S = \{Oradea, Zerind, Arad, Timisoara, Lugoj, Dobreta, \dots\}$$

The **starting** s_0 and **goal** s_g states are *elements* of the **state space**: $s_0 =$ Arad and

$$s_g = \text{Bucharest}$$

Actions can be summarised by a *function* A which maps a **state** (origin) to **another state** (destination): $A : S \to S$. Basically, a table (or a set, really . . .)

The **transition model** can be another *function*, f, which maps **an action** to a **cost** (a number): $f : A \to \mathbb{R}$

3.2 Example Problems

Just multiple examples

3.3 Search Algorithms

A **search algorithm** takes a search problem as input and returns a solution, or an indication of failure. In this chapter we consider algorithms that superimpose a **search tree** over the state-space graph, forming various paths from the initial state, trying to find a path that reaches a goal state. Each **node** in the search tree corresponds to a state in the state space and the edges in the search tree correspond to actions. The root of the tree corresponds to the initial state of the problem.

3.3.1 Generalized Search Procedure:

You are standing on the starting node

1. Check where you are standing: Is it the goal state?

2. If not, what are the nodes that can be explored here?
3. Expand the node you are in
4. Add the successor nodes into the *frontier*
5. Select (according to certain criteria - a function f) the next node to expand and move

and then repeat

3.3.2 What is a node?

A node is a *representation* of a state. It is a data structure constituting a part of a search tree:

- The state of the node
- The parent of the node
- the children of the node
- The path cost of the search (at this point)

Notice that a node is not a state, but rather a step in the search.

Terminology

- The **frontier** are those nodes I can expand
- The set of **reached** states contains both the frontier AND the expanded nodes
- node.STATE is the STATE of node
- node.PARENT is the PARENT of node

Graph properties

As many other graphs, search graphs and trees can contain redundant paths and loops. One can check the chain of parent nodes and make sure not to visit the same node twice. Coding is very different from the theoretical analysis we will do in the course. The performance of a search algorithm can be measured in different ways:

- **Completeness**: is the algorithm guaranteed to find a solution?
- **Optimality**: the solution quality. Is it optimal? (cheaper, faster, etc.)
- **Time complexity**: how long does the algorithm take? (in seconds, operations, expanded states. . .)
- **Space complexity**: how much memory do we need, for example, in the frontier or reached sets?

3.4 Uninformed Search Strategies

We use the following graph as an example graph for the graph traversing algorithms:

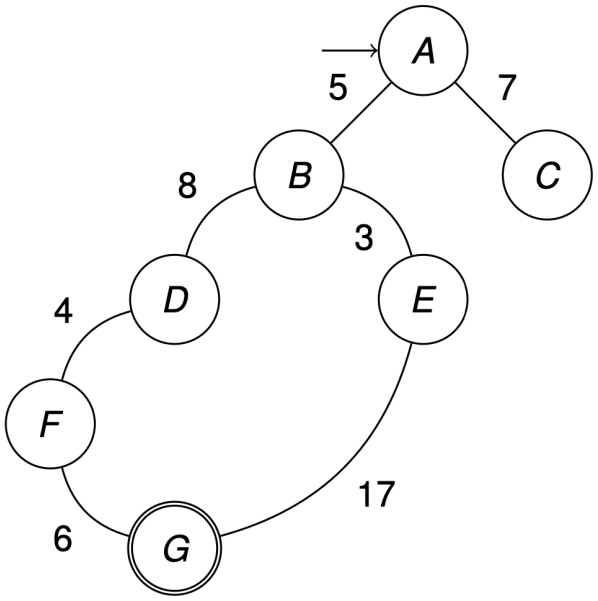


Figure 12: Example-graph

W start at A and Goal is G.

3.4.1 Breadth-first search

BFS prioritises old nodes first, and newly discovered ones last (hence the name, as it explores by breadth first). The frontier is a queue, i.e., "First In, First Out" (FIFO).

Step-by-step algorithm:

1. Add A to frontier and solution.
2. A is not goal, so Expand(A)
3. Add successors to frontier:
   - frontier=⟨B,C⟩
   - is any of those the goal?

4. Choose first element in frontier, and repeat

Note!: In BFS we check early for the goal node. That is, if we we update the frontier and find that a node in the frontier is the goal node, we move directly to that and forgo the queue data structure, and that becomes the final expanded node.

We note the following about this algorithm:

- Not optimal, unless all costs are equal
- complete: always finds a solution if space state is finite
- Time and space complexity is insane: $O(b^d)$, where b is the branching factor (number of successors to consider) and d is the depth of the solution. Note: This complexity is in relation to the algorithm performed on trees, and not on graphs where it would be the commonly know $O(VE)$

3.4.2 Depth-first search

DFS prioritises new nodes first, and previously discovered ones go last (hence the name, as it explores by depth first). The frontier is a stack, i.e., "Last In, First Out" (LIFO)

Step-by-step algorithm:

1. Add A to frontier and solution.
2. A is not goal, so Expand(A)
3. Add successors to frontier (in reverse order):
   - frontier=$\langle$B,C$\rangle$
   - Frontier now grows towards left!
4. Choose first element in frontier, and repeat

We note the following about this algorithm:

- Not necessarily optimal, returns the first viable path it finds
- Not complete: fails in infinitely deep space and spaces with loops (due to the algorithm usually being implemented as a tree-search: algorithm does not maintain visisted or closed set)
- Time complexity $O(b^m)$, and space complexity is linear $O(bm)$ where b is the branching factor and m is the maximum depth in the state space (tree version)
- One can make a smarter version of DFS with graph search (memory).Space complexity grows to exponential,and might still miss if on infinite spaces.

3.4.3 Depth-limited and Iterative deepening search (No need to study!!!)

Two other ideas lie on imposing a limit on DFS, both as tree search strategies.

- Use DFS with DepthLimit = 1
- If no solution found, then try increasing the DepthLimit iteratively until a set cutoff.
- Iterative deepening will try multiple levels and return either a solution if it exists, a failure if it does not, or a cutoff.
- A cutoff means the maximum depth we set previously was reached, so a solution might exist deeper than the levels we explored.

If we impose these limits on the DFS algorithm, we get the following:

- Always complete if solution exists and state space is finite
- Not cost optimal, unless costs are equally weighted as for BFS
- Time complexity: $O(b^d)$
- Space complexity: $O(bd)$

3.4.4 Summary of uniformed search strategies:

- They systematically navigate the search space blindly - not questioning where the goal may be in the space
- The search space is often very large

Hence, we could try to be smarter with it by using information.

3.5 Informed Search Strategies

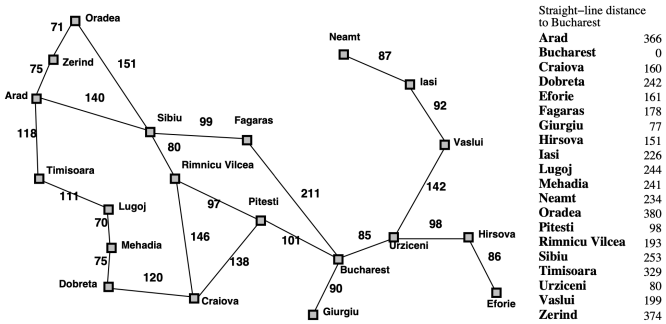
To take better informed decisions, we can use a domain-specific hint about how "desirable" a state can be. This is usually done by using a heuristic function $h(n)$, where $h : S \to \mathbb{R}$, i.e., a guessing function about an estimated remaining cost to the goal.

3.5.1 Greedy best-first search

Greedy best-first search is a form of best-first search that expands first the node with the lowest value—the node that appears to be closest to the goal—on the grounds that this is likely to lead to a solution quickly. That is, It selects nodes based only on the heuristic value and ignores the actual cost to reach the node from the start (i.e., it does not consider $g(n)$).
We can summarize this algorithm as follows:

1. Start in a state
2. Check if in goal state
3. If not, expand and update frontier using the heuristic
4. Choose the best of the given estimates and move there

Example: Using h as the straight line distance to goal



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

Figure 13:

The following explanation shows the progress of a greedy best-first search using $h(n)$ to find a path from Arad to Bucharest.

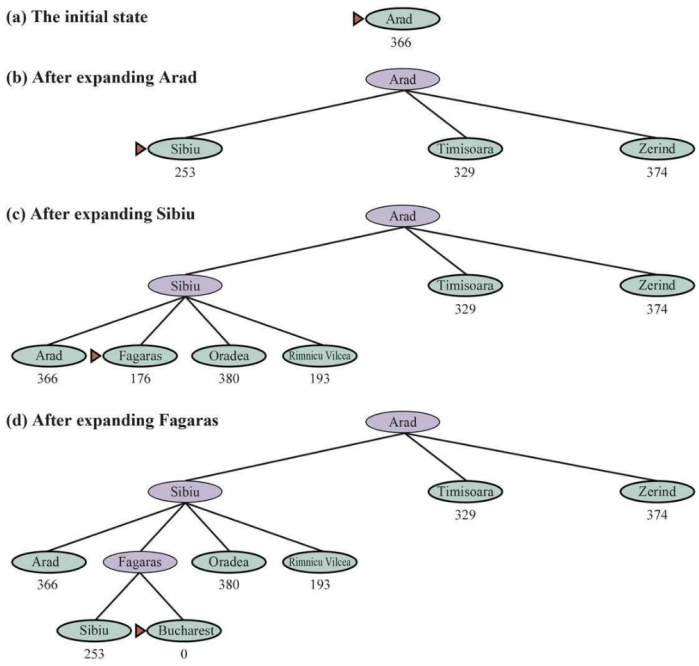


Figure 14:

The first node to be expanded from Arad will be Sibiu because the heuristic says it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras because it is now closest according to the heuristic. Fagaras in turn generates Bucharest, which is the goal

We note the following about this algorithm:

- Complete if we have a finite state space with no cycles
- Not necessarily optimal

3.5.2 A* Search

The most common informed search algorithm is A* search, a best-first search that uses the evaluation function $f(n) = g(n)+h(n)$, where $g(n)$ is the path cost from the initial state to the current node and $h(n)$ is the estimated cost of the shortest path from current node to the goal state. $f(n)$ is then the estimated cost of the cheapest solution through n to the goal. It is **complete** for positive costs, within a finite state space and an existing solution. It is **cost optimal** if certain conditions are met. Those conditions are:

- Arc costs need to be positive(They usually are)

- The heuristic function needs to be **admissible** and non-negative.

We say a heuristic h is admissible if it ***never*** overestimates the cost from a node to the goal node.
Another important (and even stronger) property of a heuristic h is consistency. A heuristic h is consistent if for every node n and all of its successors n' generated by an action a, we have $h(n) \leq c(n, a, n') + h(n')$. In other words, the estimate of a node should be less or equal than the the estimate of a descendant plus the cost of reaching there. A heuristic that is consistent is always admissible. Since a consistent heuristic is admissible, then a consistent heuristic is also always optimal. A consistent heuristic $h(n)$ ensures that the cost function $f(n) = g(n) + h(n)$ is **monotonic non-decreasing**. That means that $f(n)$ is non-decreasing along any path. A* is **optimally efficient** with a consistent heuristic. This means that any other search algorithm with the same heuristic values must expand all nodes that A* expanded. However, the main issue of A* lies on its memory use. Some ways to reduce it:

- Reference count – remove a state from reached when there are no more ways to reach it.
- Beam search – limit size of frontier to k-best candidates.
- Iterative deepening A* – gradually increase the f -cost cutoff.
- Memory-bounded A* – expand until memory is full, and then drop the worst candidate from frontier

For completeness, we also summarize this algorithm as points: We can summarize this algorithm as follows:

1. Start in a state
2. Check if in goal state
3. If not, expand and update frontier using the evaluation function $f(n) = g(n) + h(n)$.
4. Choose the node with the lowest evaluation and expand it, and adding it to the frontier, and repeat.

A generalised heuristic search

Generalised heuristic search:

$$f(n) = g(n) + wh(n) \tag{1}$$

where w is a weight defining how important the heuristic $h(n)$ is.

- With $w = 0$ you only care about the cost of the path
  - Choose the cheapest!

  - This is called uniform-cost search and it's an uninformed search.
  - It is also known as Dijkstra's algorithm.
  - complete if we have finite state space and non-negative weights
- With $w = \infty$ you only care about the estimate
  - Choose the one that seems the cheapest
  - This is Greedy Best-First search.
- with w = 1 you care equally about the path cost and the estimates
  - This is A*
- All these algorithms have $O(b^d)$ as time and space complexity

Dominance: comparing heuristics

Which of the heuristics is better? Admissible heuristics can be compared by looking at their values. An admissible heuristic h_2 it is said to dominate another admissible heuristic h_1 if for all nodes n if $h_2(n) \geq h_1(n)$. This will reflect in A* expanding fewer nodes on h_2, and thus find an optimal solution, faster.

4 Search in Complex Environments

Both informed and uninformed searching strategies are designed to explore search spaces systematically. They keep one or more paths in memory, and record which alternatives have been explored at each point along the path. The path to that goal constitutes a solution. But in most problems in the real world, the path to a solution might be irrelevant. If we only care about finding a solution, then there are better ways to search the space.

4.1 Local Search

Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached. That means they are not systematic—they might never explore a portion of the search space where a solution actually resides. However, they have two key advantages:

1. they use very little memory
2. (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable

Local search algorithms can also solve optimization problems, in which the aim is to find the best state according to an objective function.

The Search Landscape

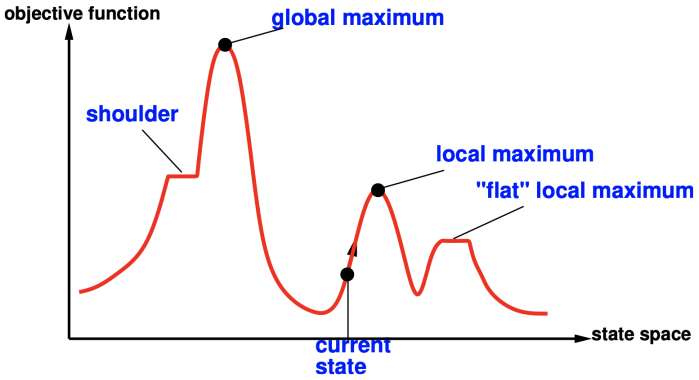


Figure 15:

Usually, the state space is referred to as the search space.

- Each point in the landscape represents a state in the search space and has "an elevation" (its $h(n)$)
- If the elevation corresponds to an objective function, then the aim is to find the highest peak (or maximum)
- If the elevation corresponds to a cost function, then we look for the lowest valley (or minimum)

Recall our search problems.

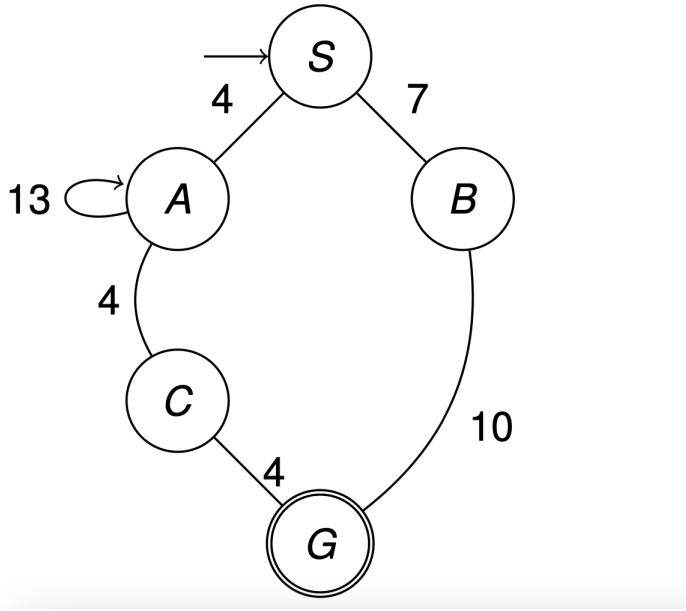


Figure 16:

- A is a neighbour of S, C and itself because those are the states than can be reached from A.
- The neighbourhood of A is then (A, C, S).
- This concept of neighbourhood is very important for local search, as we decide where to move next by looking around us.

4.1.1 Hill-climbing search

Idea: Go to the *best* spot you see now.

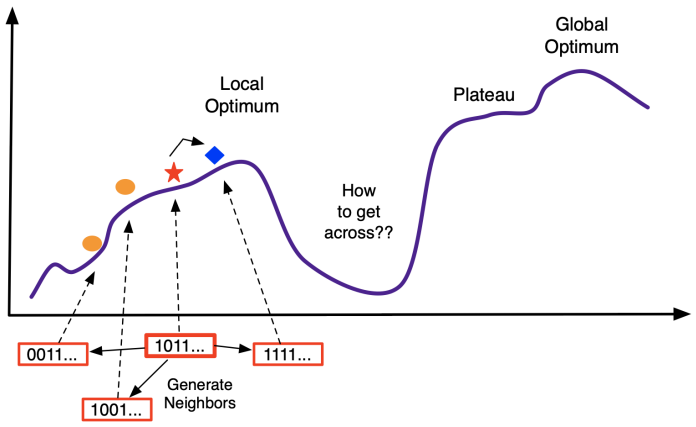


Figure 17:

Assume you are doing maximisation. You then want to climb the tallest peak. This is called **hill-climbing**. If you are **minimising** instead, then the procedure is called gradient descent as we want to move towards the direction where the difference in "height" is largest.

How to get across?

Both hill-climbing and gradient descent get stuck in local optima. How do we get out of this mess?

- Idea 1: take some not so good decisions every now and then!
  - This is what we call **stochastic local search**.
- Idea 2: Make it so that you gradually reduce the frequency of taking such "bad" decisions
  - This is the key to the **simulated annealing** algorithm
- Idea 3: Search multiple paths in batches
  - This is the key idea behind **population-based optimisation**
- Idea 4: Increase the neighbourhood size
  - For example, consider 2-moves-away adjacency instead
- Idea 5: Jump!
  - Either via long jumps when you are not doing very good
  - Or doing short hops when you are in a promising state (you do not want to miss it)

Genetic Algorithms

A well-known metaheuristic in the family of population-based optimisers is the genetic algorithm.

1. Start with a population of k randomly generated states
2. Randomly choose two parent states weighted by their fitness (objective function)
3. Generate child states by combining parent states randomly
4. Add child states to the population
5. Replace the old population by the new

This process will be repeated until a solution has been found, or until enough generations have been replaced.

4.2 Local Search in Continuous Spaces

Ikke fokus i forelesning

4.3 Search with Nondeterministic Actions

So far, we have assumed that actions are deterministic. That our intended action will **always** yield the result we expect. In the real-world, things do not always go as expected. To account for different possible outcomes, we need to come up with a contingency plan instead of a single path of actions.

4.3.1 The erratic vacuum world

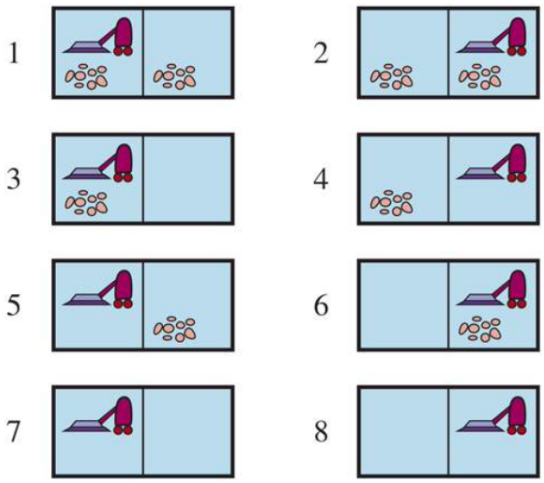


Figure 18:

When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too. When applied to a clean square the action sometimes deposits dirt on the carpet.
To provide a precise formulation of this problem, we need to generalize the notion of a transition model from Chapter 3 . Instead of defining the transition model by a RESULT function that returns a single outcome state, we use a RESULTS function that returns a set of possible outcome states. For example, in the erratic vacuum world, the Suck action in state 1 cleans up either just the current location, or both locations:
If we start in state 1, no single sequence of actions solves the problem, but the following conditional plan does:

$$[Suck, \textbf{if } State = 5 \textbf{ then}[\textbf{Right}, Suck] \textbf{ else}[]] \tag{2}$$

Here we see that a conditional plan can contain if–then–else steps; this means that solutions are trees rather than sequences.

4.3.2 AND–OR search trees

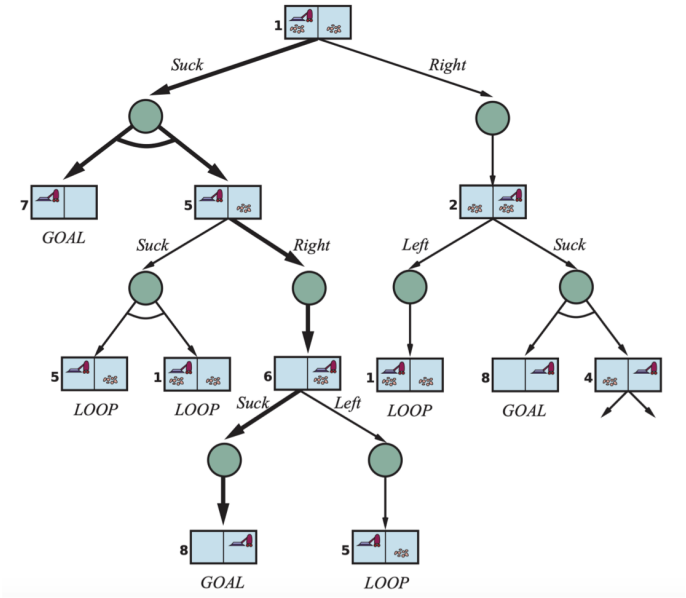


Figure 19:

OR nodes represent actions(shown as states). **AND** nodes represent outcomes(shown as circles). The solution found is shown in bold lines. Since it is a tree, we can search in it. This is called **AND-OR** search. It is recursive, with a base case of either failure or an empty plan. AND–OR graphs can be explored either breadth-first or best-first.

4.4 Search in Partially Observable Environments

Mange flere figurer og eksempler i boka

So far, we have assumed that the agent knows exactly the state of its environment. In reality, an agent receives partial (and possibly noisy) observations. Therefore, the state can only be estimated through a "belief".

4.4.1 Searching with no observation

When the agent's percepts provide no information at all, we have what is called a **sensorless** problem (or a conformant problem). Consider a sensorless version of the (deterministic) vacuum world. Assume that the agent knows the geography of its world, but not its own location or the distribution of dirt. In that case, its initial belief state is (1,2,3,4,5,6,7,8). Now, if the agent moves *Right* it will be in

one of the states (2,4,6,8). The agent as gained information without perceiving anything. After [Right,Suck] the agent will always end up in one of the states (4,8). Finally, after [Right,Suck,Left,Suck] the agent is guaranteed to reach the goal state 7, no matter what the start state. We say that the agent can **coerce** the world into state 7.

4.4.2 Searching in partially observable environments

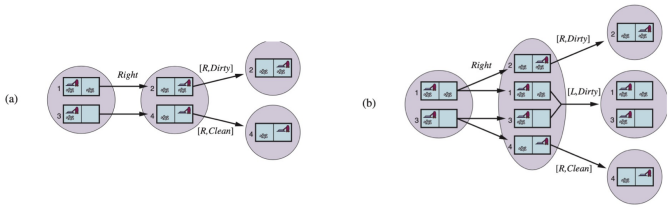


Figure 20:

The agent knows where it is and see the dirt (if any) on its spot. The transition model becomes a function of a belief state, an action, and a another belief state. In case of non-determinism (right) we consider possible outcomes on different possibilities.

4.4.3 Solving partially observable problems

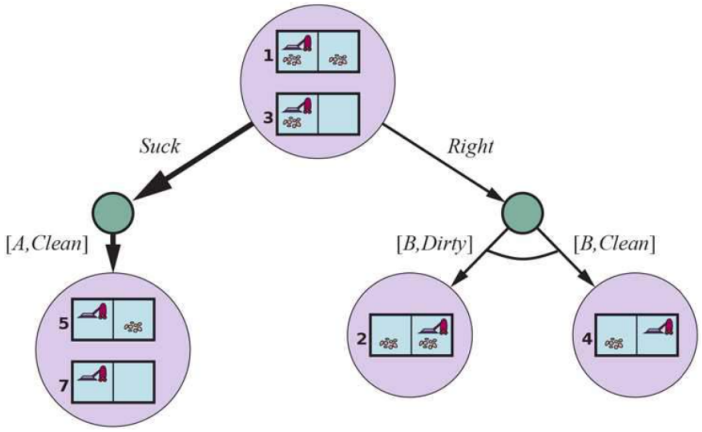


Figure 21:

Using an AND-OR tree. Notice how the nodes are now belief states. The solution is a conditional plan.

6 Constraint Satisfaction Problems

In this chapter we break open the black box by using a **factored representation** for each state: a set of **variables**, each of which has a **value**. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or **CSP**.
CSP search algorithms take advantage of the structure of states and use general rather than domain-specific heuristics to enable the solution of complex problems. The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints. CSPs have the additional advantage that the actions and transition model can be deduced from the problem description.

6.1 Defining Constraint Satisfaction Problems

Definition:
A constraint satisfaction problem consists of three components, X, D and C:

- X is a set of variables $\{X_1, ..., X_n\}$
- D is the set of domains $\{D_1, ..., D_n\}$ one for each variable $X_i \epsilon D_i$.
- C is a set of constraints that specify allowable combinations of values $\{C_1, ..., C_m\}$

A domain, D_i , consists of a set of allowable values, $\{v_1, ..., v_k\}$, for variable X_i. Different variables can have different domains of different sizes. Each constraint C_j consists of a pair $<$ scope, rel $>$, where *scope* is a tuple of variables that participate in the constraint and *rel* is a relation that defines the values that those variables can take on. A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation.
CSPs deal with assignments of values to variables, $\{X_i = v_i, X_j = v_j\}$. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is assigned a value, and a solution to a CSP is consistent, complete assignment. A partial assignment is one that leaves some variables unassigned, and a partial solution is a partial assignment that is consistent. Solving a CSP is an NP-complete problem in general, although there are important subclasses of CSPs that can be solved very efficiently.

Why formulate probles as CSPs: In atomic state-space search we can only ask: is this specific state a goal? No? What about this one? With CSPs, once we find out that a partial assignment violates a constraint, we can immediately discard further refinements of the partial assignment. Furthermore, we can see why the assignment is not a solution—we see which variables violate a constraint—so we can focus attention on the variables that matter. As a result,

many problems that are intractable for atomic state-space search can be solved quickly when formulated as a CSP.

6.1.1 Coloring problem

Assign red, green or blue to each region, but neighboring regions cannot have the same color.

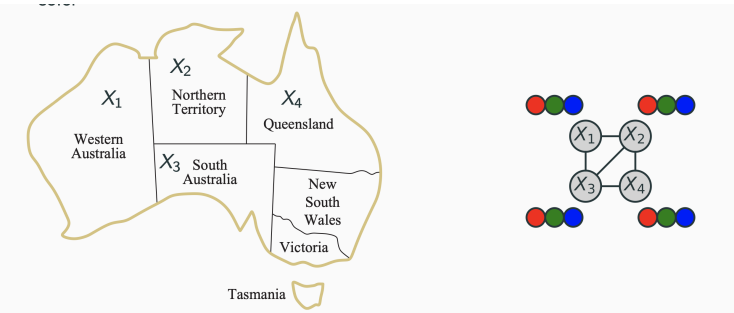


Figure 22:

- Variables: X_1, X_2, X_3, X_4
- Domains: $D_1 = ... = D_4 = (red, green, blue)$
- Constraints: $(X_1 \neq X_2, X_1 \neq X_3, X_2 \neq X_3, X_2 \neq X_4, X_3 \neq X_4)$

Note: As we can see, it can be helpful to visualize a CSP as a constraint graph. The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.

6.1.3 Variations on the CSP formalism

The simplest kind of CSP involves variables that have **discrete**, **finite domains**. A discrete domain can be **infinite**, such as the set of integers or strings. Constraint satisfaction problems with **continuous domains** are common in the real world and are widely studied in the field of operations research. The best-known category of continuous-domain CSPs is that of **linear programming problems**, where constraints must be linear equalities or inequalities. In addition to examining the types of variables that can appear in CSPs, it is useful to look at the types of constraints. The simplest type is the **unary constraint**, which restricts the value of a single variable. A **binary constraint** relates two variables. A binary CSP is one with only unary and binary constraints. A constraint involving an arbitrary number of variables is called a **global constraint**.

6.2 Constraint Propagation: Inference in CSPs

An atomic state-space search algorithm makes progress in only one way: by expanding a node to visit the successors. A CSP algorithm has choices. It can generate successors by choosing a new variable assignment, or it can do a specific type of inference called **constraint propagation**: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on. The idea is that this will leave fewer choices to consider when we make the next choice of a variable assignment. Constraint propagation may be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.
The key idea is local consistency. If we treat each variable as a node in a graph and each binary constraint as an edge, then the process of enforcing local consistency in each part of the graph cause inconsistent values to be eliminated throughout the graph.

6.2.1 Node consistency

A single variable (corresponding to a node in the CSP graph) is **node-consistent** if all the values in the variable's domain satisfy the variable's unary constraints. It is easy to eliminate all the unary constraints in a CSP by reducing the domain of variables with unary constraints at the start of the solving process. It is also possible to transform all n-ary constraints into binary ones. Because of this, some CSP solvers work with only binary constraints, expecting the user to eliminate the other constraints ahead of time. We make that assumption for the rest of this chapter, except where noted.

6.2.2 Arc consistency

A variable in a CSP is **arc-consistent** if every value in its domain satisfies the variable's binary constraints. More formally,X_i, is arc-consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j). A graph is arc-consistent if every variable is arc-consistent with every other variable.

Make (X_i, X_j) arc-consistent: Remove values from D_i to make X_i arc-consistent with X_j.

AC-3 Algorithm

- start with a queue of initial arc(s)

- while queue is not empty
  - pop an arc and make arc-consistent
  - if the domain was reduced for a variable, add neighbouring arcs to the queue to ensure their continued consistency due to the domain reduction.

Example:

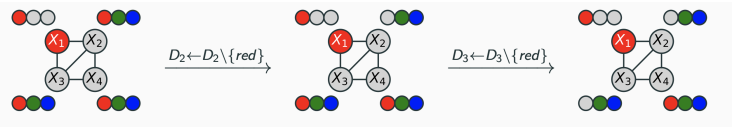


Figure 23:

Complexity:
Assume a CSP with n variables, each with domain size at most d, and with c binary constraints (arcs). Each arc (X_k, X_j) can be inserted in the queue only d times because X_i has at most d-values to delete. Checking consistency of an arc can be done in $O(d^2)$ time as we in the worst case would have to check d values of X_i against d values of X_j. There are c constraints, and handling one constraint might take d^3 operations if each possible value removal leads to a re-check of the constraint.Thus $O(cd^3)$ is the total worst-case time.

6.3 Backtracking Search for CSPs

Sometimes we can finish the constraint propagation process and still have variables with multiple possible values. In that case we have to search for a solution.

```
function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, assignment)
      if inferences ≠ failure then
        add inferences to csp
        result ← BACKTRACK(csp, assignment)
        if result ≠ failure then return result
        remove inferences from csp
      remove {var = value} from assignment
  return failure
```

Figure 24:

Whereas the uninformed search algorithms of Chapter 3 could be improved only by supplying them with domain-specific heuristics, it turns out that backtracking search can be improved using domain-independent heuristics that take advantage of the factored representation of CSPs. The following sections shows how this is done, as it also explains the functions called in the backtracking algorithm.

Algorithm in Layman Terms: The backtracking search procedure repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to extend each one into a solution via a recursive call. If the call succeeds, the solution is returned, and if it fails, the assignment is restored to the previous state, and we try the next value. If no value works then we return failure.

6.3.1 Variable and value ordering

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is static ordering: choose the variables in order, $\{X_1, X_2, ...\}$. The next simplest is to choose randomly. Neither strategy is optimal. The intuitive idea—choosing the variable with the fewest "legal" values—is called the **minimum-remaining-values** (MRV) heuristic. The MRV heuristic usually performs better than a random or static ordering, sometimes by orders of magnitude, although the results vary depending on the problem.

If all variables have the same amount of legal values, the degree heuristic comes in handy. The **degree heuristic** attempts to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables. Once a variable has been selected, the algorithm must decide on the order in which to examine its values. The least-constraining value heuristic is effective for this.

The **least-constraining-value** heuristic is effective for this. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph.

Intuitive explanation of the heuristics:
Why should variable selection be fail-first, but value selection be fail-last? Every variable has to be assigned eventually, so by choosing the ones that are likely to fail first, we will on average have fewer successful assignments to backtrack over. For value ordering, the trick is that we only need one solution; therefore it makes sense to look for the most likely values first. If we wanted to enumerate all solutions rather than just find one, then value ordering would be irrelevant.

6.3.2 Interleaving search and inference

We saw how AC-3 can reduce the domains of variables before we begin the search. But inference can be even more powerful during the course of a search:

every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reductions on the neighboring variables.

Forward Checking: One of the simplest forms of inference is called forward checking. After assigning a value to a variable, explore the domains of neighboring unassigned variables, and remove values whose assignments would violate a constraint.

For many problems the search will be more effective if we combine MRV heuristic with forward checking. We can view forward checking as an efficient way to incrementally compute the information that the MRV heuristic needs to do its job.

MAC (Maintaining Arc Consistency) Algorithm: Although forward checking detects many inconsistencies, it does not detect all of them. The problem is that it doesn't look ahead far enough. After a variable X_i is assigned a value, the INFERENCE procedure calls AC-3, but instead of a queue of all arcs in the CSP, we start with only the arcs (X_j, X_i) for all X_j that are unassigned variables that are neighbors of X_i. From there, AC-3 does constraint propagation in the usual way, and if any variable has its domain reduced to the empty set, the call to AC-3 fails and we know to backtrack immediately. We can see that MAC is strictly more powerful than forward checking because forward checking does the same thing as MAC on the initial arcs in MAC's queue; but unlike MAC, forward checking does not recursively propagate constraints when changes are made to the domains of variables.

6.3.3 Intelligent backtracking: Looking backward

The BACKTRACKING-SEARCH algorithm has a very simple policy for what to do when a branch of the search fails: back up to the preceding variable and try a different value for it. This is called **chronological backtracking** because the most recent decision point is revisited. In this subsection, we consider better possibilities.

The **backjumping** method backtracks to the most recent assignment in the conflict set. This method is easily implemented by a modification to BACKTRACK such that it accumulates the conflict set while checking for a legal value to assign. If no legal value is found, the algorithm should return the most recent element of the conflict set along with the failure indicator.

Backjumping occurs when every value in a domain is in conflict with the current assignment; but forward checking detects this event and prevents the search from ever reaching such a node! In fact, it can be shown that every branch pruned by backjumping is also pruned by forward checking. Hence, simple backjumping is redundant in a forward-checking search or, indeed, in a search that uses stronger consistency checking

6.4 Local Search for CSPs

Local search algorithms turn out to be very effective in solving many CSPs. They use a complete-state formulation where each state assigns a value to every variable, and the search changes the value of one variable at a time.

8-queens problem

function MIN-CONFLICTS(*csp*, *max_steps*) **returns** a solution or *failure*
  inputs: *csp*, a constraint satisfaction problem
    max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for *csp*
  for *i* = 1 to *max_steps* **do**
    if *current* is a solution for *csp* **then return** *current*
    var ← a randomly chosen conflicted variable from *csp*.VARIABLES
    value ← the value *v* for *var* that minimizes CONFLICTS(*csp*, *var*, *v*, *current*)
    set *var* = *value* in *current*
  return *failure*

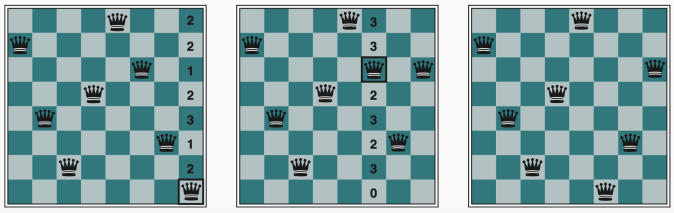


Figure 25: The MIN-CONFLICTS local search algorithm for CSPs. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

We would like to change the value to something that brings us closer to a solution; the most obvious approach is to select the value that results in the minimum number of conflicts with other variables, the **min-conflicts** heuristic.

Note: All the local search techniques from Section 4.1 are candidates for application to CSPs, and some of those have proved especially effective.

6.4 The Structure of Problems

Tree-structured CSPs:
A constraint graph is a tree when any two variables are connected by only one path. Any tree-structured CSP can be solved in time linear in the number of variables. The key is a new notion of consistency, called directional arc consistency or DAC. A CSP is defined to be directional arc-consistent under an ordering of variables $X_1, ..., X_n$ if and only if every X_i is arc-consistent with

each X_j for $j > i$.
To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree. Such an ordering is called a topological sort. Below is an example:

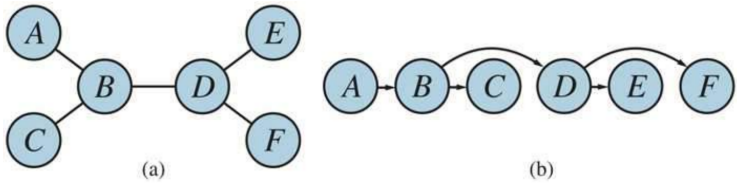


Figure 26: (a) The constraint graph of a tree-structured CSP. (b) A linear ordering of the variables consistent with the tree with as the root. This is known as a topological sort of the variables.

Any tree with nodes has edges, so we can make this graph directed arc-consistent in $O(n)$ steps, each of which must compare up to d possible domain values for two variables, for a total time of $O(nd^2)$. Once we have a directed arc-consistent graph, we can just march down the list of variables and choose any remaining value. Since each edge from a parent to its child is arc-consistent, we know that for any value we choose for the parent, there will be a valid value left to choose for the child. That means we won't have to backtrack; we can move linearly through the variables. The complete algorithm is given below:

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or *failure*
  inputs: *csp*, a CSP with components X, D, C

  $n \leftarrow$ number of variables in X
  assignment $\leftarrow$ an empty assignment
  root $\leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, *root*)
  for $j = n$ **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent **then return** *failure*
  for $i = 1$ **to** n **do**
    assignment[X_i] $\leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** *failure*
  return *assignment*

Figure 27: The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

Now that we have an efficient algorithm for trees, we can consider whether more general constraint graphs can be reduced to trees somehow.

6.5.1 Cutset Conditioning

The first way to reduce a constraint graph to a tree involves assigning values to some variables so that the remaining variables form a tree. We perform Cutset conditioning using the following **algorithm:**

For each possible assignment to the cycle cutset:

- remove values from the remaining domains that would violate a constraint with the cycle cutset
- if the remaining tree-structured CSP has a solution, return the solution together with the assignment to the cycle cutset

Cycle cutset: This is a subset, S, of the CSP's variables such that the constraint graph becomes a tree after its removal.

6.5.2 Tree Decomposition

The second way to reduce a constraint graph to a tree is based on constructing a tree decomposition of the constraint graph: a transformation of the original graph into a tree where each node in the tree consists of a set of variables. A tree decomposition must satisfy these three requirements:

- Every variable in the original problem appears in at least one of the tree nodes.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the tree nodes.
- If a variable appears in two nodes in the tree, it must appear in every node along the path connecting those nodes.

Once we have a tree-structured graph, we can apply TREE-CSP-SOLVER to get a solution in $O(nd^2)$ time, n where is the number of tree nodes and d is the size of the largest domain. But note that in the tree, a domain is a set of tuples of values, not just individual values.

The algorithm for tree decomposition is given as follows:

- Decompose the original graph into a tree where each node consists of overlapping subproblems that are solved independently
- Solve the tree-structured CSP.

A given graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible. (Putting all the variables into one node is technically a tree, but is not helpful.) The tree width of a tree decomposition of a graph is one less than the size of the largest node; the tree width of the graph itself is defined to be the minimum width among all its tree decompositions. If a graph has tree width, w, then the problem can be solved in $O(nd^{w+1})$ time given the corresponding tree decomposition. Hence, CSPs with constraint graphs of bounded tree width are solvable in polynomial time. Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

7 Logical Agents

Humans, it seems, know things; and what they know helps them do things. In AI, knowledge-based agents use a process of reasoning over an internal representation of knowledge to decide what actions to take.

7.1 Knowledge-Based Agents

The central component of a **knowledge-based** agent is its knowledge base, or KB. A knowledge base is a set of **sentences**. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. When the sentence is taken as being given without being derived from other sentences, we call it an **axiom**.

An **Inference engine** is a set of procedures that use the representational language to infer new facts from known ones as well as to answer a variety of KB queries. We have two important operations on the KB:

- ***Tell***: add new knowledge to KB
- ***Ask***: ask questions about the knowledge in the KB. Questions are "asked"/triggered in two ways:
  - A direct question from the user that doesn't require reasoning but just **retrieval** from the KB
  - A question representing a lack of knowledge required to solve a problem. This knowledge is implicit in the KB and need to be **inferred**.

7.2 The Wumpus World

The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has

only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only redeeming feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

The precise definition of the task environment is given by the PEAS description:

- **PERFORMANCE MEASURE:** $+1000$ for climbing out of the cave with the gold, -1000 for falling into a pit or being eaten by the wumpus, -1 for each action taken, and -10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
- **ENVIRONMENT:** A 4x4 grid of rooms, with walls surrounding the grid. The agent always starts in the square labeled $[1, 1]$, facing to the east. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
- **ACTUATORS:** The agent can move *Forward, TurnLeft by 90 degrees*, or *TurnRight by 90 degrees*. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square $[1, 1]$.
- **SENSORS:** The agent has five sensors, each of which gives a single bit of information:
  - In the squares directly (not diagonally) adjacent to the wumpus, the agent will perceive a *Stench*.
  - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
  - In the square where the gold is, the agent will perceive a *Glitter*.
  - When an agent walks into a wall, it will perceive a *Bump*.
  - When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

  The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [*Stench,Breeze,None,None,None*].

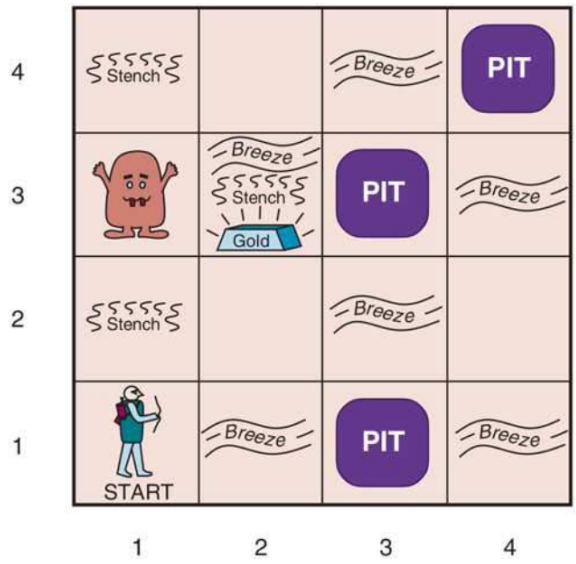


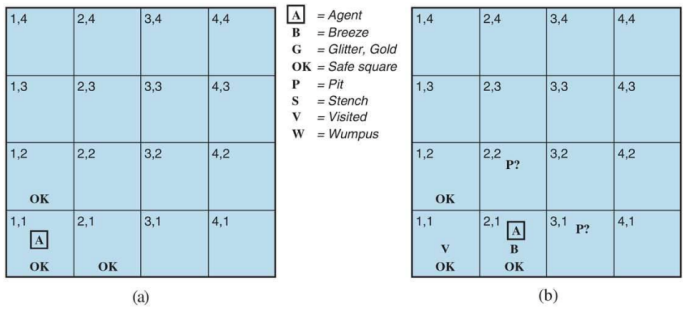
Figure 28: Typical Wumpus World

Environment characteristics:

- **Deterministic?:** Yes. Outcomes are exactly specified
- **Static?:** Yes. Wumpus and pits do not move
- **Discrete?:** Yes
- **Single agent?:** Yes, assuming wumpus as a natural phenomena
- **Fully observable?:** No, only local perception
- **Episodic?:** No, previous actions affect the current and next actions.

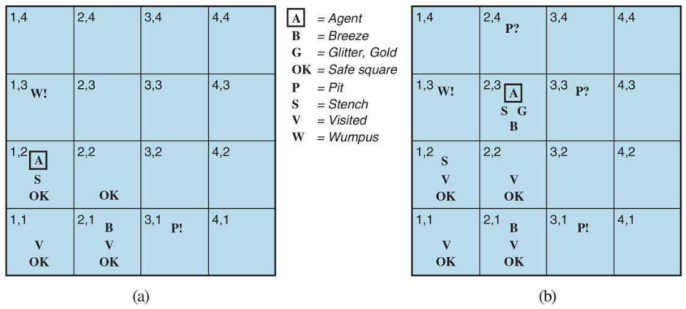
Let us watch a knowledge-based wumpus agent exploring the environment:

Figure 7.3



The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None,None,None,None,None*]. (b) After moving to [2,1] and perceiving [*None,Breeze,None,None,None*].

Figure 7.4



Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [*Stench,None,None,None,None*]. (b) After moving to [2,2] and then [2,3], and perceiving [*Stench,Breeze,Glitter,None,None*].

Figure 29: Exploring wumpus world

7.3 Logic

In Section 7.1 , we said that knowledge bases consist of sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: "$x + y = 4$" is a well-formed sentence, whereas "$x4y =$" is not.

A logic must also define the **semantics**, or meaning, of sentences. The semantics defines the **truth** of each sentence with respect to each **possible world**. For example, the semantics for arithmetic specifies that the sentence "$x+y = 4$" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no "in between." When we need to be precise, we use

the term **model** in place of "possible world." If a sentence a is true in model m, we say that m **satisfies** a or sometimes m **is a model of** a. We use the notation $M(a)$ to mean the set of all models of a.
Now that we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences, which is the idea that a sentence follows logically from another sentence.
Entailment: $<SentenceA> \models <SentenceB>$

- A entails B
- B follows from A
- B is true whenever A is true

7.4 Propositional Logic: A Very Simple Logic

We now present **propositional logic**. We describe its syntax (the structure of sentences) and its semantics (the way in which the truth of sentences is determined). From these, we derive a simple, syntactic algorithm for logical inference that implements the semantic notion of entailment. Everything takes place in the wumpus world.

7.4.1 Syntax

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false. There are two proposition symbols with fixed meanings: *True* is the always-true proposition and *False* is the always-false proposition. **Complex sentences** are constructed from simpler sentences, using parentheses and operators called **logical connectives.** There are five connectives in common use: $\neg, \wedge, \vee, \Leftrightarrow, \rightarrow$

Compound sentences: constructed from atomic and/or other compound sentences via connectives:

- If S is a sentence, $\neg S$ is a sentence (negation)
- If S_1 and S_2 are sentences, $S_1 \wedge S_2$ is a sentence (conjunction)
- If S_1 and S_2 are sentences, $S_1 \vee S_2$ is a sentence (disjunction)
- If S_1 and S_2 are sentences, $S_1 \Rightarrow S_2$ is a sentence (implication)
- If S_1 and S_2 are sentences, $S_1 \Leftrightarrow S_2$ is a sentence (biconditional)

7.4.2 Semantics

Semantics of atomic sentences are determined according to their truth values wrt interpretations. An **interpretation** maps symbols to one of the two values: True (T), or False (F), depending on whether the symbol is **satisfied** in the "world". Semantics of compositional sentences are determined using the standard rules of logic for connectives:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
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

Figure 30: Logic for connectives

7.4.3 A simple knowledge base

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We need the following symbols for each location:

- $P_{x,y}$ is true if there is a pit in $[x, y]$
- $W_{x,y}$ is true if there is a wumpus in $[x, y]$ dead or alive.
- $B_{x,y}$ is true if there is a breeze in $[x, y]$
- $S_{x,y}$ is true if there is a stench in $[x, y]$
- $L_{x,y}$ is true if the agent is in location $[x, y]$

We label each sentence so that we can refer to them:

- There is no pit in [1,1]:
$$R_1 : \neg P_{1,1} \tag{3}$$
- A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:
$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}) \tag{4}$$
$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1}) \tag{5}$$
- The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in.
$$R_4 : \neg B_{1,1} \tag{6}$$
$$R_5 : B_{2,1} \tag{7}$$

7.4.4 A simple inference procedure

Our goal now is to decide whether $KB \models \alpha$ for some sentence α. Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. We do this with the following truth table:

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

A truth table constructed for the knowledge base given in the text. KB is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

7.5 Propositional Theorem Proving

In this section, we show how entailment can be done by **theorem proving**—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models.

Before we plunge into the details of theorem-proving algorithms, we will need some additional concepts related to entailment. The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write $\alpha \equiv \beta$

The second concept we will need is **validity**. A sentence is valid if it is true in all models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as **tautologies**, they are *necessarily* true. Because the sentence *True* is true in all models, every valid sentence is logically equivalent to *True*. From our definition of entailment, we can derive the **deduction theorem**:

For any sentences α and β ($\alpha \models \beta$), if and only if the sentence ($\alpha \Rightarrow \beta$) is valid.

The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, some model. For example, the knowledge base given earlier, $(R_1 \vee R_2 \vee R_3 \vee R_4 \vee R_5)$ is satisfiable because there are three models in which it is true, as shown in the truth table.

7.5.1 Inference and proofs

This section covers **inference rules** that can be applied to derive a proof, a chain of conclusions that leads to the desired goal. The best-known rule is called **Modus Ponens** and is written:

$$\frac{\alpha \Rightarrow \beta, \alpha}{\alpha} \tag{8}$$

Another useful inference rule is **And-Elimination**, which says that, from a conjunction, any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha} \tag{9}$$

7.5.2 Proof by resolution

$$\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}$$

Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

Unit resolution inference rule:

$$\frac{l_1 \vee ... \vee l_k, m}{l_1 \vee ... \vee l_{i-1} \vee l_{i+1} \vee ... \vee l_k} \tag{10}$$

A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF**. We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \wedge P_{2,1})$ into CNF:

1. Eliminating $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$:

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}) \tag{11}$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}) \tag{12}$$

3. CNF requires $\neg$ to appear only in literals, so we "move $\neg$ inwards" by repeated application of the following equivalences $\neg(\neg\alpha) \equiv \alpha$ (double-negation elimination)

$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \tag{13}$$

$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \tag{14}$$

In the example, we require just one application of the last rule:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}) \tag{15}$$

4. Now we have a sentence containing nested $\wedge$ and $\vee$ operators applied to literals. We apply the distributivity law, distributing $\vee$ over $\wedge$ wherever possible.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}) \tag{16}$$

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

Based on the conversion we just showed, we can make a basic resolution algorithm:

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  while true do
    for each pair of clauses C_i, C_j in clauses do
      resolvents ← PL-RESOLVE(C_i, C_j)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

A simple resolution algorithm for propositional logic. PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

To conclude our discussion of resolution, we now show why PL-RESOLUTION is complete. To do this, we introduce the resolution closure RC(S) of a set of clauses , which is the set of all clauses derivable by repeated application of the resolution rule to clauses in or their derivatives. The resolution closure is what PL-RESOLUTION computes as the final value of the variable clauses.

The completeness theorem for resolution in propositional logic is called the **ground resolution** theorem: If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

7.5.3 Horn clauses and definite clauses

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm. One such restricted form is the **definite clause**, which is a disjunction of literals of which *exactly one is positive*. For example, the clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ is a definite clause.

Slightly more general is the **Horn clause**, which is a disjunction of literals of which at most one is positive. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called **goal clauses**. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause.

7.5.4 Forward and backward chaining

The forward-chaining algorithm PL-FC-ENTAILS? determines if a single proposition symbol q—the query—is entailed by a knowledge base of definite clauses.

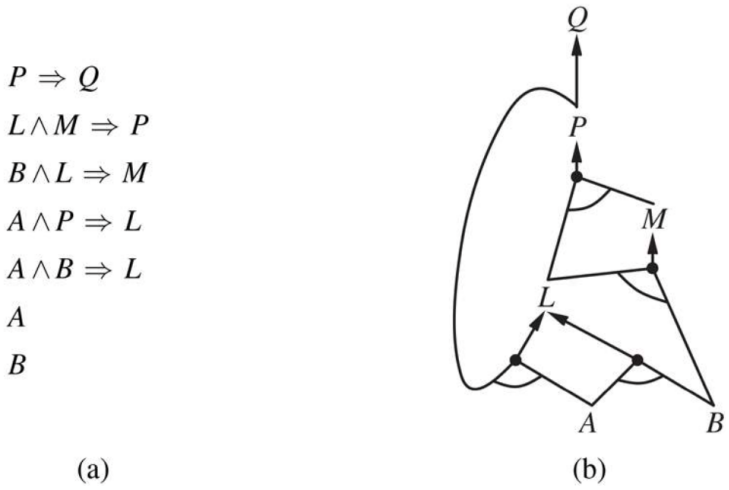
```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is initially the number of symbols in clause c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  queue ← a queue of symbols, initially symbols known to be true in KB

  while queue is not empty do
    p ← POP(queue)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to queue
  return false
```

The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are not yet proven. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

Forward chaining basic idea: Whenever the premises of a rule are satisfied, infer the conclusion.

Backward chaining basic idea: To prove the fact that appears in the conclusion of a rule prove the premises of the rule.

Forward chaining example:



(a) A set of Horn clauses. (b) The corresponding AND–OR graph.

8 First-Order Logic

8.1 Representation Revisited

Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation. Propositional logic has a third property that is desirable in representation languages, namely, compositionality. In a compositional language, the meaning of a sentence is a function of the meaning of its parts.

8.1.2 Combining the best of formal and natural languages

When we look at the syntax of natural language, the most obvious elements are nouns and noun phrases that refer to **objects** (squares, pits, wumpuses) and verbs and verb phrases along with adjectives and adverbs that refer to **relations** among objects (is breezy, is adjacent to, shoots). Some of these relations are **functions** which is relations in which there is only one "value" for a given "input." It is easy to start listing examples of objects, relations, and functions:

- Objects: people, houses, numbers, theories
- Relations: these can be unary relations or properties such as red, round, bogus, prime, multistoried ..., or more general n-ary relations such as brother of, bigger than, inside,
- Functions: father of, best friend, third inning of, one more than, beginning of ...

The language of first-order logic, whose syntax and semantics we define in the next section, is built around objects and relations.

The primary difference between propositional and first-order logic lies in the **ontological commitment** made by each language—that is, what it assumes about the nature of reality. Mathematically, this commitment is expressed through the nature of the formal models with respect to which the truth of sentences is defined. For example, propositional logic assumes that there are facts that either hold or do not hold in the world. Each fact can be in one of two states—true or false—and each model assigns true or false to each proposition symbol (see Section 7.4.2). First-order logic assumes more; namely, that the world consists of objects with certain relations among them that do or do not hold. The formal models are correspondingly more complicated than those for propositional logic.

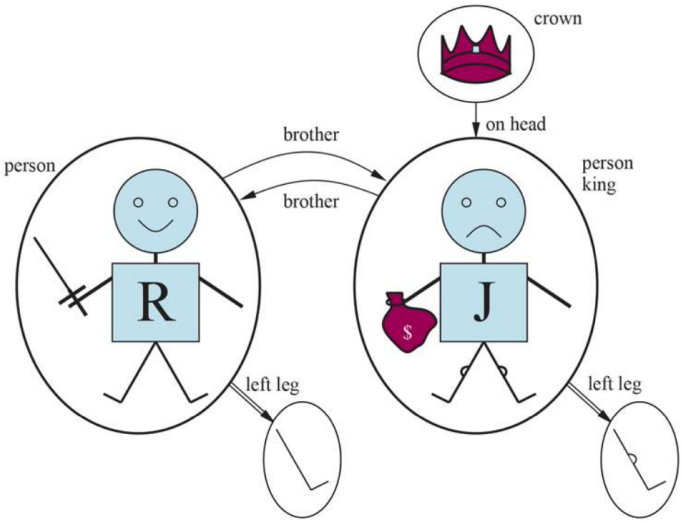
This ontological commitment is a great strength of logic (both propositional and first-order), because it allows us to start with true statements and infer other true statements. But in the real world, many propositions have vague boundaries: Is Vienna a large city? Does this restaurant serve delicious food? Is that person tall? It depends who you ask, and their answer might be "kind of." One response is **Fuzzy logic**, which makes the ontological commitment that propositions have a **degree of truth** between 0 and 1. For example, the sentence "Vienna is a large city" might be true to degree 0.8 in fuzzy logic, while "Paris is a large city" might be true to degree 0.9. This corresponds better to our intuitive conception of the world, but it makes it harder to do inference.

Various special-purpose logics make still further ontological commitments; for example, **temporal logic** assumes that facts hold at particular times and that those times (which may be points or intervals) are ordered. Thus, special-purpose logics give certain kinds of objects (and the axioms about them) "first class" status within the logic, rather than simply defining them within the knowledge base. **Higher-order logic** views the relations and functions referred to by first-order logic as objects in themselves. This allows one to make assertions about all relations. Unlike most special-purpose logics, higher-order logic is strictly more expressive than first-order logic, in the sense that some sentences of higher-order logic cannot be expressed by any finite number of first-order logic sentences.

8.2 Syntax and Semantics of First-Order Logic

8.2.1 Models for first-order logic

The **domain** of a model is the set of objects or **domain elements** it contains. The domain is required to be nonempty, which means, every possible world must contain at least one object. Mathematically speaking, it doesn't matter what these objects are. All that matters is how many there are in each particular model. For pedagogical purposes we'll use a concrete example.



A model containing five objects, two binary relations (brother and on-head), three unary relations (person, king, and crown), and one unary function (left-leg).

The figure shows a model with five objects: Richard the Lionheart, King of England from 1189 to 1199; his younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; and a crown. The objects in the model may be related in various ways. In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of **tuples** of objects that are related. Thus, the brotherhood relation in this model is the set (<Richard the lionheart, King John> , <King John, Richard the lionheart>). The crown is on the kings head so the "on head" relation contains just one tuple <The crown, King John>. "brother" and "on head" are binary relations. That is, they relate pairs of objects. The model also contains unary relations, or properties: the "person" property is true of both Richard and John; the "king" property is true only of John, and the "crown" property is true only of the crown.

8.2.2 Symbols and interpretations

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} & \rightarrow & (\textit{Sentence}) \\
& \mid & \neg \textit{Sentence} \\
& \mid & \textit{Sentence} \wedge \textit{Sentence} \\
& \mid & \textit{Sentence} \vee \textit{Sentence} \\
& \mid & \textit{Sentence} \Rightarrow \textit{Sentence} \\
& \mid & \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
& \mid & \textit{Quantifier Variable}, \ldots \textit{Sentence} \\
\\
\textit{Term} & \rightarrow & \textit{Function}(\textit{Term}, \ldots) \\
& \mid & \textit{Constant} \\
& \mid & \textit{Variable} \\
\\
\textit{Quantifier} & \rightarrow & \forall \mid \exists \\
\textit{Constant} & \rightarrow & A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\textit{Predicate} & \rightarrow & \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} & \rightarrow & \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\textsc{Operator Precedence} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: constant symbols, which stand for objects; predicate symbols, which stand for relations; and function symbols, which stand for functions. We adopt the convention that these symbols will begin with uppercase letters. For example, we might use the constant symbols Richard and John; the predicate symbols Brother, OnHead, Person, King, and Crown; and the function symbol LeftLeg. As with proposition symbols, the choice of names is entirely up to the user.

In addition to its objects, relations, and functions, each model includes an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.

8.2.3 Terms

A **term** is a logical expression that refers to an object. Constant symbols are terms, but it is not always convenient to have a distinct symbol to name every object. In English we might use the expression "King John's left leg" rather than giving a name to his leg. This is what function symbols are for: instead of using a constant symbol, we use LeftLeg(John).

In the general case, a complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol. It is important to remember that a complex term is just a complicated kind of name. It is not a "subroutine call" that "returns a value." There is no LeftLeg subroutine that takes a person as input and returns a leg. We can reason about left legs (e.g., stating the general rule that everyone has one and then deducing that John must have one) without ever providing a definition of LeftLeg. This is something that cannot be done with subroutines in programming languages.

8.2.4 Atomic sentences

Now that we have terms for referring to objects and predicate symbols for referring to relations, we can combine them to make **atomic sentences** that state facts. An atomic sentence (or **atom** for short) is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as $Brother(Richard, John)$. This states that Richard the Lionheart is the brother of King John. Atomic sentences can have complex terms as arguments. Thus,

$$Married(Father(Richard), Mother(John)) \tag{17}$$

states that Richard the Lionheart's father is married to King John's mother (again, under a suitable interpretation). An atomic sentence is **true** in a given model if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.

8.2.5 Complex sentences

We can use logical connectives to construct more **complex sentences**, with the same syntax and semantics as in propositional calculus. Here are four sentences that are true in the model of king John, Richard the lionheart and the crown, under our intended interpretation:

$$\begin{aligned} \neg Brother(LeftLeg(Richard), John) \\ Brother(Richard, John) \wedge Brother(John, Richard) \\ King(Richard) \vee King(Richard) \\ \neg King(Richard) \implies King(John) \end{aligned} \tag{18}$$

8.2.6 Quantifiers

Once we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. **Quantifiers** let us do this. First-order logic contains two standard quantifiers, called *universal* and *existential*.

Universal quantification ($\forall$)

"All kings are persons," is written in first-order logic as $\forall x\ King(x) \Rightarrow Person(x)$. The symbol x is called a **variable**. By convention, variables are lowercase letters. A variable is a term all by itself, and as such can also serve as the argument of a function. For example, $LeftLeg(x)$. A term with no variables is called a **ground term**.

Existential quantification ($\exists$)

Universal quantification makes statements about every object. Similarly, we can make a statement about some object without naming it, by using an **existential quantifier**. To say, for example, that King John has a crown on his head, we write

$$\exists x\ Crown(x) \wedge OnHead(x, John) \tag{19}$$

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. For example, "Brothers are siblings" can be written as

$$\forall x\ \forall y\ Brothers(x, y) \Rightarrow Siblings(x, y) \tag{20}$$

Connections between $\forall$ and $\exists$

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa:

$$\forall x \neg Likes(x, Parsnips) \tag{21}$$

is equivalent to

$$\neg \exists x\ Likes(x, Parsnips) \tag{22}$$

8.2.7 Equality

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms as described earlier. We can use the equality symbol to signify that two terms refer to the same object. For example,

$$Father(John) = Henry \tag{23}$$

says that the object referred to by Father(John) and the object referred to by Henry are the same.

8.2.8 Database semantics

Suppose that we believe that Richard has two brothers, John and Geoffrey.7 We could write

$$Brother(John, Richard) \wedge Brother(Geoffrey, Richard) \tag{24}$$

but that wouldn't completely capture the state of affairs. First, this assertion is true in a model where Richard has only one brother. We need to add $John \neq Geoffrey$. Second, the sentence doesn't rule out models in which Richard has many more brothers besides John and Geoffrey. Thus, the correct translation of "Richard's brothers are John and Geoffrey" is as follows:

$$\begin{aligned} &Brother(John, Richard) \wedge Brother(Geoffrey, Richard) \wedge John \neq \\ &Geoffrey \wedge \forall x\, Brother(x, Richard) \implies (x = John \vee x = Geoffrey) \end{aligned} \tag{25}$$

This logical sentence seems much more cumbersome than the corresponding English sentence. But if we fail to translate the English properly, our logical reasoning system will make mistakes. Can we devise a semantics that allows a more straightforward logical sentence?

One proposal that is very popular in database systems works as follows. First, we insist that every constant symbol refer to a distinct object, the **unique-names assumption**. Second, we assume that atomic sentences not known to be true are in fact false, the **closed-world assumption**. Finally, we invoke **domain closure**, meaning that each model contains no more domain elements than those named by the constant symbols.

Under the resulting semantics, Equation (24) does indeed state that Richard has exactly two brothers, John and Geoffrey. We call this **database semantics** to distinguish it from the standard semantics of first-order logic.

8.3 Using First-Order Logic

Now that we have defined an expressive logical language, let's learn how to use it. In this section, we provide example sentences in some simple **domains**. In knowledge representation, a domain is just some part of the world about which we wish to express some knowledge.

8.3.1 Assertions and queries in first-order logic

Sentences are added to a knowledge base using $Tell$, exactly as in propositional logic. Such sentences are called **assertions**. For example, we can assert that John is a king, Richard is a person, and all kings are persons:

$$\begin{aligned} Tell(KB, King(John)) \\ Tell(KB, Person(Richard)) \\ Tell(KB, \forall x\ King(x) \Rightarrow Person(x)) \end{aligned} \tag{26}$$

We can ask questions of the knowledge base using Ask. For example,

$$Ask(KB, King(John)) \tag{27}$$

returns true. Questions asked with Ask are called **queries** or **goals**. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively.

If we want to know what value of x makes the sentence true, we will need a different function, which we call $AskVars$,

$$AskVars(KB, Person(x)) \tag{28}$$

which yields a stream of answers. In this case there will be two answers: $(x/John)$ and $(x/Richard)$. Such an answer is called a **substitution** or **binding list**.

8.3.2 The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent." Clearly, the objects in our domain are people. Unary predicates include Male and Female, among others. Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother, Sister, Child, Daughter, Son, Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt, and Uncle. We use functions for Mother and Father, because every person has exactly one of each of these.

We can go through each function and predicate, writing down what we know in terms of the other symbols. For example, one's mother is one's parent who is female:

$$\forall m, c\,(Mother(c) = m \iff Female(m) \wedge Parent(m, c)).$$

One's husband is one's male spouse:

$$\forall w, h\,(Husband(h, w) \iff Male(h) \wedge Spouse(h, w)).$$

Parent and child are inverse relations:

$$\forall p, c\,(Parent(p, c) \iff Child(c, p)).$$

A grandparent is a parent of one's parent:

$$\forall g, c\,(Grandparent(g, c) \iff \exists p\,(Parent(g, p) \wedge Parent(p, c))).$$

A sibling is another child of one's parent:

$$\forall x, y \, (Sibling(x, y) \iff x \neq y \wedge \exists p \, (Parent(p, x) \wedge Parent(p, y))).$$

Each of these sentences can be viewed as an **axiom** of the kinship domain.

Not all logical sentences about a domain are axioms. Some are **theorems**, that is, they are entailed by the axioms. For example, consider the assertion that siblinghood is symmetric:

$$\forall x, y \; Sibling(x, y) \Leftrightarrow Sibling(y, x) \tag{29}$$

Is this an axiom or a theorem? In fact, it is a theorem that follows logically from the axiom that defines siblinghood. If we *Ask* the knowledge base this sentence, it should return true.

8.3.3 Numbers, sets, and lists

Numbers are perhaps the most vivid example of how a large theory can be built up from a tiny kernel of axioms. We describe here the theory of natural numbers or nonnegative integers. We need a predicate $NatNum$ that will be true of natural numbers; we need one constant symbol, 0; and we need one function symbol, (successor). The **Peano axioms** define natural numbers and addition.8 Natural numbers are defined recursively:

$$\begin{aligned} NatNum(0) \\ \forall n \; NatNum(n) \Rightarrow NatNum(S(n)) \end{aligned} \tag{30}$$

That is, 0 is a natural number, and for every object n, if n is a natural number, then $S(n)$ is a natural number. So the natural numbers are $0, S(0), S(S(0))$, and so on. We also need axioms to constrain the successor function:

$$\forall n \, 0 \neq S(n).$$
$$\forall m, n \, m \neq n \implies S(m) \neq S(n).$$

Now we can define addition in terms of the successor function:

$$\forall m \, NatNum(m) \implies +(0, m) = m.$$
$$\forall m, n \, NatNum(m) \wedge NatNum(n) \implies +(S(m), n) = S(+(m, n)).$$

The first of these axioms says that adding 0 to any natural number m gives m itself. Notice the use of the binary function symbol + in the term $+(m, 0)$; in ordinary mathematics, the term would be written $m + 0$ using infix notation. (The notation we have used for first-order logic is called **prefix**.) To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write $S(n)$ as $n + 1$, so the second axiom becomes

$$\forall m, n\, NatNum(m) \wedge NatNum(n) \implies (m+1)+n = (m+n)+1.$$

The domain of **sets** is also fundamental to mathematics as well as to commonsense reasoning. (In fact, it is possible to define number theory in terms of set theory.) We want to be able to represent individual sets, including the empty set. We need a way to build up sets from elements or from operations on other sets. We will want to know whether an element is a member of a set and we will want to distinguish sets from objects that are not sets. One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adding something to a set:

$$\forall s\, Set(s) \iff (s = \{\}) \vee (\exists x, s_2\, Set(s_2) \wedge s = Add(x, s_2)).$$

2. The empty set has no elements added into it. In other words, there is no way to decompose $\{\}$ into a smaller set and an element:

$$\neg\exists x, s\, Add(x, s) = \{\}.$$

3. Adding an element already in the set has no effect:

$$\forall x, s\, x \in s \iff s = Add(x, s).$$

4. The only members of a set are the elements that were added into it. We express this recursively, saying that x is a member of s if and only if s is equal to some element y added to some set s_2, where either y is the same as x or x is a member of s_2:

$$\forall x, s\, x \in s \iff \exists y, s_2\, (s = Add(y, s_2)) \wedge (x = y \vee x \in s_2).$$

5. A set is a subset of another set if and only if all of the first set's members are members of the second set:

$$\forall s_1, s_2\, s_1 \subseteq s_2 \iff (\forall x\, x \in s_1 \implies x \in s_2).$$

6. Two sets are equal if and only if each is a subset of the other:

$$\forall s_1, s_2\, (s_1 = s_2) \iff (s_1 \subseteq s_2 \wedge s_2 \subseteq s_1).$$

7. An object is in the intersection of two sets if and only if it is a member of both sets:

$$\forall x, s_1, s_2\, x \in (s_1 \cap s_2) \iff (x \in s_1 \wedge x \in s_2).$$

8. An object is in the union of two sets if and only if it is a member of either set:

$$\forall x, s_1, s_2\, x \in (s_1 \cup s_2) \iff (x \in s_1 \vee x \in s_2).$$

Lists are similar to sets. The differences are that lists are ordered and the same element can List appear more than once in a list.

8.3.4 The wumpus world

Some propositional logic axioms for the wumpus world were given in Chapter 7. The first-order axioms in this section are much more concise, capturing in a natural way exactly what we want to say.

Recall that the wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what. We use integers for time steps. A typical percept sentence would be

$$Percept([Stench, Breeze, Glitter, None, None], 5) \tag{31}$$

Here, Percept is a binary predicate, and Stench and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

$$Turn(Right),\ Turn(Left),\ Forward,\ Shoot,\ Grab,\ Climb \tag{32}$$

To determine which is best, the agent program executes the query

$$AskVars(KB, BestAction(a, 5)) \tag{33}$$

which returns a binding list such as $[a/Grab]$ The agent program can then return as the action to take. Simple "reflex" behavior can also be implemented by quantified implication sentences. For example, we have

$$\forall t\ Glitter(t) \Rightarrow BestAction(Grab, t) \tag{34}$$

Adjacency of any two squares can be defined as:

$$\begin{aligned} &\forall x, y, a, b\ Adjacent([x, y], [a, b]) \\ \iff\ &(x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1)). \end{aligned} \tag{35}$$

8.4 Knowledge Engineering in First-Order Logic

The preceding section illustrated the use of first-order logic to represent knowledge in three simple domains. This section describes the general process of knowledge-base construction. a process called **knowledge engineering**. A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain.

8.4.1 The knowledge engineering process

Knowledge engineering projects vary widely in content, scope, and difficulty, but all such projects include the following steps:

1. **IDENTIFY THE QUESTIONS**. The knowledge engineer must delineate the range of questions that the knowledge base will support and the kinds of facts that will be available for each specific problem instance. For example, does the wumpus knowledge base need to be able to choose actions, or is it required only to answer questions about the contents of the environment? Will the sensor facts include the current location? The task will determine what knowledge must be represented in order to connect problem instances to answers.

2. **ASSEMBLE THE RELEVANT KNOWLEDGE.** The knowledge engineer might already be an expert in the domain, or might need to work with real experts to extract what they know, a process called knowledge acquisition. At this stage, the knowledge is not represented formally. The idea is to understand the scope of the knowledge base, as determined by the task, and to understand how the domain actually works. For the wumpus world, which is defined by an artificial set of rules, the relevant knowledge is easy to identify. (Notice, however, that the definition of adjacency was not supplied explicitly in the wumpus-world rules.)

3. **DECIDE ON A VOCABULARY OF PREDICATES, FUNCTIONS, AND CONSTANTS.** That is, translate the important domain-level concepts into logic-level names. This involves many questions of knowledge-engineering style. Like programming style, this can have a significant impact on the eventual success of the project. For example, should pits be represented by objects or by a unary predicate on squares? Should the agent's orientation be a function or a predicate? Should the wumpus's location depend on time? Once the choices have been made, the result is a vocabulary that is known as the ontology of the domain. The word ontology means a particular theory of the nature of being or existence. The ontology determines what kinds of things exist, but does not determine their specific properties and interrelationships.

4. **ENCODE GENERAL KNOWLEDGE ABOUT THE DOMAIN.** The knowledge engineer writes down the axioms for all the vocabulary terms. This pins down (to the extent possible) the meaning of the terms, enabling the expert to check the content. Often, this step reveals misconceptions or gaps in the vocabulary that must be fixed by returning to step 3 and iterating through the process.

5. **ENCODE A DESCRIPTION OF THE PROBLEM INSTANCE.** If the ontology is well thought out, this step is easy. It involves writing simple atomic sentences about instances of concepts that are already part of the ontology. For a logical agent, problem instances are supplied by the sensors, whereas a "disembodied" knowledge base is given sentences in the same way that traditional programs are given input data.

6. **POSE QUERIES TO THE INFERENCE PROCEDURE AND GET ANSWERS**. This is where the reward is: we can let the inference

procedure operate on the axioms and problem-specific facts to derive the facts we are interested in knowing. Thus, we avoid the need for writing an application-specific solution algorithm.

7. **DEBUG AND EVALUATE THE KNOWLEDGE BASE.** Alas, the answers to queries will seldom be correct on the first try. More precisely, the answers will be correct for the knowledge base as written, assuming that the inference procedure is sound, but they will not be the ones that the user is expecting. For example, if an axiom is missing, some queries will not be answerable from the knowledge base. A considerable debugging process could ensue.

8.4.2 The electronic circuits domain

Triviellt fra Krets og Datdig

9 Inference in First-Order Logic

In this chapter, we describe algorithms that can answer any answerable first-order logic question.

9.1 Propositional vs. First-Order Inference

One way to do first-order inference is to convert the first-order knowledge base to propositional logic and use propositional inference, which we already know how to do. A first step is eliminating universal quantifiers. For example, suppose our knowledge base contains the standard folkloric axiom that all greedy kings are evil:

$$\forall x \ (\text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x)) \,.$$

From that we can infer any of the following sentences:

$$\text{King}(\mathit{John}) \wedge \text{Greedy}(\mathit{John}) \Rightarrow \text{Evil}(\mathit{John})$$
$$\text{King}(\mathit{Richard}) \wedge \text{Greedy}(\mathit{Richard}) \Rightarrow \text{Evil}(\mathit{Richard})$$
$$\text{King}(\text{Father}(\mathit{John})) \wedge \text{Greedy}(\text{Father}(\mathit{John})) \Rightarrow \text{Evil}(\text{Father}(\mathit{John})).$$
$$\vdots$$

In general, the rule of **Universal Instantiation** (UI for short) says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for a universally quantified variable. To write out the inference rule formally, we use the notion of *substitutions* introduced in Section 8.3. Let $\text{SUBST}(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written

$$\frac{\forall v\, \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

Similarly, the rule of **Existential Instantiation** replaces an existentially quantified variable with a single *new constant symbol*. The formal statement is as follows: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the knowledge base,

$$\frac{\exists v\, \alpha}{\text{SUBST}(\{v/k\}, \alpha)}.$$

Basically, the existential sentence says there is some object satisfying a condition, and applying the existential instantiation rule just gives a name to that object. In logic, the new name is called a **Skolem constant**. Whereas Universal Instantiation can be applied many times to the same axiom to produce many different consequences, Existential Instantiation need only be applied once, and then the existentially quantified sentence can be discarded.

9.1.1 Reduction to propositional inference

We now show how to convert any first-order knowledge base into a propositional knowledge base. The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of all possible instantiations. For example, suppose our knowledge base contains just the sentences

$$\forall x\ (\text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x))$$

$$\text{King}(John)$$

$$\text{Greedy}(John)$$

$$\text{Brother}(Richard, John).$$

and that the only objects are *John* and *Richard*. We apply UI to the first sentence using all possible substitutions, $\{x/John\}$ and $\{x/Richard\}$. We obtain

$$\text{King}(John) \wedge \text{Greedy}(John) \Rightarrow \text{Evil}(John)$$

$$\text{King}(Richard) \wedge \text{Greedy}(Richard) \Rightarrow \text{Evil}(Richard).$$

Next replace ground atomic sentences, such as $\text{King}(John)$, with proposition symbols, such as $JohnIsKing$. Finally, apply any of the complete propositional algorithms to obtain conclusions such as $JohnIsEvil$, which is equivalent to $\text{Evil}(John)$.

9.2 Unification and First-Order Inference

Generalized Modus Ponens: For atomic sentences p_i, p'_i, and q, where there is a substitution θ such that $\text{SUBST}(\theta, p'_i) = \text{SUBST}(\theta, p_i)$ for all i,

$$\frac{p'_1, p'_2, \ldots, p'_n, \ (p_1 \wedge p_2 \wedge \cdots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}.$$

There are $n+1$ premises to this rule: the n atomic sentences p'_i and the one implication. The conclusion is the result of applying the substitution θ to the consequent q. For our example:

$$p'_1 \text{ is } \text{King}(John) \quad p_1 \text{ is } \text{King}(x)$$
$$p'_2 \text{ is } \text{Greedy}(y) \quad p_2 \text{ is } \text{Greedy}(x)$$
$$\theta \text{ is } \{x/John, y/John\} \quad q \text{ is } \text{Evil}(x)$$
$$\text{SUBST}(\theta, q) \text{ is } \text{Evil}(John).$$

Generalized Modus Ponens is a **lifted** version of Modus Ponens—it raises Modus Ponens from ground (variable-free) propositional logic to first-order logic. The key advantage of lifted inference rules over propositionalization is that they make only those substitutions that are required to allow particular inferences to proceed.

9.2.1 Unification

Lifted inference rules require finding substitutions that make different logical expressions look identical. This process is called **unification** and is a key component of all first-order inference algorithms. The UNIFY algorithm takes two sentences and returns a **unifier** for them (a substitution) if one exists:

$$\text{Unify}(p, q) = \theta \text{ where } \text{SUBST}(\theta, p) = \text{SUBST}(\theta, q).$$

Let us look at some examples of how UNIFY should behave. Suppose we have a query AskVars(Knows($John, x$)): whom does John know? Answers to this query can be found by finding all sentences in the knowledge base that unify with Knows($John, x$). Here are the results of unification with four different sentences that might be in the knowledge base:

$$\text{Unify}(\text{Knows}(John, x), \text{Knows}(John, Jane)) = \{x/Jane\}$$
$$\text{Unify}(\text{Knows}(John, x), \text{Knows}(y, Bill)) = \{x/Bill, y/John\}$$
$$\text{Unify}(\text{Knows}(John, x), \text{Knows}(y, \text{Mother}(y))) = \{y/John, x/\text{Mother}(John)\}$$
$$\text{Unify}(\text{Knows}(John, x), \text{Knows}(x, Elizabeth)) = \textit{failure}.$$

The last unification fails because x cannot take on the values *John* and *Elizabeth* at the same time. Now, remember that Knows($x, Elizabeth$) means "Everyone knows Elizabeth," so we *should* be able to infer that John knows

Elizabeth. The problem arises only because the two sentences happen to use the same variable name, x. The problem can be avoided by **standardizing apart** one of the two sentences being unified, which means renaming its variables to avoid name clashes. For example, we can rename x in $\text{Knows}(x, Elizabeth)$ to x_{17} (a new variable name) without changing its meaning. Now the unification will work:

$$\text{Unify}(\text{Knows}(John, x), \text{Knows}(x_{17}, Elizabeth)) = \{x/Elizabeth, x_{17}/John\}.$$

Every unifiable pair of expressions has a single **most general unifier (MGU)** that is unique up to renaming and substitution of variables. For example, $\{x/John\}$ and $\{y/John\}$ are considered equivalent, as are $\{x/John, y/John\}$ and $\{x/John, y/x\}$.

9.2.2 Storage and retrieval

Underlying the TELL, ASK, and ASKVARS functions used to inform and interrogate a knowledge base are the more primitive STORE and FETCH functions. STORE(s) stores a sentence s into the knowledge base and FETCH(q) returns all unifiers such that the query q unifies with some sentence in the knowledge base. The problem we used to illustrate unification—finding all facts that unify with $\text{Knows}(John, x)$—is an instance of FETCHING.

The simplest way to implement STORE and FETCH is to keep all the facts in one long list and unify each query against every element of the list. Such a process is inefficient, but it works. The remainder of this section outlines ways to make retrieval more efficient.

We can make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. For example, there is no point in trying to unify $\text{Knows}(John, x)$ with $\text{Brother}(Richard, John)$. We can avoid such unifications by **indexing** the facts in the knowledge base. A simple scheme called **predicate indexing** puts all the Knows facts in one bucket and all the Brother facts in another. The buckets can be stored in a hash table for efficient access.

9.3 Forward Chaining

9.3.1 First-order definite clauses

First-order definite clauses are disjunctions of literals of which *exactly one* is positive. That means a definite clause is either atomic, or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. Existential quantifiers are not allowed, and universal quantifiers are left implicit: if you see an x in a definite clause, that means there is an implicit $\forall x$ quantifier. A typical first-order definite clause looks like this:

$$\text{King}(x) \wedge \text{Greedy}(x) \Rightarrow \text{Evil}(x),$$

but the literals King($John$) and Greedy(y) also count as definite clauses. First-order literals can include variables, so Greedy(y) is interpreted as "everyone is greedy" (the universal quantifier is implicit).

Let us put definite clauses to work in representing the following problem:

> The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

First, we will represent these facts as first-order definite clauses:
". . . it is a crime for an American to sell weapons to hostile nations":

$$\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x).$$

"Nono . . . has some missiles." The sentence $\exists x\, \text{Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$ is transformed into two definite clauses by Existential Instantiation, introducing a new constant M_1:

$$\text{Owns}(\text{Nono}, M_1)$$

$$\text{Missile}(M_1).$$

"All of its missiles were sold to it by Colonel West":

$$\text{Missile}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono}).$$

We will also need to know that missiles are weapons:

$$\text{Missile}(x) \Rightarrow \text{Weapon}(x)$$

and we must know that an enemy of America counts as "hostile":

$$\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x).$$

"West, who is American . . .":

$$\text{American}(\text{West}).$$

"The country Nono, an enemy of America . . .":

$$\text{Enemy}(\text{Nono}, \text{America}).$$

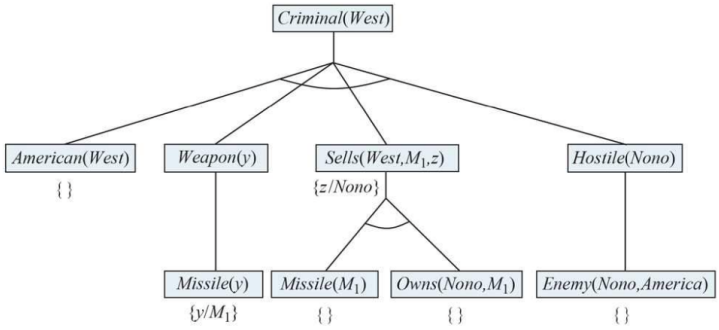
This knowledge base happens to be a **Datalog** knowledge base: Datalog is a language consisting of first-order definite clauses with no function symbols. Datalog gets its name because it can represent the type of statements typically made in relational databases. The absence of function symbols makes inference much easier.

9.4 Backward Chaining

The second major family of logical inference algorithms uses backward chaining over definite clauses. These algorithms work backward from the goal, chaining through rules to find known facts that support the proof.

9.4.1 A backward-chaining algorithm

Backward chaining is a kind of AND/OR search—the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in the *lhs* of a clause must be proved.



Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove *Criminal(West)*, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding subgoal. Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL-BC-ASK gets to the last conjunct, originally $Hostile(z)$, z is already bound to *Nono*.

Backward chaining, as we have written it, is clearly a depth-first search algorithm. This means that its space requirements are linear in the size of the proof. It also means that backward chaining (unlike forward chaining) suffers from problems with repeated states and incompleteness. Despite these limitations, backward chaining has proven to be popular and effective in logic programming languages.

9.5 Resolution

The last of our three families of logical systems, and the only one that works for any knowledge base, not just definite clauses, is **resolution**.

9.5.1 Conjunctive normal form for first-order logic

The first step is to convert sentences to conjunctive normal form (CNF)—that is, a conjunction of clauses, where each clause is a disjunction of literals. In CNF, literals can contain variables, which are assumed to be universally quantified. For example, the sentence

$$\forall x, y, z \ (\text{American}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x))$$

becomes, in CNF,

$$\neg\text{American}(x) \vee \neg\text{Weapon}(y) \vee \neg\text{Sells}(x, y, z) \vee \neg\text{Hostile}(z) \vee \text{Criminal}(x).$$

The key is that *Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.*

The procedure for conversion to CNF is similar to the propositional case. The principal difference arises from the need to eliminate existential quantifiers. We illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

$$\forall x \, [\forall y \, \text{Animal}(y) \Rightarrow \text{Loves}(x, y)] \Rightarrow \exists y \, \text{Loves}(y, x).$$

The steps are as follows:

- **ELIMINATE IMPLICATIONS**: Replace $P \Rightarrow Q$ with $\neg P \vee Q$. For our sample sentence, this needs to be done twice:

$$\forall x \, [\neg\forall y \, \text{Animal}(y) \vee \text{Loves}(x, y)] \vee \exists y \, \text{Loves}(y, x).$$

$$\forall x \, [\forall y \, \neg\text{Animal}(y) \vee \text{Loves}(x, y)] \vee \exists y \, \text{Loves}(y, x).$$

- **MOVE $\neg$ INWARDS**: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have:

$$\neg\forall x \, p \text{ becomes } \exists x \, \neg p$$

$$\neg\exists x \, p \text{ becomes } \forall x \, \neg p.$$

  Our sentence goes through the following transformations:

$$\forall x \, [\exists y \ (\neg \, (\neg\text{Animal}(y) \vee \text{Loves}(x, y)))] \vee \exists y \, \text{Loves}(y, x).$$

$$\forall x \, [\exists y \ (\neg\neg\text{Animal}(y) \wedge \neg\text{Loves}(x, y))] \vee \exists y \, \text{Loves}(y, x).$$

$$\forall x \, [\exists y \ (\text{Animal}(y) \wedge \neg\text{Loves}(x, y))] \vee \exists y \, \text{Loves}(y, x).$$

  Notice how a universal quantifier ($\forall y$) in the premise of the implication has become an existential quantifier. The sentence now reads "Either there is some animal that x doesn't love, or (if this is not the case) someone loves x." Clearly, the meaning of the original sentence has been preserved.

- **STANDARDIZE VARIABLES**: For sentences like $(\exists x \, P(x)) \vee (\exists x \, Q(x))$ that use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have:

$$\forall x \, [\exists y \, \text{Animal}(y) \wedge \neg\text{Loves}(x, y)] \vee \exists z \, \text{Loves}(z, x).$$

- **SKOLEMIZE**: Skolemization is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule of Section 9.1: translate $\exists x\, P(x)$ into $P(A)$, where A is a new constant. However, we can't apply Existential Instantiation to our sentence above because it doesn't match the pattern $\exists x\, \alpha$; only parts of the sentence match the pattern. If we blindly apply the rule to the two matching parts we get:

  $$\forall x\, [\text{Animal}(A) \wedge \neg\text{Loves}(x, A)] \vee \text{Loves}(B, x),$$

  which has the wrong meaning entirely: it says that everyone either fails to love a particular animal A or is loved by some particular entity B. In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person. Thus, we want the Skolem entities to depend on x:

  $$\forall x\, [\text{Animal}(F(x)) \wedge \neg\text{Loves}(x, F(x))] \vee \text{Loves}(G(x), x).$$

  Here F and G are *Skolem functions*. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears. As with Existential Instantiation, the Skolemized sentence is satisfiable exactly when the original sentence is satisfiable.

- **DROP UNIVERSAL QUANTIFIERS**: At this point, all remaining variables must be universally quantified. Therefore, we don't lose any information if we drop the quantifier:

  $$[\text{Animal}(F(x)) \wedge \neg\text{Loves}(x, F(x))] \vee \text{Loves}(G(x), x).$$

- **DISTRIBUTE $\vee$ OVER $\wedge$**:

  $$[\text{Animal}(F(x)) \vee \text{Loves}(G(x), x)] \wedge [\neg\text{Loves}(x, F(x)) \vee \text{Loves}(G(x), x)]\,.$$

  This step may also require flattening out nested conjunctions and disjunctions.

9.5.2 The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule. Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one *unifies* with the negation of the other. Thus, we have

$$\frac{\ell_1 \vee \cdots \vee \ell_k,\ m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. For example, we can resolve the two clauses

$$\text{Animal}(F(x)) \vee \text{Loves}(G(x), x) \quad \text{and} \quad \neg\text{Loves}(u, v) \vee \neg\text{Kills}(u, v)$$

by eliminating the complementary literals $\text{Loves}(G(x), x)$ and $\neg\text{Loves}(u, v)$, with the unifier $\theta = \{u/G(x), v/x\}$, to produce the *resolvent* clause

$$\text{Animal}(F(x)) \vee \neg\text{Kills}(G(x), x).$$

This rule is called the *binary resolution* rule because it resolves exactly two literals. The binary resolution rule by itself does not yield a complete inference procedure. The full resolution rule resolves subsets of literals in each clause that are unifiable. An alternative approach is to extend *factoring*—the removal of redundant literals—to the first-order case. Propositional factoring reduces two literals to one if they are *identical*; first-order factoring reduces two literals to one if they are *unifiable*. The unifier must be applied to the entire clause. The combination of binary resolution and factoring is complete.

Summary of Logic for Quick Look-Up in Problem Solving:

Propositional Logic Summary

Key Concepts

Syntax

- **Symbols:**
  - **Constants:** True, False
  - **Proposition symbols:** $P, Q, R, \ldots$
  - **Connectives:** $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
- **Sentences:**
  - **Atomic:** Single propositions (e.g., P, Q).
  - **Compound:** Built using connectives (e.g., $\neg P \wedge Q$).

Semantics

- Truth values assigned to propositions: True or False.
- The truth value of compound sentences is determined by standard truth tables for connectives:

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$
False	False	True	False	False	True
False	True	True	False	True	True
True	False	False	False	True	False
True	True	False	True	True	True

Inference Rules

Rules for deriving conclusions from premises:

1. **Modus Ponens:**
$$\frac{\alpha \Rightarrow \beta, \alpha}{\beta}$$
2. **And-Elimination:**
$$\frac{\alpha \wedge \beta}{\alpha} \quad \text{or} \quad \frac{\alpha \wedge \beta}{\beta}$$
3. **Modus Tollens:**
$$\frac{\alpha \Rightarrow \beta, \neg\beta}{\neg\alpha}$$
4. **Disjunctive Syllogism:**
$$\frac{\alpha \vee \beta, \neg\alpha}{\beta}$$
5. **And-Introduction:**
$$\frac{\alpha, \beta}{\alpha \wedge \beta}$$
6. **Or-Introduction:**
$$\frac{\alpha}{\alpha \vee \beta}$$
7. **Hypothetical Syllogism:**
$$\frac{\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\alpha \Rightarrow \gamma}$$
8. **Resolution Rule:**
$$\frac{\alpha \vee \beta, \neg\alpha \vee \gamma}{\beta \vee \gamma}$$

Proof Techniques

- **Entailment:** $KB \models \alpha$ means α is true in all models where KB is true.
- **Model Checking:**
  - Build a truth table for all possible models of KB.
  - Verify if α is true in all models where KB is true.
- **Proof by Contradiction:**
  - To prove $KB \models \alpha$, show that $KB \wedge \neg\alpha$ is unsatisfiable.
- **Resolution:**
  - Convert KB and $\neg\alpha$ to **Conjunctive Normal Form (CNF)**.
  - Use the Resolution Rule repeatedly to derive a contradiction.

Conversion to CNF

1. Eliminate $\Leftrightarrow$: Replace $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
2. Eliminate $\Rightarrow$: Replace $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
3. Move $\neg$ inward using De Morgan's Laws:

$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta), \quad \neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta)$$

4. Distribute $\vee$ over $\wedge$:

$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$$

Algorithms for Inference

- **Forward Chaining:**
  - Start with known facts in KB.
  - Apply inference rules to derive all possible conclusions until the query is proven.
- **Backward Chaining:**
  - Start with the query.
  - Work backward to check if the query can be derived from the KB.

Satisfiability and Validity

Satisfiability:

- A propositional formula is **satisfiable** if there exists at least one assignment of truth values to its variables that makes the formula true.
- Example: The formula $P \vee \neg Q$ is satisfiable because it can be true when:
  - $P = \text{True}, Q = \text{True}$, or
  - $P = \text{True}, Q = \text{False}$, or
  - $P = \text{False}, Q = \text{False}$.
- **Unsatisfiable formulas:** A formula is unsatisfiable if no truth assignment makes it true.
- Example: $P \wedge \neg P$ is unsatisfiable because P cannot be both true and false at the same time.

Validity:

- A propositional formula is **valid** if it is true for all possible truth assignments.

- Example: $P \vee \neg P$ is valid because it is always true, regardless of whether P is true or false (this is an instance of the *Law of the Excluded Middle*).
- Valid formulas are sometimes called **tautologies**.

Relationship Between Validity and Satisfiability:

- A formula is valid if and only if its negation is unsatisfiable.
- Example: $P \vee \neg P$ is valid, and $\neg(P \vee \neg P)$ (equivalent to $P \wedge \neg P$) is unsatisfiable.
- A formula is unsatisfiable if and only if its negation is valid.

First-Order Logic (FOL)

First-Order Logic (FOL) is more expressive than Propositional Logic, incorporating objects, relations, and quantifiers to represent knowledge.

Syntax of FOL

- **Constants:** Represent objects in the domain (e.g., *John*, *NTNU*).
- **Predicates:** Represent relationships or properties (e.g., *Brother(John, Mike)*, *At(x, NTNU)*).
- **Functions:** Map objects to other objects (e.g., *FatherOf(x)*).
- **Variables:** Represent arbitrary objects (e.g., x, y).
- **Connectives:** $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$.
- **Quantifiers:**
  - Universal Quantifier ($\forall$): $\forall x\, P(x)$ (e.g., "All x satisfy $P(x)$").
  - Existential Quantifier ($\exists$): $\exists x\, P(x)$ (e.g., "Some x satisfies $P(x)$").

Semantics of FOL

- **Domains:** Set of all objects under consideration.
- **Interpretation:** Assigns meaning to constants, predicates, and functions.
- **Models:** A model satisfies a sentence if it evaluates to true in the given interpretation.

Key Concepts

- **Atomic Sentences:** $Predicate(term_1, ..., term_n)$ (e.g., *Loves(John, Mary)*).
- **Complex Sentences:** Formed using connectives and quantifiers.
- **Equality:** $x = y$ asserts x and y refer to the same object.
- **Functions vs. Relations:**
  - Functions map objects (e.g., *FatherOf(John)*).
  - Relations describe connections between objects (e.g., *Brother(John, Mike)*).

Inference in FOL

- **Reduction to Propositional Logic:**
  - **Universal Instantiation (UI):**
    - Removes a universal quantifier ($\forall$) by substituting a specific constant for the quantified variable.
    - Example: From $\forall x\, Loves(John, x)$ (John loves everyone), we can instantiate:

      $$Loves(John, Mary),\ Loves(John, Max),\ \text{etc.}$$

    - This works for any constant in the domain, allowing reasoning about specific cases.
  - **Existential Instantiation (EI):**
    - Removes an existential quantifier ($\exists$) by substituting a new constant or function (called a Skolem constant or function) for the variable.
    - Example: From $\exists x\, Loves(x, Mary)$ (someone loves Mary), we can instantiate:

      $$Loves(C1, Mary),$$

      where $C1$ is a new constant representing "someone" (a placeholder individual).
  - These steps are crucial for reasoning in FOL, as they simplify quantified statements into concrete terms.
- **Generalized Modus Ponens (GMP):**
  - A more flexible version of Modus Ponens that works with FOL.
  - **Formula**:

    $$\frac{(P_1 \wedge P_2 \wedge \cdots \wedge P_n) \Rightarrow Q,\ P_1',\ P_2',\ \ldots,\ P_n'}{Q'}$$

    Where:

    - $P_1, P_2, \ldots, P_n$: Premises in the rule.
    - $P'_1, P'_2, \ldots, P'_n$: Facts matching the premises after unification.
    - Q: Conclusion in the rule.
    - Q': Result of applying the unifier θ to Q.
  - Example:
    - Rule: $\forall x\,(Human(x) \Rightarrow Mortal(x))$.
    - Fact: $Human(Socrates)$.
    - Instantiate the rule: $Human(Socrates) \Rightarrow Mortal(Socrates)$.
    - Apply Modus Ponens: Conclude $Mortal(Socrates)$.

- **Unification:**
  - A process to make two logical expressions identical by finding a substitution for their variables.
  - **Unification Formula**:

$$\text{UNIFY}(P(t_1, t_2, \ldots), P(u_1, u_2, \ldots)) = \theta,$$

    where θ is a substitution such that $t_i = u_i$ for all i.
  - Example:
    - Given $Loves(John, x)$ and $Loves(y, Mary)$.
    - Unification finds the substitution $\theta = \{x/Mary, y/John\}$ to make the sentences identical:

$$Loves(John, Mary).$$

  - Unification is crucial in reasoning processes like resolution, where you need to match predicates in different clauses.

Conversion to CNF (Conjunctive Normal Form)

To use resolution in FOL, all sentences must be in CNF. Here's how to convert any FOL sentence to CNF:

1. **Eliminate implications and biconditionals:**
   - Replace $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.
   - Replace $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
2. **Move negations inward:**
   - Use De Morgan's Laws:

$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta), \quad \neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta).$$

   - Push $\neg$ down to the atomic level (e.g., individual predicates).

3. **Standardize variables apart:**
   - Ensure all variables in the formula are uniquely named to avoid conflicts during substitution.
   - Example: In $\forall x\, Loves(x, Mary) \land \forall x\, Hates(x, Mary)$, rename variables:
   $$\forall x\, Loves(x, Mary) \land \forall y\, Hates(y, Mary).$$
4. **Skolemize existential quantifiers:**
   - Replace $\exists x$ with a Skolem constant (if no universal quantifiers) or a Skolem function (if there are universal quantifiers).
   - Example:
   $$\exists x\, Loves(x, Mary) \equiv Loves(C1, Mary).$$
   $$\forall y\, \exists x\, Loves(x, y) \equiv \forall y\, Loves(F(y), y),$$
   where $F(y)$ is a Skolem function.
5. **Drop universal quantifiers:**
   - Universal quantifiers are implicit in CNF.
   - Example: $\forall x\, Loves(x, Mary)$ becomes:
   $$Loves(x, Mary).$$
6. **Distribute $\lor$ over $\land$:**
   - Apply distributive laws to form a conjunction of disjunctions.
   - Example: $(P \lor (Q \land R)) \equiv (P \lor Q) \land (P \lor R)$.

Resolution Refutation

To prove entailment in FOL using resolution:

1. **Negate the goal:**
   - Add the negation of the query to the knowledge base.
   - Example: To prove $Mortal(Socrates)$, add $\neg Mortal(Socrates)$ to KB.
2. **Convert all sentences to CNF:**
   - Use the steps outlined above to convert KB and the negated goal to CNF.

3. **Apply the resolution rule: Resolution Rule**
   The resolution rule is an inference rule that operates on two clauses and derives a new clause by eliminating a pair of complementary literals.

$$\frac{P(x) \vee A,\ \neg P(y) \vee B}{A \vee B\ [\theta]}$$

   Where:

   - $P(x)$ and $\neg P(y)$ are complementary literals (one is the negation of the other).
   - A and B are disjunctions of literals in the two clauses.
   - θ is a unifier that makes $P(x)$ and $\neg P(y)$ identical by substituting variables as necessary.

   Explanation: The resolution rule removes $P(x)$ and $\neg P(y)$ from the two clauses and combines the remaining literals (A and B) into a new clause. This rule is used in the resolution refutation method to prove a query by deriving a contradiction (empty clause $\perp$).

   - Resolve pairs of clauses to eliminate literals and derive new clauses.
   - Continue resolving until:
     - You derive an empty clause (contradiction), proving the query is true.
     - No further resolution is possible, proving the query is false.

4. **Example:**

   - $KB = \{\forall x\,(Human(x) \Rightarrow Mortal(x)), Human(Socrates)\}$.
   - Query: Prove $Mortal(Socrates)$.
   - Steps:
     (a) Negate the goal: $\neg Mortal(Socrates)$.
     (b) Convert to CNF:
       - $Human(x) \Rightarrow Mortal(x) \equiv \neg Human(x) \vee Mortal(x)$.
       - $Human(Socrates)$ remains as is.
       - $\neg Mortal(Socrates)$ remains as is.
     (c) Resolve:
       - Resolve $Human(Socrates)$ with $\neg Human(x) \vee Mortal(x)$, substituting $x = Socrates$, to get $Mortal(Socrates)$.
       - Resolve $Mortal(Socrates)$ with $\neg Mortal(Socrates)$ to derive the empty clause.

5 Adversarial Search and Games

In this chapter we cover **competitive environments**, in which two or more agents have conflicting goals, giving rise to **adversarial search** problems. Rather than deal with the chaos of real-world skirmishes, we will concentrate on games, such as chess, Go, and poker

5.1 Game Theory

There are at least three stances we can take towards multi-agent environments. The first stance, appropriate when there are a very large number of agents, is to consider them in the aggregate as an economy, allowing us to do things like predict that increasing demand will cause prices to rise, without having to predict the action of any individual agent.

Second, we could consider adversarial agents as just a part of the environment—a part that makes the environment nondeterministic. But if we model the adversaries in the same way that, say, rain sometimes falls and sometimes doesn't, we miss the idea that our adversaries are actively trying to defeat us, whereas the rain supposedly has no such intention.

The third stance is to explicitly model the adversarial agents with the techniques of adversarial game-tree search. That is what this chapter covers. We begin with a restricted class of games and define the optimal move and an algorithm for finding it: minimax search, a generalization of AND–OR search. We show that **pruning** makes the search more efficient by ignoring portions of the search tree that make no difference to the optimal move. For nontrivial games, we will usually not have enough time to be sure of finding the optimal move (even with pruning); we will have to cut off the search at some point.

For each state where we choose to stop searching, we ask who is winning. To answer this question we have a choice: we can apply a heuristic **evaluation function** to estimate who is winning based on features of the state, or we can average the outcomes of many fast simulations of the game from that state all the way to the end.

5.1.1 Two-player zero-sum games

The games most commonly studied within AI (such as chess and Go) are what game theorists call deterministic, two-player, turn-taking, perfect information, zero-sum games. "Perfect information" is a synonym for "fully observable,"1 and "zero-sum" means that what is good for one player is just as bad for the other: there is no "win-win" outcome. For games we often use the term move as a synonym for "action" and position as a synonym for "state."

We will call our two players MAX and MIN, for reasons that will soon become

obvious. MAX moves first, and then the players take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A game can be formally defined with the following elements:

- S_0: The *initial state*, which specifies how the game is set up at the start.
- TO-MOVE(s): The player whose turn it is to move in state s.
- ACTIONS(s): The set of legal moves in state s.
- RESULT(s, a): The *transition model*, which defines the state resulting from taking action a in state s.
- IS-TERMINAL(s): A *terminal test*, which is true when the game is over and false otherwise. States where the game has ended are called *terminal states*.
- UTILITY(s, p): A *utility function* (also called an objective function or payoff function), which defines the final numeric value to player p when the game ends in terminal state s. In chess, the outcome is a win, loss, or draw, with values 1, 0, or 1/2. Some games have a wider range of possible outcomes—for example, the payoffs in backgammon range from 0 to 192.

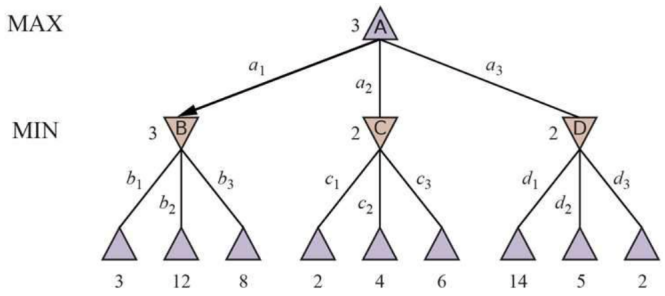
The initial state, ACTIONS function, and RESULT function define the **state space graph**—a graph where the vertices are states, the edges are moves and a state might be reached by multiple paths. We can superimpose a **search tree** over part of that graph to determine what move to make. We define the complete **game tree** as a search tree that follows every sequence of moves all the way to a terminal state. The game tree may be infinite if the state space itself is unbounded or if the rules of the game allow for infinitely repeating positions.

5.2 Optimal Decisions in Games

MAX wants to find a sequence of actions leading to a win, but MIN has something to say about it. This means that MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves. In games that have a binary outcome (win or lose), we could use AND–OR search to generate the conditional plan. In fact, for such games, the definition of a winning strategy for the game is identical to the definition of a solution for a nondeterministic planning problem: in both cases the desirable outcome must be guaranteed no matter what the "other side" does. For games with multiple outcome scores, we need a slightly more general algorithm called **minimax search**.

Consider the trivial game in the figure below. The possible moves for MAX at the root node are labeled a_1, a_2, a_3. The possible replies to a_1 for MIN are

b_1, b_2, b_3, and so on. This particular game ends after one move each by MAX and MIN.



A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

Given a game tree, the optimal strategy can be determined by working out the **minimax value** of each state in the tree, which we write as Minimax(s). The minimax value is the utility (for MAX) of being in that state, *assuming that both players play optimally from there to the end of the game*. The minimax value of a terminal state is just its utility. In a non-terminal state, MAX prefers to move to a state of maximum value when it is MAX's turn to move, and MIN prefers a state of minimum value (that is, minimum value for MAX and thus maximum value for MIN). So we have:

$$\text{Minimax}(s) = \begin{cases} \text{Utility}(s, \text{MAX}) & \text{if IS-TERMINAL}(s) \\ \max_{a \in \text{ACTIONS}(s)} \text{Minimax}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MAX} \\ \min_{a \in \text{ACTIONS}(s)} \text{Minimax}(\text{RESULT}(s, a)) & \text{if TO-MOVE}(s) = \text{MIN} \end{cases}$$

The terminal nodes on the bottom level get their utility values from the game's UTILITY function. The first MIN node, labeled , has three successor states with values 3, 12, and 8, so its minimax value is 3. Similarly, the other two MIN nodes have minimax value 2. The root node is a MAX node; its successor states have minimax values 3, 2, and 2; so it has a minimax value of 3. We can also identify the **minimax decision** at the root: action a_1 is the optimal choice for MAX because it leads to the state with the highest minimax value.

5.2.1 The minimax search algorithm

Now that we can compute MINIMAX(s), we can turn that into a search algorithm that finds the best move for MAX by trying all actions and choosing the one whose resulting state has the highest MINIMAX value.

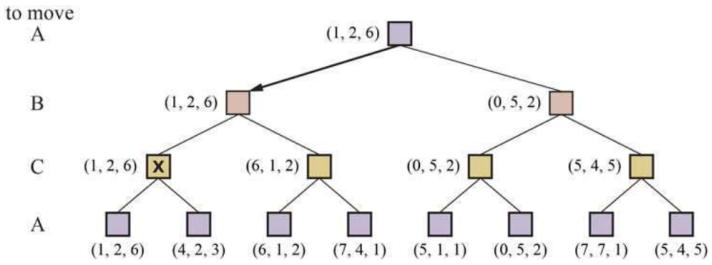
```
function MINIMAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

It is a recursive algorithm that proceeds all the way down to the leaves of the tree and then **backs up** the minimax values through the tree as the recursion unwinds. The minimax algorithm performs a complete depth-first exploration of the game tree.

5.2.2 Optimal decisions in multiplayer games

Many popular games allow more than two players. Let us examine how to extend the minimax idea to multiplayer games. This is straightforward from the technical viewpoint, but raises some interesting new conceptual issues. First, we need to replace the single value for each node with a vector of values. For example, in a three-player game with players A, B, and C, a vector$[v_A, v_B, v_C]$ is associated with each node. For terminal states, this vector gives the utility of the state from each player's viewpoint. The simplest way to implement this is to have the UTILITY function return a vector of utilities.

Now we have to consider nonterminal states. Consider the node marked X in the game tree shown in the figure below. In that state, player C chooses what to do. The two choices lead to terminal states with utility vectors $\langle v_A = 1, v_B = 2, v_C = 6\rangle$ and $\langle v_A = 4, v_B = 2, v_C = 3\rangle$. Since 6 is bigger than 3, C should choose the first move. This means that if state X is reached, subsequent play will lead to a terminal state with utilities $\langle v_A = 1, v_B = 2, v_C = 6\rangle$. Hence, the backed-up value of X is this vector. In general, the backed-up value of a node n is the utility vector of the successor state with the highest value for the player choosing at n.

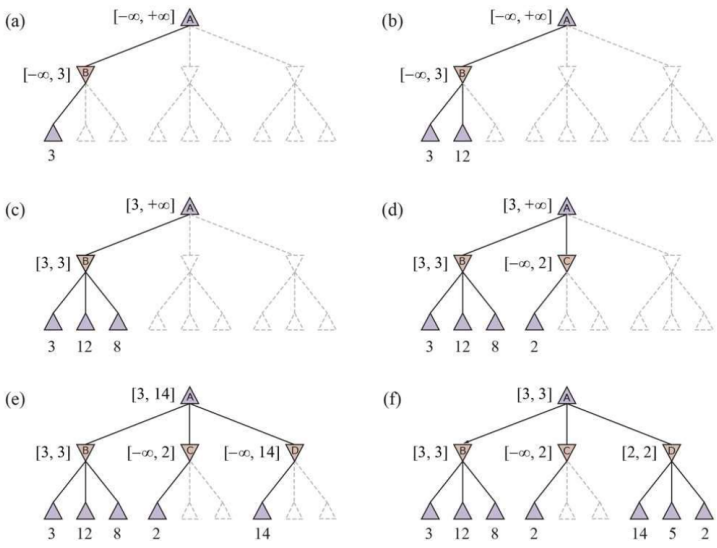


The first three ply of a game tree with three players (*A, B, C*). Each node is labeled with values from the viewpoint of each player. The best move is marked at the root.

5.2.3 Alpha–Beta Pruning

The number of game states is exponential in the depth of the tree. No algorithm can completely eliminate the exponent, but we can sometimes cut it in half, computing the correct minimax decision without examining every state by **pruning** large parts of the tree that make no difference to the outcome. The particular technique we examine is called **alpha–beta pruning**.

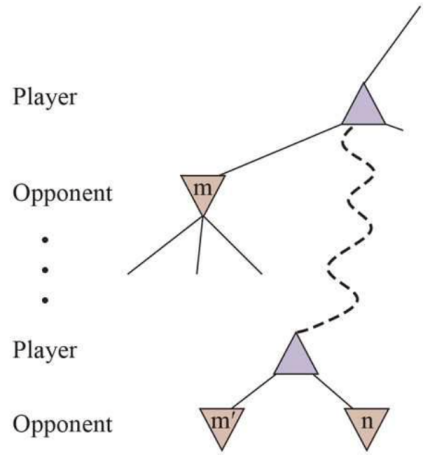
Consider again the two-ply game tree. Let's go through the calculation of the optimal decision once more, this time paying careful attention to what we know at each point in the process. The outcome is that we can identify the minimax decision without ever evaluating two of the leaf nodes.



At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid

this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successor states, so the value of B is exactly 3. Now we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, C, which is a MIN node, has a value of *at most* 2. But we know that B is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successor states of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative (i.e., 3), so we need to keep exploring D's successor states. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree(see figure below), such that Player has a choice of moving to n. If Player has a better choice either at the same level, or at any point higher up in the tree, then Player will never move to n. So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.



The general case for alpha–beta pruning. If m or m' is better than n for Player, we will never get to n in play.

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Alpha–beta pruning gets its name from the two extra parameters in MAX-VALUE(state, α, β) (see *Figure 5.7*) that describe bounds on the backed-up values that appear anywhere

along the path:

α = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX. Think: α = "at least."

β = the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Think: β = "at most."

```
function ALPHA-BETA-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state, −∞, +∞)
  return move

function MAX-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a), α, β)
    if v2 > v then
      v, move ← v2, a
      α ← MAX(α, v)
    if v ≥ β then return v, move
  return v, move

function MIN-VALUE(game, state, α, β) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a), α, β)
    if v2 < v then
      v, move ← v2, a
      β ← MIN(β, v)
    if v ≤ α then return v, move
  return v, move
```

Alpha–beta search updates the values of α and β as it goes along and prunes the remaining branches at a node (i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α or β value for MAX or MIN, respectively.

5.2.4 Move ordering

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. For example, in the pruning example e) and f), we could not prune any successors of D at all because the worst successors (from the point of view of MIN) were generated first. If the third successor of D had been generated first, with value 2, we would have been able to prune the other two successors. This suggests that it might be worthwhile to try to first examine the successors that are likely to be best. Obviously we cannot achieve perfect move ordering—in that case the ordering function could be used to play a perfect game! But we can often get fairly close. For chess, a fairly simple ordering

function (such as trying captures first, then threats, then forward moves, and then backward moves) gets you to within about a factor of 2 of the best-case result.

Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit. The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move through a process of **iterative deepening**. First, search one ply deep and record the ranking of moves based on their evaluations. Then search one ply deeper, using the previous ranking to inform move ordering; and so on. The increased search time from iterative deepening can be more than made up from better move ordering. The best moves are known as **killer moves**, and to try them first is called the killer move heuristic.

In game tree search, repeated states can occur because of **transpositions**, different permutations of the move sequence that end up in the same position, and the problem can be addressed with a **transposition table** that caches the heuristic value of states.

5.3 Heuristic Alpha–Beta Tree Search

To make use of our limited computation time, we can cut off the search early and apply a heuristic **evaluation function** to states, effectively treating nonterminal nodes as if they were terminal. In other words, we replace the UTILITY function with EVAL, which estimates a state's utility. We also replace the terminal test by a **cutoff test**, which must return true for terminal states, but is otherwise free to decide when to cut off the search, based on the search depth and any property of the state that it chooses to consider. That gives us the formula H-MINIMAX(s, d) for the heuristic minimax value of state s at search depth d:

$$\text{H-Minimax}(s, d) =$$

$$\begin{cases} \textsc{Eval}(s, \textsc{Max}) & \text{if } \textsc{Is-Cutoff}(s, d) \\ \max_{a \in \textsc{Actions}(s)} \text{H-Minimax}(\textsc{Result}(s, a), d+1) & \text{if } \textsc{To-Move}(s) = \textsc{Max} \\ \min_{a \in \textsc{Actions}(s)} \text{H-Minimax}(\textsc{Result}(s, a), d+1) & \text{if } \textsc{To-Move}(s) = \textsc{Min} \end{cases}$$

5.3.1 Evaluation functions

A heuristic evaluation function EVAL(s, p) returns an *estimate* of the expected utility of state s to player p, return an estimate of the distance to the goal. For terminal states, it must be that EVAL(s, p) = UTILITY(s, p) and for nonterminal states, the evaluation must be somewhere between a loss and a win:

$$\textsc{Utility}(\text{loss}, p) \leq \textsc{Eval}(s, p) \leq \textsc{Utility}(\text{win}, p).$$

Most evaluation functions work by calculating various **features** of the state—for example, in chess, we would have features for the number of white pawns, black pawns, white queens, black queens, and so on. The features, taken together, define various categories or equivalence classes of states: the states in each category have the same values for all the features. For example, one category might contain all two-pawn versus one-pawn endgames. Any given category will contain some states that lead (with perfect play) to wins, some that lead to draws, and some that lead to losses.

The evaluation function does not know which states are which, but it can return a single value that estimates the *proportion* of states with each outcome. For example, suppose our experience suggests that 82% of the states encountered in the two-pawns versus one-pawn category lead to a win (utility +1); 2% to a loss (0), and 16% to a draw (1/2). Then a reasonable evaluation for states in the category is the **expected value**:

$$(0.82 \times +1) + (0.02 \times 0) + (0.16 \times 1/2) = 0.90.$$

In principle, the expected value can be determined for each category of states, resulting in an evaluation function that works for any state.

In practice, this kind of analysis requires too many categories and hence too much experience to estimate all the probabilities. Instead, most evaluation functions compute separate numerical contributions from each feature and then *combine* them to find the total value. For centuries, chess players have developed ways of judging the value of a position using just this idea. For example, introductory chess books give an approximate **material value** for each piece: each pawn is worth 1, a knight or bishop is worth 3, a rook 5, and the queen 9. Other features such as "good pawn structure" and "king safety" might be worth half a pawn, say. These feature values are then simply added up to obtain the evaluation of the position.

Mathematically, this kind of evaluation function is called a *weighted linear function* because it can be expressed as

$$\textsc{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s) = \sum_{i=1}^{n} w_i f_i(s),$$

where each f_i is a feature of the position (such as "number of white bishops") and each w_i is a weight (saying how important that feature is). The weights should be normalized so that the sum is always within the range of a loss (0) to a win (+1).

5.3.2 Cutting off search

The next step is to modify ALPHA-BETA-SEARCH so that it will call the heuristic EVAL function when it is appropriate to cut off the search. We replace the two lines in the function that mention IS-TERMINAL with the following line:

if game.IS-CUTOFF(state, depth) then return game.EVAL(state, player), null

We also must arrange for some bookkeeping so that the current depth is incremented on each recursive call. The most straightforward approach to controlling the amount of search is to set a fixed depth limit so that IS-CUTOFF(state, depth) returns true for all depth greater than some fixed depth d(as well as for all terminal states). The depth is chosen so that a move is selected within the allocated time.

This simple approach can lead to errors due to the approximate nature of the evaluation function. The evaluation function should be applied only to positions that are **quiescent**, that is, positions in which there is no pending move (such as a capturing the queen) that would wildly swing the evaluation. For nonquiescent positions the IS-CUTOFF returns false, and the search continues until quiescent positions are reached. This extra **quiescence search** is sometimes restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

The horizon effect is more difficult to eliminate. It arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by the use of delaying tactics. One strategy to mitigate the horizon effect is to allow **singular extensions**, moves that are "clearly better" than all other moves in a given position, even when the search would normally be cut off at that point. This makes the tree deeper, but because there are usually few singular extensions, the strategy does not add many total nodes to the tree, and has proven to be effective in practice.

5.3.3 Forward pruning

Alpha–beta pruning prunes branches of the tree that can have no effect on the final evaluation, but **forward pruning** prunes moves that appear to be poor moves, but might possibly be good ones. Thus, the strategy saves computation time at the risk of making an error. In Shannon's terms, this is a Type B strategy. Clearly, most human chess players do this, considering only a few moves from each position (at least consciously).

One approach to forward pruning is **beam search**: on each ply, consider only a "beam" of the best moves (according to the evaluation function) rather than considering all possible moves. Unfortunately, this approach is rather dangerous because there is no guarantee that the best move will not be pruned away.

Another technique, **late move reduction**, works under the assumption that move ordering has been done well, and therefore moves that appear later in the list of possible moves are less likely to be good moves. But rather than pruning them away completely, we just reduce the depth to which we search these moves, thereby saving time. If the reduced search comes back with a value above the current value, we can re-run the search with the full depth.

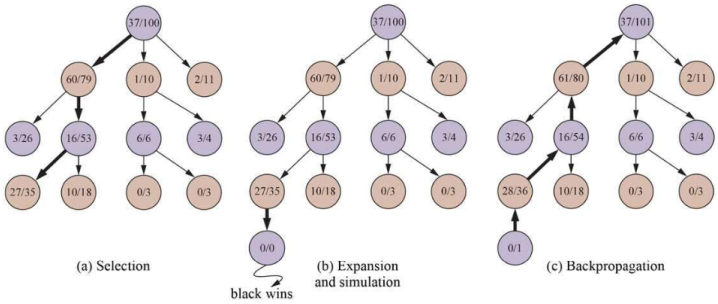
5.4 Monte Carlo Tree Search

The game of Go illustrates two major weaknesses of heuristic alpha–beta tree search: First, Go has a branching factor that starts at 361, which means alpha–beta search would be limited to only 4 or 5 ply. Second, it is difficult to define a good evaluation function for Go because material value is not a strong indicator and most positions are in flux until the endgame. In response to these two challenges, modern Go programs have abandoned alpha–beta search and instead use a strategy called **Monte Carlo tree search (MCTS).**

The basic MCTS strategy does not use a heuristic evaluation function. Instead, the value of a state is estimated as the average utility over a number of **simulations** of complete games starting from the state. A simulation (also called a **playout** or **rollout**) chooses moves first for one player, than for the other, repeating until a terminal position is reached. At that point the rules of the game (not fallible heuristics) determine who has won or lost, and by what score. For games in which the only outcomes are a win or a loss, "average utility" is the same as "win percentage."

How do we choose what moves to make during the playout? To get useful information from the playout we need a **playout policy** that biases the moves towards good ones. For Go and other games, playout policies have been successfully learned from self-play by using neural networks. Sometimes game-specific heuristics are used. Given a playout policy, we next need to decide two things: from what positions do we start the playouts, and how many playouts do we allocate to each position? The simplest answer, called **pure Monte Carlo search**, is to do N simulations starting from the current state of the game, and track which of the possible moves from the current position has the highest win percentage.

For some stochastic games this converges to optimal play as increases, but for most games it is not sufficient—we need a **selection policy** that selectively focuses the computational resources on the important parts of the game tree. It balances two factors: **exploration** of states that have had few playouts, and **exploitation** of states that have done well in past playouts, to get a more accurate estimate of their value. Monte Carlo tree search does that by maintaining a search tree and growing it on each iteration of the following four steps.



One iteration of the process of choosing a move with Monte Carlo tree search (MCTS) using the upper confidence bounds applied to trees (UCT) selection metric, shown after 100 iterations have already been done. In (a) we select moves, all the way down the tree, ending at the leaf node marked (27/35) (for 27 wins for black out of 35 playouts). In (b) we expand the selected node and do a simulation (playout), which ends in a win for black. In (c), the results of the simulation are back-propagated up the tree.

- **SELECTION**: Starting at the root of the search tree, we choose a move (guided by the selection policy), leading to a successor node, and repeat that process, moving down the tree to a leaf. a) in the figure above shows a search tree with the root representing a state where white has just moved, and white has won 37 out of the 100 playouts done so far. The thick arrow shows the selection of a move by black that leads to a node where black has won 60/79 playouts. This is the best win percentage among the three moves, so selecting it is an example of exploitation. But it would also have been reasonable to select the 2/11 node for the sake of exploration—with only 11 playouts, the node still has high uncertainty in its valuation, and might end up being best if we gain more information about it. Selection continues on to the leaf node marked 27/35.

- **EXPANSION**: We grow the search tree by generating a new child of the selected node; b) shows the new node marked with 0/0. (Some versions generate more than one child in this step.)

- **SIMULATION**: We perform a playout from the newly generated child node, choosing moves for both players according to the playout policy. These moves are *not* recorded in the search tree. In the figure, the simulation results in a win for black.

- **BACK-PROPAGATION**: We now use the result of the simulation to update all the search tree nodes going up to the root. Since black won the playout, black nodes are incremented in both the number of wins and the number of playouts, so 27/35 becomes 28/26 and 60/79 becomes 61/80. Since white lost, the white nodes are incremented in the number of playouts only, so 16/53 becomes 16/54 and the root 37/100 becomes 37/101.

We repeat these four steps either for a set number of iterations, or until the allotted time has expired, and then return the move with the highest number

of playouts.

One very effective selection policy is called "upper confidence bounds applied to trees" or **UCT**. The policy ranks each possible move based on an upper confidence bound formula called **UCB1**. For a node n, the formula is:

$$\text{UCB1}(n) = \frac{U(n)}{N(n)} + C \times \sqrt{\frac{\log N(\text{PARENT}(n))}{N(n)}}$$

where $U(n)$ is the total utility of all playouts that went through node n, $N(n)$ is the number of playouts through node n, and $\text{PARENT}(n)$ is the parent node of n in the tree. Thus, $\frac{U(n)}{N(n)}$ is the exploitation term: the average utility of n. The term with the square root is the exploration term: it has the count $N(n)$ in the denominator, which means the term will be high for nodes that have only been explored a few times. In the numerator it has the log of the number of times we have explored the parent of n. This means that if we are selecting n some non-zero percentage of the time, the exploration term goes to zero as the counts increase, and eventually the playouts are given to the node with highest average utility.

C is a constant that balances exploitation and exploration. There is a theoretical argument that C should be $\sqrt{2}$, but in practice, game programmers try multiple values for C and choose the one that performs best. (Some programs use slightly different formulas; for example, ALPHAZERO adds in a term for move probability, which is calculated by a neural network trained from past self-play.) With $C = 1.4$, the 60/79 node in *Figure 5.10* has the highest UCB1 score, but with $C = 1.5$, it would be the 2/11 node.

The algorithm below shows the complete UCT MCTS algorithm. When the iterations terminate, the move with the highest number of playouts is returned. You might think that it would be better to return the node with the highest average utility, but the idea is that a node with 65/100 wins is better than one with 2/3 wins, because the latter has a lot of uncertainty. In any event, the UCB1 formula ensures that the node with the most playouts is almost always the node with the highest win percentage, because the selection process favors win percentage more and more as the number of playouts goes up.

```
function MONTE-CARLO-TREE-SEARCH(state) returns an action
  tree ← NODE(state)
  while IS-TIME-REMAINING() do
    leaf ← SELECT(tree)
    child ← EXPAND(leaf)
    result ← SIMULATE(child)
    BACK-PROPAGATE(result, child)
  return the move in ACTIONS(state) whose node has highest number of playouts
```

The Monte Carlo tree search algorithm. A game tree, *tree*, is initialized, and then we repeat a cycle of SELECT / EXPAND / SIMULATE / BACK-PROPAGATE until we run out of time, and return the move that led to the node with the highest number of playouts.

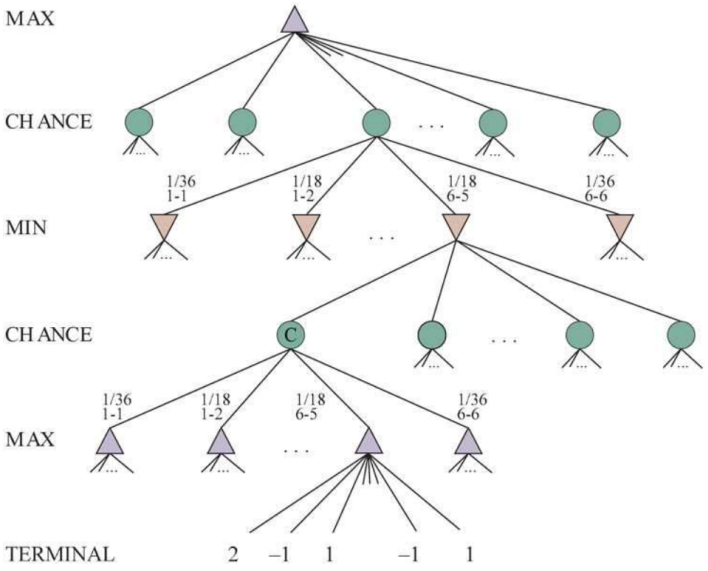
It is also possible to combine aspects of alpha–beta and Monte Carlo search. For example, in games that can last many moves, we may want to use **early playout termination**, in which we stop a playout that is taking too many moves, and either evaluate it with a heuristic evaluation function or just declare it a draw.

Monte Carlo search can be applied to brand-new games, in which there is no body of experience to draw upon to define an evaluation function. As long as we know the rules of the game, Monte Carlo search does not need any additional information. The selection and playout policies can make good use of hand-crafted expert knowledge when it is available, but good policies can be learned using neural networks trained by self-play alone.

Monte Carlo search has a disadvantage when it is likely that a single move can change the course of the game, because the stochastic nature of Monte Carlo search means it might fail to consider that move. In other words, Type B pruning in Monte Carlo search means that a vital line of play might not be explored at all. Monte Carlo search also has a disadvantage when there are game states that are "obviously" a win for one side or the other (according to human knowledge and to an evaluation function), but where it will still take many moves in a playout to verify the winner.

5.5 Stochastic Games

Stochastic games bring us a little closer to the unpredictability of real life by including a random element, such as the throwing of dice. Backgammon is a typical stochastic game that combines luck and skill. A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in the figure below. The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability. There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls. The six doubles (1–1 through 6–6) each have a probability of 1/36, so we say $P(1-1) = 1/36$. The other 15 distinct rolls each have a 1/18 probability.



Schematic game tree for a backgammon position.

The next step is to understand how to make correct decisions. Obviously, we still want to pick the move that leads to the best position. However, positions do not have definite minimax values. Instead, we can only calculate the **expected value** of a position: the average over all possible outcomes of the chance nodes.

This leads us to the **expectiminimax** value for games with chance nodes, a generalization of the minimax value for deterministic games. Terminal nodes and MAX and MIN nodes work exactly the same way as before (with the caveat that the legal moves for MAX and MIN will depend on the outcome of the dice roll in the previous chance node). For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

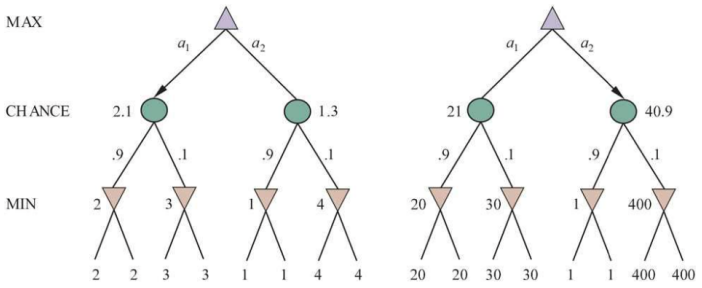
$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s, \textsc{Max}) & \text{if } \textsc{Is-Terminal}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{Max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{To-Move}(s) = \textsc{Min} \\ \sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{To-Move}(s) = \textsc{Chance} \end{cases}$$

where r represents a possible dice roll (or other chance event) and $\textsc{Result}(s, r)$ is the same state as s, with the additional fact that the result of the dice roll is r.

5.5.1 Evaluation functions for games of chance

As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf. One might think that evaluation functions for games such as backgammon should be just like evaluation functions for chess—they just need to give higher values to better positions. But in fact, the presence of chance nodes means that one has to be more careful about what the values mean.

The figure below shows what happens: with an evaluation function that assigns the values [1, 2, 3, 4] to the leaves, move a_1 is best; with values [1, 20, 30, 400], move a_2 is best. Hence, the program behaves totally differently if we make a change to some of the evaluation values, even if the preference order remains the same.



An order-preserving transformation on leaf values changes the best move.

It turns out that to avoid this problem, the evaluation function must return values that are a positive linear transformation of the **probability** of winning (or of the expected utility, for games that have outcomes other than win/lose). This relation to probability is an important and general property of situations in which uncertainty is involved.

5.6 Partially Observable Games

Bobby Fischer declared that "chess is war," but chess lacks at least one major characteristic of real wars, namely, **partial observability**. In the "fog of war," the whereabouts of enemy units is often unknown until revealed by direct contact. As a result, warfare includes the use of scouts and spies to gather information and the use of concealment and bluff to confuse the enemy.Partially observable games share these characteristics and are thus qualitatively different from the games in the preceding sections.

In deterministic partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent.

Game theory

Game theory: Systematic study of strategic interactions among rational individuals.
Rational individual: Has well-defined objectives and implements the best available strategy to pursue them.

Nash equilibrium for two-player simultaneous action games

- $\pi_p(s, a)$: strategy for player $p \in \{1, 2\}$, i.e., probability of taking action a at state s.
  - In this lecture:
    - In all but the last section, the state is empty and is therefore omitted.
    - Each player has only two actions, thus the probability of taking the first action is defined to be π_p, and the probability of taking the second action is then $1 - \pi_p$.
- $E_p(\pi_1, \pi_2)$: expected value for player p of the given players' strategies.

The specific strategies (π_1^*, π_2^*) form a Nash equilibrium if:

$$E_1(\pi_1^*, \pi_2^*) \geq E_1(\pi_1, \pi_2^*) \quad \text{for all possible strategies } \pi_1$$

$$E_2(\pi_1^*, \pi_2^*) \geq E_2(\pi_1^*, \pi_2) \quad \text{for all possible strategies } \pi_2$$

and no player will then have an incentive to change his/her strategy.

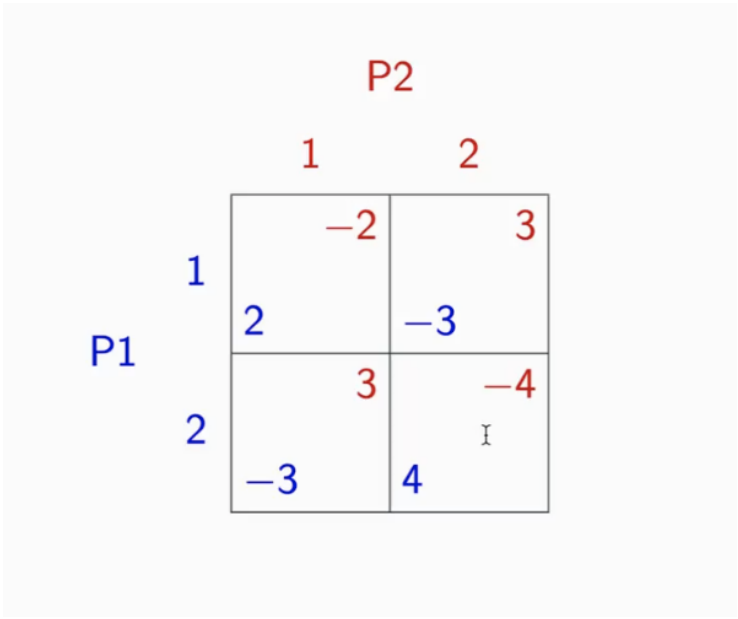
Pure vs mixed strategy Nash equilibria

- **Pure strategy**: selects a predetermined action, i.e., $\pi_p \in \{0, 1\}$.
- **Mixed strategy**: selects actions according to a probability distribution, i.e., $\pi_p \in [0, 1]$.
  - A pure strategy can be seen as a special case of a mixed strategy.

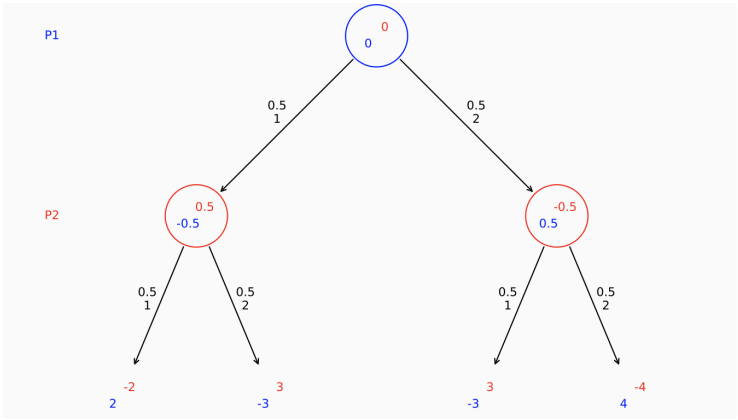
Mixed strategies

Two-finger Morra

Two players show either one or two fingers on a count of three(similar to rock, paper, scissors). If both players show an equal amount, P1 wins. If they show different amount of fingers, P2 wins. Showing two fingers has higher stakes for both players as the amount your losing will increase. It is best described through the payoff utility matrix below.



Below is the game tree for two-finger Morra:

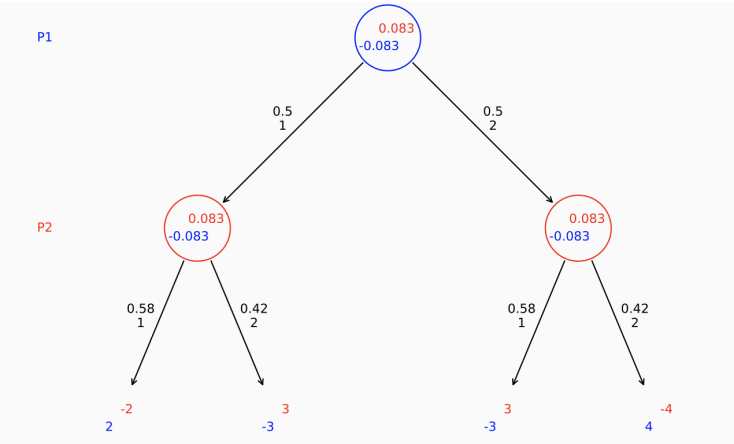


We can see from the expected value after P1 has made its move(numbers within the red circles) that he can exploit P2 by always choosing two fingers. The question then becomes: how can P2 play so that P1 cannot increase its expected value(How can P2 stop P1 form exploiting P2).
Mixed strategy for P2 with probability π_2 of choosing 1, where the expected values of P1 in the red nodes are set equal:

$$2\pi_2 - 3(1-\pi_2) = -3\pi_2 + 4(1-\pi_2)$$

$$\pi_2 = \frac{7}{12}$$

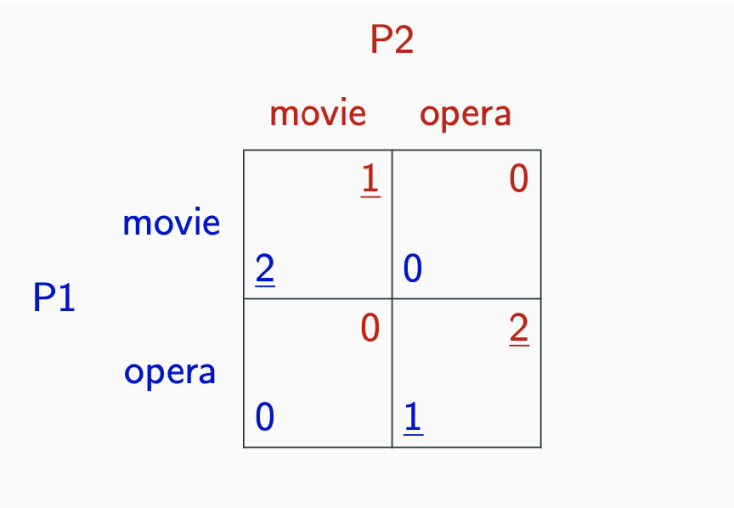
This results in the following game tree for mixed strategies for P2:



From the tree we see that what move P1 chooses is irrelevant because the expected values after P1 makes a move is equal in both cases.

A mixed strategy for P1 would yield the exact same results in favor of P1 as it did for P2. The result after both players uses the mixed strategy is therefor a Nash equilibrium.

Battle of the Sexes



Not a zero-sum game. Two pure strategy Nash equilibria: P1 and P2 choose movie, and P1 and P2 choose opera.

Mixed strategy for $P2$ with probability π_2 of choosing movie, where the expected values of $P1$ in the red nodes are set equal:

$$2\pi_2 + 0(1-\pi_2) = 0\pi_2 + 1(1-\pi_2)$$

$$\pi_2 = \frac{1}{3}$$

Mixed strategy for $P1$ with probability π_1 of choosing movie, where the expected values of $P2$ in the blue nodes are set equal:

$$1\pi_1 + 0(1-\pi_1) = 0\pi_1 + 2(1-\pi_1)$$

$$\pi_1 = \frac{2}{3}$$

Pure strategies

Prisoner's dilemma

		P2: Defect	P2: Stay silent
P1: Defect		−5 / −5	−10 / 0
P1: Stay silent		0 / −10	−1 / −1

From the matrix one can see that regardless of what the other person chooses, the best outcome is always defecting. For example if P2 stays silent, defecting gives P1 0 years in prison, while staying silent results in 1 year in prison. If P2 defects, defecting gives P1 5 years in prison, while staying silent results in 10 years in prison.
Defecting is a pure strategy Nash equilibrium.

Repeated games

Backward induction: when the number of rounds is fixed, finite and known

Players will play 10 rounds of prisoner's dilemma. What will the players do?
What will the players do at the tenth round? This is the same as playing the game once, and the answer is therefore, defect.
What will the players do at the ninth round? This is the same as playing the game once, and the answer is therefore, defect.
Etc.

Infinite rounds

We cannot use the sum of the expected values of each game:

$$\sum_{t=0}^{\infty} E_p(\pi_1(s_t), \pi_2(s_t))$$

where state s_t contains all the previous actions prior to time t.

For example, both the strategies

$$\pi_1(s_t) = \pi_2(s_t) = 1 \quad \forall s_t \quad \text{(always defect)}$$

and

$$\pi_1(s_t) = \pi_2(s_t) = 0 \quad \forall s_t \quad \text{(always stay silent)}$$

would then get the sum $-\infty$.

However, we can use a discount factor $\delta \in (0, 1)$:

$$\sum_{t=0}^{\infty} \delta^t E_p(\pi_1(s_t), \pi_2(s_t))$$

where a small δ will lead to discounting all but the first few games.

Grim trigger strategy

Let the state s_t be defined as all the previous actions:

$$s_t = ((a_{1,0}, a_{2,0}), (a_{1,1}, a_{2,1}), \ldots, (a_{1,t-1}, a_{2,t-1}))$$

where $a_{1,0}, a_{2,0}$ are player 1's and player 2's actions (1: defect, 0: stay silent) at the first round.

The grim trigger strategy for example for $P1$ is then:

$$\pi_1(s_t) = \begin{cases} 1, & \text{if } t > 0 \wedge \sum_{i=0}^{t-1} a_{2,i} > 0 \\ 0, & \text{otherwise} \end{cases}$$

i.e., defect if the opponent has defected previously; otherwise, stay silent.

If both players follow the grim trigger strategies, the discount factor determines if, for example, player 1 has no incentive to change his/her strategy:

$$\sum_{t=0}^{\infty} \delta^t E_1(0,0) \geq \delta^0 E_1(1,0) + \sum_{t=1}^{\infty} \delta^t E_1(1,1)$$

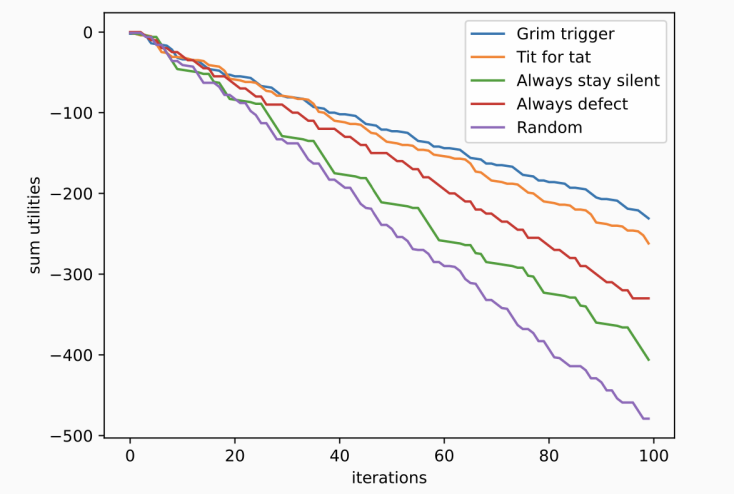
$$-1\delta^0 - 1\delta^1 - 1\delta^2 - \cdots \geq 0\delta^0 - 5\delta^1 - 5\delta^2 - 5\delta^3 - \ldots$$

$$\frac{-1}{1-\delta} \geq 5 - \frac{5}{1-\delta}$$

$$\delta \geq \frac{1}{5}$$

i.e., if both players follow the grim strategy and use a discount factor $\delta \geq \frac{1}{5}$, they are in a Nash equilibrium.

Unknown number of rounds



The tit for tat strategy is to repeat the opponent's last action, for example for $P1$:

$$\pi_1(s_t) = \begin{cases} 0, & \text{if } t = 0 \\ a_{2,t-1}, & \text{otherwise} \end{cases}$$

23 Natural Language Processing

NLP Defenition: NLP is a subfield in Artificial Intelligence (AI) and Computational Linguistics, focuses on

- The interaction between computers and humans' (natural) languages.
- Developing algorithms to interpret, and generate responses to/in human languages, which should be meaningful and useful.

Key Applications of NLP:

- Machine Translation: Statistical-based or Neural-based
- Sentiment Analysis: e.g., customer feedback analysis
- Chatbots and AI assistants: e.g., ChatGPT
- Text Summarization and Classification: e.g., identification of a document main topic
- Paraphrasing and Grammar correction: e.g., Grammarly, ChatGPT
- Auto-completion: e.g., the auto-complete in Google Search or mobile devices
- Named Entity Recognition (NER): e.g., identification of locations, organization or personal names
- Part-of-Speech tagging: e.g., recognizing the structure of phrases such as nouns, pronouns, verbs, etc..
- Document Clustering: e.g., grouping documents based on similarity

Text preprocessing is a crucial step in NLP that involves transforming raw text into a clean and structured format suitable for analysis and modelling. The main goals are:

- **Noise Reduction:** Removing irrelevant or redundant information
- **Normalization:** Standardizing text data to a consistent format, which helps in reducing the complexity
- **Tokenization:** Splitting the text into smaller units (tokens) such as words or subwords, which serve as the basic units for analysis

An example:

- Sentence 1: "Text preprocessing is essential for NLP!". Tokens: ["Text", "preprocessing", "is", "essential", "for", "NLP"]
- Sentence 2: "Text preprocessing helps improve NLP models." Tokens: ["Text", "preprocessing", "helps", "improve", "NLP", "models"]

- Combined Tokens from both sentences: [”Text”, ”preprocessing”, ”is”, ”essential”, ”for”, ”NLP”, ”Text”, ”preprocessing”, ”helps”, ”improve”, ”NLP”, ”models”]
- Tokens after lowercasing (Normalization): [”text”, ”preprocessing”, ”is”, ”essential”, ”for”, ”nlp”, ”text”, ”preprocessing”, ”helps”, ”improve”, ”nlp”, ”models”](Further normalization also include stemming and lemmatization)
- Removing Stop Words (Noise Reduction): Not applied for some tasks such as auto- completion
  - Stop Words list: [”is”, ”for”]
  - Tokens after removing Stop Words: [”text”, ”preprocessing”, ”essential”, ”nlp”, ”text”, ”preprocessing”, ”helps”, ”improve”, ”nlp”, ”models”]

Token	**Frequency**
text	2
preprocessing	2
essential	1
nlp	2
helps	1
improve	1
models	1

Table 1: Frequency Distribution Table

23.1 Language Models

We define a **language model** as a probability distribution describing the likelihood of any string. Such a model should say that “Do I dare disturb the universe?” has a reasonable probability as a string of English, but “Universe dare the I disturb do?” is extremely unlikely. With a language model, we can predict what words are likely to come next in a text, and thereby suggest completions for an email or text message. We can compute which alterations to a text would make it more probable, and thereby suggest spelling or grammar corrections.

23.1.1 The bag-of-words model

This section revisits the naive Bayes model, casting it as a full language model. That means we don’t just want to know what category is most likely for each sentence; we want a joint probability distribution over all sentences and categories. That suggests we should consider all the words in the sentence. Given a

sentence consisting of the words $w_1, w_2, \ldots, w_N$ (which we will write as $w_{1:N}$), the naive Bayes formula gives us

$$P(\text{Class} \mid w_{1:N}) = \alpha\, P(\text{Class}) \prod_j P(w_j \mid \text{Class}).$$

The application of naive Bayes to strings of words is called the **bag-of-words model**. It is a generative model that describes a process for generating a sentence: Imagine that for each category (business, weather, etc.) we have a bag full of words (you can imagine each word written on a slip of paper inside the bag; the more common the word, the more slips it is duplicated on). To generate text, first select one of the bags and discard the others. Reach into that bag and pull out a word at random; this will be the first word of the sentence. Then put the word back and draw a second word. Repeat until an end-of-sentence indicator (e.g., a period) is drawn.

This model is **clearly wrong**: it falsely assumes that each word is independent of the others, and therefore it does not generate coherent English sentences. But it does allow us to do classification with good accuracy using the naive Bayes formula.
We can learn the prior probabilities needed for this model via supervised training on a body or **corpus** of text, where each segment of text is labeled with a class. A corpus typically consists of at least a million words of text, and at least tens of thousands of distinct vocabulary words.

From a corpus we can estimate the prior probability of each category, $P(\text{Class})$, by counting how common each category is. We can also use counts to estimate the conditional probability of each word given the category, $P(w_j \mid \text{Class})$. For example, if we've seen 3000 texts and 300 of them were classified as *business*, then we can estimate

$$P(\text{Class} = \text{business}) \approx \frac{300}{3000} = 0.1.$$

And if within the *business* category we have seen 100,000 words and the word "stocks" appeared 700 times, then we can estimate

$$P(\text{stocks} \mid \text{Class} = \text{business}) \approx \frac{700}{100,000} = 0.007.$$

Estimation by counting works well when we have high counts (and low variance), but we will see a better way to estimate probabilities when the counts are low.

Note it is not trivial to decide what a word is. Is "aren't" one word, or should it be broken up as "aren/'/t" or "are/n't," or something else? The process of dividing a text into a sequence of words is called **tokenization**.

23.1.2 N-gram word models

N-grams: the model calculates the statistical representation of contiguous sequences of N items (Tokens) from a given text or speech. The items can be words, characters, or other units of text.
In the sentence "Text preprocessing is essential for NLP!":

- 1-grams (Unigrams) representations: ["text", "preprocessing", "is", "essential", "for", "nlp"]
- 2-grams (Bigrams) representations: ["text preprocessing", "preprocessing is", "is essential", "essential for", "for nlp"]
- 3-grams (Trigrams) representations: ["text preprocessing is", "preprocessing is essential", "is essential for", "essential for nlp"]

The Bigrams Language Model: For an auto-completion task of the input: "text pre"

Probability of Next Token w_{i+1} Given w_i

$$P(w_{i+1} \mid w_i) = \frac{\text{Count}(w_i, w_{i+1})}{\text{Count}(w_i)}$$

- Calculate Probabilities:
  - For "preprocessing" after "text":

    $$P(\text{"preprocessing"} \mid \text{"text"}) = \frac{\text{Count}(\text{"text"}, \text{"preprocessing"})}{\text{Count}(\text{"text"})} = \frac{2}{2} = 1$$

  - For "helps" after "preprocessing":

    $$P(\text{"helps"} \mid \text{"preprocessing"}) = \frac{\text{Count}(\text{"preprocessing"}, \text{"helps"})}{\text{Count}(\text{"preprocessing"})} = \frac{1}{2} = 0.5$$

Given this calculation, the auto-complete suggests, for Input "text pre":

- Suggestion 1: "text preprocessing" (high probability, based on bigram count)
- Suggestion 2: "preprocessing helps" (lower probability but valid)

To generalize the calculations on any number N of contiguous sequences:

$$P(w_{1:N}) = \prod_{j=1}^{N} P(w_j \mid w_{1:j-1}).$$

N-gram models work well for classifying newspaper sections, as well as for other classification tasks such as **spam detection** (distinguishing spam email from non-spam), **sentiment analysis** (classifying a movie or product review as positive or negative) and **author attribution** (Hemingway has a different style and vocabulary than Faulkner or Shakespeare).

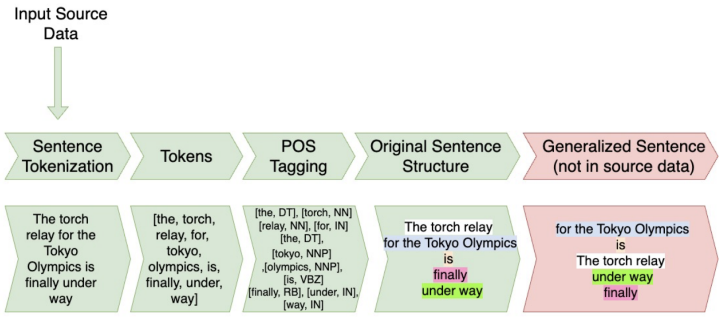
23.1.3 Other n-gram models

An alternative to an n-gram word model is a character-level model in which the probability of each character is determined by the $n-1$ previous characters. This approach is helpful for dealing with unknown words, and for languages that tend to run words together.

Another possibility is the **skip-gram** model, in which we count words that are near each other, but skip a word (or more) between them.

23.1.6 Part-of-speech (POS) tagging

A statistical language model such as N-grams suffers when it needs to generalize beyond the data it calculates frequencies from. The model can auto-complete a sentence only based on what was available in its source dataset. To help overcoming the generalization limitation, more knowledge should be modelled as well. Hence, a structured representation of the model source dataset is recommended. **POS** is an example for such structured information. One basic way to categorize words is by their part of speech (POS), also called **lexical category** or tag: noun, verb, adjective, and so on. Parts of speech allow language models to capture generalizations such as "adjectives generally come before nouns in English."



23.3 Parsing

Parsing is the process of analyzing a string of words to uncover its phrase structure, according to the rules of a grammar. We can think of it as a search for a valid parse tree whose leaves are the words of the string.

To avoid this source of inefficiency we can use **dynamic programming:** every time we analyze a substring, store the results so we won't have to reanalyze it later.

23.6 Natural Language Tasks

Speech recognition is the task of transforming spoken sound into text. We can then perform further tasks (such as question answering) on the resulting text. Current systems have a word error rate of about 3% to 5% (depending on details of the test set), similar to human transcribers. The challenge for a system using speech recognition is to respond appropriately even when there are errors on individual words.

Text-to-speech synthesis is the inverse process—going from text to sound. The challenge is to pronounce each word correctly, and to make the flow of each sentence seem natural, with the right pauses and emphasis. Another area of development is in synthesizing different voices—starting with a choice between a generic male or female voice, then allowing for regional dialects, and even imitating celebrity voices

Machine translation transforms text in one language to another. Systems are usually trained using a bilingual corpus: a set of paired documents, where one member of the pair is in, say, English, and the other is in, say, French. The documents do not need to be annotated in any way; the machine translation system learns to align sentences and phrases and then when presented with a novel sentence in one language, can generate a translation to the other.

Information extraction is the process of acquiring knowledge by skimming a text and looking for occurrences of particular classes of objects and for relationships among them. A typical task is to extract instances of addresses from Web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation.

Information retrieval is the task of finding documents that are relevant and important for a given query. Internet search engines such as Google and Baidu perform this task billions of times a day.

Question Answering is a different task, in which the query really is a question, such as "Who founded the U.S. Coast Guard?" and the response is not a ranked list of documents but rather an actual answer: "Alexander Hamilton."

24 Deep Learning for Natural Language Processing

24.1 Word Embeddings

Word Embeddings: a representation of words that does not require manual feature engineering, but allows for generalization between related words:

- "colorless" and "ideal" are both adjectives, a syntax that can be learned automatically
- "cat" and "kitten" are both felines, a semantic that can be learned automatically
- "awesome" has opposite sentiment to "cringeworthy", a sentiment that can be learned automatically

Traditionally, a word can be represented by a vector that hopefully capture its contextual meaning and how to capture the latter from the words that often come within its context:

- Following the linguist John R. Firth's (1957) maxim, "You shall know a word by the company it keeps"
- We would get better generalization if we reduced this to a smaller-size vector, perhaps with just a few hundred dimensions. We call this smaller, dense vector: a word embedding

Word Embeddings are learned automatically. They capture implicit relationships also, such as aunt to niece is same as uncle to nephew. They solve the computational overload in statistical NLP approaches.

11 Automated Planning

11.1 Definition of Classical Planning

Classical planning is defined as the task of finding a sequence of actions to accomplish a goal in a discrete, deterministic, static, fully observable environment. In response to the limitations of problem solving agents and propositional logical agents, planning researchers have invested in a **factored representation** using a family of languages called **PDDL**, the Planning Domain Definition Language. This allows us to express all $4Tn^2$ actions with a single action schema, and does not need domain-specific knowledge. Basic PDDL can handle classical planning domains, and extensions can handle non-classical domains that are continuous, partially observable, concurrent, and multi-agent.

In PDDL, a **state** is represented as a conjunction of ground atomic fluents. Recall that "ground" means no variables, "fluent" means an aspect of the world that changes over time, and "ground atomic" means there is a single predicate, and if there are any arguments, they must be constants. PDDL uses **database semantics**: the closed-world assumption means that any fluents that are not mentioned are false.

An **action schema** represents a family of ground actions. For example, here is an action schema for flying a plane from one location to another:

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

The schema consists of the action name, a list of all the variables used in the schema, a precondition and an effect. The **precondition** and the **effect** are each conjunctions of literals (positive or negated atomic sentences). A ground action a is **applicable** in state s if s entails the precondition of a; that is, if every positive literal in the precondition is in s and every negated literal is not.

The result of executing applicable action a in state s is defined as a state s' which is represented by the set of fluents formed by starting with s, removing the fluents that appear as negative literals in the action's effects (what we call the **delete list** or DEL(a)), and adding the fluents that are positive literals in the action's effects (what we call the **add list** or ADD(a)).

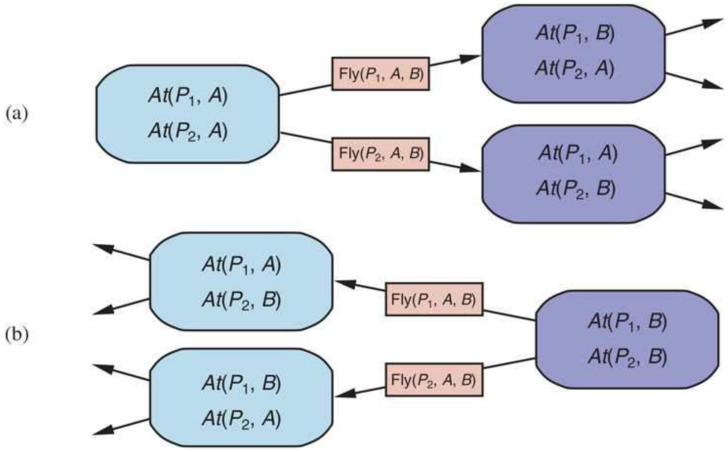
11.1.1 Example domain: Air cargo transport

11.1.2 Example domain: The spare tire problem

11.1.3 Example domain: The blocks world

11.2 Algorithms for Classical Planning

The description of a planning problem provides an obvious way to search from the initial state through the space of states, looking for a goal. A nice advantage of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state. A third possibility is to translate the problem description into a set of logic sentences, to which we can apply a logical inference algorithm to find a solution.



Two approaches to searching for a plan. (a) Forward (progression) search through the space of ground states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through state descriptions, starting at the goal and using the inverse of the actions to search backward for the initial state.

11.2.1 Forward state-space search for planning

We can solve planning problems by applying any of the heuristic search algorithms.
How to plan:

1. Determine all actions applicable
2. Ground the actions by replacing any variable with constants
3. Choose an action to apply
4. Determine the new state of the world and update the knowledge based according to the action description
5. Repeat this process until the goal state is reached

Forward search can have a very large branching factor. Many applicable dumb actions that do not progress towards our goal. The search algorithms we have covered can waste a lot of time here. It needs a good (domain-specific) heuristic or pruning procedure to work efficiently.

11.2.2 Backward search for planning

How to plan:

1. Choose a relevant action that satisfies (some) goal propositions
2. Make a new goal by applying an action a backwards:
    - DEL satisfied conditions of goal

- ADD preconditions of a
- Keep unsolved goal propositions

3. Repeat until the goal is satisfied by the start state

Backward search is not guided towards any specific subgoal. The order in which we try to achieve the subgoals (and do search) matters. It impacts the efficiency of the search. A wrong order can make the plan unfeasible.

11.3 Heuristics for Planning

Neither forward nor backward search is efficient without a good heuristic function. By definition, there is no way to analyze an atomic state, and thus it requires some ingenuity by an analyst (usually human) to define good domain-specific heuristics for search problems with atomic states. But planning uses a factored representation for states and actions, which makes it possible to define good domain-independent heuristics.

Recall that an admissible heuristic can be derived by defining a relaxed problem that is easier to solve. General idea:

- Add edges and group nodes (subplans)
- Ignore restrictions (all or some)
- Ignore negative fluents
- Weigh actions to have preferred actions
- Find serialisable subplans

11.3.1 Domain-independent pruning

Factored representations make it obvious that many states are just variants of other states. Consider a case where all states are symmetric: choosing one over another makes no difference, and thus a planner should only consider one of them. This is the process of **symmetry reduction:** we prune out of consideration all symmetric branches of the search tree except for one. For many domains, this makes the difference between intractable and efficient solving.

Another possibility is to do forward pruning, accepting the risk that we might prune away an optimal solution, in order to focus the search on promising branches. We can define a **preferred action** as follows: First, define a relaxed version of the problem, and solve it to get a **relaxed plan**. Then a preferred action is either a step of the relaxed plan, or it achieves some precondition of the relaxed plan.

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has **serializable**

subgoals if there exists an order of subgoals such that the planner can achieve them in that order without having to undo any of the previously achieved subgoals. As an example, if there is a room with n light switches, each controlling a separate light, and the goal is to have them all on, then we don't have to consider permutations of the order; we could arbitrarily restrict ourselves to plans that flip switches in, say, ascending order.

11.3.2 State abstraction in planning

A relaxed problem leaves us with a simplified planning problem just to calculate the value of the heuristic function. Many planning problems have 10000 states or more, and relaxing the actions does nothing to reduce the number of states, which means that it may still be expensive to compute the heuristic. Relaxations that decrease the number of states by forming a state abstraction—a many-to-one mapping from states in the ground representation of the problem to the abstract representation is the solution. Examples of relaxation:

- State abstraction: group subtasks into one bigger, more abstract task.
  - All packages for Trondheim (from Oslo) count as one big package
  - Worry about getting those packages to Trondheim first. You can worry about individual deliveries afterwards.
  - Decompose costs: $Cost(P) = Cost(Pi) + Cost(Pj)$ where i, j are subgoals.
- Ignore restrictions: 'a perfect plan if this road r were not closed'. We will try finding an alternative route for the r segment later.
- Serialisable subplans: achieving a subgoal (putting on shoe1) does not interfere with other goals (putting on shoe2)

11.4 Hierarchical Planning

Here, we concentrate on the idea of hierarchical decomposition, an idea that pervades almost all attempts to manage complexity. For example, complex software is created from a hierarchy of subroutines and classes; armies, governments and corporations have organizational hierarchies. The key benefit of hierarchical structure is that at each level of the hierarchy, a computational task, military mission, or administrative function is reduced to a small number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small.

11.4.1 High-level actions

The basic formalism we adopt to understand hierarchical decomposition comes from the area of **hierarchical task networks** or HTN planning. For now we assume full observability and determinism and a set of actions, now called

primitive actions, with standard precondition–effect schemas. The key additional concept is the **high-level action** or HLA. Each HLA has one or more possible **refinements**, into a sequence of actions, each of which may be an HLA or a primitive action.

11.5 Planning and Acting in Nondeterministic Domains

11.5.1 Sensorless planning

I need to make sure that all preconditions are met. I will then carry out all operations that will lead me to the goal. Example: paint a chair and a table with the same colour. How?

$$[RemoveLid(Can), Paint(Chair, Can), Paint(Table, Can)] \tag{36}$$

11.5.2 Contingent planning

I need to make sure to know where I am by looking around. Then, and depending on where I am, I will carry out necessary operations conditionally. Example: paint a chair and a table with the same colour. How?

$$\begin{aligned}\text{ContingencyPlan} = [&\text{RemoveLid(Can), LookAt(Can),}\\ &\text{if Color(Table}, c) \wedge \text{Color(Can}, c) \text{ then Paint(Chair, can)}\\ &\text{else if Color(Chair}, c) \wedge \text{Color(Can}, c) \text{ then Paint(Table, can)}\\ &\text{else [Paint(Chair, Can), Paint(Table, Can)]]}\end{aligned} \tag{37}$$

11.5.3 Online planning

The world is dynamic, and it can change in unpredictable ways.

- Action monitoring: check that all preconditions hold
- Plan monitoring: check if the plan can succeed (or the goal is satisfiable)
- Goal monitoring: check if there is a better goal

JSP: heuristic approach

- Assign the earliest and latest possible start for each action.
- Then duration of the plan is that of the critical path. i.e., the path with the longest duration. There cannot be a shorter plan than this.
- The JSP is an NP-problem. The optimisation version (finding how) is an NP-hard problem.

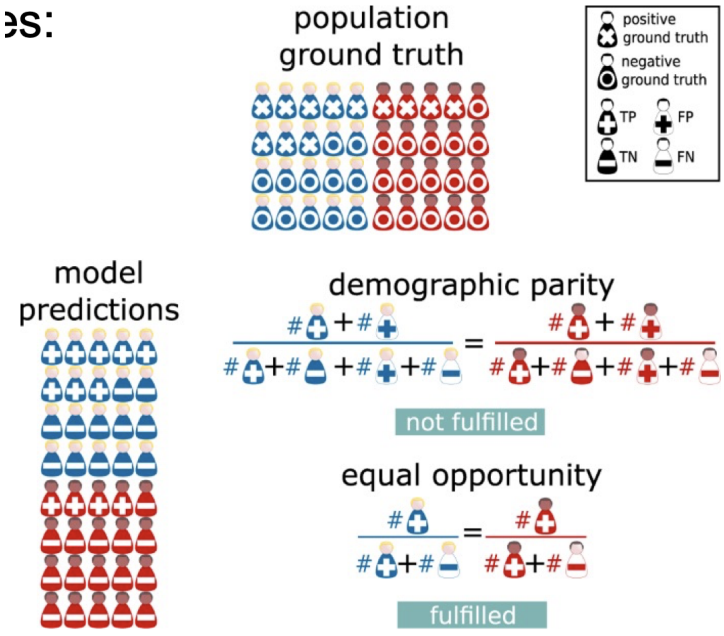
27 Philosophy, Ethics, and Safety of AI

The Philosophy of AI

Fairness and Bias in AI

Philosophers claim that a machine that acts intelligently would not be actually thinking, but would be only a simulation of thinking. Most AI researchers are not concerned with the distinction.
The Importance of Understanding the Terminologies:

- Individual Fairness
- Group Fairness (Demographic Parity)
- Fairness Through Unawareness
- Equal Outcome
- Equal Opporotunity
- Equal Impact

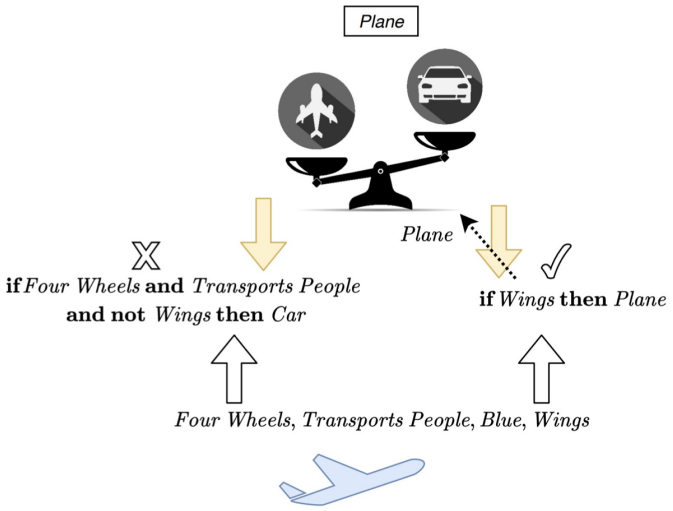


Trust and Transparency

Being fair is not even enough, an AI system needs to convince users that it did the Job in a fair way:

- Can we trust an AI to suggest launching a rocket without justification for its decision?
- A self-driven car must be technically tested in a particular manner to ensure safety.
- How the AI accesses the data must be governed responsibly.
- We need to measure the AI uncertainty.

An example of a transparent AI model for classification :



- Suppose we need an AI model to classify transportation related documents
- The model is then transparent if it can provide why it did particular classifications
- One approach is to trace the text features (important words) that describe the kind of transportation
- Then, the model can assign its output to some supportive statements that describe how/ why such output was concluded
- Tsetlin Machines (TMs) is a good example for transparent classification
- TMs' main building blocks are learning automata, similar to finite state automata but with learning mechanisms
- Each automaton can be assigned to a feature in the data, trying to learn either to include that feature or exclude it in its classification decision

- The include/ exclude means that a feature with a particular value in a document can be a proof of its class, while having another value can also be a proof
- The TMs conduct some calculations to estimate the probabilities for inclusion/ exclusion over the features
- In TMs, data features must be converted into binary. E.g., One-Hot-Encoding vector
- Hence, each feature will have either 1, or 0, means a True or False value, respectively
- The TMs final output can then be propositional logic clauses in the form: X1 AND X2 AND NOT X3, for the features X1, X2, X3

Uncertainty Quantification

A crucial characteristic of modern AI systems in the industry. Important in risk management, being transparent about the risk of a decision. We need to know how likely/ unlikely the AI calculated probabilities are calculated with confidence. The key is to conduct simulations and monitor the variations of the AI model on multiple runs/ scenarios.

Example of two AI agents (Reinforcement Learning) with different uncertainty on the same task (Misinformation Mitigation):
We can calculate the Shannon Entropy for two probability distributions from the two models:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$$

where:

- $H(X)$ is the entropy.
- $p(x_i)$ is the probability of outcome x_i.
- $\log_2$ denotes the logarithm to base 2.

Ethical Dilemmas in Large Language Models (LLMs)

Hallucination rate is currently high.
The paradox of Cyber Security threats and improved LLM architecture.

- Use input text to the LLM to search external documents.
- External documents can be updated without training.
- Reduce hallucination.

- However, vulnerable to poisoning of the external knowledge.

Mis(Dis)information in Large Language Models (LLMs):

- Training a threat model, not for good.
- Unintentionally generation of misinformation.
- Data Pollution is a main reason
- Intentionally produce disinformation

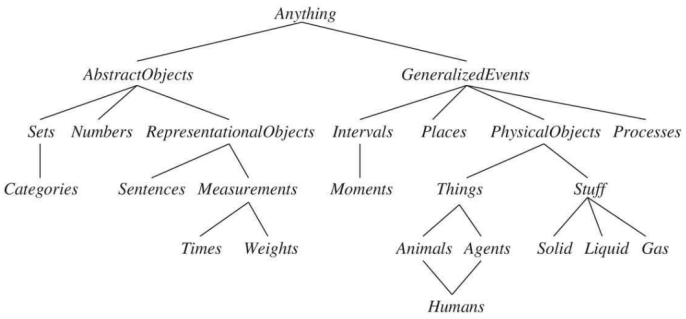
10 Knowledge Representation

Knowledge Representation Concept (First-Order-Logic). First-Order Logic (FOL) is a formal system used in mathematics, philosophy, linguistics, and computer science. It is a powerful tool for representing and reasoning about objects and their relationships:

- Variables: Represent objects in the domain.
- Constants: Represent specific objects
- Predicates: Represent properties or relationships between objects
- Quantifiers: Existential and universal quantifiers to express statements about some or all objects
- Functions: Map objects to other objects
- Logical Connectives: Such as AND, OR, NOT, IMPLIES, and EQIVALENT.

10.1 Ontological Engineering

Ontologies are formal representations of a set of concepts within a domain and the relationships between concepts. They leverage the formal definitions from FOL to model complex domains in a structured and interpretable way:



The upper ontology of the world, showing the topics to be covered later in the chapter. Each link indicates that the lower concept is a specialization of the upper one. Specializations are not necessarily disjoint—a human is both an animal and an agent. We will see in Section 10.3.2 why physical objects come under generalized events.

Word Embeddings Representation in LLMs

Neural Network-based approach improves a language model and helps reducing manual feature engineering in the source datasets and allow for more generalization through some techniques. Word Embeddings: a representation of words that does not require manual feature engineering, but allows for generalization between related words:

- "colorless" and "ideal" are both adjectives, a syntax that can be learned automatically
- "cat" and "kitten" are both felines, a semantic that can be learned automatically
- "awesome" has opposite sentiment to "cringeworthy", a sentiment that can be learned automatically

Ethical Representation in LLMs (Word Embeddings Representation)

Ethical representations in word embeddings aim to mitigate biases and ensure fair and just representation of concepts in language models.

Scenario:

You are developing a word embedding model to be used in a job recommendation system. The goal is to ensure that the embeddings do not perpetuate gender biases, particularly in job-related contexts.

Problem:

Traditional word embeddings like Word2Vec and GloVe have been found to exhibit gender biases. For instance, embeddings might associate "man" with "computer programmer" and "woman" with "homemaker," reflecting and potentially reinforcing stereotypes.

Steps for Ethical Representation:

1. **Identify Biases:**
    - Use techniques like the Word Embedding Association Test (WEAT) to quantify biases in the embeddings. This involves comparing associations between gendered words (e.g., "man," "woman") and occupation-related words (e.g., "engineer," "nurse").
2. **Debiasing Techniques:**
    - **Hard Debiasing:**
        - Identify a gender subspace using pairs of gendered words (e.g., "he-she," "man-woman").
        - Project occupation-related words to remove their components along this subspace.
    - **Soft Debiasing:**
        - Modify the training algorithm to reduce bias gradually while maintaining the utility of the embeddings.

After hard debiasing, non-gender-specific concepts (in black) are more equidistant to genders:

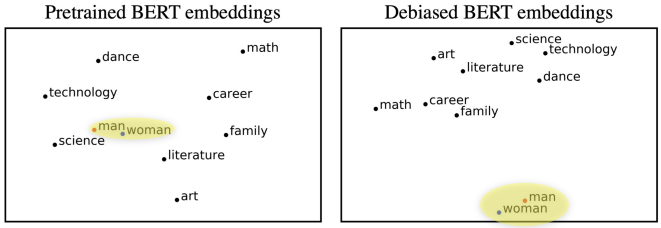


Figure: Plots of average sentence representations of a word across its sentence templates before (left) and after (right) debiasing. After debiasing, non gender-specific concepts (in black) are more equidistant to genders – Authors: Liang, Paul Pu, et al. 2020

Anonymized Ontologies

Data anonymization techniques are essential for protecting personal privacy while retaining the utility of the data for analysis and machine learning. **Common techniques used in anonymization:**

- Data Masking
    - Original Data: 1234-5678-9012-3456
    - Masked Data: XXXX-XXXX-XXXX-3456
- Pseudonymization
    - Original Data: John Doe
    - Pseudonymized: User12345

- Swapping
  - Original Data: Person A: 01/01/1980, Person B: 02/02/1990
  - Swapped: Person A: 02/02/1990, Person B: 01/01/1980
- Noise Addition
  - Original Data: $50,000
  - Noise Added: $50,000 + random_value
- Encryption
  - Original Data: MySecretData
  - Encrypted Data: U2FsdGVkX1+P/4nB6UoF0J6H...

Eksamensoppgaver etter tema:

Miscelleneous short questions:

- 2023: 1 (Done)
- 2022: 1 (Done)
- 2021: 6 (Done)
- 2020: 6 (Not relevant)
- 2019: 1 (Done)
- 2019: 7 (Done)
- 2018: 3 (Done)
- 2018: 8 (Done)
- 2017: 6 (Done)
- 2017: 7 (Done)
- 2015: 1 (Done)
- 2015: 6 (Done)
- 2014: 1 (Done)
- 2014: 2 (Done)
- 2013: 1 (Done)
- 2012: 2 (Done) (Gjort)

Intelligent Agents: (Done)

- 2022: 2 (Done)(Gjortx2)
- 2016: 1 (Done)(Gjort)
- 2013: 2 (Done)(Gjort)
- 2012: 1 (Done)(Gjort)

Uniformed and informed search

- 2023: 2 (Done)(Gjortx2)
- 2022: 3 (Lite relevant til flervalg)
- 2021: 2 (Done)(Gjort)
- 2019: 3 (Done)(Gjort)
- 2018: 4 (Done)(Gjort)
- 2018: 5 (Done)(Gjort)
- 2017: 4 (Done)(Gjort)
- 2016: 3 (Done)
- 2015: 7 (Done)(Gjort)
- 2014: 6 (Done)(Gjort)
- 2013: 5 (Done)
- 2013: 6 (Done)
- 2012: 5 (Done)(Gjort)

Search in complex environments

- 2023: 7 (Done)(Gjort)

Constraint satisfaction problem

- 2023: 4 (Done) (Lik som 2018 med bedre LF)(Gjortx2)
- 2022: 4 (Se senere)
- 2021: 3 (Se senere)
- 2019: 4 (Done)(Gjort)
- 2018: 1 (Vanskelig, se på senere)

- 2017: 1 (Done)(Gjort)
- 2016: 4 (Done)(Gjort)
- 2012: 7 (Done)(Gjort)

Logic

- 2023: 3 (Done) (Gjortx2)
- 2022: 5 (Done) (Gjort)
- 2021: 1 (Done) (Gjort)
- 2019: 2 (Done) (Gjort)
- 2018: 6 (Done) (Gjort)
- 2018: 9 (Done) (Gjort)
- 2017: 2 (Done) (Gjort)
- 2017: 5 (Done) (Gjort)
- 2016: 2 (Done) (Check again later) (Gjort)
- 2015: 3 (Done) (Gjort)
- 2014: 3 (Done) (Gjort, men gjør igjen senere)
- 2014: 4 (Done) (Gjort)
- 2013: 3 (Done) (Gjort)
- 2013: 4 (Done) (Gjort)
- 2012: 3 (Done) (Gjort, men gjør igjen senere)
- 2012: 6 (Done) (Gjort)

Adversarial search

- 2023: 6 (Done) (Gjortx2)
- 2021: 5 (-———-)
- 2019: 6 (Done) (Gjort)
- 2018: 2 (-———-)(Gjort)
- 2017: 3 (-———-)(Gjort)
- 2015: 8 (Done) (Gjort)

- 2014: 5 (Done) (Gjort)
- 2013: 8 (Done) (Gjort)
- 2012: 4 (Done) (Gjort)

Game theory

(Skip)

- 2024: 5
- 2019: 5
- 2018: 7

Natural Language Processing

- 2016: 6 (Done: Not all subtasks relevant)(Gjort)
- 2014: 7 (Done) (Gjort)

Planning

- 2023: 8 (Done) (Gjortx2)
- 2022: 6 (Kaos)(Hahah gleder meg)
- 2021: 4 (Done) (Gjort)
- 2016: 5 (Done) (Gjort)
- 2015: 5 (Not curriculum)
- 2013: 10 (Not curriculum)
- 2012: 8 (Not curriculum)

4.5 Mixed problems:

- 2015: 2
- 2015: 4
- 2013: 9
- 2012: 9
- 2012: 10(Gjort)

5 Summarizing problems to go through:

- BFS (Done)
- DFS (Done)
- UCS (Done)
- A* (Done)
- GBFS (Done)
- Do problem for search where cycles are present to note which works and not (Done)
- CSP: All problems (Done)
- Adverserial search: Minimax, Alpha-Beta Pruning, Expected minimax (Done)
- Game Theory: Look at all examples and learn calculations from lecture note (Done)
- PPDL representation, and how to use in forward and backward inference, as well as partial ordering (Done)
- Propositional Logic: All tasks (difficult)
- Read through Compendium on topics not in calculations: Check summary lecture for important topics here (NLP etc.)