AI Notes

Riccardo Cappi

January 2024

0.1 Disclaimer

These are just my notes that I used to prepare for the exam. So, probably, there will be both spelling and conceptual errors. Feel free to contact me at riccardo.cappi@studenti.unipd.it if you find any errors. This is the github repo where you can find the latex files of the notes: `https://github.com/riccardocappi/Computer-Science-notes`

Contents

Chapter 1

Lec 01 - Rational Agents

1.1 Intelligent (or rational) agent

An **agent** is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. A human agent has eyes, ears, and other organs for sensors and hands, legs, vocal tract, and so on for actuators. A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators. A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

In the context of the course, a **rational agent**, tries to achieve its goals as much as possible given the available information.

Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

$$f : P^* \to A$$

Among all the classes of environments and tasks, we look for the agent (or class of agents) with the best performance. We can imagine tabulating the agent function that describes any given agent; for most agents, this would be a very large table, infinite, unless we place a bound on the length of percept sequences we want to consider. Computational limitations prevent the realization of perfect rationality.

Internally, the agent function for an artificial agent will be implemented by an **agent program**. 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.

To illustrate these ideas, we use a very simple example, **the vacuum-cleaner world**: This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck; otherwise, move to the other square. This agent function can be represented as a table:

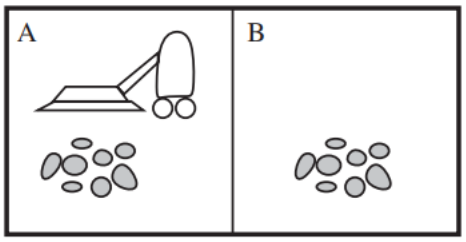


Figure 2.2 A vacuum-cleaner world with just two locations.

Percept sequence	Action
$[A, Clean]$	*Right*
$[A, Dirty]$	*Suck*
$[B, Clean]$	*Left*
$[B, Dirty]$	*Suck*
$[A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Dirty]$	*Suck*
⋮	⋮
$[A, Clean], [A, Clean], [A, Clean]$	*Right*
$[A, Clean], [A, Clean], [A, Dirty]$	*Suck*
⋮	⋮

Figure 2.3 Partial tabulation of a simple agent function for the vacuum-cleaner world shown in Figure 2.2.

Various vacuum-world agents can be defined simply by filling in the right hand column in various ways. The obvious question, then, is this: What is the right way to fill out the table?

1.2 Rationality

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. A **rational agent** chooses any action that maximizes the expected value of the performance measure given the sequence of perceptions obtained up to the current instant.

Obviously, there is not one fixed performance measure for all tasks and agents; typically, a designer will devise one appropriate to the circumstances:

- +1 for each clean space in time T ?
- +1 for each clean space per instant of time, -1 for movement ?
- ...

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. Rationality maximizes expected performance, while perfection maximizes actual performance. A rational agent should be able to **gather information** (exploration), **learn** as much as possible from what it perceives and be **autonomous**.

1.3 PEAS

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.

Therefore, in designing an agent, we had to specify the performance measure, the environment, and the agent's actuators and sensors. We group all these under the heading of the **task environment**. For the acronymically minded, we call this the **PEAS** (Performance, Environment, Actuators, Sensors) description. In designing an agent, the first step must always be to specify the task environment as fully as possible.

Consider, for example, the task of designing an automated taxi:

Agent Type	Performance Measure	Environment	Actuators	Sensors
Taxi driver	Safe, fast, legal, comfortable trip, maximize profits	Roads, other traffic, pedestrians, customers	Steering, accelerator, brake, signal, horn, display	Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard

Figure 2.4 PEAS description of the task environment for an automated taxi.

1.3.1 Tasks environments

The range of task environments that might arise in AI is obviously vast. We can, however, identify a fairly small number of dimensions along which task environments can be categorized:

- **Fully observable** vs. **partially observable**: Whether an agent's sensors give it access to the complete state of the environment at each point in time or not.
- **Single agent** vs. **multiagent**
- **Deterministic** vs. **stochastic**: Whether the next state of the environment is completely determined by the current state and the action executed by the agent or not.
- **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. Many classification tasks are episodic. In sequential environments, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential.
- **Static** vs. **dynamic**: 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. Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing.

- **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 (excluding the clock). Chess also has a discrete set of percepts and actions. Taxi driving is a continuous-state and continuous-time problem. Input from digital cameras is discrete, strictly speaking, but is typically treated as representing continuously varying intensities and locations.
- **Known** vs. **unknown**: In a known environment, the outcomes (or outcome probabilities if the environment is stochastic) 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. Note that the distinction between known and unknown environments is not the same as the one between fully and partially observable environments. It is quite possible for a known environment to be partially observable; for example, in solitaire card games, I know the rules but am still unable to see the cards that have not yet been turned over. Conversely, an unknown environment can be fully observable—in a new video game, the screen may show the entire game state but I still don't know what the buttons do until I try them.

Task Environment	Observable	Agents	Deterministic	Episodic	Static	Discrete
Crossword puzzle	Fully	Single	Deterministic	Sequential	Static	Discrete
Chess with a clock	Fully	Multi	Deterministic	Sequential	Semi	Discrete
Poker	Partially	Multi	Stochastic	Sequential	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Sequential	Static	Discrete
Taxi driving	Partially	Multi	Stochastic	Sequential	Dynamic	Continuous
Medical diagnosis	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Image analysis	Fully	Single	Deterministic	Episodic	Semi	Continuous
Part-picking robot	Partially	Single	Stochastic	Episodic	Dynamic	Continuous
Refinery controller	Partially	Single	Stochastic	Sequential	Dynamic	Continuous
Interactive English tutor	Partially	Multi	Stochastic	Sequential	Dynamic	Discrete

Figure 2.6 Examples of task environments and their characteristics.

The type of environment largely determines the design of the agent. The real world is, of course, partially observable, stochastic, sequential, dynamic, continuous, multi-agent.

1.4 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 architecture**:

$$agent = architecture + program$$

Obviously, the program we choose has to be one that is appropriate for the architecture. If the program is going to recommend actions like Walk, the architecture had better have legs. The architecture might be just an ordinary PC.

1.4.1 Agent programs

The agent programs that we'll design have the same skeleton: they take the current percept as input from the sensors and return an action to the actuators. We can design a table-driven agent as follows:

```
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 2.7 The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns an action each time. It retains the complete percept sequence in memory.

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.

Four kinds of agents can be defined in general:

- Simple reflex agents;
- Model-based reflex agents;
- Goal-based agents; and
- Utility-based agents

They differ in the internal components they use to generate actions. All these types of agents can be made into agents who learn.

Simple reflex agents

The simplest kind of agent is the simple reflex agent. These agents select actions on the basis of the current percept, ignoring the rest of the percept history. Their behavior is defined by a set of condition-action rules, for example, **if** car-in-front-is-braking **then** initiate-braking. One example of this kind of agent is the vacuum-agent presented before.

Simple reflex agents have the admirable property of being simple, but they turn out to be of limited intelligence. If the environment is partially observable it can fail.

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. First, we need some information about how the world evolves independently of the agent. Second, we need some information about how the agent's own actions affect the world. This knowledge about "how the world works" is called a **model** of the world. An agent that uses such a model is called a **model-based agent**. For example, an automated taxi may not be able to see around the large truck that has stopped in front of it and can only guess about what may be causing the hold-up. Thus, uncertainty about the current state may be unavoidable, but the agent still has to make a decision.

Goal-based agents

Knowing something about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. In other words, as well as a current state description, the agent needs some sort of goal information that describes situations that are desirable, for example, being at the passenger's destination. The agent program can combine this with the model to choose actions that achieve the goal.

Goal-based agents are more **flexible** because they can automatically adapt their relevant behaviors to suit new conditions. For example, If it starts to rain, the agent can update its knowledge of how effectively its brakes will operate. For the reflex agent, on the other hand, we would have to rewrite many condition–action rules.

Utility-based agents

Goals alone are not enough to generate high-quality behavior in most environments. For example, many action sequences will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others. 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.

We have already seen that a performance measure assigns a score to any given sequence of environment states, so it can easily distinguish between more and less desirable ways of getting to the taxi's destination. An agent's **utility function** is essentially an internalization of the performance measure. If the internal utility function and the external performance easure are in agreement, then an agent that chooses actions to maximize its utility will be rational according to the external performance measure.

Like goal-based agents, a utility-based agent has many advantages in terms of flexibility and learning.

Furthermore, in two kinds of cases, goals are inadequate but a utility-based agent can still make rational decisions. First, when there are conflicting goals, only some of which can be achieved (for example, speed and safety), the utility function specifies the appropriate **tradeoff**. Second, when there are several goals that the agent can aim for, none of which can be achieved with certainty, utility provides a way in which the likelihood of success can be weighed against the importance of the goals.

Learning agents

We have described agent programs with various methods for selecting actions. We have not, so far, explained how the agent programs come into being.

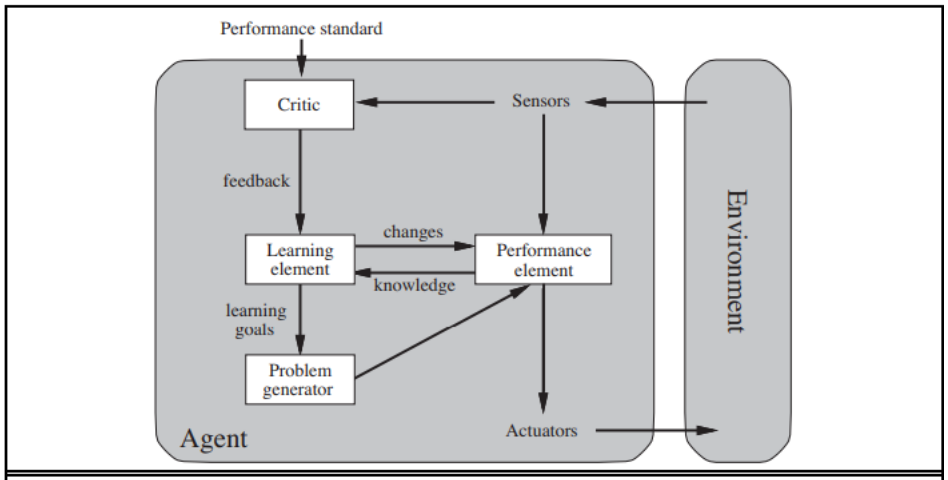


Figure 2.15 A general learning agent.

A learning agent can be divided into four conceptual components. The most important distinction is between the **learning element**, which is responsible for making improvements, and the **performance element**, which is responsible for electing 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. The critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The last component of the learning agent is the **problem generator**. It is responsible for suggesting actions that will lead to new and informative experiences. Basically, it is necessary to improve the generalization of the agent's behaviors.

Representing the environment

We can place the environment representations along an axis of increasing complexity and expressive power: **atomic**, **factored**, and **structured**.

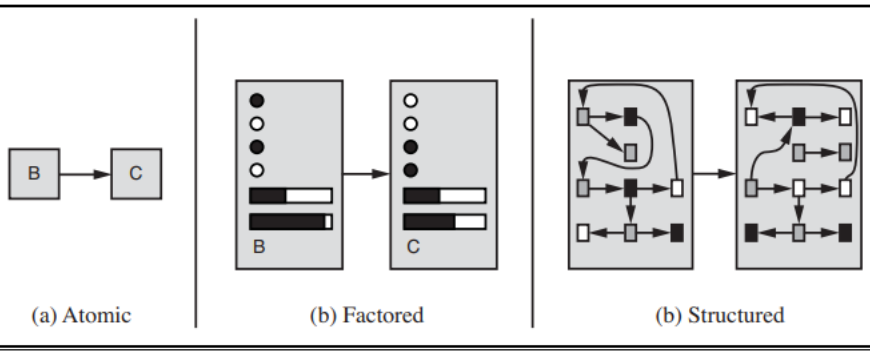


Figure 2.16 Three ways to represent states and the transitions between them. (a) Atomic representation: a state (such as B or C) is a black box with no internal structure; (b) Factored representation: a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed 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.

- In an **atomic representation** each state of the world is indivisible.

- **factored representation**: splits up each state into a fixed set of variables or attributes, each of which can have a value.
- **structured representation**: in which objects and their various and varying relationships can be described explicitly

Chapter 2

Lec 02 - Problem Solving Agent

2.1 Solving Problems by Searching

We'll describe one kind of **goal-based** agent called a problem-solving agent. Our discussion of problem solving begins with precise definitions of **problems** and their **solutions** and give several examples to illustrate these definitions. We then describe several general-purpose search algorithms that can be used to solve these problems. We will see several **uninformed** search algorithms, algorithms that are given no information about the problem other than its definition. Let's start with an example:

Imagine an agent in the city of Arad, Romania, enjoying a touring holiday. Now, suppose the agent has a nonrefundable ticket to fly out of Bucharest the following day. In that case, it makes sense for the agent to adopt the **goal** of getting to Bucharest by driving across various cities. Therefore, the **problem formulation** can be defined as follows:

- Formulate goal: be in Bucharest
- Formulate problems:
  - states: various cities
  - actions: drive between cities
- Solution: sequence of cities; e.g. Arad, Sibiu, Fagaras, Bucharest

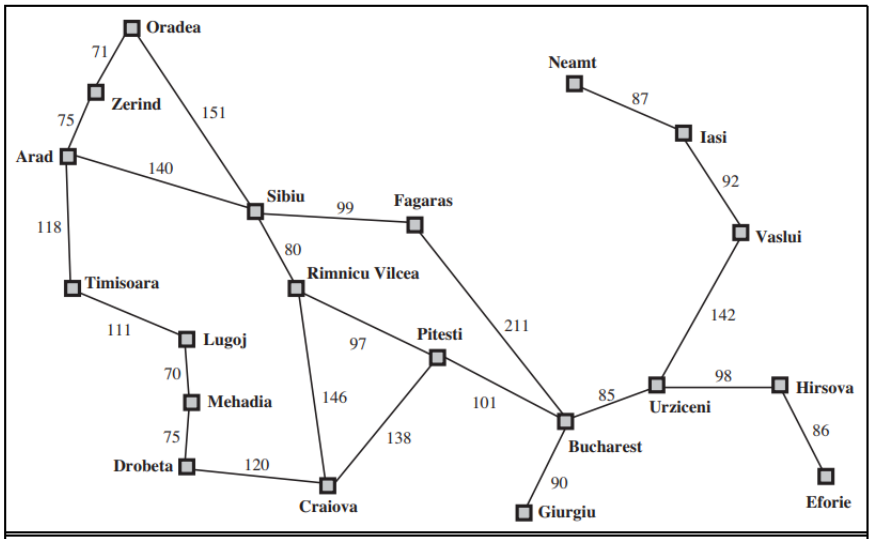


Figure 3.2 A simplified road map of part of Romania.

We assume that the environment is **observable**, so the agent always knows the current state. For the agent driving in Romania, it's reasonable to suppose that each city on the map has a sign indicating its presence to arriving drivers. We also assume the environment is **discrete**, so at any given state there are only finitely many actions to choose from. This is true for navigating in Romania because each city is connected to a small number of other cities. We will assume the environment is **known**, so the agent knows which states are reached by each action. (Having an accurate map suffices to meet this condition for navigation problems.) Finally, we assume that the environment is **deterministic**, so each action has exactly one outcome.

The process of looking for a sequence of actions that reaches the goal is called **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

A problem can be defined formally by four components:

- The **initial state** that the agent starts in. For example, the initial state for our agent in Romania might be described as *In(Arad)*.

- A description of what each action does; the formal name for this is the **transition model**. We also use the term **successor** to refer to any state reachable from a given state by a single action. This can be described by a set $E(x)$ of action-state pairs, e.g. $E(In(Arad)) = \{< Arad \rightarrow Zerind, Zerind >, ...\}$. An alternative formulation for the successor function can be the actions that can be performed in a given state.

- The **goal test**, which determines whether a given state is a goal state. The goal can be explicit, e.g. the singleton set [1] $\{In(Bucharest)\}$, or implicit, an abstract property rather than an explicitly enumerated set of states.

- A **path cost** function that assigns a numeric cost to each path. The problem-solving agent chooses a cost function that reflects its own performance measure. For the agent trying to get to Bucharest, time is of the essence, so the cost of a path might be its length in kilometers. The **step cost** of taking action a in state s to reach state s' is denoted by $c(s, a, s')$, assumed to be ≥ 0.

The preceding elements define a problem and can be gathered into a single data structure that is given as input to a problem-solving algorithm. A solution to a problem is an action sequence that leads from the initial state to a goal state. Solution quality is measured by the path cost function, and an **optimal solution** has the lowest path cost among all solutions.

2.2 Selecting a State Space

Real world is absurdly complex. Compare the simple state description we have chosen, $In(Arad)$, to an actual crosscountry trip, where the state of the world includes so many things: the traveling companions, the current radio program, the scenery out of the window, the proximity of law enforcement officers, etc. All these considerations are left out of our state descriptions because they are irrelevant to the problem of finding a route to Bucharest. The process of removing detail from a representation is called **abstraction**.

In addition to abstracting the state description, we must abstract the actions themselves.

Now consider a solution to the abstract problem: for example, the path from Arad to Sibiu to Rimnicu Vilcea to Pitesti to Bucharest. This abstract solution corresponds to a large number of more detailed paths.

The choice of a good abstraction thus involves removing as much detail as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

2.3 Searching for Solutions

A solution is an action sequence, so search algorithms work by considering various possible action sequences. The possible action sequences starting at the initial state form a **search tree** with the initial state at the root; the branches are actions and the **nodes** correspond to states in the **state space** of the problem.

[1] There can be multiple goal states

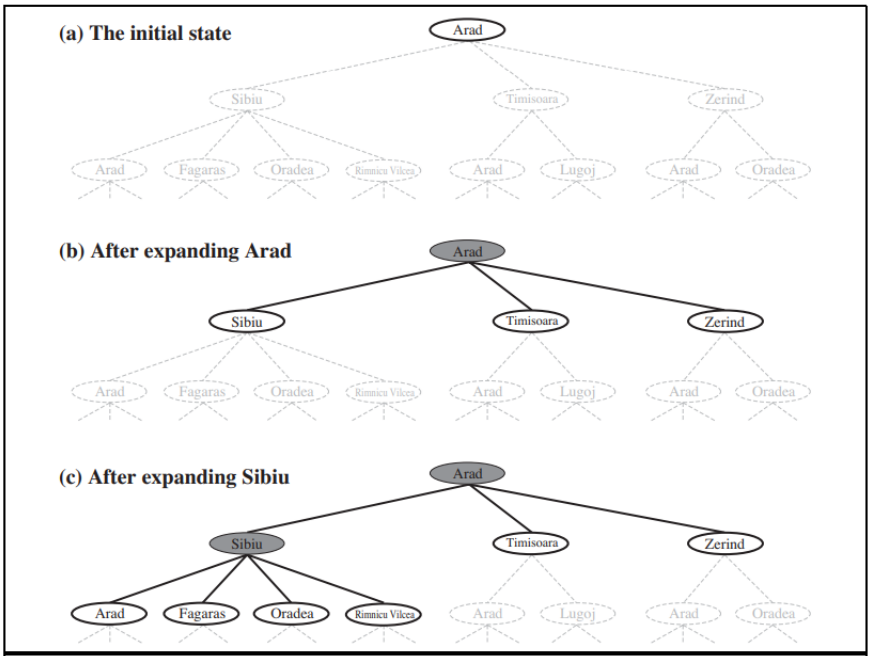


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

We start from the root and, at each step, we need to check whether we reached the goal state. Then, we need to consider taking various actions. We do this by **expanding** the current state, that is, applying each legal action to the current state, thereby **generating** a new set of states. In the case of the figure above, we add three branches from the parent node $In(Arad)$ leading to three new child nodes: $In(Sibiu)$, $In(Timisoara)$, and $In(Zerind)$. Now we must choose which of these three possibilities to consider further.

This is the essence of search, following up one option now and putting the others aside for later. The set of all leaf nodes available for expansion at any given point is called the **frontier**. The process of expanding nodes on the frontier continues until either a solution is found or there are no more states to expand.

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Search algorithms all share the basic structure above; they vary primarily according to how they choose which state to expand next, the so-called search strategy.

In the figure representing the search tree of the *driving through Romania* problem we can notice that it includes the path from Arad to Sibiu and back to Arad again. We say that $In(Arad)$ is a **repeated state** in the search tree, generated in this case by a **loopy path**.

The way to avoid exploring redundant paths is to remember where one has been. To do this, we augment the TREE-SEARCH algorithm with a data structure called the **explored set** (also known as the closed list), which remembers every expanded node. Newly generated nodes that match previously generated nodes can be discarded instead of being added to the frontier. This new algorithm is called GRAPH-SEARCH.

Clearly, the search tree constructed by the GRAPH-SEARCH algorithm contains at most one copy of each state, so we can think of it as growing a tree directly on the state-space graph. Furthermore, the frontier **separates** the state-space graph into the explored region and the unexplored region, so that every path from the initial state to an unexplored state has to pass through a state in the frontier.

Up to now, we have not been very careful to distinguish between nodes and states, but in writing detailed algorithms it's important to make that distinction. A node is a bookkeeping data structure used to represent the search tree. It includes *parent, children, depth, path cost*. A state corresponds to a configuration of the world. Thus, nodes are on particular paths whereas states are not. Furthermore, two different nodes can contain the same world state if that state is generated via two different search paths.

Now that we have nodes, we need somewhere to put them. The frontier needs to be stored in such a way that the search algorithm can easily choose the next node to expand according to its preferred strategy. The appropriate data structure for this is a **queue**.

2.4 Uninformed Search Strategies

Uniformed Search Strategies include the following:

- **Breadth-first search:** Breadth-first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. The frontier is implemented using a FIFO queue.
- **Uniform-cost search:** The idea is to expand least-cost unexpanded node. This is done by storing the frontier as a priority queue ordered by path cost.
- **Depth-first search:** Depth-first search always expands the deepest node in the current frontier of the search tree. the frontier is implemented as a LIFO queue.
- **Iterative deepening depth-first search:** Iterative deepening search (or iterative deepening depth-first search) which repeatedly applies depth first search limiting the depth of the tree. It does this by gradually increasing the limit, first 0, then 1, then 2, and so on, until a goal is found.
- **Bidirectional search:** The idea behind bidirectional search is to run two simultaneous searches—one forward from the initial state and the other backward from the goal—hoping that the two searches meet in the middle.

We can evaluate an algorithm's performance in four ways:

- **Completeness:** Is the algorithm guaranteed to find a solution when there is one?
- **Optimality:** Does the strategy find the optimal solution in terms of cost?
- **Time complexity:** How long does it take to find a solution?
- **Space complexity:** How much memory is needed to perform the search?

The evaluation of the previously presented strategies is the following:

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Notes: [a]complete is b finite; [b]complete if step costs $\geq \epsilon > 0$;
[c]optimal if step costs are all identical; [d]if both directions use breadth-first search.

where:

- b is the branching factor (how many successor a node has)
- d is the depth of the shallowest solution
- m is the maximum depth of the search tree
- l is the depth limit

We can easily see that **Breadth-first search** is complete; if the shallowest goal node is at some finite depth d, breadth-first search will eventually find it after generating all shallower nodes. Note that as soon as a goal node is generated, we know it is the shallowest goal node because all shallower nodes must have been generated already and failed the goal test. However, the shallowest goal node is not necessarily the optimal one. Furthermore, every node generated remains in memory, making the space complexity $O(b^d)$

On the other hand, **Uniform-cost search** is optimal in general. First, we observe that whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found. Then, because step costs are non-negative, paths never get shorter as nodes are added.

The properties of **depth-first search** depend strongly on whether the graph-search or tree-search version is used. The graph-search version, which avoids repeated states and redundant paths, is complete in finite state spaces because it will eventually expand every node. The tree-search version, on the other hand, is not complete (it may get stuck in a loop). The time complexity of depth-first tree search is $O(b^m)$. Note that m itself can be much larger than d. So far, depth-first search seems to have no clear advantage over breadth-first search, so why do we include it? The reason is the space complexity. For a graph search, there is no advantage, but a depth-first tree search needs to store only a single path from the root to a leaf node, along with the remaining unexpanded sibling nodes for each node on the path. Once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored.

We can combine the completeness of BFS and the space saving of DFS with the **Iterative deepening search**. This strategy may seem wasteful because states are generated multiple times. It turns out this is not too costly. The reason is that in a search tree with the same (or nearly the same) branching factor at each level, most of the nodes are in the bottom level, so it does not matter much that the upper levels are generated multiple times.

Chapter 3

Lec 03 - Informed Search

3.1 Problem Solving: Informed Search

The **Informed Search** strategy exploits a priori information on the problem to be solved to perform the search. The general approach we consider is called best-first search, in which a node is selected for expansion based on an evaluation function, $f(n)$. The evaluation function is constructed as an estimate of "desirability", so the node with the highest desirability is expanded first. Then, the frontier can be implemented as a queue sorted in decreasing order of desirability.

The choice of f determines the search strategy. Most best-first algorithms include as a component of f a heuristic function, denoted $h(n)$.

$$h(n) = \textit{estimate of cost from } n \textit{ to the closest goal}$$

Heuristic functions are the most common form in which additional knowledge of the problem is imparted to the search algorithm.

Special cases of best-first algorithms are:

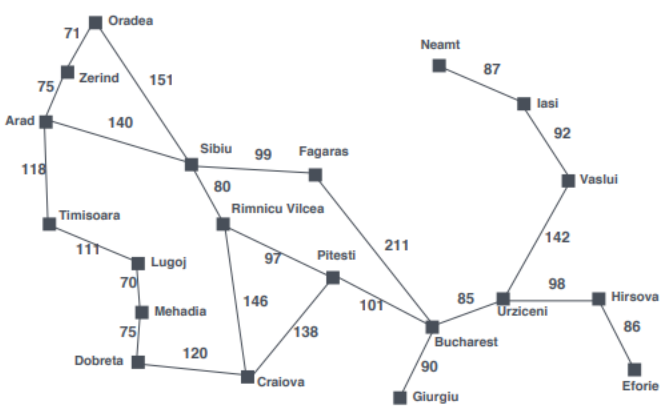
- Greedy search
- A* search

3.2 Greedy search

Greedy best-first search tries to expand the node that is closest to the goal, on the grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function; that is, $f(n) = h(n)$.

In the Romania route-finding problem the heuristic function can be defined as the straight-line distance between a city and Bucharest.

$$h_{SLD}(n) = \textit{straight-line distance from } n \textit{ to Bucharest}$$



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

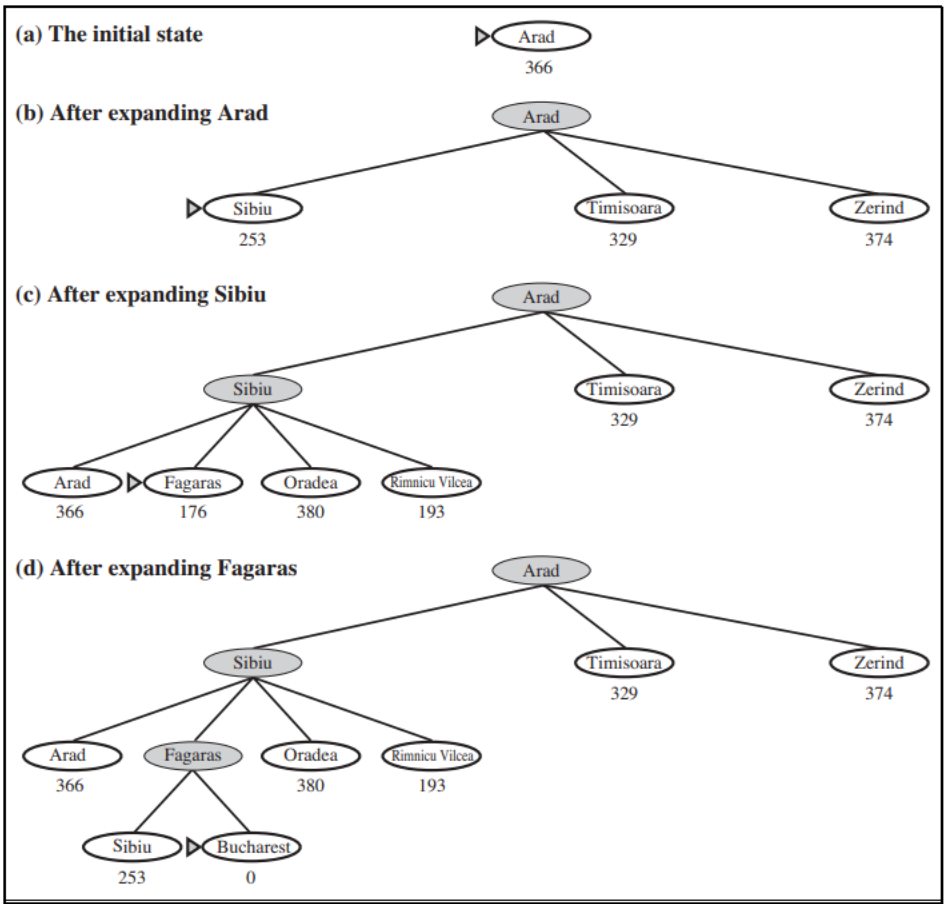


Figure 3.23 Stages in a greedy best-first tree search for Bucharest with the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

The evaluation of the greedy best-first tree search is the following:

- **Complete:** The algorithm is not complete; in fact, it can be stuck in loops (e.g. $In(Iasi) \rightarrow In(Neamt) \rightarrow In(Iasi) \rightarrow In(Neamt) \rightarrow$).

- **Time:** The worst-case time and space complexity for the tree version is $O(b^m)$, where m is the maximum depth of the search space. With a good heuristic function, however, the complexity can be reduced substantially.

- **Space:** $O(b^m)$ because it keeps all nodes in memory.

- **Optimal:** It's not optimal since it is not complete.

The graph search version is complete in finite spaces, but not in infinite ones.

3.3 A* search

The most widely known form of best-first search is called **A* search**. It evaluates nodes by combining $g(n)$, the cost to reach the node n, and $h(n)$, the cost to get from n to the goal:

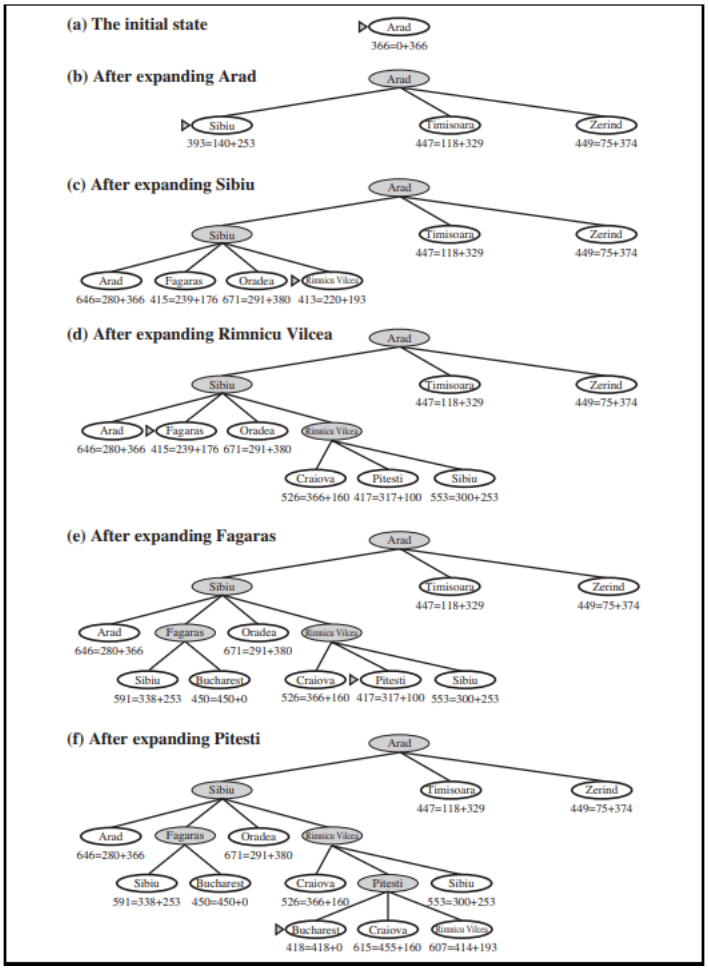
$$f(n) = g(n) + h(n)$$

Since $g(n)$ gives the path cost from the start node to node n, and $h(n)$ is the estimated cost of the cheapest path from n to the goal, we have

$$f(n) = \textit{estimated cost of the cheapest solution through } n$$

Thus, if we are trying to find the cheapest solution, a reasonable thing to try first is the node with the lowest value of $g(n) + h(n)$.

A* search uses an **admissible heuristic**, that is, it never overestimates the cost to reach the goal; $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost from n. It also requires $h(n) \geq 0$, so $h(G) = 0$ for any goal G.



Theorem: the tree-search version of A* is optimal if $h(n)$ is admissible.

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

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

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

The property above does not work for the graph-search version of A*. This is because there is a risk of discarding a repeated occurrence of a state that is on an optimal path (since graph-search does not visit an already visited node).

A solution to this problem is to use **consistent** heuristics: A heuristic $h(n)$ is consistent if, for every node n and every successor n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n':

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

This is a form of the general **triangle inequality**, which stipulates that each side of a triangle cannot be longer than the sum of the other two sides. Here, the triangle is formed by n, n', and the goal G_n closest to n.

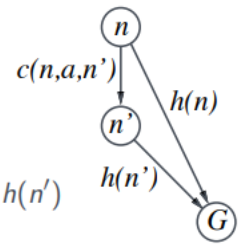
Theorem: e the graph-search version of A* is optimal if $h(n)$ is **consistent**.

Proof: if h is consistent, we have:

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

If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$



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

From the preceding observation, it follows that the sequence of nodes expanded by A* using GRAPH-SEARCH is in non-decreasing order of $f(n)$. Hence, the first goal node selected for expansion must be an optimal solution because f is the true cost for goal nodes (which have $h = 0$) and all later goal nodes will be at least as expensive.

The fact that f-costs are nondecreasing along any path also means that we can draw **contours** in the state space, just like the contours in a topographic map.

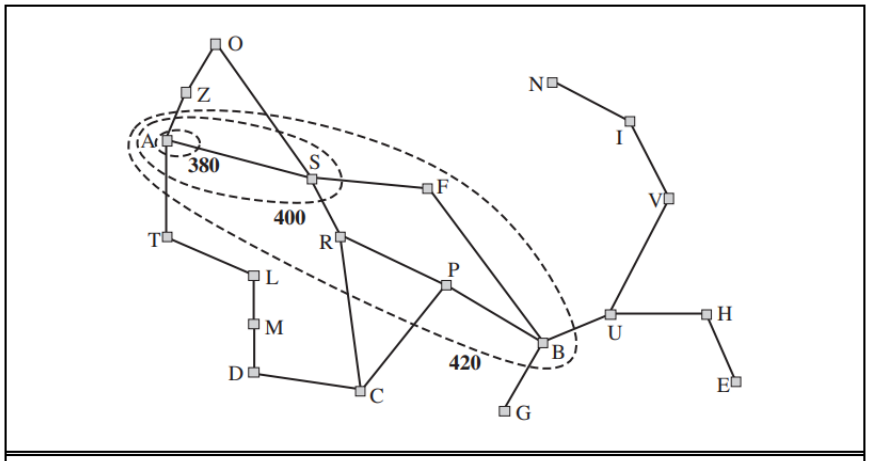


Figure 3.25 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have f-costs less than or equal to the contour value.

Properties of A*:

- **Complete:** Yes, unless there are infinitely many nodes with $f \leq f(G)$.
- **Time:** Exponential in [relative error in $h\times$ length of solution].
- **Space:** Keeps all nodes in memory.

- **Optimal:** Yes (it follows from the theorems). It expands all nodes with $f(n) < C^*$, where C^* is is the cost of the optimal solution path. It might then expand some of the nodes right on the "goal contour" (where $f(n) = C^*$) before selecting a goal node. It expands no nodes with $f(n) > C^*$

A* is not only optimal: no other optimal algorithm is guaranteed to expand fewer nodes than A* (except possibly through tie-breaking among nodes with $f(n) = C^*$). In fact, any algorithm that does not expand all nodes with $f(n) < C^*$ runs the risk of missing the optimal solution.

3.4 Iterative Deepening A*

A* is exponential in space and therefore not applicable to many real-world problems. The main problem is that the frontier grows exponential in size even if most of the nodes will never be extracted ($f(n) > C^*$). In order to reduce the use of memory of A* we can exploit the same idea behind Iterative Deepening Search.

The idea is to iteratively increase the f-value, just as Iterative Deepening Search increases the depth level. However, the f-value is a real number and cannot just be incremented by 1. Therefore, at iteration $i + 1$, we can use the best f-value of nodes that were not inserted in the frontier at iteration i.

This procedure is called Iterative Deepening A* and it works as follows:

1. a **cutoff** value for f is defined (initially equal to 0)
2. during an iteration, a node n is **not** inserted in the frontier if its f-value is larger than the cutoff value ($f(n) > cutoff$).
3. when the queue is empty (and no goal found) a new iteration is started with a new cutoff value.
4. the new cutoff value is given by the minimum f-value among all the nodes not inserted in the queue.

Chapter 4

Lec 04 - Informed Search II

4.1 Recursive Best First Search

Recursive best-first search (RBFS) is a simple recursive algorithm that attempts to RECURSIVE mimic the operation of standard best-first search, but using only **linear space**.

Its structure is similar to that of a recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the *f-limit* variable to keep track of the *f-value* of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the *f-value* of each node along the path with a **backed-up value**, that is, the best *f-value* of its children. In this way, RBFS remembers the *f-value* of the best leaf in the forgotten subtree and can therefore decide whether it's worth reexpanding the subtree at some later time.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

Figure 3.26 The algorithm for recursive best-first search.

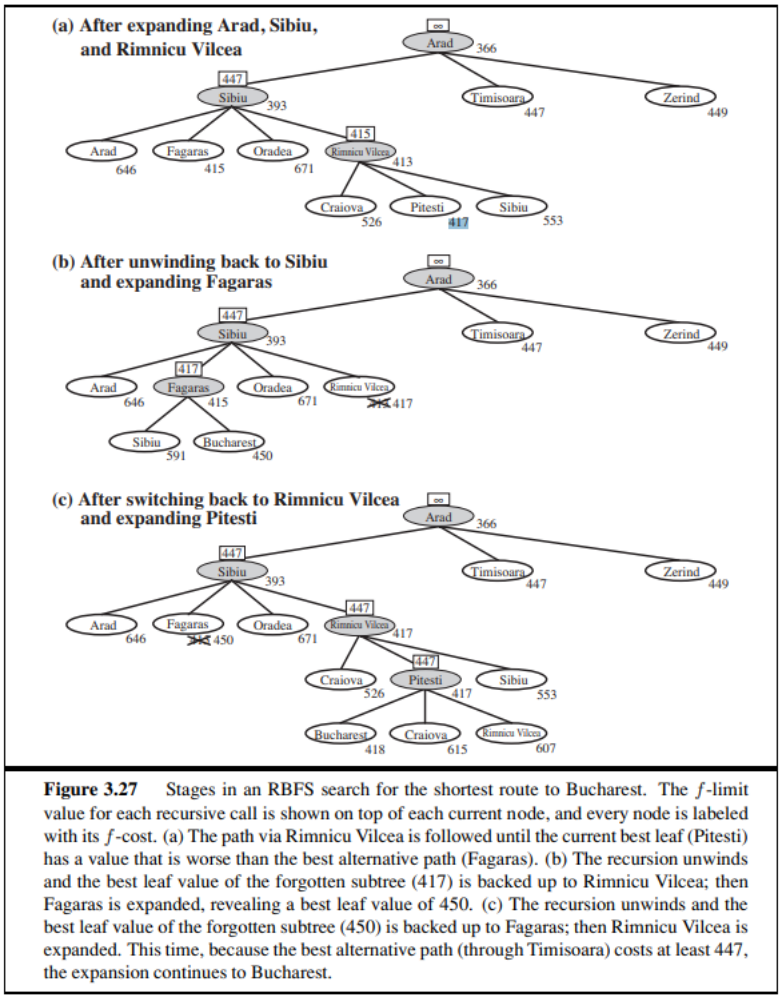


Figure 3.27 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node, and every node is labeled with its f-cost. (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras). (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea; then Fagaras is expanded, revealing a best leaf value of 450. (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed up to Fagaras; then Rimnicu Vilcea is expanded. This time, because the best alternative path (through Timisoara) costs at least 447, the expansion continues to Bucharest.

Just like A*, RBFS is optimal if the heuristic function $h(n)$ is admissible. Moreover, it has space complexity $O(bd)$ and time complexity which is exponential in the worst case.

The main problem of RBFS (as well as IDA*) is the use of too little memory. In fact, it may happen that the algorithm reexpand forgotten subtrees many times to recreate the best path. Unlike A*, RBFS does not store in the frontier each expanded node, but only the nodes along a single path from the root to a leaf node.

In general, the less memory you use, more repetitions you have to do in time. The algorithm MA* (memory-bounded A*) and SMA* (simplified memory-bounded A*) **fully use all available memory**.

4.2 Simplified MA*

Simplified MA* behaves just like A*, expanding the best node until the memory is full. At this point, it cannot add a new node to the search tree without dropping an old one. SMA* always drops the worst leaf node, that is, the one with the highest *f-value* (last node in the queue). Like RBFS, SMA* then backs up the value of the removed node to its parent. In this way, the ancestor of a forgotten subtree knows the quality of the best path in that subtree. With this information, SMA* regenerates the subtree only when all other

paths have been shown to look worse than the path it has forgotten.

What if all the leaf nodes have the same *f-value*? To avoid selecting the same node for deletion and expansion, SMA* expands the newest best leaf and deletes the oldest worst leaf. These coincide when there is only one leaf, but in that case, the current search tree must be a single path from root to leaf that fills all of memory. If the leaf is not a goal node, then even if it is on an optimal solution path, that solution is **not reachable with the available memory**.

SMA* properties:

- **complete** only if a solution can be kept in memory
- **optimal** only if any optimal solution is reachable (otherwise it returns the best reachable solution).

4.3 Heuristic functions

The performance of heuristic search algorithms depends on the quality of the heuristic function. Let's look, as an example, at heuristics for the 8-puzzle

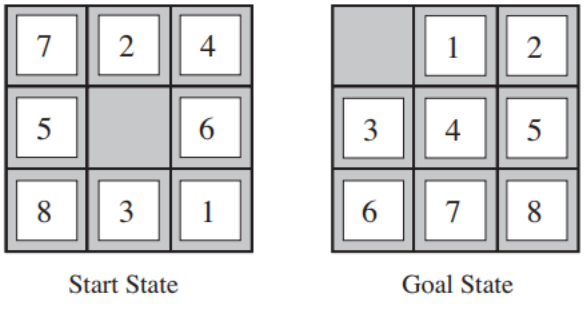


Figure 3.28 A typical instance of the 8-puzzle. The solution is 26 steps long.

If we want to find the shortest solutions by using A*, we need a heuristic function that never overestimates the number of steps to the goal. There is a long history of such heuristics for the 8-puzzle; here are two commonly used candidates:

- $h_1(n)$ = number of misplaced tiles. The start state above would have $h_1(S) = 6$.
- $h_2(n)$ = total Manhattan distance, that is, the sum of the distances of the tiles from their goal positions. The state start S qould have $h_2(S) = 4 + 0 + 3 + 3 + 1 + 0 + 2 + 1 = 14$.

h_1 is faster to compute than h_2, however, if $h_2(n) \geq h_1(n)$ for all n (both admissible), then h_2 **dominates** h_1. Domination translates directly into efficiency: A* using h_2 will never expand more nodes than $A*$ using h_1 (except possibly for some nodes with $f(n) = C^*$). Recall that $A*$ expand all nodes with $f(n) < C^*$ (which is mandatory for any optimal algorithm). This is the same as saying that every node with $h(n) < C^* - g(n)$ will surely be expanded. But because h_2 is at least as big as h_1 for all nodes, every node that is surely expanded by A* search with h_2 will also surely be expanded with h_1, and h_1 might cause other nodes to be expanded as well. Hence, it is generally better to use a heuristic function with higher value, provided it is consistent and that the computation time for the heuristic is not too long.

Admissible heuristics can be derived from the exact solution cost of a **relaxed** version of the problem. A relaxed problem is a problem with fewer restrictions on the actions than the original.

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

The key point is that the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem. Hence, the cost of an optimal solution to a relaxed problem is an **admissible** heuristic for the original problem.

For example, we can derive a lower bound on the shortest tour for the Travelling Salesman Problem (TSP) using a Minimum Spanning Tree (which can be computed in $O(n^2)$).

4.4 Iterative Improvements Algorithms

If the path to the goal does not matter, we might consider a different class of algorithms, ones that do not worry about paths at all. **Local search algorithms** operate using a single current node (rather than multiple paths) and generally move only to neighbors of that node. In this case, the **state space** is the set of *complete* configurations and the task is to find the **optimal** configuration (e.g. TSP), or to find a configuration satisfying constraints (e.g. timetable).

Local search algorithms have two key advantages: (1) they use very little memory, usually a constant amount, since the paths followed by the search are not retained; and (2) they can often find reasonable solutions in large or infinite (continuous) state spaces for which systematic algorithms are unsuitable.

In addition to finding goals, local search algorithms are useful for solving pure optimization problems, in which the aim is to find the best state according to an **objective function**.

To understand local search, we find it useful to consider the state-space landscape. A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function). If elevation corresponds to cost, then the aim is to find the lowest valley, a **global minimum**; if elevation corresponds to an objective function, then the aim is to find the highest peak—a global maximum. Local search algorithms explore this landscape. A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

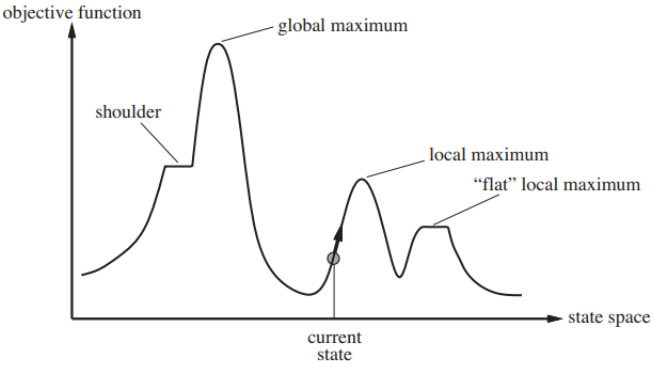


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill-climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text.

4.4.1 Hill-climbing

The **hill-climbing** search algorithm is simply a loop that continually moves in the direction of increasing value, that is, uphill. At each step the current node is replaced by the best neighbor, i.e., the neighbor with the highest **value** (or lowest heuristic cost if a heuristic is used). It terminates when it reaches a "peak" where no neighbor has a higher value.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum

  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
    neighbor ← a highest-valued successor of current
    if neighbor.VALUE ≤ current.VALUE then return current.STATE
    current ← neighbor
```

To illustrate hill climbing, we will use the **8-queens problem**. Local search algorithms typically use a complete-state formulation, where each state has 8 queens on the board, one per column. The successors of a state are all possible states generated by moving a single queen to another square in the same column (so each state has $8 \times 7 = 56$ successors). The heuristic cost function h is the number of pairs of queens that are attacking each other, either directly or indirectly. The global minimum of this function is zero, which occurs only at perfect solutions.

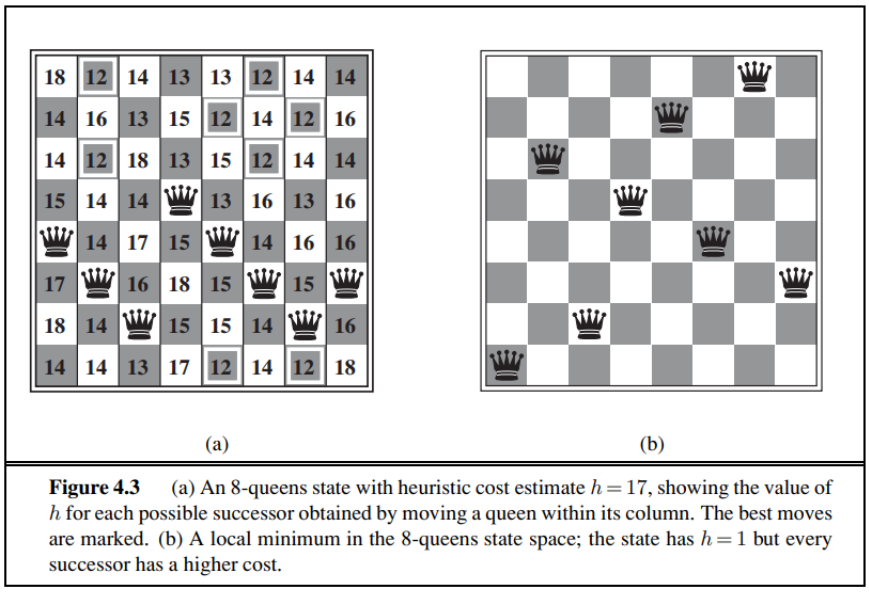


Figure 4.3 (a) An 8-queens state with heuristic cost estimate $h = 17$, showing the value of h for each possible successor obtained by moving a queen within its column. The best moves are marked. (b) A local minimum in the 8-queens state space; the state has $h = 1$ but every successor has a higher cost.

Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state. Unfortunately, hill climbing often gets stuck for the following reasons:

- **Local maxima:** a local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
- **Ridges:** Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux:** a plateau is a flat area of the state-space landscape. It can be a flat local maximum, from which no uphill exit exists, or a shoulder, from which progress is possible.

A possible solution to the **plateaux** problem is to allow **sideways move**, i.e, move to a state with same h value, in the hope that the plateau is really a shoulder. In fact, if we are on a flat maxima, an infinite loop will occur. The solution to above problem is to put a limit on the number of consecutive sideways moves allowed.

Many variants of hill climbing have been invented to face the problem of **local maxima**. **Stochastic hill climbing** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move. This usually converges more slowly than steepest ascent, but in some state landscapes, it finds better solutions. **Random-restart hill climbing** conducts a series of hill-climbing searches from randomly generated initial states. It can be proved that if p is the probability to find an optimal solution for a single search, the expected number of searches needed to find an optimal solution is $1/p$.

Going back to the 8-queens problem, Starting from a randomly generated 8-queens state, **standard** steepest-ascent hill climbing gets stuck 86% of the time, solving only 14% of problem instances. It works quickly, taking just 4 steps on average when it succeeds and 3 when it gets stuck, not bad for a state space with $8^8 \approx 17$ million states.

Hill-climbing with **sideways moves** (≤ 100 consecutive moves) raises the percentage of problem instances solved to 94%. On the average it takes around 21 steps to find a solution, otherwise around 64 steps for finding suboptimal solution.

Random-restart hill climbing, for 8-queens instances with no sideways moves allowed ($p \approx 0.14$), needs roughly 7 iterations to find a goal (6 failures and 1 success). The expected number of steps is the cost of one successful iteration plus $(1 - p)/p$ times the cost of failure, or roughly 22 steps in all. When we allow sideways moves, the optimal solution is found with probability $p = 0.94$ of times, thus around 1.06 searches to find optimal solution. Total expected number of steps: $63(1p)/p + 21 = 25.08$.

Chapter 5

Lec 05 - Informed Search III

5.1 Simulated annealing

A hill-climbing algorithm that never makes "downhill" moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. **Simulated annealing** is such an algorithm. In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      ΔE ← next.VALUE – current.VALUE
      if ΔE > 0 then current ← next
      else current ← next only with probability e^(ΔE/T)
```

The simulated annealing algorithm is a version of stochastic hill climbing where some downhill moves are allowed. Downhill moves are accepted readily early in the annealing schedule and then less often as time goes on. The schedule input determines the value of the temperature T as a function of time.

The innermost loop of the simulated-annealing algorithm picks a random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move, that is, the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad" moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases. If the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

5.2 Local beam search

The local beam search algorithm keeps track of k states rather than just one. It begins with k randomly generated states. At each step, all the successors of all k states are generated. If any one is a goal, the algorithm halts. Otherwise, it selects the k best successors from the **complete** list and repeats.

At first sight, a local beam search with k states might seem to be nothing more than running k random restarts in parallel instead of in sequence. However, the two algorithms are quite different. In a random-restart search, each search process runs independently of the others. In a local beam search, useful information is passed among the parallel search threads.

In its simplest form, local beam search can suffer from a lack of diversity among the k states—they can quickly become concentrated in a small region of the state space, making the search little more than an expensive version of hill climbing. A variant called stochastic beam search, analogous to stochastic hill climbing, helps alleviate this problem. Instead of choosing the best k from the the pool of candidate successors, stochastic beam search chooses k successors at random, with the probability of choosing a given successor being an increasing function of its value.

5.3 Genetic algorithms

A **genetic algorithm** (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state.

Like beam searches, GAs begin with a set of k randomly generated states, called the **population**. Each state, or **individual**, is represented as a string over a finite alphabet, most commonly, a string of 0s and 1s. The production of the next generation of states works as follows:

1. Each state is rated by the objective function, or (in GA terminology) the **fitness function**. A fitness function should return higher values for better states.

2. for $i = 1$ to $size(population)$:

   (a) A pair is selected at random for reproduction (the probability of being chosen for reproducing is directly proportional to the fitness score). A **crossover** point is chosen randomly from the positions in the string.

   (b) A new child is generated starting from the parents according to the crossover point c. Basically, given n the length of the representation string, the child is generated concatenating the substring of the first parent from 1 to c and the substring of the second parent from $c + 1$ to n.

   (c) Finally, the child is subject to random mutation with a small independent probability and is then added to the new population.

3. Then, the old population is updated to the new population.

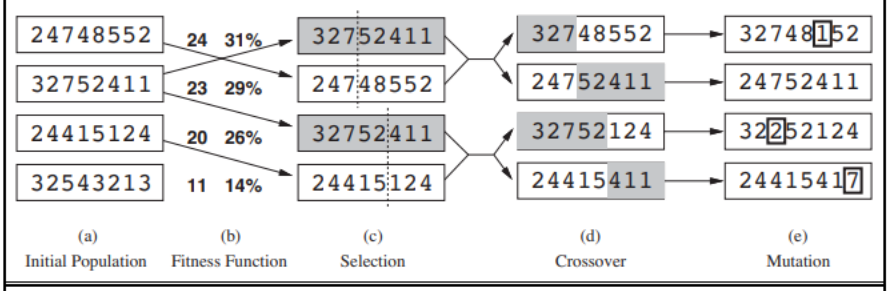


Figure 4.6 The genetic algorithm, illustrated for digit strings representing 8-queens states. The initial population in (a) is ranked by the fitness function in (b), resulting in pairs for mating in (c). They produce offspring in (d), which are subject to mutation in (e).

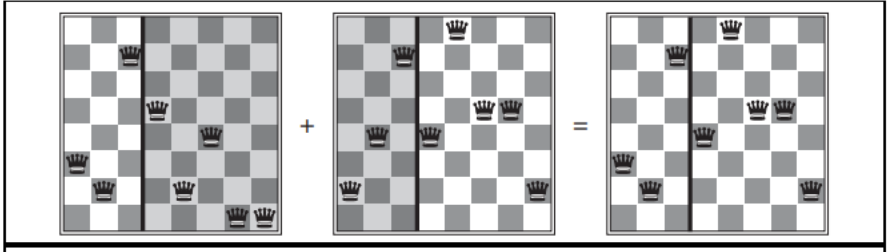


Figure 4.7 The 8-queens states corresponding to the first two parents in Figure 4.6(c) and the first offspring in Figure 4.6(d). The shaded columns are lost in the crossover step and the unshaded columns are retained.

5.4 Continuous Spaces: Gradient Descent

Yet none of the algorithms we have described (except for first-choice hill climbing and simulated annealing) can handle continuous state and action spaces, because they have infinite branching factors.

The idea is to minimize/maximize an objective function exploiting its gradient with respect to a set of parameters. The gradient vector can be interpreted as the direction and rate of fastest increase of the function. The gradient is the zero vector at a point if and only if the point is a stationary point.

This minimization problem can be solved using gradient descent. Starting from a random configuration of θ, each parameter is updated in the following way:

$$\theta_{k+1} = \theta_k - \eta \nabla f(\theta_k)$$

where:

- $\nabla f(\theta_k)$ is the partial derivative of the function in θ_k.
- The parameter $\eta > 0$ is known as the *learning rate*.

The derivative term $\frac{\partial}{\partial \theta_k} f(\theta_k)$ can be:

- ≥ 0 it means that the function is increasing, so we are decreasing θ_k in the *right direction*.
- ≤ 0 it means that the function is decreasing, so we are increasing θ_k in the *right direction*

If η is too small, gradient descent can be slow. Anyway, if it is too large, it can overshoot the minimum (fail to converge).

5.5 Online Search

So far we have concentrated on agents that use **offline search** algorithms. They compute a complete solution before setting foot in the real world and then execute the solution. In contrast, when the environment is not completely observable, or it is dynamic/semidynamic, the agent needs to interact with the environment to extract information. An **online search** agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action. Online search is a necessary idea for unknown environments, where the agent does not know what states exist or what its actions do. In this state of ignorance, the agent faces an **exploration problem**.

5.5.1 Online Search Problems

We assume a deterministic and fully observable environment, however the agent knows only the following:

- ACTIONS(s), which returns a list of actions allowed in state s;
- The step-cost function $c(s, a, s')$, note that this cannot be used until the agent knows that s' is the outcome;
- GOAL-TEST(s).

If some actions are irreversible the online search might accidentally reach a **dead-end state**. In general, this is not avoidable (not even for safely explorable state spaces).

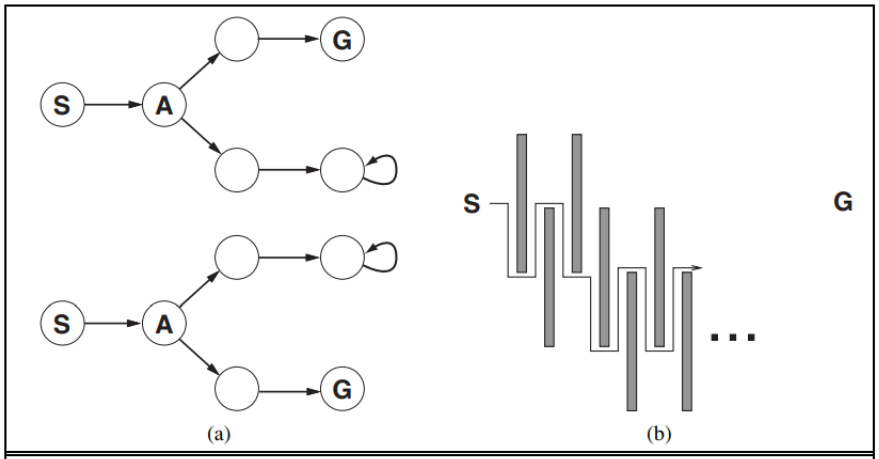


Figure 4.20 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal. Whichever choice the agent makes, the adversary blocks that route with another long, thin wall, so that the path followed is much longer than the best possible path.

Typically, the agent's objective is to reach a goal state while minimizing cost. (Another possible objective is simply to explore the entire environment). The cost is the total path cost of the path that the agent actually travels. It is common to compare this cost with the path cost of the path the agent would follow *if it knew the search space in advance*, that is, the actual shortest path (or shortest complete exploration). In the language of online algorithms, this is called the **competitive ratio**; we would like it to be as small as possible.

5.5.2 Depth-first Online Search

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment. The current map is used to decide where to go next. Offline algorithms such as A* can expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions. An online algorithm, on the other hand, can discover successors only for a node that it physically occupies. To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a **local** order.

Depth-first search has exactly this property because (except when backtracking) the next node expanded is a child of the previous node expanded.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in untried) then untried[s'] ← ACTIONS(s')
  if s is not null then
      result[s, a] ← s'
      add s to the front of unbacktracked[s']
  if untried[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

Figure 4.21 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action.

This agent stores its map in a table, RESULT$[s, a]$, that records the state resulting from executing action a in state s. Whenever an action from the current state has not been explored, the agent tries that action. The difficulty comes when the agent has tried all the actions in a state. In offline depth-first search, the state is simply dropped from the queue; in an online search, the agent has to backtrack physically. To achieve that, the algorithm keeps a table that lists, for each state, the predecessor states to which the agent has not yet backtracked. If the agent has run out of states to which it can backtrack, then its search is complete.

5.6 Random Search

Like depth-first search, hill-climbing search has the property of locality in its node expansions. However, random restarts cannot be used, because the agent cannot transport itself to a new state. Instead of random restarts, one might consider using a **random walk** to explore the environment. A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried. It is easy to prove that a random walk will eventually find a goal or complete its exploration, provided that the space is finite. On the other hand, the process can be very slow. Augmenting hill climbing with memory rather than randomness turns out to be a more effective approach.

5.6.1 LRTA* Search

The basic idea is to store a "current best estimate" $H(s)$ of the cost to reach the goal from each state that has been visited. $H(s)$ starts out being just the heuristic estimate $h(s)$ and is updated as the agent gains experience in the state space.

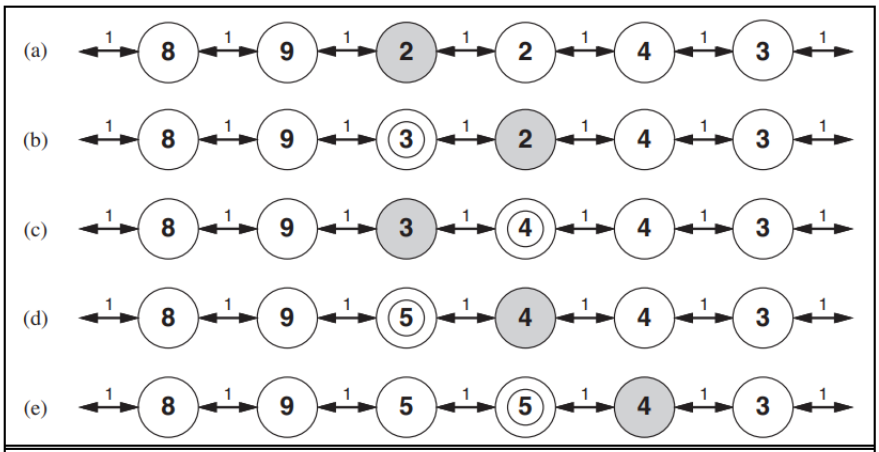


Figure 4.23 Five iterations of LRTA* on a one-dimensional state space. Each state is labeled with $H(s)$, the current cost estimate to reach a goal, and each link is labeled with its step cost. The shaded state marks the location of the agent, and the updated cost estimates at each iteration are circled.

The agent should follow what seems to be the best path to the goal given the current cost estimates for its neighbors. The estimated cost to reach the goal through a neighbor s' is the cost to get to s' plus the estimated cost to get to a goal from there, that is, $c(s, a, s') + H(s')$. In the example, there are two actions, with estimated costs 1+9 and 1+2, so it seems best to move right.

An agent implementing this scheme, which is called learning real-time A* (**LRTA***). It builds a map of the environment in the result table. It updates the cost estimate for the state it has just left and then chooses the "apparently best" move according to its current cost estimates. One important detail is that actions that have not yet been tried in a state s are always assumed to lead immediately to the goal with the least possible cost, namely $h(s)$.

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
              H, a table of cost estimates indexed by state, initially empty
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null
      result[s, a] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

Figure 4.24 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

Chapter 6

Lec 06 - Adversarial Search I

6.1 Adversarial Search

Mathematical game theory, a branch of economics, views any multiagent environment as a game, provided that the impact of each agent on the others is "significant," regardless of whether the agents are cooperative or competitive. In AI, the most common games are of a rather specialized kind, what game theorists call deterministic, turn-taking, two-player, zero-sum games of **perfect information** (such as chess). In our terminology, this means deterministic, fully observable environments in which two agents act alternately and in which the utility values at the end of the game are always equal and opposite. For example, if one player wins a game of chess, the other player necessarily loses. It is this opposition between the agents' utility functions that makes the situation adversarial. There exist also **imperfect information** games, such as bridge, poker, scrabble, and games with non-deterministic outcomes, e.g. backgammon or monopoly.

In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state, a terminal state that is a win. In adversarial search, the solution is a strategy specifying a move for every possible opponent reply. Algorithms that deal with games also need to consider the time limits of the game itself (e.g. chess). Therefore, they require the ability to make some decision even when calculating the optimal decision is infeasible (approximation).

We first consider games with two players, whom we call MAX and MIN for reasons that will soon become obvious. MAX moves first, and then they 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.

The initial state, the legal moves in each state and the results of the moves form the so called **game tree**, where the nodes are game states and the edges are moves.

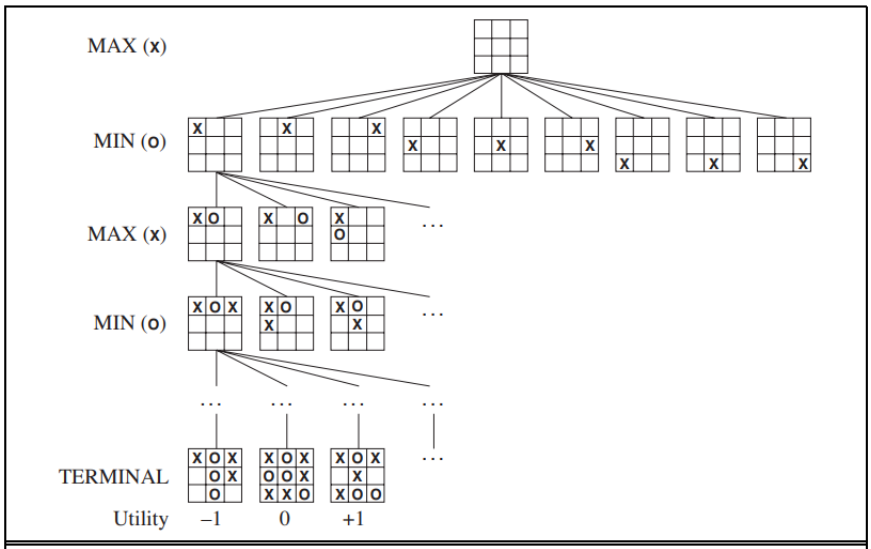


Figure 5.1 A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

The figure above shows part of the game tree for tic-tac-toe. From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing an O until we reach leaf nodes corresponding to terminal states such that one player has three in a row or all the squares are filled. The number on each leaf node indicates the **utility value** (provided by the **utility function**) of the terminal state; high values are assumed to be good for MAX and bad for MIN and viceversa. For tic-tac-toe the game tree is relatively small, fewer than $9! = 362,880$ terminal nodes. But for chess there are over 10^{40} nodes.

A utility function (also called an objective function or payoff function), defines the final numeric value for a game that ends in terminal state s for a player p. A **zero-sum** game is (confusingly) defined as one where the total payoff to all players is the same for every instance of the game. Chess is zero-sum because every game has payoff of either $0 + 1$, $1 + 0$ or $1/2 + 1/2$.

6.2 Minimax

Given a game tree, the optimal strategy can be determined from the **minimax** value of each node, which we write as MINIMAX(n). The minimax value of a node is the utility (for MAX) of being in the corresponding state, **assuming that both players play optimally** from there to the end of the game. Obviously, the minimax value of a terminal state is just its utility. Furthermore, given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value. So we have the following:

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

This definition of optimal play for MAX assumes that MIN also plays **optimally**, it maximizes the worst-case outcome for MAX. What if MIN does not play optimally? Then, it is easy to show that MAX will do

even better. Other strategies against suboptimal opponents may do better than the minimax strategy, but these strategies necessarily do worse against optimal opponents.

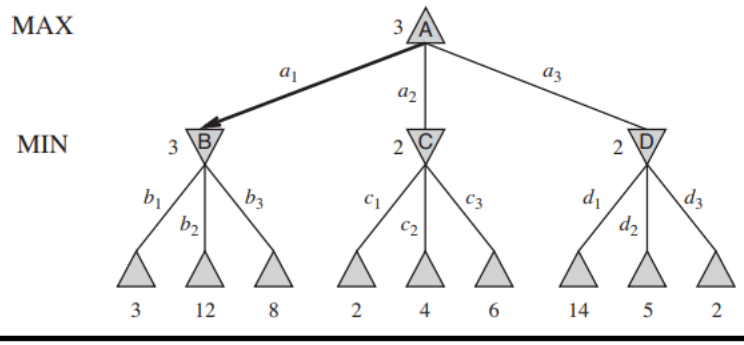


Figure 5.2 A two-ply game tree. The △ nodes are "MAX nodes," in which it is MAX's turn to move, and the ▽ nodes are "MIN nodes." The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX's best move at the root is a_1, because it leads to the state with the highest minimax value, and MIN's best reply is b_1, because it leads to the state with the lowest minimax value.

The **minimax algorithm** computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are backed up through the tree as the recursion unwinds.

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

function MIN-VALUE(state) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← ∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s, a)))
  return v
```

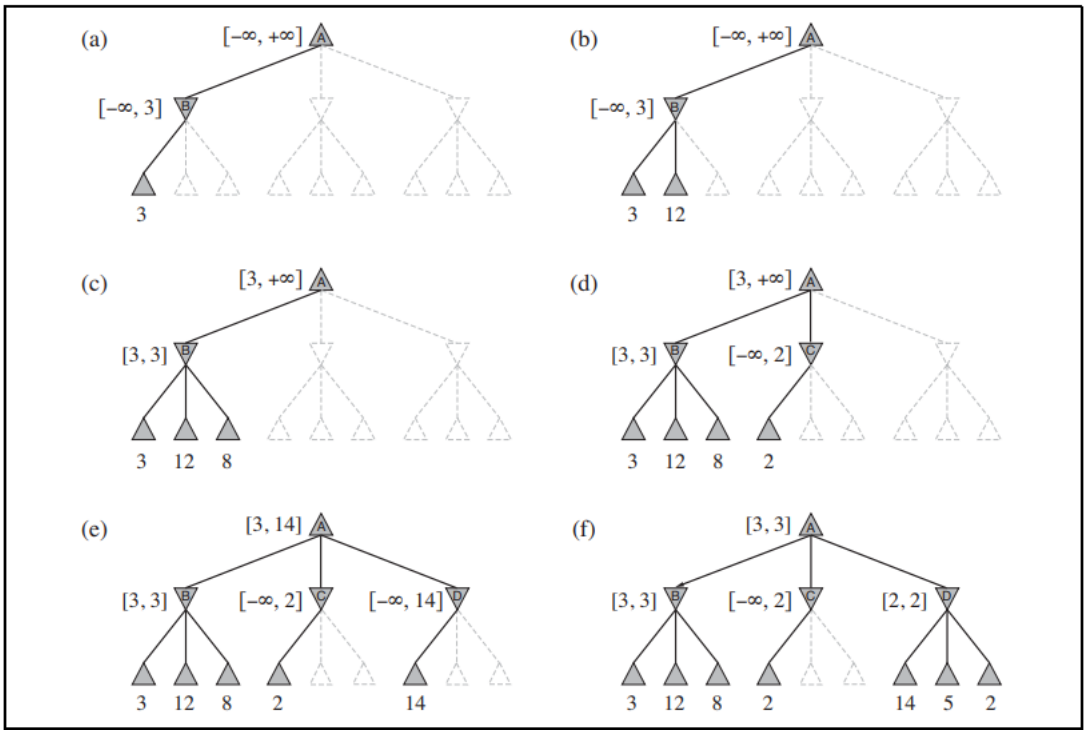
Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation $\text{argmax}_{a \in S} f(a)$ computes the element a of set S that has the maximum value of $f(a)$.

The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity is $O(b^m)$. The space

complexity is $O(bm)$. For real games, of course, the time cost is totally impractical, but this algorithm serves as the basis for the mathematical analysis of games and for more practical algorithms.

6.3 Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. Unfortunately, we can't eliminate the exponent, but it turns out we can effectively cut it in half. The trick is that it is possible to compute the **correct** minimax decision without looking at every node in the game tree. This particular technique is called **alpha–beta pruning**.



Alpha–beta pruning gets its name from the following two parameters 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.
- 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.

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.

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. A "good" ordering of the moves improves the effectiveness of pruning. In particular, the perfect order would be:

- for MAX: from highest to lowest values;
- for MIN: from lowest to highest values.

In this way, when MIN needs to examine its successors he knows that as soon as he founds a state with lower value than α, he can prune the remaining nodes (the same is valid for MAX). With perfect order it turns out that alpha–beta needs to examine only $O(b^{m/2})$ nodes to pick the best move (which means we can double the search depth).

Chapter 7

Lec 07 - Adversarial Search II

7.1 Resource Limits

The minimax algorithm generates the entire game search space, whereas the alpha–beta algorithm allows us to prune large parts of it. However, alpha–beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time (typically a few minutes at most). The solution is to alter minimax or alpha-beta in two ways: replace the utility function by a heuristic **evaluation function** EVAL, which estimates the position's utility, and replace the terminal test by a **cutoff test** that decides when to apply EVAL (e.g. depth limit).

7.2 Evaluation functions

An evaluation function returns an estimate of the expected utility of the game from a given position, just as the heuristic functions return an estimate of the distance to the goal. It should be clear that the performance of a game-playing program depends strongly on the quality of its evaluation function. How exactly do we design good evaluation functions?

First, the evaluation function should order the terminal states in the same way as the true utility function: states that are wins must evaluate better than draws, which in turn must be better than losses. Second, the computation must not take too long. Third, for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.

7.2.1 Quiescence Search

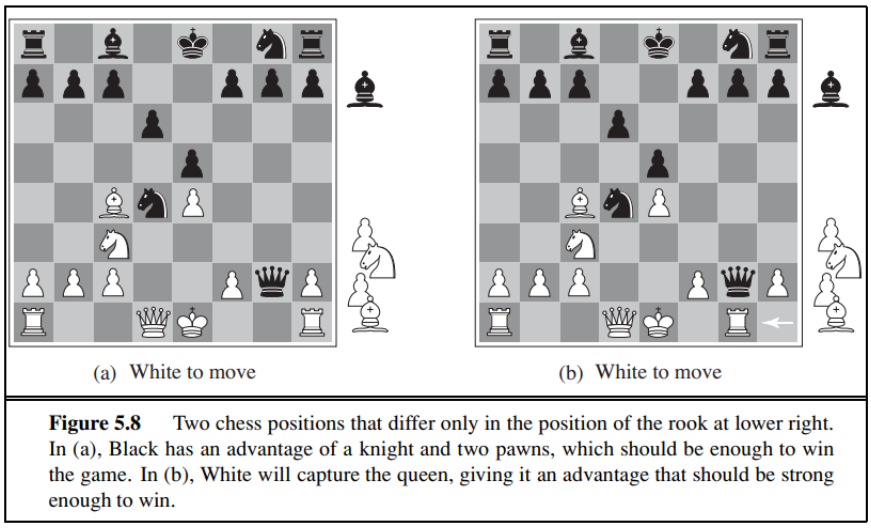


Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

Suppose the program searches to the depth limit, reaching the position in the figure above, where Black is ahead by a knight and two pawns. If the EVAL function was based only on the number of pieces on the board, it would declare that the state is a probable win by Black. But White's next move captures Black's queen with no compensation. Hence, the position is really won for White, but this can be seen only by looking ahead one more ply. Obviously, a more sophisticated cutoff test is needed. The evaluation function should be applied only to positions that are **quiescent**, that is, unlikely to exhibit wild swings in value in the near future. Non-quiescent positions can be expanded further until quiescent positions are reached.

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 delaying tactics. One strategy to mitigate the horizon effect is the **singular extension**, a move that is "clearly better" than all other moves in a given position. Once discovered anywhere in the tree in the course of a search, this singular move is remembered. When the search reaches the normal depth limit, the algorithm checks to see if the singular extension is a legal move; if it is, the algorithm allows the move to be considered. This makes the tree deeper, but because there will be few singular extensions, it does not add many total nodes to the tree.

7.3 Non-deterministic Games

In real life, many unpredictable external events can put us into unforeseen situations. Many games mirror this unpredictability by including a random element, such as the throwing of dice.

Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the legal moves. Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be. That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe. A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes.

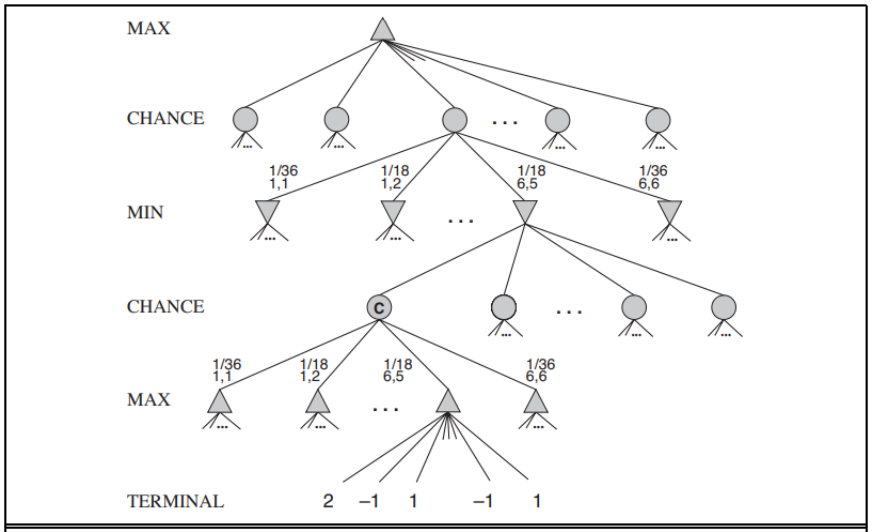


Figure 5.11 Schematic game tree for a backgammon position.

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 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 generalize the minimax value for deterministic games to an **expecti-minimax value** for games with chance nodes. Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before. 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:

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

where r represents a possible dice roll (or other chance event).

Note that alpha-beta pruning is still possible for non-deterministic game trees. In fact, for chance nodes, α and β are computed by multiplying the value of each MAX and MIN nodes by the respective probability. Stronger pruning can be obtained if possible to bound the leaves values.

7.3.1 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 scores to better positions. But in fact, the presence of chance nodes means that one has to be more careful about what the evaluation values mean.

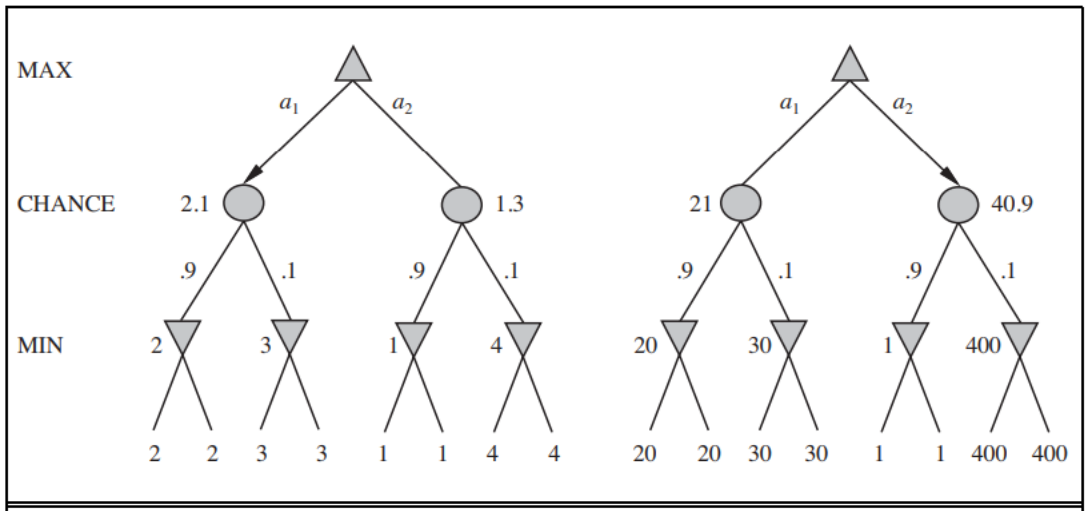


Figure 5.12 An order-preserving transformation on leaf values changes the best move.

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 in the scale of some evaluation values! It turns out that to avoid this sensitivity, the evaluation function must be a positive linear transformation of the probability of winning from a position (or, more generally, of the expected utility of the position).

Because expectiminimax is also considering all the possible dice-roll sequences, it will take $O(b^m n^m)$ where n is the number of distinct rolls. Even if the search depth is limited to some small depth d, the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance. In backgammon n is 21 and b is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles.

7.4 Partially Observable Games

Partially Observable Games are games with hidden information, for example card games, where opponent's initial cards are unknown.

At first sight, it might seem that these card games are just like dice games: the cards are dealt randomly and determine the moves available to each player, but all the "dice" are rolled at the beginning! Even though this analogy turns out to be incorrect, it suggests an effective algorithm: consider all possible deals of the invisible cards; solve each one as if it were a fully observable game; and then choose the move that has the best outcome averaged over all the deals. Suppose that each deal s occurs with probability $P(s)$; then the move we want is:

$$argmax_a \sum_s P(s)\text{MINIMAX}(\text{RESULT}(s,a))$$

Now, in most card games, the number of possible deals is rather large. Solving even one deal is quite difficult, so solving ten million is out of the question. Instead, we resort to a Monte Carlo approximation: instead of adding up all the deals, we take a random sample of N deals, where the probability of deal s appearing in the sample is proportional to $P(s)$:

$$argmax_a \frac{1}{N} \sum_{i=1}^{N} \text{MINIMAX}(\text{RESULT}(s_i,a))$$

Chapter 8

Lec 08 - Logical Agents

8.1 Logical Agents

Logical agents simulate the process of **reasoning** operating on internal **representations** of knowledge. The problem-solving agents seen so far know things, but only in a very limited, inflexible sense. The knowledge of what the actions do is hidden inside the domain-specific code of the transition model, which defines the result of a move. We will see that **logic** provides a possible formalism to support knowledge-based agents.

8.2 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** in a formal language. Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world.

There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively. Both operations may involve inference, that is, deriving new sentences from old.

Agents can be viewed at the **knowledge level**, that is, what they know, regardless of how implemented, or can be viewed at the implementation level, data structures in KB and algorithms that manipulate them.

function KB-AGENT(*percept*) **returns** an *action*
  persistent: KB, a knowledge base
    t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(*percept*, t))
  action $\leftarrow$ ASK(KB, MAKE-ACTION-QUERY(t))
  TELL(KB, MAKE-ACTION-SENTENCE(*action*, t))
  $t \leftarrow t + 1$
  return *action*

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

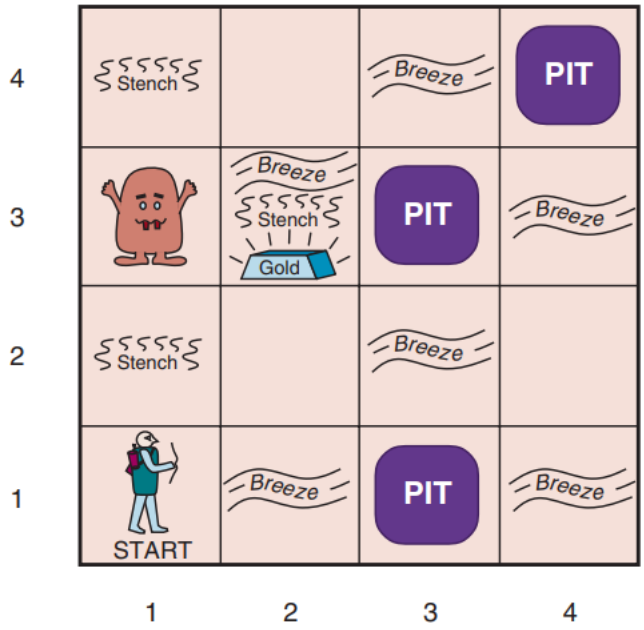
The figure above shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, KB, which may initially contain some **background knowledge**.

Each time the agent program is called, it does three things. First, it TELLs the knowledge base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action. In order to do so, the agent must be able to:

- Represent states, actions, etc.
- Incorporate new percepts
- Update internal representations of the world
- Deduce hidden properties of the world
- Deduce appropriate actions

8.3 The Wumpus world

In this section we describe an environment in which knowledge-based agents can show their worth. 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 mitigating feature of this bleak environment is the possibility of finding a heap of gold.



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 4×4 grid of rooms. The agent always starts in the square labeled $[1, 1]$, facing to the right. 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.

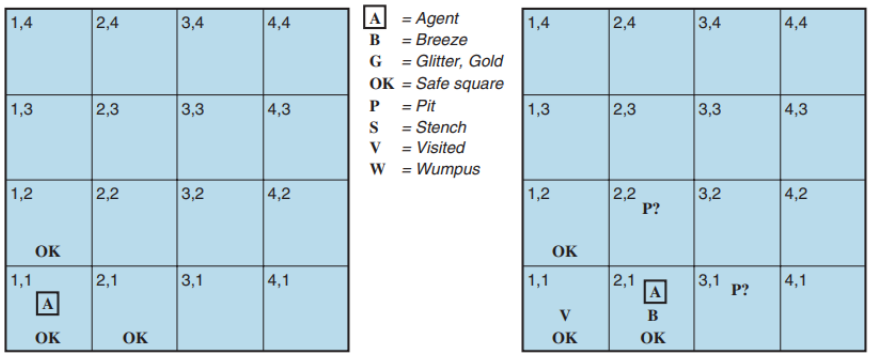
- **Actuators:** The agent can move *Forward*, *TurnLeft* by 90°, or *TurnRight* by 90°. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing (only one available). 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:

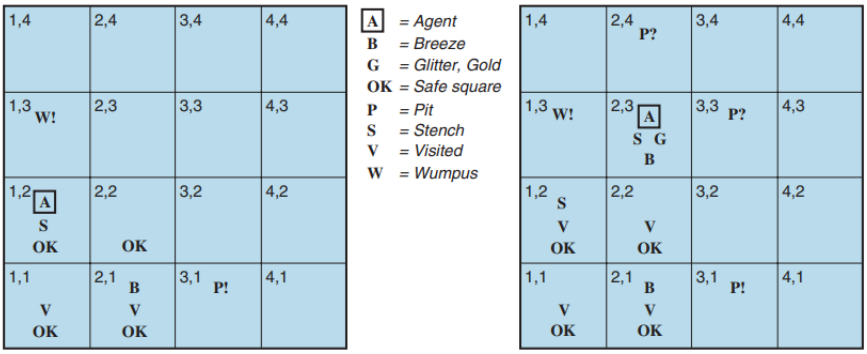
  - In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
  - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
  - In the square where the gold is, the agent will perceive a *Glitter*.
  - When an agent 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.

We can characterize the wumpus environment along the various dimensions seen previously. It is discrete, static, and single-agent. It is sequential, because rewards may come only after many actions are taken. It is partially observable, because some aspects of the state are not directly perceivable. For an agent in the environment, the main challenge is its initial ignorance of the configuration of the environment. Overcoming this ignorance seems to require logical reasoning.

Let us watch a knowledge-based wumpus agent exploring the environment using an informal knowledge representation language consisting of writing down symbols in a grid:



(a) (b)



(a) (b)

Note that in each case for which the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct if the available information is correct. we'll describe how to build logical agents that can represent in a **formal way** information and draw conclusions.

8.4 Propositional Logic

We now present a simple but powerful logic called **propositional logic**. The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single proposition symbol (P, Q, R, ... but can be whatever). 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 **logical connectives**.

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \mid [\textit{Sentence}] \\
&\mid \neg \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
\text{OPERATOR PRECEDENCE} &: \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

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 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

8.5 Models

As we said previously, the 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. 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.

When we need to be precise, we use the term **model** in place of "possible world." If a sentence α is true in model m, we say that m **satisfies** α or sometimes m **is a model of** α. We use the notation $M(\alpha)$ to mean the set of all models of α. Basically, a model assigns truth values to proposition symbols of a sentence.

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, —the idea that a sentence follows logically from another sentence. In mathematical notation, we write:

$$\alpha \models \beta$$

to mean that the sentence α entails the sentence β. The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write

$$\alpha \models \beta \text{ if and only if } M(\alpha) \subseteq M(\beta)$$

Note that the direction of the $\subseteq$ means that α is a stronger assertion than β.

We can apply the same kind of analysis to the wumpus-world reasoning example given previously. The agent has detected nothing in $[1,1]$ and a breeze in $[2,1]$. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested (among other things) in whether the adjacent squares $[1,2]$, $[2,2]$, and $[3,1]$ contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are $2^3 = 8$ possible models.

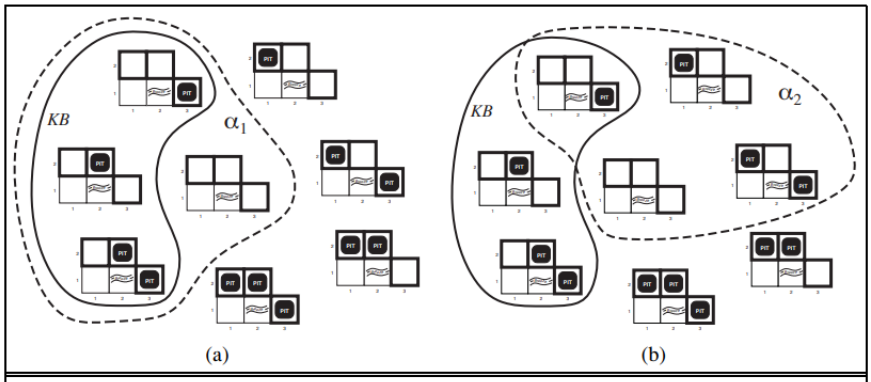


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

The KB can be thought of as a set of sentences or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows; for example, the KB is false in any model in which $[1,2]$ contains a pit, because there is no breeze in $[1,1]$. There are in fact just three models in which the KB is true, and these are shown surrounded by a solid line in the figure above.

Now let us consider two possible conclusions:

- α_1 = "There is no pit in $[1,2]$."

- α_2 = "There is no pit in $[2, 2]$."

By inspection, we see the following: in every model in which KB is true, α_1 is also true. Hence $KB \vDash \alpha_1$: there is no pit in $[1, 2]$. We can also see that in some models in which KB is true, α_2 is false. Hence, the agent cannot conclude that there is no pit in $[2, 2]$.

In understanding entailment and inference, it might help to think of the set of all consequences of KB as a haystack and of α as a needle. Entailment is like the needle being in the haystack; inference is like finding it. This distinction is embodied in some formal notation: if an inference algorithm i can derive α from KB, we write:

$$KB \vdash_i \alpha$$

which is pronounced "α is derived from KB by i".

An inference algorithm that derives only entailed sentences is called **sound**: whenever $KB \vdash_i \alpha$, it is also true that $KB \vDash \alpha$. Note that, as we did with the example before, enumerating all possible models to check that α is true in all models in which KB is true is sound. The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed.

If KB is true in the real world, then any sentence α derived from KB by a sound inference procedure is also true in the real world.

8.6 A simple knowledge base

Now we can construct a knowledge base for the wumpus world, focusing on its immutable aspects. For now, we need the following symbols for each $[x, y]$ 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 the agent perceives a breeze in $[x, y]$.
- $S_{x,y}$ is true if the agent perceives a stench in $[x, y]$.

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

- There is no pit in $[1, 1]$: $R_1 : \neg P_{1,1}$.
- 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:
  - $R_2 : B_{1,1} \iff (P_{1,2} \vee P_{2,1})$
  - $R_3 : B_{2,1} \iff (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
- 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, leading up to the situation in the figure above (Figure 7.5):
  - $R_4 : \neg B_{1,1}$
  - $R_5 : B_{2,1}$

Our goal now is to decide whether $KB \vDash \alpha$ for some sentence α. For example, is $\neg P_{1,2}$ entailed by our KB? 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.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	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*

Figure 7.9 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].

Returning to our wumpus-world example, the relevant proposition **symbols** are $B_{1,1}, B_{2,1}, P_{1,1}, P_{1,2}, P_{2,1}, P_{2,2}$, and $P_{3,1}$. With seven symbols, there are $2^7 = 128$ possible models; in three of these, KB is true. Note that this is just a "snapshot" of the situation shown in the Figure 7.5. Obviously, as the agent goes on, the KB changes. In the three model where KB is *True*, $\neg P_{1,2}$ is also *True*. Hence, there is no pit in $[1, 2]$. On the other hand, $P_{2,2}$ is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in $[2, 2]$. Note that KB is true if R_1 through R_5 are true (KB can be expressed as a sentence $R_1 \wedge ... \wedge R_5$), which occurs in just 3 of the 128 rows.

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

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, { })

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
     if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
     else return true // when KB is false, always return true
  else do
     P ← FIRST(symbols)
     rest ← REST(symbols)
     return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
             and
             TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword "**and**" is used here as a logical operation on its two arguments, returning *true* or *false*.

The algorithm above performs a recursive enumeration of a finite space of assignments to symbols. The algorithm is **sound** because it implements directly the definition of entailment, and **complete** because it works for any KB and α and always terminates; there are only finitely many models to examine.

Basically, the algorithm constructs the truth table seen above using a recursive implementation. In particular, it creates a binary tree where the nodes are the symbols in the KB and the branches are truth assignment for that symbol (*True* or *False*). It starts from the first symbol and recursively produces all the possible assignment for all the possible symbols. The paths from the root to the leaves corresponds to the rows of the table above, i.e. the models. Once the algorithm arrives at a leaf, it checks whether the sentence α holds within the corresponding model (path). The algorithm returns always *True* if the KB is *False* w.r.t the model because we are only interested to see if a sentence is *True* when the KB is *True*. These operations correspond to look at the rows of the table (models) where KB is *True* and check whether α is also *True*. If the sentence α is *True* for all the models where also KB is *True* (i.e. all the leaves *returns True*), then α can be deduces from KB. Note that if all the leaves return *True*, when the algorithm ascents from recursion, the *True* value is propagated towards the root by the logical *and* and eventually the algorithm returns *True*. For the same reason, it is sufficient that just one leaf returns *False* that the whole algorithm returns *False* (this is why the algorithms always returns *True* when KB is *False*).

Note that If KB and α contain n symbols in all, then there are 2^n models. Thus, the time complexity of the algorithm is $O(2^n)$ (The space complexity is only $O(n)$ because the enumeration is depth-first). Unfortunately, propositional entailment is co-NP-complete.

Chapter 9

Lec 09 - Logical Agents II

9.1 Propositional Theorem Proving

So far, we have shown how to determine entailment by model checking: enumerating models and showing that the sentence must hold in all models. In this section, we show how entailment can be done by **theorem proving**, that is, applying rules of **inference** directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models.

Inference rules 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**.

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

The notation means that, whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred. For example, if $(WumpusAhead \wedge WumpusAlive) \Rightarrow Shoot$ and $(WumpusAhead \wedge WumpusAlive)$ are given, then $Shoot$ can be inferred. These techniques typically require translation of sentences into a **normal form**.

Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to be represented with a restricted form which enables more efficient inference algorithms. One such restricted form is the **Horn form**, which is a conjunction of Horn clauses. A Horn clause is defined as follows:

- proposition symbol; or
- (conjunction of symbols) $\Rightarrow$ symbol.

$$C \wedge (B \Rightarrow A) \wedge (C \wedge B \Rightarrow A)$$

In Horn form, the premise is called the **body** and the conclusion is called the **head**. A sentence consisting of a single positive literal, such as $P_{1,1}$, is called a **fact** $(True \Rightarrow P_{1,1})$.

Knowledge bases containing only Horn clauses are interesting for the following reasons:

- Inference with Horn clauses can be done through the forward-chaining and backwardchaining algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow.
- Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base.

9.2 Forward and Backward Chaining

The **forward-chaining** algorithm determines if a single proposition symbol q, the query, is entailed by a knowledge base of Horn clauses. It begins from known facts in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if $L_{1,1}$ and $Breeze$ are known and $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$ is in the knowledge base, then $B_{1,1}$ can be added. This process continues until the query q is added or until no further inferences can be made.

```
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 the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p ← POP(agenda)
    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 agenda
  return false
```

Figure 7.15 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 as yet unknown. 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 $Q \Rightarrow P$.

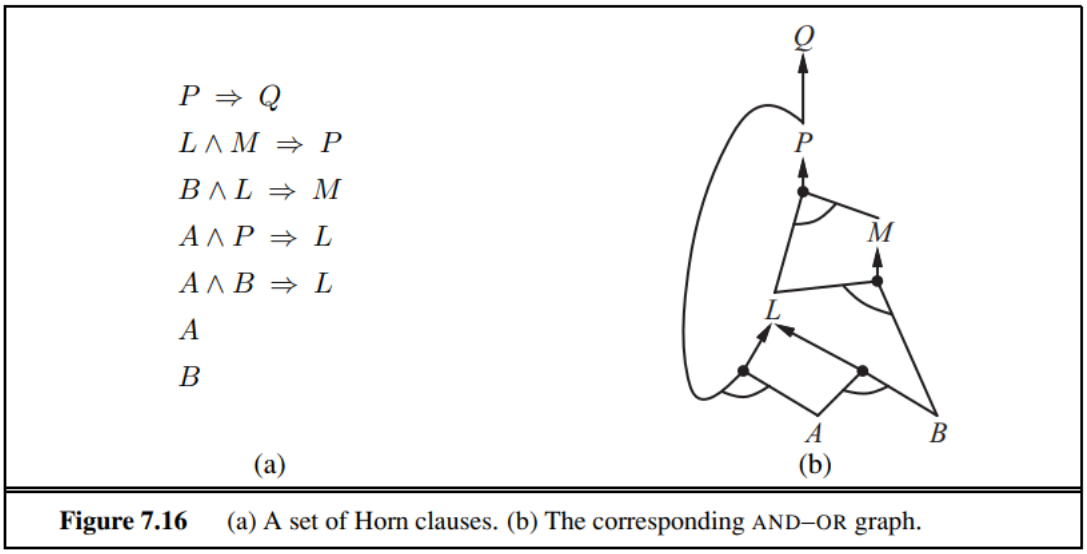
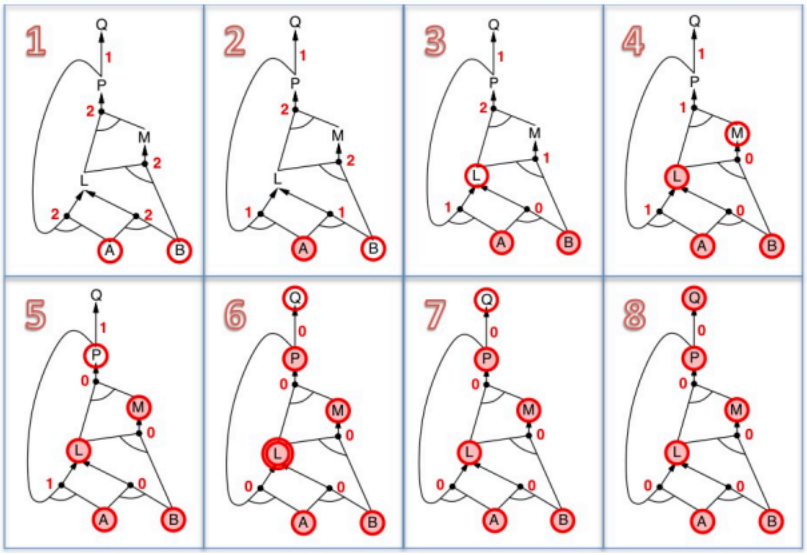


Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

The best way to understand the algorithm is through an example showed in the figure above. It shows a simple knowledge base of Horn clauses with A and B as known facts represented as an AND-OR graph. In AND–OR graphs, multiple links joined by an arc indicate a conjunction, that is, every link must be proved. While multiple links without an arc indicate a disjunction, any link can be proved.

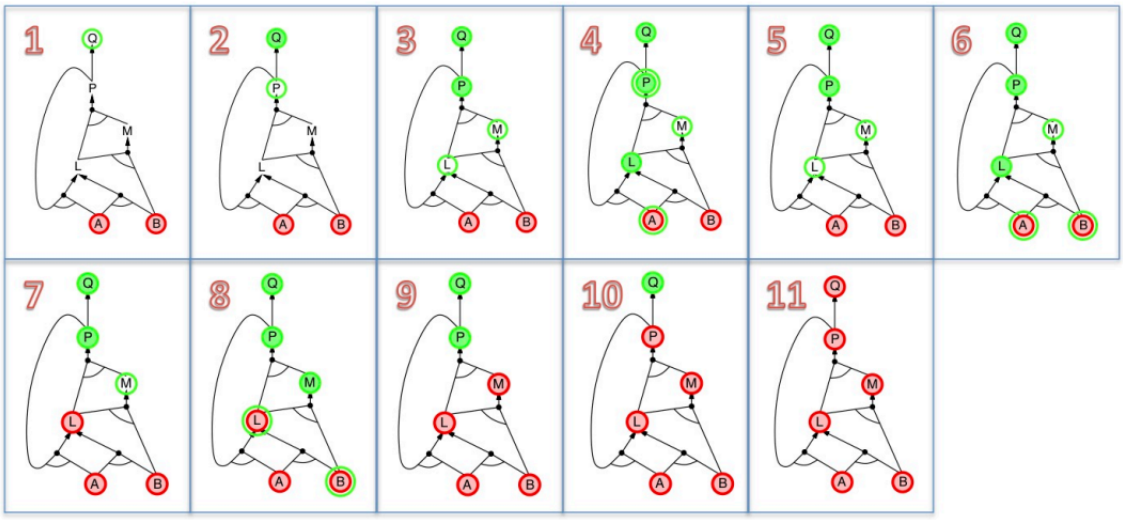


Forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived.

The easiest way to see this is to consider the final state of the *inferred* table (after the algorithm reaches a **fixed point** where no new inferences are possible). The table contains *true* for each symbol inferred during the process, and *false* for all other symbols. We can view the table as a logical model; moreover, every Horn clause in the original KB is true in this model. To see this, assume the opposite, namely that some clause $a_1 \wedge ... \wedge a_k \Rightarrow b$ is *false* in the model. Then $a_1 \wedge ... \wedge a_k$ must be *true* in the model and b must be *false* in the model. But this contradicts our assumption that the algorithm has reached a fixed point. We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines model of the

original KB. Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence q must be inferred by the algorithm.

The **backward-chaining** algorithm, as its name suggests, works backward from the query. If the query q is known to be true, then no work is needed. Otherwise, the algorithm finds those implications in the knowledge base whose conclusion is q. If all the premises of one of those implications can be proved true (by backward chaining), then q is true.



Forward chaining is an example of the general concept of **data-driven** reasoning, that is, reasoning in which the focus of attention starts with the known data. However, it may do lots of work that is irrelevant to the goal.

Backward chaining is a form of **goal-directed** reasoning. It is appropriate for problem solving and for answering specific questions such as "What shall I do now?" and "Where are my keys?" Often, the cost of backward chaining is much less than linear in the size of the knowledge base, because the process touches only relevant facts.

9.3 Resolution

The current section introduces a single inference rule, **resolution**, that yields a complete inference algorithm when coupled with any complete search algorithm. The resolution rule applies only to disjunctions of literals (clauses), so it would seem to be relevant only to knowledge bases and queries consisting of clauses. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that every sentence of propositional logic is logically equivalent to a **conjunction of clauses**. A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF**.

The resolution rule is the following:

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

where l_i and m_j are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses except the two complementary literals. For example:

$$\frac{P_{1,3} \vee P_{2,2}, \qquad \neg P_{2,2}}{P_{1,3}}$$

means that if there is a pit in $[1,3]$ or in $[2,2]$ and the pit is not in $[2,2]$, then the pit is in $[1,3]$.

The soundness of the resolution rule can be seen easily by considering the literal l_i that is complementary to literal m_j in the other clause. If l_i is true, then m_j is false, and hence $m_1 \vee ... \vee m_{j-1} \vee m_{j+1} \vee ... \vee m_n$ must be true, because $m_1 \vee ... \vee m_n$ is **given**. If l_i is false, then $l_1 \vee ... \vee l_{i-1} \vee l_{i+1} \vee ... \vee l_k$ must be true, because $l_1 \vee ... \vee l_k$ is given. Therefore, every derived sentence is also entailed.

What is more surprising about the resolution rule is that it forms the basis for a family of complete inference procedures. A resolution-based theorem prover can, for any sentences α and β in propositional logic, decide whether $\alpha \models \beta$.

9.3.1 A resolution algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction, that is, to show that $KB \models \alpha$, we show that $KB \wedge \neg\alpha$ is unsatisfiable.

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 $\leftarrow$ the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
  new $\leftarrow$ { }
  loop do
    for each pair of clauses C_i, C_j **in** *clauses* **do**
      resolvents $\leftarrow$ PL-RESOLVE(C_i, C_j)
      if *resolvents* contains the empty clause **then return** *true*
      new $\leftarrow$ *new* $\cup$ *resolvents*
    if *new* $\subseteq$ *clauses* **then return** *false*
    clauses $\leftarrow$ *clauses* $\cup$ *new*

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

First, $(KB \wedge \neg\alpha)$ is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:

- there are no new clauses that can be added, in which case KB does not entail α; or,
- two clauses resolve to yield the empty clause, in which case KB entails α.

Note that the empty clause, a disjunction of no disjuncts, is equivalent to *False*, because a disjunction is true only if at least one of its disjuncts is true.

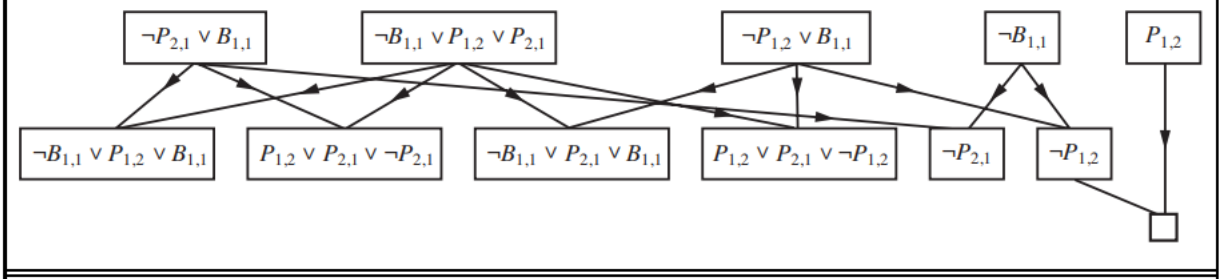


Figure 7.13 Partial application of PL-RESOLUTION to a simple inference in the wumpus world. $\neg P_{1,2}$ is shown to follow from the first four clauses in the top row.

We can apply the resolution procedure to a very simple inference in the wumpus world. When the agent is in $[1,1]$, there is no breeze, so there can be no pits in neighboring squares. The relevant knowledge base is:

$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

When we convert $(KB \wedge \neg\alpha)$ into CNF, we obtain the clauses shown at the top of the figure above. The second row of the figure shows clauses obtained by resolving pairs in the first row. Then, when $P_{1,2}$ is resolved with $\neg P_{1,2}$, we obtain the empty clause, shown as a small square. Therefore, we can deduce that $KB \vDash \neg P_{1,2}$.

9.3.2 Completeness of Resolution Algorithm

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 S, which is the set of all clauses derivable by repeated application of the resolution rule to clauses in S 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.*

This theorem is proved by demonstrating its contrapositive: if the closure $RC(S)$ does **not** contain the empty clause, then S is satisfiable. In fact, we can construct a model for S with suitable truth values for $P_1, ..., P_k$. The construction procedure is as follows:

For i from 1 to k,

- If a clause in $RC(S)$ contains the literal $\neg P_i$ and all its other literals are false under the assignment chosen for $P_1, ...P_{i-1}$, then assign false to P_i.
- Otherwise, assign true to P_i.

This assignment to $P_1, ..., P_k$ is a model of S. To see this, assume the opposite, that, at some stage i in the sequence, assigning symbol P_i causes some clause C to become false. For this to happen, it must be the case that all the other literals in C must already have been falsified by assignments to $P_1, ..., P_{i-1}$. Thus, C must now look like either $(false \vee false \vee ...false \vee P_i)$ or like $(false \vee false \vee ...false \vee \neg P_i)$. If just one of these two is in $RC(S)$, then the procedure will assign the appropriate truth value to P_i to make C true, so C can only be falsified if both of these clauses are in $RC(S)$. Now, since $RC(S)$ is closed under resolution, it

will contain the resolvent of these two clauses, and that resolvent will have all of its literals already falsified by the assignments to $P_1, ..., P_{i-1}$. This contradicts our assumption that the first falsified clause appears at stage i. Hence, we have proved that the construction never falsifies a clause in $RC(S)$, that is, it produces a model of $RC(S)$.

Chapter 10

Lec 10 - Logical Agents III- First Order Logic

10.1 First Order Logic

We showed how a knowledge-based agent could represent the world in which it operates and deduce what actions to take. We used propositional logic as our representation language because it sufficed to illustrate the basic concepts of logic and knowledge-based agents. 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. However, propositional logic lacks the expressive power to concisely describe an environment with many objects.

We can adopt the foundation of propositional logic—a declarative, compositional semantics that is context-independent and unambiguous—and build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks. Whereas propositional logic assumes world contains facts, **first-order logic** (like natural language) assumes the world contains:

- **Objects**: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries ...
- **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, part of, has color, occurred after, owns, comes between, ...
- **Functions**: father of, best friend, third inning of, one more than, beginning of ...

Indeed, almost any assertion can be thought of as referring to objects and properties or relations.

10.2 Syntax of FOL: Basic elements

The basic syntactic elements of first-order logic are the symbols that stand for objects, relations, and functions:

- **variable symbols** $x, y, z, ...$
- **constant symbols** which stand for objects $KingJohn, 2, UCB, ...$

- **function symbols**, which stand for functions.
- **predicate symbols**, which stand for relations.

Another important part of the syntax are the logical connectives:

- **Propositional connectives**: $\wedge, \vee, \neg, \Rightarrow$
- **Quantifiers:** $\forall, \exists$

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

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

10.2.1 Terms

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object. For example, 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. A term can also be a constant or a variable.

10.2.2 Atomic sentences

Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make **atomic sentences** that state facts. An **atomic sentence** is formed from a predicate symbol optionally followed by a parenthesized list of terms, such as $Brother(Richard, John)$. We can use the equality symbol to signify that two terms refer to the same object. For example, $Father(John) = Henry$. Atomic sentences can have complex terms as arguments. Thus, $Married(Father(Richard), Mother(John))$.

10.2.3 Complex sentences

We can use logical connectives to construct more **complex sentences**, with the same syntax and semantics as in propositional calculus:

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \wedge Brother(John, Richard)$$

10.3 Models for first-order logic

The models of a logical language are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values. Models for first-order logic are much more interesting. First, they have objects in them! The **domain** of a model is the set of objects or domain elements it contains. The objects in a model may be related in various ways. Formally speaking, a relation is just the set of tuples of objects that are related.

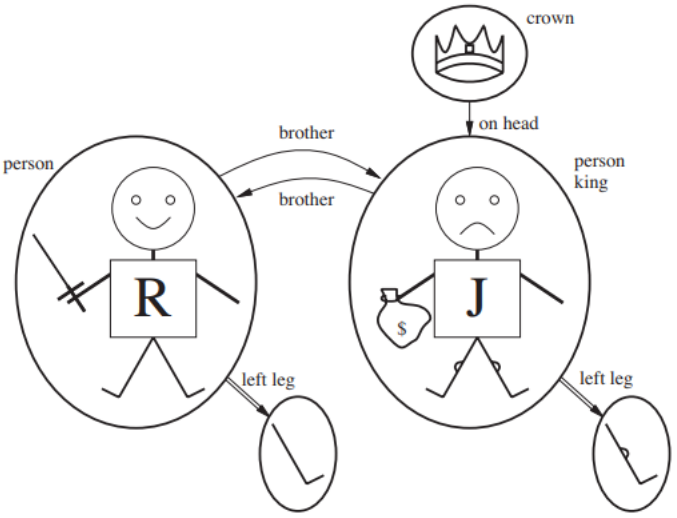


Figure 8.2 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

The figure above 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 crown is on King John's head, so the "on head" relation contains just one tuple, $< thecrown, KingJohn >$. The "brother" and "on head" relations 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 (presumably because Richard is dead at this point); and the "crown" property is true only of the crown. So far, we have described the elements that populate models for first-order logic. The other essential part of a model is the link between those elements and the vocabulary of the logical sentences.

As we said previously, constant symbols stand for objects, predicate symbols stand for relations and function symbols stand for functions. Each predicate and function symbol comes with an **arity** that fixes the number of arguments. As in propositional logic, every model must provide the information required to determine if any given sentence is true or false. Thus, 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. One possible interpretation for our example is as follows:

- **Constants**: *Richard* refers to Richard the Lionheart and *John* refers to the evil King John.
- **Predicates**: *Brother* refers to the brotherhood relation, *OnHead* refers to the "on head" relation that holds between the crown and King John; *Person*, *King*, and *Crown* refer to the sets of objects that are persons, kings, and crowns.
- **Functions:** $LeftLeg$ refers to the "left leg" function.

There are many other possible interpretations, of course. For example, one interpretation maps *Richard* to the crown and *John* to King John's left leg. There are five objects in the model, so there are 25 possible interpretations just for the constant symbols *Richard* and *John*.

For example, the atomic sentence $Brother(Richard, John)$ states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John. Instead, the atomic sentence *Married(Father (Richard), Mother (John))* states that Richard the Lionheart's father is married to King John's mother.

*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.*

Here are some complex sentences that are true in the model under our intended interpretation:

$$\neg Brother(LeftLeg(Richard), John)$$
$$Brother(Richard, John) \wedge Brother(John, Richard)$$
$$King(Richard) \wedge King(John)$$
$$\neg King(Richard) \Rightarrow King(John).$$

In summary, a model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects.

Because the number of possible models is unbounded, checking entailment by the enumeration of all possible models is not feasible for first-order logic (unlike propositional logic). Even if the number of objects is restricted, the number of combinations can be very large.

10.4 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**.

10.4.1 Universal quantification

The **universal quantifier** $\forall$ is of the form $\forall < variables >< sentence >$. For example, "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.

Intuitively, the sentence $\forall x\ P$, where P is any logical expression, says that P is true for every object x. More precisely, $\forall x\ P$ is true in a given model if P is true in all possible extended interpretations constructed from the interpretation given in the model, where each extended interpretation specifies a domain element to which x refers. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king $\Rightarrow$ Richard the Lionheart is a person.

King John is a king $\Rightarrow$ King John is a person.

Richard's left leg is a king $\Rightarrow$ Richard's left leg is a person.

John's left leg is a king $\Rightarrow$ John's left leg is a person.

The crown is a king $\Rightarrow$ the crown is a person

Typically, $\Rightarrow$ is the main connective with $\forall$.

10.4.2 Existential quantification

Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an **existential quantifier**.

To say, for example, that King John has a crown on his head, we write:

$$\exists x\ Crown(x) \wedge OnHead(x, John)$$

Intuitively, the sentence $\exists x\ P$ says that P is true for at least one object x. More precisely, $\exists x\ P$ is true in a given model if P is true in at least one extended interpretation that assigns x to a domain element. That is, at least one of the following is true:

Richard the Lionheart is a crown $\wedge$ Richard the Lionheart is on John's head;

King John is a crown $\wedge$ King John is on John's head;

Richard's left leg is a crown $\wedge$ Richard's left leg is on John's head;

John's left leg is a crown $\wedge$ John's left leg is on John's head;

The crown is a crown $\wedge$ the crown is on John's head

Typically, $\wedge$ is the main connective with $\exists$.

10.4.3 Properties of quantifiers

We will often want to express more complex sentences using multiple quantifiers:

- $\forall x \forall y$ is the same as $\forall y \forall x$;
- $\exists x \exists y$ is the same as $\exists y \exists x$;
- $\exists x \forall y$ is **not** the same as $\forall y \exists x$. In fact, $\exists y \forall x\ Loves(x, y)$ says that "There is a person who loves everyone in the world". On the other hand, $.\forall x(\exists y Loves(x, y))$ says that "Everyone in the world is loved by at least one person".

- **Quantifier duality**: each can be expressed using the other:

$$\forall x\ Likes(x, IceCream) \neg \exists x\ \neg Likes(x, IceCream)$$
$$\exists x\ Likes(x, Broccoli) \neg \forall x\ \neg Likes(x, Broccoli)$$

10.5 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:

$$TELL(KB, King(John)).$$
$$TELL(KB, Person(Richard)).$$
$$TELL(KB, \forall x King(x) \Rightarrow Person(x)).$$

We can ask questions of the knowledge base using ASK. For example, $ASK(KB, King(John))$ returns true. Questions asked with ASK are called **queries** or **goals**.

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

$$ASKVARS(KB, Person(x))$$

and 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**.

10.6 Inference in First Order Logic

This section and the next introduce the ideas underlying modern logical inference systems. We begin with some simple inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers. These rules lead naturally to the idea that first-order inference can be done by converting the knowledge base to propositional logic and using propositional inference, which we already know how to do. In the future, we will provide an obvious shortcut, leading to inference methods that manipulate first-order sentences directly.

10.6.1 Universal Instantiation (UI)

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard folkloric axiom stating that all greedy kings are evil:

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$

Then it seems quite permissible to infer any of the following sentences:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$$
$$King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John))$$
.
.
.

The rule of **Universal Instantiation** (**UI**) says that we can infer any sentence obtained by substituting a ground term (a term without variables) for the variable. Basically, Every instantiation of a universally quantified sentence is entailed by it. To write out the inference rule formally, we use the notion of substitutions introduced previously. Let $Subst(\theta, \alpha)$ denote the result of applying the substitution θ to the sentence α. Then the rule is written as:

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

for any variable v and ground term g. For example, the three sentences given earlier are obtained with the substitutions $\{x/John\}$, $\{x/Richard\}$, and $\{x/Father(John)\}$.

10.6.2 Existential Instantiation

In the rule for **Existential Instantiation**, the variable is replaced by 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,

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

For example, from the sentence

$$\exists x\ Crown(x) \wedge OnHead(x, John)$$

we can infer the sentence

$$Crown(C1) \wedge OnHead(C1, John)$$

as long as C_1 does not appear elsewhere in the knowledge base. 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. Of course, that name must not already belong to another object. C_1 is called **Skolem constant**. Existential Instantiation is a special case of a more general process called **skolemization**, which we will cover in the future.

Whereas Universal Instantiation can be applied many times to add new sentences to KB, Existential Instantiation can be applied once, and then the existentially quantified sentence can be discarded. For example, we no longer need $\exists x\ Kill(x, Victim)$ once we have added the sentence $Kill(Murderer, Victim)$ to KB. Strictly speaking, the new knowledge base is not logically equivalent to the old.

10.6.3 Reduction to propositional inference

Once we have rules for *inferring nonquantified sentences from quantified sentences*, it becomes possible to reduce first-order inference to propositional inference.

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:

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John).$$

Then we apply UI to the first sentence using all possible ground-term substitutions from the vocabulary of the knowledge base, in this case, $\{x/John\}$ and $\{x/Richard\}$. We obtain:

$$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$$
$$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard),$$

and we discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences, $King(John)$, $Greedy(John)$, and so on, as proposition symbols. Therefore, we can apply any of the complete propositional algorithms described previously to obtain conclusions such as $Evil(John)$.

However, there is a problem. When the knowledge base includes a function symbol, the set of possible ground-term substitutions is infinite! For example, if the knowledge base mentions the $Father$ symbol, then infinitely many nested terms such as $Father(Father(Father(John)))$ can be constructed.

Fortunately, there is a famous theorem due to Jacques Herbrand (1930) to the effect that if a sentence is entailed by the original, first-order knowledge base, then there is a proof involving just a finite subset of the propositionalized knowledge base. Since any such subset has a maximum depth of nesting among its ground terms, we can find the subset by first generating all the instantiations with constant symbols ($Richard$ and $John$), then all terms of depth 1 ($Father(Richard)$ and $Father(John)$), then all terms of depth 2, and so on, until we are able to construct a propositional proof of the entailed sentence.

We have sketched an approach to first-order inference via propositionalization that is **complete**, that is, any entailed sentence can be proved. On the other hand, we do not know until the proof is done that the sentence is entailed! What happens when the sentence is not entailed? Alan Turing (1936) and Alonzo Church (1936) both proved, in rather different ways, that *the question of entailment for first-order logic is semidecidable, that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every nonentailed sentence.*

Chapter 11

Lec 11 - Unification and Generalized Modus Ponens

11.1 Problems with propositionalization

The propositionalization approach is rather inefficient. For example, given the query $Evil(x)$ and the knowledge base

$$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John),$$

it seems obvious the inference of $Evil(John)$, but propositionalization generates sentences such as $King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$ that are **irrelevant**. With p k-ary predicates and n constants, there are $p \cdot n^k$ instantiations!

11.2 Unification

The inference that John is evil, that is, that $\{x/John\}$ solves the query $Evil(x)$, works like this: to use the rule that greedy kings are evil, find some x such that x is a king and x is greedy, and then infer that this x is evil.

More generally, if there is some substitution θ that makes each of the conjuncts of the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying θ. In this case, the substitution $\theta = \{x/John\}$ achieves that aim.

We can actually make the inference step do even more work. Suppose that instead of knowing $Greedy(John)$, we know that everyone is greedy:

$$\forall y\ Greedy(y)$$

Then we would still like to be able to conclude that $Evil(John)$. In this case, applying the substitution $\{x/John, y/John\}$ to the implication premises $King(x)$ and $Greedy(x)$ and the knowledge-base sentences $King(John)$ and $Greedy(y)$ will make them identical. Thus, we can infer the conclusion of the implication.

This process is called **unification** and is a key component of all first-order inference algorithms. Unification finds a substitution that make different logical expressions look identical. The UNIFY algorithm takes two sentences and returns a unifier for them 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:

$$\begin{aligned}
&\text{UNIFY}(Knows(John, x),\ Knows(John, Jane)) = \{x/Jane\} \\
&\text{UNIFY}(Knows(John, x),\ Knows(y, Bill)) = \{x/Bill, y/John\} \\
&\text{UNIFY}(Knows(John, x),\ Knows(y, Mother(y))) = \{y/John, x/Mother(John)\} \\
&\text{UNIFY}(Knows(John, x),\ Knows(x, Elizabeth)) = fail\ .
\end{aligned}$$

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 $Knows(x, Elizabeth)$ to x_{17} (a new variable name) without changing its meaning. Now the unification will work:

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

There is one more complication: we said that UNIFY should return a substitution that makes the two arguments look the same. But there could be more than one such unifier. For example, $UNIFY(Knows(John, x), Knows(y, z))$ could return $\{y/John, x/z\}$ or $\{y/John, x/John, z/John\}$. We say that the first unifier is more general than the second, because it places fewer restrictions on the values of the variables. It turns out that, for every unifiable pair of expressions, there is a single **most general unifier** (or **MGU**) that is unique up to renaming and substitution of variables.

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
          y, a variable, constant, list, or compound expression
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

An algorithm for computing most general unifiers is shown in the figure above. The process is simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. There is one expensive step: when matching a variable against a complex term, one must check whether the variable itself occurs inside the term; if it does, the match fails because no consistent unifier can be constructed. For example, $S(x)$ can't unify with $S(S(x))$. This so called **occur check** makes the complexity of the entire algorithm quadratic in the size of the expressions being unified.

11.3 Generalized Modus Ponens (GMP)

For atomic sentences p_i, p_i' , and q, where there is a substitution θ, such that $SUBST(\theta, p_i') = SUBST(\theta, p_i)$, for all i,

$$\frac{p_1',\ \ p_2',\ \ \ldots,\ \ p_n',\ \ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\text{SUBST}(\theta, q)}$$

This inference rule is called **Generalized Modus Ponens** (**GMP**). 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' is $King(John)$ $\quad$ p_1 is $King(x)$
p_2' is $Greedy(y)$ $\quad$ p_2 is $Greedy(x)$
θ is $\{x/John, y/John\}$ $\quad$ q is $Evil(x)$
SUBST(θ, q) is $Evil(John)$.

Generalized Modus Ponens is a **sound** inference rule. We need to show that $p_1', ..., p_n'$, $(p_1 \wedge ... \wedge p_n \Rightarrow q) \vDash Subst(\theta, q)$, provided that $Subst(\theta, p_i') = Subst(\theta, p_i)$ for all i. First, we observe that, for any sentence p (whose variables are assumed to be universally quantified) and for any substitution θ,

$$p \vDash Subst(\theta, p)$$

holds by Universal Instantiation. It holds in particular for a θ that satisfies the conditions of the Generalized Modus Ponens rule. Thus, from $p_1', ..., p_n'$ we can infer

$$Subst(\theta, p_i') \wedge ... \wedge Subst(\theta, p_n')$$

and from the implication $p_1 \wedge ... \wedge p_n \Rightarrow q$ we can infer

$$Subst(\theta, p_1) \wedge ... \wedge Subst(\theta, p_n) \Rightarrow Subst(\theta, q).$$

Now, θ in Generalized Modus Ponens is defined so that $Subst(\theta, p_i') = Subst(\theta, p_i)$, for all i; therefore the first of these two sentences matches the premise of the second exactly. Hence, $Subst(\theta, q)$ follows by Modus Ponens.

11.4 First-order definite clauses

A first order definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. Unlike propositional literals, first-order literals can include variables, in which case those variables are **assumed to be universally quantified**. (Typically, we omit universal quantifiers when writing definite clauses.) Consider 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."

We will prove that West is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the problem.

"... it is a crime for an American to sell weapons to hostile nations":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x). \tag{11.1}$$

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

$$Owns(Nono, M1) \tag{11.2}$$

$$Missile(M1) \tag{11.3}$$

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

$$Missile(x) \wedge Owns(Nono, x) \Rightarrow Sells(West, x, Nono). \tag{11.4}$$

We will also need to know that missiles are weapons:

$$Missile(x) \Rightarrow Weapon(x). \tag{11.5}$$

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

$$Enemy(x, America) \Rightarrow Hostile(x). \tag{11.6}$$

"West, who is American ...":

$$American(West) \tag{11.7}$$

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

$$Enemy(Nono, America) \tag{11.8}$$

This knowledge base contains no function symbols and is therefore an instance of the class of **Datalog** knowledge bases. Datalog is a language that is restricted to 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. We will see that the absence of function symbols makes inference much easier.

11.5 Forward Chaining Algorithm

The first forward-chaining algorithm we consider is a simple one. Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered (assuming that just one answer is required) or no new facts are added. We use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (1), (4), (5), (6). Two iterations are required:

- On the first iteration, rule (1) has unsatisfied premises.
  Rule (4) is satisfied with $\{x/M_1\}$, and $Sells(West, M_1, Nono)$ is added to the known facts. Note that Rule (4) is satisfied because the substitution $\{x/M_1\}$ makes its premises equivalent to known facts in the KB.
  Rule (5) is satisfied with $\{x/M_1\}$, and $Weapon(M_1)$ is added.
  Rule (6) is satisfied with $\{x/Nono\}$, and $Hostile(Nono)$ is added.

- On the second iteration, rule (1) is satisfied with $\{x/West, y/M_1, z/Nono\}$, and $Criminal(West)$ is added.

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    new ← { }
    for each rule in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new then
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

Figure 9.3 A conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB. The function STANDARDIZE-VARIABLES replaces all variables in its arguments with new ones that have not been used before.

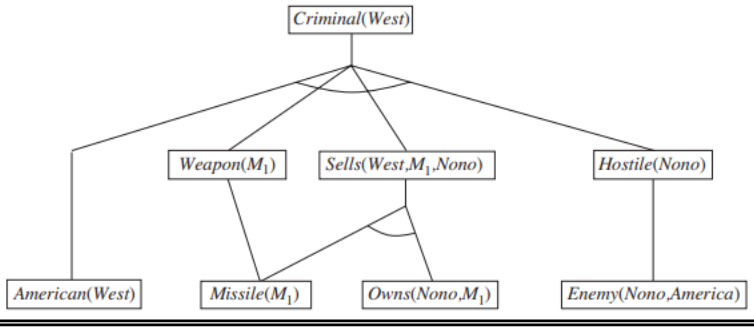


Figure 9.4 The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.

Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process.

Forward chaining is **sound**, because every inference is just an application of Generalized Modus Ponens, which is sound. Second, it is **complete** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses.

For Datalog knowledge bases, which contain no function symbols, forward chaining terminates in poly iterations. In fact, there can be no more than pn^k distinct ground facts, where k is the maximum arity (number of arguments) of any predicate, p be the number of predicates, and n be the number of constant symbols.

For general definite clauses with function symbols, If the query has no answer, the algorithm could fail to terminate in some cases. This is unavoidable since entailment in first-order logic is semidecidable.

11.5.1 Efficiency of forward chaining

The forward-chaining algorithm shown in the figure above is designed for ease of understanding rather than for efficiency of operation. There are three possible sources of inefficiency. First, the "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive. Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration. Finally, the algorithm might generate many facts that are irrelevant to the goal.

The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. However, it turns out that this problem is NP-Hard. Forward chaining can be extend to deal with these problems, but we will not see these extensions.

Chapter 12

Lec 12 - Backward-chaining and Prolog

12.1 Backward Chaining

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

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ'

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first,rest ← FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ'' in FOL-BC-AND(KB, rest, θ') do
        yield θ''
```

Figure 9.6 A simple backward-chaining algorithm for first-order knowledge bases.

The figure above shows a backward-chaining algorithm for definite clauses. FOL-BC-ASK(KB, goal) will be proved if the knowledge base contains a clause of the form $lhs \Rightarrow goal$, where lhs (left-hand side) is a list of conjuncts. 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. FOL-BC-OR works by fetching all clauses that might unify with the goal, standardizing the variables in the clause to be brand-new variables, and then, if the rhs of the clause does indeed unify with the goal, proving every conjunct in the lhs, using FOL-BC-AND. That function in turn works by proving

each of the conjuncts in turn, keeping track of the accumulated substitution as we go.

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**. This could be fixed by checking current goal against every goal on stack. This algorithm is widely used for **logic programming**.

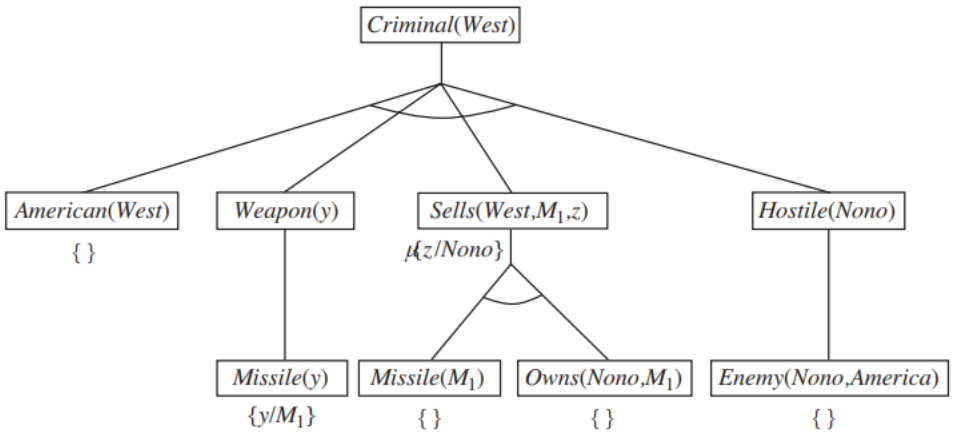


Figure 9.7 Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove $Criminal(West)$, we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. 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 $Nono$.

12.2 Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative ideal described before: that systems should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge.

Prolog is the most widely used logic programming language. Prolog programs are sets of definite clauses written in a notation somewhat different from standard first-order logic. Prolog uses uppercase letters for variables and lowercase for constants, the opposite of our convention for logic. Commas separate conjuncts in a clause, and the clause is written "backwards" from what we are used to. instead of $A \wedge B \Rightarrow C$ in Prolog we have $C :- A, B$.

The execution of Prolog programs is done through depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base. There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved" by executing code rather than doing further inference.

The notation $[E|L]$ denotes a list whose first element is E and whose rest is L. Here is a Prolog program for $append(X, Y, Z)$, which succeeds if list Z is the result of appending lists X and Y:

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z).
```

In English, we can read these clauses as (1) appending an empty list with a list Y produces the same list Y and (2) $[A|Z]$ is the result of appending $[A|X]$ onto Y, provided that Z is the result of appending X onto Y. We can ask the query $append(X, Y, [1, 2])$: what two lists can be appended to give $[1, 2]$? We get back the solutions:

```
X=[]    Y=[1,2];
X=[1]   Y=[2];
X=[1,2] Y=[]
```

Chapter 13

Lec 13 - Resolution for First Order Logic

13.1 Resolution

The last of our three families of logical systems is based on **resolution**. In resolution for first order logic 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

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

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

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

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

$$[Animal(F(x)) \vee \neg Kills(G(x), x)].$$

Then, the resolution steps can be applied to $CNF(KB \wedge \neg\alpha)$ as we did for propositional logic.

13.2 Conversion to CNF

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF), that is, a conjunction of clauses, where each clause is a disjunction of literals. Literals can contain variables, which are assumed to be universally quantified. Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.

We illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or

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

The steps are as follows:

- **Eliminate implications:**

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

- **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:

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

- **Standardize variables**: each quantifier should use a different one:

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

- **Skolemize: Skolemization** is the process of removing existential quantifiers by elimination. In the simple case, it is just like the Existential Instantiation rule seen previously: 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. If we blindly apply the rule to the two matching parts we get

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

  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. Thus, we want the Skolem entities to depend on x and z:

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

  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.

- **Drop universal quantifiers:** At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers:

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

- **Distribute $\vee$ over $\wedge$**:

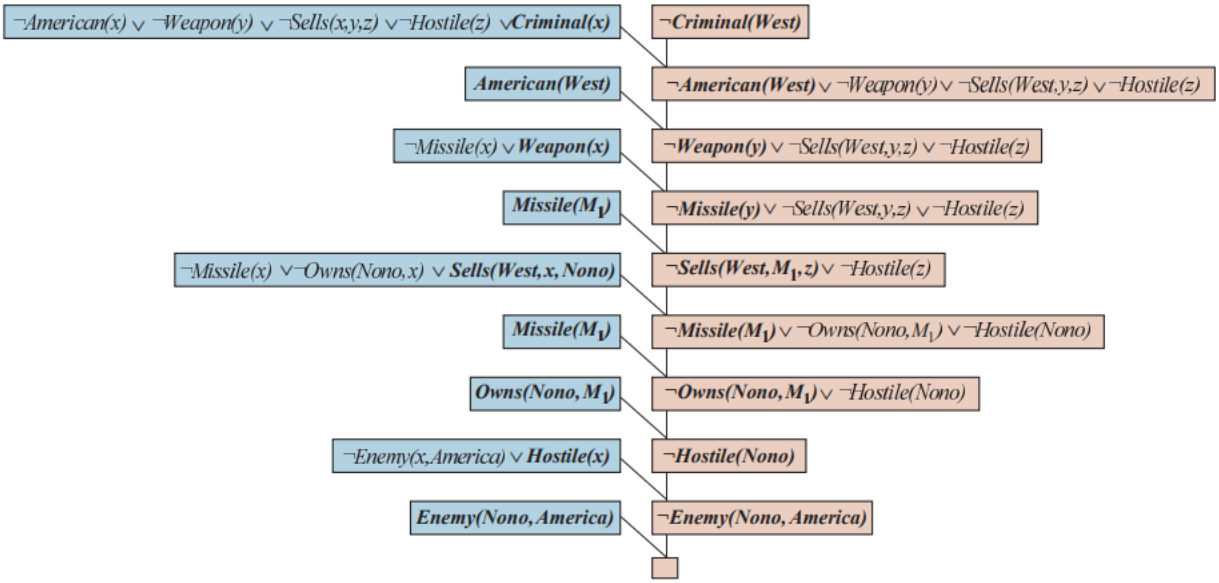
$$[Animal(F(x)) \vee Loves(G(z),x)] \wedge [\neg Loves(x,F(x)) \vee Loves(G(z),x)].$$

The sentence is now in CNF and consists of two clauses.

13.3 Example proofs

Resolution proves that $KB \models \alpha$ by proving $KB \wedge \neg\alpha$ unsatisfiable, that is, by deriving the empty clause. The algorithmic approach is identical to the propositional case. We give two example proofs. The first is the crime example presented in the previous sections. The sentences in CNF are:

$$\neg American(x) \lor \neg Weapon(y) \lor \neg Sells(x, y, z) \lor \neg Hostile(z) \lor Criminal(x)$$
$$\neg Missile(x) \lor \neg Owns(Nono, x) \lor Sells(West, x, Nono)$$
$$\neg Enemy(x, America) \lor Hostile(x)$$
$$\neg Missile(x) \lor Weapon(x)$$
$$Owns(Nono, M_1) \qquad Missile(M_1)$$
$$American(West) \qquad Enemy(Nono, America) .$$



The resolution proof is shown in the figure above. Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. This is characteristic of resolution on Horn clause knowledge bases. In fact, the clauses along the main spine correspond exactly to the consecutive values of the goals variable in the backward-chaining algorithm.

Our second example makes use of Skolemization and involves clauses that are not definite clauses. This results in a somewhat more complex proof structure. In English, the problem is as follows:

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.
Did Curiosity kill the cat?

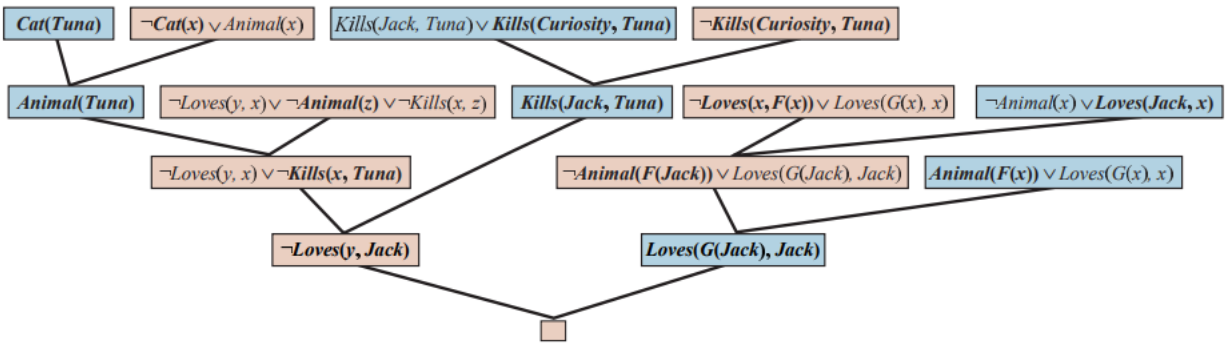
First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

A. $\forall x\ [\forall y\ \ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ \ Loves(y, x)]$
B. $\forall x\ [\exists z\ \ Animal(z) \wedge Kills(x, z)] \Rightarrow [\forall y\ \neg Loves(y, x)]$
C. $\forall x\ \ Animal(x) \Rightarrow Loves(Jack, x)$
D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
E. $Cat(Tuna)$
F. $\forall x\ \ Cat(x) \Rightarrow Animal(x)$
¬G. $\neg Kills(Curiosity, Tuna)$

Now we apply the conversion procedure to convert each sentence to CNF:

A1. $Animal(F(x)) \vee Loves(G(x), x)$
A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$
B. $\neg Loves(y, x) \vee \neg Animal(z) \vee \neg Kills(x, z)$
C. $\neg Animal(x) \vee Loves(Jack, x)$
D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$
E. $Cat(Tuna)$
F. $\neg Cat(x) \vee Animal(x)$
¬G. $\neg Kills(Curiosity, Tuna)$

The resolution proof that Curiosity killed the cat is given in the figure below:



The proof answers the question "Did Curiosity kill the cat?" but often we want to pose more general questions, such as "Who killed the cat?". Then, the goal is $\exists w\, Kills(w, Tuna)$, which, when negated, becomes $\neg Kills(w, Tuna)$ in CNF. Repeating the proof with the new negated goal, we obtain a similar proof tree, but with the substitution w/Curiosity in one of the steps. Unfortunately, resolution can produce **nonconstructive proofs** for existential goals. For example, $\neg Kills(w, Tuna)$ resolves with $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$ to give $Kills(Jack, Tuna)$, which resolves again with $\neg Kills(w, Tuna)$ to yield the empty clause. Notice that w has two different bindings in this proof; resolution is telling us that, yes, someone killed Tuna, either Jack or Curiosity. This is no great surprise!

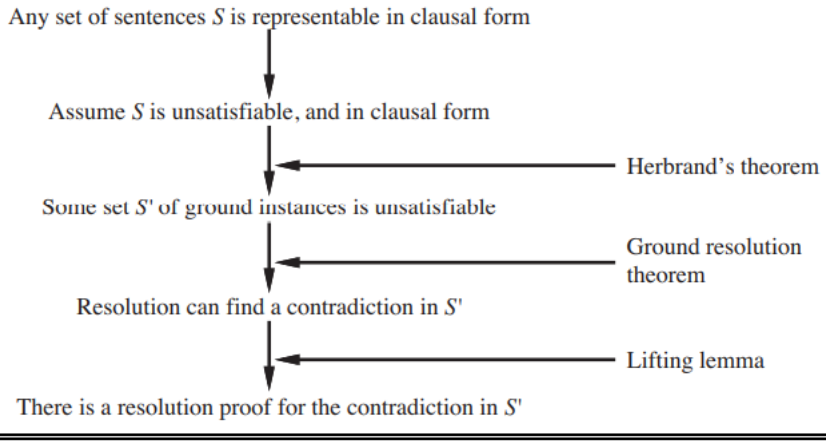


Figure 9.13 Structure of a completeness proof for resolution.

13.4 Resolution strategies

We know that repeated applications of the resolution inference rule will eventually find a proof if one exists. In this subsection, we examine strategies that help find proofs efficiently.

- **Unit preference/clause**: prefers to do resolutions where one of the sentences is a single literal. The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorter clauses.
- **Unit resolution:** is a restricted form of resolution in which every resolution step **must** involve a unit clause. Unit resolution is incomplete in general, but complete for Horn clauses. Unit resolution proofs on Horn clauses resemble forward chaining.
- **Set of support:** every resolution step involve at least one element of a special set of clauses (the set of support); the resolvent is then added into the set of support. Incomplete if the wrong set of support is chosen.
- **Input resolution:** In this strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. Complete for knowledge bases that are in Horn form, but incomplete in the general case. Input resolution has the characteristic shape of a single "spine" with single sentences combining onto the spine. The **linear resolution** strategy is a slight generalization that allows P and Q to be resolved together either if P is in the original KB or if P is an ancestor of Q in the proof tree. Linear resolution is complete.
- **Subsumption:** eliminates all sentences that are subsumed by (that is, more specific than) an existing sentence in the KB. For example, if $P(x)$ is in the KB, then there is no sense in adding $P(A)$ and even less sense in adding $P(A) \vee Q(B)$. Subsumption helps keep the KB small and thus helps keep the search space small.

Chapter 14

Lec 14 - Dealing with Uncertainty

14.1 Uncertainty

Agents may need to handle **uncertainty**, whether due to partial observability, nondeterminism, or a combination of the two. Suppose, for example, that an automated taxi has the goal of delivering a passenger to the airport on time. The agent forms a plan, A_{90}, that involves leaving home 90 minutes before the flight departs and driving at a reasonable speed. Even though the airport is only about 5 miles away, a logical taxi agent will not be able to conclude with certainty that "Plan A_{90} will get us to the airport in time." Instead, it reaches the weaker conclusion "Plan A_{90} will get us to the airport in time, as long as the car doesn't break down or run out of gas, and I don't get into an accident, and there are no accidents on the bridge, and the plane doesn't leave early, and no meteorite hits the car, and ... ." None of these conditions can be deduced for sure, so the plan's success cannot be inferred. Other plans, such as A_{180}, might increase the agent's belief that it will get to the airport on time, but also increase the likelihood of a long wait. *The right thing to do, the rational decision, therefore depends on both the relative importance of various goals and the likelihood that, and degree to which, they will be achieved.*

A logical agent believes each sentence to be true or false or has no opinion, whereas a probabilistic agent may have a numerical degree of belief between 0 (for sentences that are certainly false) and 1 (certainly true). **Probability** provides a way of summarizing the uncertainty that comes from our

- **laziness:** failure to enumerate exceptions, qualifications, etc.
- **ignorance**: lack of relevant facts, initial conditions, etc.

Probabilities relate propositions to one's own state of knowledge, e.g. $P(A_{25}|$ no reported accidents$) = 0.06$. These are not claims of some probabilistic tendency in the current situation, but might be learned from past experience of similar situations. Probabilities of propositions change with new evidence:

$$P(A_{25}|\text{ no reported accidents, 5 a.m.}) = 0.15$$

Consider again the A_{90} plan for getting to the airport. Suppose it gives us a 97% chance of catching our flight. Does this mean it is a rational choice? Not necessarily: there might be other plans, such as A_{180}, with higher probabilities. If it is vital not to miss the flight, then it is worth risking the longer wait at the airport. What about A_{1440}, a plan that involves leaving home 24 hours in advance? In most circumstances, this is not a good choice, because although it almost guarantees getting there on time, it involves an intolerable wait.

To make such choices, an agent must first have **preferences** between the different possible outcomes of the various plans. We use **utility theory** to represent and reason with preferences. Utility theory says that every state has a degree of usefulness, or utility, to an agent and that the agent will prefer states with

higher utility. Preferences, as expressed by utilities, are combined with probabilities in the general theory of rational decisions called **decision theory**:

$$Decision\ theory = Probability\ theory + Utility\ theory$$

14.2 Probability notation

For our agent to represent and use probabilistic information, we need a formal language.

- Probabilities such as $P(Cavity = true) = 0.1$ and $P(Weather = sunny) = 0.72$ are called **prior** or **unconditional** probabilities. They refer to degrees of belief in propositions in the absence of any other information.
- Most of the time, we have some information, usually called **evidence**, that has already been revealed. In this case we have the **conditional** or **posterior** probability. For example, $P(cavity|toothache) = 0.6$. This assertion does **not** mean "Whenever toothache is true, conclude that cavity is true with probability 0.6". Rather it means "Whenever toothache is true and we have no further information, conclude that cavity is true with probability 0.6." The extra condition is important; for example, if we had the further information that the dentist found no cavities, we definitely would not want to conclude that cavity is true with probability 0.6; instead we need to use $P(cavity|toothache \wedge \neg cavity) = 0$. Note also that new evidence may be irrelevant, allowing simplification, e.g., $P(cavity|toothache, InterWin) = P(cavity|toothache) = 0.6$.

  Mathematically speaking, conditional probabilities are defined in terms of unconditional probabilities as follows: for any events a and b, we have

  $$P(a|b) = \frac{P(a \wedge b)}{P(b)}$$

  The definition of conditional probability can be written in a different form called the **product rule**:

  $$P(a \wedge b) = P(a|b)P(b)$$

- Variables in probability theory are called **random variables** and their names begin with an uppercase letter. Every random variable has a **domain**, the set of possible values it can take on. A **Probability distribution** gives values for all possible assignments to a random variable. E.g., if $Weather$ can take values in
  $< sunny, rain, cloudy, snow >$, then

  $$\mathbf{P}(Weather) =< 0.72, 0.1, 0.08, 0.1 >$$

  Normalized, i.e., sums to 1. For continuous variables, it is not possible to write out the entire distribution as a vector, because there are infinitely many values. Instead, we can define the probability that a random variable takes on some value x as a parameterized function of x. We call this a **probability density function**.

- In addition to distributions on single variables, we need notation for distributions on multiple variables. The **Joint probability distribution** for a set of random variables gives the probabilities of all combinations of the values of those random variables (i.e., every sample point). For example, $\mathbf{P}(Weather, Cavity)$ is a 4×2 table of probabilities:

Weather =	sunny	rain	cloudy	snow
Cavity = true	0.144	0.02	0.016	0.02
Cavity = false	0.576	0.08	0.064	0.08

Every question about a domain can be answered by the joint distribution because every event is a sum of sample points.

Any joint probability distribution over many random variables may be decomposed into conditional distributions.

$$P(\mathrm{x}^{(1)},\ldots,\mathrm{x}^{(n)}) = P(\mathrm{x}^{(1)})\Pi_{i=2}^{n} P(\mathrm{x}^{(i)} \mid \mathrm{x}^{(1)},\ldots,\mathrm{x}^{(i-1)}).$$

This observation is known as the **chain rule** of probability.

14.3 Inference by enumeration

Inference by enumeration means doing inference using only full joint distributions. **Probabilistic inference** is the computation of posterior probabilities for query propositions given observed evidence.

We begin with a simple example: a domain consisting of just the three Boolean variables *Toothache*, *Cavity*, and *Catch* (the dentist's nasty steel probe catches in my tooth). The full joint distribution is a $2 \times 2 \times 2$ table.

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

The probability associated with a proposition is defined to be the sum of the probabilities of the atomic events in which it holds: For any proposition ϕ

$$P(\phi) = \sum_{w \in \phi} P(w)$$

For example, when rolling fair dice, we have $P(Total = 11) = P((5,6)) + P((6,5)) = 1/36 + 1/36 = 1/18$.

Then, we can calculate the probability $P(cavity \vee toothache)$ as follows:

$$P(cavity \vee toothache) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28$$

We simply identify those events in which the proposition is true and add up their probabilities.

	toothache		¬ *toothache*	
	catch	¬ *catch*	*catch*	¬ *catch*
cavity	**.108**	**.012**	**.072**	**.008**
¬ *cavity*	**.016**	**.064**	**.144**	**.576**

One particularly common task is to extract the distribution over some subset of variables or a single variable. For example, adding the entries in the first row gives the unconditional or **marginal probability** of cavity:

$$P(cavity) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2$$

This process is called **marginalization**, or **summing out**, because we sum up the probabilities for each possible value of the other variables. We can write the following general marginalization rule for any sets of variables $\mathbf{Y}$ and $\mathbf{Z}$:

$$\mathbf{P}(\mathbf{Y}) = \sum_{\mathbf{z} \in \mathbf{Z}} \mathbf{P}(\mathbf{Y}, \mathbf{z})$$

where $\sum_{\mathbf{z} \in \mathbf{Z}}$ means to sum over all the possible combinations of values of the set of variables $\mathbf{Z}$.

$$\mathbf{P}(Cavity) = \sum_{\mathbf{z} \in \{Catch, Toothache\}} \mathbf{P}(Cavity, \mathbf{z})$$

We can also compute conditional probabilities, for example:

$$\begin{aligned} P(cavity|toothache) &= \frac{P(cavity \wedge toothache)}{P(toothache)} \\ &= \frac{0.108 + 0.012}{0.108 + 0.012 + 0.016 + 0.064} \\ &= 0.6 \end{aligned}$$

Just to check, we can also compute the probability that there is no cavity, given a toothache:

$$\begin{aligned} P(\neg cavity|toothache) &= \frac{P(\neg cavity \wedge toothache)}{P(toothache)} \\ &= \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} \\ &= 0.4 \end{aligned}$$

The two values sum to 1.0, as they should. Notice that in these two calculations the term $1/P(toothache)$ remains constant, no matter which value of $Cavity$ we calculate. In fact, it can be viewed as a normalization constant for the distribution $\mathbf{P}(Cavity|toothache)$, ensuring that it adds up to 1. With this notation, we can write the two preceding equations in one:

$$\begin{aligned} &\mathbf{P}(Cavity \mid toothache) = \alpha\, \mathbf{P}(Cavity, toothache) \\ &= \alpha\, [\mathbf{P}(Cavity, toothache, catch) + \mathbf{P}(Cavity, toothache, \neg catch)] \\ &= \alpha\, [\langle 0.108, 0.016 \rangle + \langle 0.012, 0.064 \rangle] = \alpha\, \langle 0.12, 0.08 \rangle = \langle 0.6, 0.4 \rangle\ . \end{aligned}$$

Where α is the normalization constant. In other words, we can calculate $\mathbf{P}(Cavity|toothache)$ even if we don't know the value of $P(toothache)$. We temporarily forget about the factor $1/P(toothache)$ and add up the values for *cavity* and $\neg$*cavity*, getting 0.12 and 0.08. Those are the correct relative proportions, but they don't sum to 1, so we normalize them by dividing each one by 0.12 + 0.08, getting the true probabilities of 0.6 and 0.4.

From the example, we can extract a general inference procedure. We begin with the case in which the query involves a single variable, X (Cavity in the example). Let $\mathbf{E}$ be the list of evidence variables (just $Toothache$ in the example), let $\mathbf{e}$ be the list of observed values for them, and let $\mathbf{Y}$ be the remaining unobserved (hidden) variables (just $Catch$ in the example). The query is $\mathbf{P}(X|\mathbf{e})$ and can be evaluated as:

$$\mathbf{P}(X|\mathbf{e}) = \alpha \mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y})$$

where the summation is over all possible $\mathbf{y}$s (i.e., all possible combinations of values of the unobserved variables $\mathbf{Y}$).

Given the full joint distribution to work with, this inference procedure can answer probabilistic queries for discrete variables. It does not scale well, however: for a domain described by n Boolean variables, it requires an input table of size $O(2^n)$ and takes $O(2^n)$ time to process the table.

14.4 Independence

Two variables X and Y are **independent** iff

$$\mathbf{P}(X|Y) = \mathbf{P}(X) \text{ or } \mathbf{P}(Y|X) = \mathbf{P}(Y) \text{ or } \mathbf{P}(X,Y) = \mathbf{P}(X)\mathbf{P}(Y)$$

Let us expand the full joint distribution seen before by adding a fourth variable, $Weather$. The full joint distribution then becomes $\mathbf{P}(Toothache, Catch, Cavity, Weather)$, which has $2 \times 2 \times 2 \times 4 = 32$ entries. We may ask how these variables are related, for example, how are $P(toothache, catch, cavity, cloudy)$ and $P(toothache, catch, cavity)$ related? We can use the product rule:

$$P(toothache, catch, cavity, cloudy) = P(cloudy|toothache, catch, cavity)P(toothache, catch, cavity)$$

Now, it seems safe to say that the weather does not influence the dental variables. Therefore, the following assertion seems reasonable:

$$P(cloudy|toothache, catch, cavity) = P(cloudy)$$

From this, we can deduce:

$$P(toothache, catch, cavity, cloudy) = P(cloudy)P(toothache, catch, cavity)$$

A similar equation exists for every entry in $\mathbf{P}(Toothache, Catch, Cavity, Weather)$. In fact, we can write the general equation:

$$\mathbf{P}(Toothache, Catch, Cavity, Weather) = \mathbf{P}(Toothache, Catch, Cavity)\mathbf{P}(Weather).$$

Therefore, the weather is independent of one's dental problems.

If the complete set of variables can be divided into independent subsets, then the full joint distribution can be **factored** into separate joint distributions on those subsets. In a more practical vein, the independence of dentistry and meteorology is a good thing, because otherwise the practice of dentistry might require intimate knowledge of meteorology, and vice versa. When they are available, then, independence assertions can help in reducing the size of the domain representation and the complexity of the inference problem. Unfortunately, clean separation of entire sets of variables by independence is quite rare.

Chapter 15

Lec 15 - Dealing with Uncertainty II

15.1 Conditional independence

The notion of independence provides a clue of how we can simplify expressions, but it needs refining. It would be nice if $Toothache$ and $Catch$ were independent, but they are not: if the probe catches in the tooth, then it is likely that the tooth has a cavity and that the cavity causes a toothache. These variables *are* independent, however, *given the presence or the absence of a cavity.* Each is directly caused by the cavity, but neither has a direct effect on the other: toothache depends on the state of the nerves in the tooth, whereas the probe's accuracy depends on the dentist's skill, to which the toothache is irrelevant. Mathematically, this property is written as:

$$\mathbf{P}(toothache \wedge catch|Cavity) = \mathbf{P}(toothache|Cavity)\mathbf{P}(catch|Cavity).$$

This equation expresses the **conditional independence** of *toothache* and *catch* given $Cavity$. The general definition of conditional independence of two variables X and Y , given a third variable Z, is

$$\mathbf{P}(X, Y|Z) = \mathbf{P}(X|Z)\mathbf{P}(Y|Z)$$

In the dentist domain, for example, it seems reasonable to assert conditional independence of the variables $Toothache$ and $Catch$, given $Cavity$:

$$\mathbf{P}(Toothache, Catch|Cavity) = \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)$$

Notice that this assertion is somewhat stronger than the previous one, which asserts independence only for specific values of $Toothache$ and $Catch$.

Equivalently, conditional independence may be stated as:

$$\mathbf{P}(X|Y, Z) = \mathbf{P}(X|Z)$$

Proof of the equivalent definition [edit]

$$P(A, B \mid C) = P(A \mid C)P(B \mid C)$$

$$\text{iff } \frac{P(A,B,C)}{P(C)} = \left(\frac{P(A,C)}{P(C)}\right)\left(\frac{P(B,C)}{P(C)}\right) \quad \text{(definition of conditional probability)}$$

$$\text{iff } P(A,B,C) = \frac{P(A,C)P(B,C)}{P(C)} \quad \text{(multiply both sides by } P(C)\text{)}$$

$$\text{iff } \frac{P(A,B,C)}{P(B,C)} = \frac{P(A,C)}{P(C)} \quad \text{(divide both sides by } P(B,C)\text{)}$$

$$\text{iff } P(A \mid B, C) = P(A \mid C) \quad \text{(definition of conditional probability)} \therefore$$

We have shown that absolute independence assertions allow a decomposition of the full joint distribution into much smaller pieces. It turns out that the same is true for conditional independence assertions. For example, we can write out the joint distribution $\mathbf{P}(Toothache, Catch, Cavity)$ using the product rule as follows:

$$\mathbf{P}(Toothache, Catch, Cavity) = \mathbf{P}(Toothache, Catch|Cavity)P(Cavity)$$

Then, we can apply the conditional independence of $Toothache$ and $Catch$, given $Cavity$:

$$= \mathbf{P}(Toothache|Cavity)\mathbf{P}(Catch|Cavity)\mathbf{P}(Cavity)$$

In this way, the original large table is decomposed into three smaller tables. The original table has seven independent numbers ($2^3 = 8$ entries in the table, but they must sum to 1, so 7 are independent). The smaller tables contain five independent numbers. Going from seven to five might not seem like a major triumph, but the point is that, for n symptoms that are all conditionally independent given $Cavity$, the size of the representation grows as $O(n)$ instead of $O(2^n)$.

That means that conditional independence assertions can allow probabilistic systems to scale up; moreover, they are much more commonly available than absolute independence assertions.

15.2 Bayes' Rule

The product rule we defined before can actually be written in two forms:

$$P(a \wedge b) = P(a|b)P(b) \quad \text{and} \quad P(a \wedge b) = P(b|a)P(a)$$

Equating the two right-hand sides and dividing by $P(a)$, we get

$$P(b|a) = \frac{P(a|b)P(b)}{P(a)}$$

This equation is known as **Bayes' rule**. This simple equation underlies most modern AI systems for probabilistic inference.

The more general case of Bayes' rule for multivalued variables can be written in distribution form:

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} = \alpha\mathbf{P}(X|Y)\mathbf{P}(Y)$$

It allows us to compute the single term $P(b|a)$ in terms of three terms: $P(a|b)$, $P(b)$, and $P(a)$. Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these

three numbers and need to compute the fourth. Often, we perceive as evidence the effect of some unknown cause and we would like to determine that cause. In that case, Bayes' rule becomes:

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

The conditional probability $P(effect|cause)$ quantifies the relationship in the causal direction, whereas $P(cause|effect)$ describes the diagnostic direction.

For example, a doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 70% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1%. Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have:

$$P(m|s) = \frac{P(s|m)P(m)}{P(s)} = \frac{0.7 \times 1/50,000}{0.01} = 0.0014$$

That is, we expect less than 1 in 700 patients with a stiff neck to have meningitis. We can avoid assessing the prior probability of the evidence (here, $P(s)$) by instead computing a posterior probability for each value of the query variable (here, m and $\neg m$):

$$\mathbf{P}(M|s) = \alpha < P(s|m)P(m), P(s|\neg m)P(\neg m) > .$$

Thus, to use this approach we need to estimate $P(s|\neg m)$ instead of $P(s)$. There is no free lunch, sometimes this is easier, sometimes it is harder.

What happens when we have two or more pieces of evidence? For example, what can a dentist conclude if her nasty steel probe catches in the aching tooth of a patient?

$$\mathbf{P}(Cavity|toothache \wedge catch)$$

We can try using Bayes' rule to reformulate the problem:

$$\mathbf{P}(Cavity|toothache \wedge catch) = \alpha\mathbf{P}(toothache \wedge catch|Cavity)\mathbf{P}(Cavity).$$

For this reformulation to work, we need to know the conditional probabilities of the conjunction $toothache \wedge catch$ for each value of $Cavity$. That might be feasible for just two evidence variables, but again it does not scale up. If there are n possible evidence variables (X rays, diet, oral hygiene, etc.), then there are 2^n possible combinations of observed values for which we would need to know conditional probabilities.

If we apply the conditional independence of $Toothache$ and $Catch$, given $Cavity$, it becomes:

$$= \alpha\mathbf{P}(toothache|Cavity)\mathbf{P}(catch|Cavity)\mathbf{P}(Cavity)$$

As we said before, using this simplification, the size of the representation grows as $O(n)$.

In general, if we want to estimate

$$\mathbf{P}(Cause|Effect_1, ..., Effect_n)$$

We can use the Bayes' rule to reformulate the problem as:

$$= \alpha\mathbf{P}(Effect_1, ..., Effect_n|Cause)\mathbf{P}(Cause)$$

However, it is computational unfeasible to compute $\mathbf{P}(Effect_1, ..., Effect_n|Cause)$, therefore we need to rely on some simplification. For example, if we assume that all the effects are conditionally independent,

given the cause, i.e, $\mathbf{P}(Effect_1, ..., Effect_n|Cause) = \prod_i \mathbf{P}(Effect_i|Cause)$, we can express the problem as:

$$\mathbf{P}(Cause|Effect_1, ..., Effect_n) = \alpha\mathbf{P}(Cause)\prod_i \mathbf{P}(Effect_i|Cause)$$

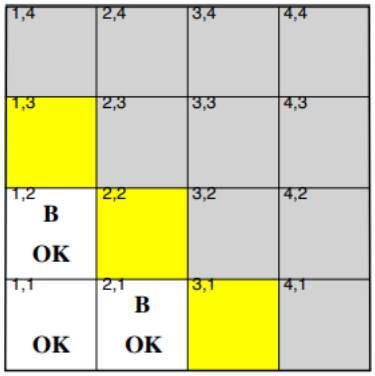
Formally, the full joint distribution can be written as:

$$\mathbf{P}(Cause, Effect_1, ..., Effect_n) = \mathbf{P}(Cause)\prod_i \mathbf{P}(Effect_i|Cause)$$

Such a probability distribution is called a **naive Bayes model**, "naive" because it is often used (as a simplifying assumption) in cases where the "effect" variables are not actually conditionally independent given the cause variable. In practice, naive Bayes systems can work surprisingly well, even when the conditional independence assumption is not true.

15.3 The Wumpus World Revisited

We can combine of the ideas in this chapter to solve probabilistic reasoning problems in the wumpus world. Uncertainty arises in the wumpus world because the agent's sensors give only partial information about the world.



For example, the figure above shows a situation in which each of the three reachable squares, $[1, 3], [2, 2]$, and $[3, 1]$, might contain a pit. Pure logical inference can conclude nothing about which square is most likely to be safe, so a logical agent might have to choose randomly. We will see that a probabilistic agent can do much better than the logical agent.

Our aim is to calculate the probability that each of the three squares contains a pit. step is to identify the set of random variables we need:

- We want one Boolean variable P_{ij} for each square, which is true iff square $[i, j]$ actually contains a pit.
- We also have Boolean variables B_{ij} that are true iff square $[i, j]$ is breezy. We include these variables only for the observed squares, in this case, $[1, 1], [1, 2]$, and $[2, 1]$.

The next step is to specify the full joint distribution:

$$\mathbf{P}(P_{1,1}, ..., P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1})$$

Applying the product rule, we have

$$= \mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1}|P_{1,1}, ..., P_{4,4})\mathbf{P}(P_{1,1}, ..., P_{4,4})$$

The first term is 1 if the breezes are adjacent to the pits and 0 otherwise. The second term is the prior probability of a pit configuration. Each square contains a pit with probability 0.2, independently of the other squares; hence,

$$\mathbf{P}(P_{1,1},...,P_{4,4}) = \prod_{i,j=1,1}^{4,4} \mathbf{P}(P_{i,j}) \tag{15.1}$$

For a particular configuration with exactly n pits, $P(P_{1,1},...,P_{4,4}) = 0.2^n \times 0.8^{16-n}$

We know the following facts:

- $b = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1}$
- $known = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$

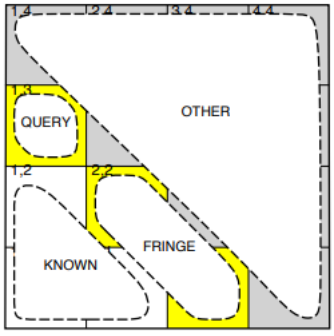
We are interested in answering queries such as $\mathbf{P}(P_{1,3}|known, b)$: how likely is it that $[1,3]$ contains a pit, given the observations so far?

To answer this query, we can follow the standard approach summing over entries from the full joint distribution. Let $Unknown$ be the set of $P_{i,j}$ variables for squares other than the $Known$ squares and the query square $[1,3]$. Then we have:

$$\mathbf{P}(P_{1,3}|known, b) = \sum_{unknown} \mathbf{P}(P_{1,3}, unknown, known, b)$$

Basically, it sums over all the possible combinations of the values of the unknown squares. There are 12 unknown squares; hence the summation contains $2^{12} = 4096$ terms. In general, the summation grows exponentially with the number of squares.

Surely, one might ask, aren't the other squares irrelevant? How could $[4,4]$ affect whether $[1,3]$ has a pit? Indeed, this intuition is correct. Let $Frontier$ be the pit variables (other than the query variable) that are adjacent to visited squares, in this case just $[2,2]$ and $[3,1]$. Also, let $Other$ be the pit variables for the other unknown squares; in this case, there are 10 other squares.



The key insight is that the observed breezes are conditionally independent of the other variables, given the known, frontier, and query variables. To use the insight, we manipulate the query formula into a form in which the breezes are conditioned on all the other variables, and then we apply conditional independence:

$$
\begin{aligned}
&\mathbf{P}(P_{1,3} \mid known, b) \\
&= \alpha \sum_{unknown} \mathbf{P}(P_{1,3}, known, b, unknown) \qquad \text{(by Equation (13.9))} \\
&= \alpha \sum_{unknown} \mathbf{P}(b \mid P_{1,3}, known, unknown)\mathbf{P}(P_{1,3}, known, unknown) \\
&\qquad\qquad \text{(by the product rule)} \\
&= \alpha \sum_{frontier} \sum_{other} \mathbf{P}(b \mid known, P_{1,3}, frontier, other)\mathbf{P}(P_{1,3}, known, frontier, other) \\
&= \alpha \sum_{frontier} \sum_{other} \mathbf{P}(b \mid known, P_{1,3}, frontier)\mathbf{P}(P_{1,3}, known, frontier, other) \ ,
\end{aligned}
$$

where the final step uses conditional independence: b is independent of *other* given *known*, $P_{1,3}$, and $frontier$. Now, the first term in this expression does not depend on the *Other* variables, so we can move the summation inward:

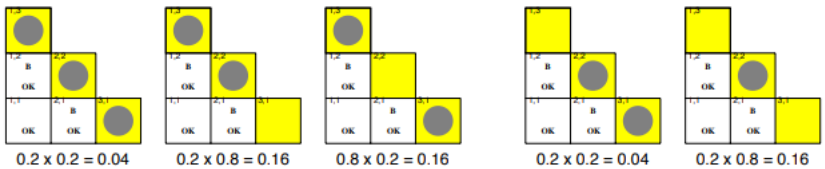
$$
\begin{aligned}
&\mathbf{P}(P_{1,3} \mid known, b) \\
&= \alpha \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier) \sum_{other} \mathbf{P}(P_{1,3}, known, frontier, other) \ .
\end{aligned}
$$

By independence, as in Equation (1), the prior term can be factored, and then the terms can be reordered:

$$
\begin{aligned}
&\mathbf{P}(P_{1,3} \mid known, b) \\
&= \alpha \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier) \sum_{other} \mathbf{P}(P_{1,3})P(known)P(frontier)P(other) \\
&= \alpha\, P(known)\mathbf{P}(P_{1,3}) \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier)P(frontier) \sum_{other} P(other) \\
&= \alpha' \mathbf{P}(P_{1,3}) \sum_{frontier} \mathbf{P}(b \mid known, P_{1,3}, frontier)P(frontier) \ ,
\end{aligned}
$$

where the last step folds $P(known)$ into the normalizing constant and uses the fact that $\sum_{other} P(other) = 1$. The use of independence and conditional independence has completely eliminated the other squares from consideration.

Notice that the expression $\mathbf{P}(b|known, P_{1,3}, frontier)$ is 1 when the frontier is consistent with the breeze observations, and 0 otherwise. Thus, for each value of $P_{1,3}$, we sum over the logical models for the frontier variables that are consistent with the known facts.



Basically, we have that, when there is a pit in $[1, 3]$, $frontier$ is either $p_{2,2} \wedge p_{3,1}$, or $p_{2,2} \wedge \neg p_{3,1}$ or $\neg p_{2,2} \wedge p_{3,1}$. $P(p_{2,2} \wedge p_{3,1}) = 0.2 \times 0.2$ because, by assumption, each square contains a pit with probability 0.2. For the

same reason, $P(p_{2,2} \wedge \neg p_{3,1}) = 0.2 \times 0.8$ and $P(\neg p_{2,2} \wedge p_{3,1}) = 0.8 \times 0.2$ respectively. The same reasoning applies also for $\neg p_{1,3}$

Then, we have:

$$\mathbf{P}(P_{1,3}|known, b) = \alpha' < 0.2(0.04 + 0.16 + 0.16), 0.8(0.04 + 0.16) > \approx < 0.31, 0.69 >$$

That is, $[1, 3]$ (and $[3, 1]$ by symmetry) contains a pit with roughly 31% probability. A similar calculation shows that $[2, 2]$ contains a pit with roughly 86% probability. The wumpus agent should definitely avoid $[2, 2]$!

Chapter 16

Lec 16 - Bayesian Networks

16.1 Bayesian Networks

This section introduces a data structure called a **Bayesian network** to represent the dependencies among variables. Bayesian networks can represent essentially any full joint probability distribution and in many cases can do so very concisely.

A Bayesian network is a directed graph in which each node is annotated with quantitative probability information. The full specification is as follows:

- Each node corresponds to a random variable, which may be discrete or continuous.
- A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y , X is said to be a parent of Y. The graph has no directed cycles (and hence is a directed acyclic graph, or DAG).
- Each node X_i has a conditional probability distribution $\mathbf{P}(X_i|Parents(X_i))$that quantifies the effect of the parents on the node.

The topology of the network, the set of nodes and links, specifies the conditional independence relationships that hold in the domain. In the simplest case, each conditional distribution is represented as a **conditional probability table** (**CPT**) giving the distribution over X_i for each combination of parent values.

Now consider the following example. You have a new burglar alarm installed at home. It is fairly reliable at detecting a burglary, but also responds on occasion to minor earthquakes. You also have two neighbors, John and Mary, who have promised to call you at work when they hear the alarm. John nearly always calls when he hears the alarm, but sometimes confuses the telephone ringing with the alarm and calls then, too. Mary, on the other hand, likes rather loud music and often misses the alarm altogether. Given the evidence of who has or has not called, we would like to estimate the probability of a burglary.

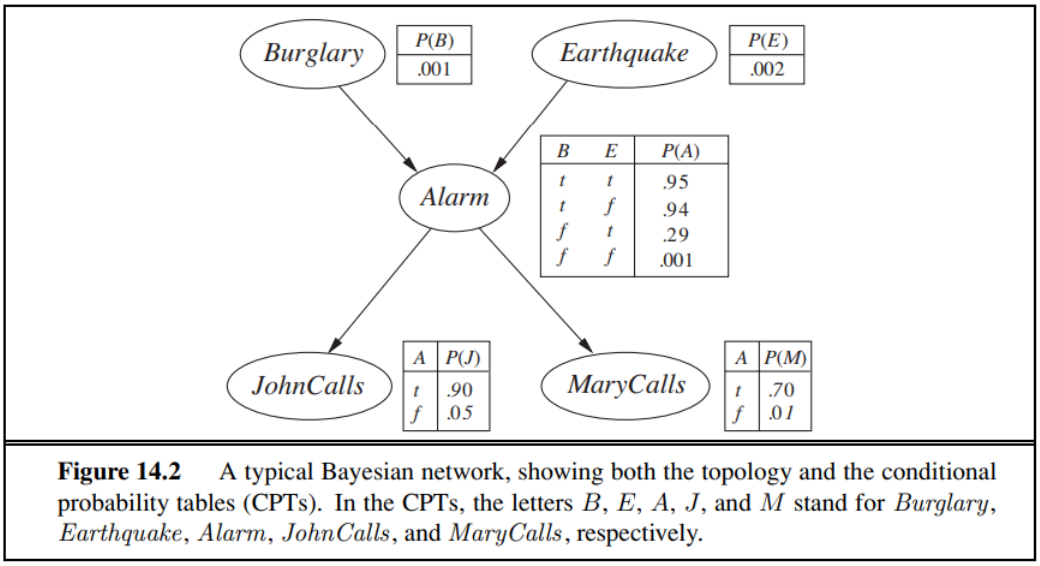


Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

The figure above shows a Bayesian network for this domain. Each row of a CPT must sum to 1, because the entries represent an exhaustive set of cases for the variable. For Boolean variables, once you know that the probability of a true value is p, the probability of false must be $1 - p$, so we often omit the second number, as we do in the figure above. In general, a table for a Boolean variable with k Boolean parents contains 2^k independently specifiable probabilities. A node with no parents has only one row, representing the prior probabilities of each possible value of the variable. If a variable X can take m different values, the number of parameters required to represent a distribution over X is given by:

$$(m-1) \prod_{xp \in Parent(X)} m_{xp}$$

where m_{xp} is the number of values that the parent xp of X can take.

To store the values of a complete network over n Boolean variables we require $O(n \cdot 2^k)$ numbers. If we are able to find a good factorization of the full joint probability distribution exploiting conditional independence (i.e. keeping k small), the number of required parameters grows linearly with n.

16.2 Global semantics

The previous section described what a network is, but not what it means. There are two ways in which one can understand the semantics of Bayesian networks. The first is to see the network as a representation of the joint probability distribution. The second is to view it as an encoding of a collection of conditional independence statements.

One way to define what the network means is to define the way in which it represents a specific joint distribution over all the variables.

A generic entry in the joint distribution is the probability of a conjunction of particular assignments to each variable, such as $P(X_1 = x_1 \wedge ... \wedge X_n = x_n)$. We use the notation $P(x_1, ..., x_n)$ as an abbreviation for this. The value of this entry is given by the formula:

$$P(x_1, ..., x_n) = \prod_{i=1}^{n} P(x_i | parents(X_i))$$

where $parents(X_i)$ denotes the values of $Parents(X_i)$ that appear in $x_1, ..., x_n$. Basically, it defines the full joint distribution as the product of the local conditional distributions.

To illustrate this, we can calculate the probability that the alarm has sounded, but neither a burglary nor an earthquake has occurred, and both John and Mary call. We multiply entries from the joint distribution (using single-letter names for the variables):

$$\begin{aligned} P(j, m, a, \neg b, \neg e) &= P(j|a)P(m|a)P(a|\neg b \wedge \neg e)P(\neg b)P(\neg e) \\ &= 0.90 \times 0.70 \times 0.001 \times 0.999 \times 0.998 = 0.000628. \end{aligned}$$

If a Bayesian network is a representation of the joint distribution, then it too can be used to answer any query, by summing all the relevant joint entries as we saw previously.

16.3 Complexity of exact inference

In the previous chapters we saw that any conditional probability can be computed by summing terms from the full joint distribution. More specifically, a query $\mathbf{P}(X|\mathbf{e})$ can be answered using

$$\mathbf{P}(X|\mathbf{e}) = \alpha\mathbf{P}(X, \mathbf{e}) = \alpha \sum_{\mathbf{y}} \mathbf{P}(X, \mathbf{e}, \mathbf{y})$$

Now, a Bayesian network gives a complete representation of the full joint distribution. More specifically, we have seen that the joint distribution can be written as products of conditional probabilities from the network. Therefore, a query can be answered using a Bayesian network by computing sums of products of conditional probabilities from the network.

Consider the query $\mathbf{P}(Burglary|JohnCalls = true, MaryCalls = true)$. The hidden variables for this query are $Earthquake$ and $Alarm$. Then, we have:

$$\mathbf{P}(B|j, m) = \alpha\mathbf{P}(B, j, m) = \alpha \sum_{e} \sum_{a} \mathbf{P}(B, j, m, e, a)$$

The semantics of Bayesian networks then gives us an expression in terms of CPT entries. For simplicity, we do this just for $Burglary = true$:

$$P(b|j, m) = \alpha \sum_{e} \sum_{a} P(b)P(e)P(a|b, e)P(j|a)P(m|a) \tag{16.1}$$

The complexity of exact inference in Bayesian networks depends strongly on the structure of the network. The burglary network we presented previously belongs to the family of networks in which there is at most one undirected path between any two nodes in the network. These are called **singly connected** networks or **polytrees**, and they have a particularly nice property: The time and space complexity of exact inference in polytrees is linear in the size of the network. Here, the size is defined as the number of CPT entries; if the number of parents of each node is bounded by a constant k, then the complexity will also be linear in the number of nodes, i.e., $O(n \cdot 2^k)$ for Boolean variables. Note that the complexity of an algorithm for computing Equation (1) is $O(n \cdot 2^n)$, but there are techniques that can speed up the computation which we won't see.

For **multiply connected** networks exact inference can have exponential time and space complexity in the worst case, even when the number of parents per node is bounded (NP-Hard problem).

16.4 Inference by stochastic simulation

Given the intractability of exact inference in large, multiply connected networks, it is essential to consider approximate inference methods. The basic idea is to draw N samples from a sampling distribution S and

compute an approximate posterior probability $\hat{\mathbf{P}}$. Then, we expect $\hat{\mathbf{P}}$ and the true probability $\mathbf{P}$ to converge as N increases. There exist severs approaches:

- **Direct sampling:** The idea is to sample each variable in turn, in topological order.
- **Rejection sampling**
- **Likelihood weighting**
- **Markov chain Monte Carlo (MCMC)**

Chapter 17

Lec 17 - Machine Learning I

17.1 What is Machine Learning

A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. Basically, Machine Learning is the field of study that gives computers the ability to learn without being explicitly programmed. In fact, we use Machine Learning when its impossible to **exactly formalise** the problem (and so to give an algorithmic solution) or when formulating a solution it's very complex and cannot be done manually.

There are three ingredients for a learning algorithm:

- The Task
- The Performance Measure
- The Experience

A **task** is usually described in terms of how the machine learning algorithm should process an example. There are several different tasks:

- Classification
- Regression
- Machine translation
- ...

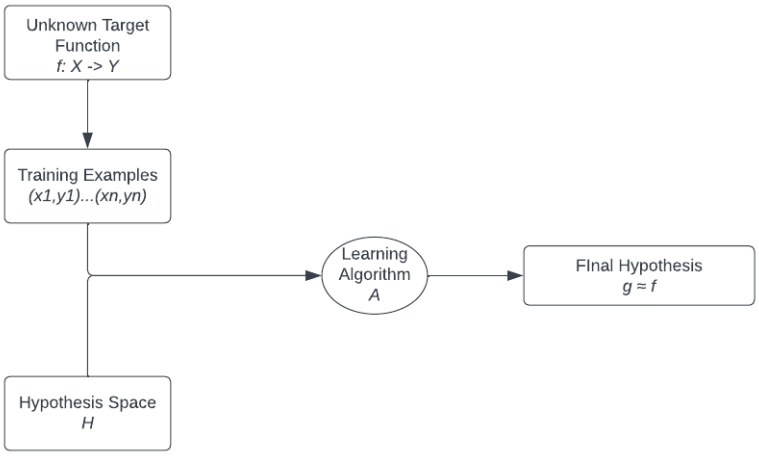
The **performance measure** depends on the task and measures how good is our model trained using the learning algorithm. For example: for a classification task, a good performance measure can be the **accuracy**, i.e., the proportion of examples for which the model produces the correct output. For a regression task, the performance measure can be the **mean squared error** (MSE), i.e., the average of the squares of the errors.

The **experience** is the set of data on which we train our model. These data can be real-valued, discrete or mixed and they can be obtained once for all (batch learning), or acquired incrementally by interacting with the environment (on-line learning).

There are several ways in which data can be used to learn.

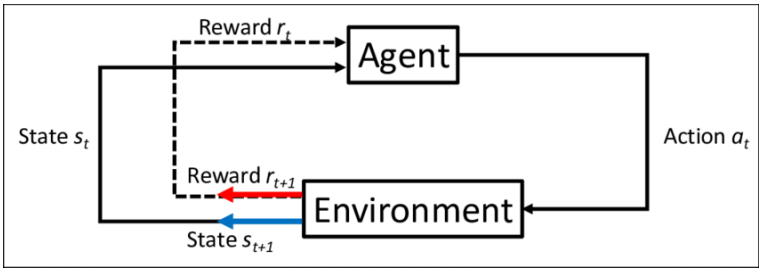
17.1.1 Main Learning Paradigms

- **Supervised Learning:**
  - **Goal:** give the *right answer* for each example in the data.
  - Given a training set $\{(x^{(i)}, y^{(i)})\}$ we look for a function $h(\cdot)$ which is able to map in a predictive way $x^{(i)}$'s to $y^{(i)}$'s. It's called supervised learning because there is an expert that provides a *supervision* assigning a label $y^{(i)}$ to each input $x^{(i)}$
  - **Output:** Classification, regression.
  - Use cases: Object recognition, Predicting pandemic, ...



The *Training examples* are generated according to the *Target function* f (unknown). Once we have this set of pairs, we choose the *hypothesis space*. The *learning algorithm* (e.g a Neural Network) searches in the hypothesis space for a function g that approximates the target function f.

- **Unsupervised Learning:**
  - **Goal:** Find regularities / patterns on the data
  - Given examples $\{x^{(i)}\}$, discover regularities on the whole input domain.
  - There is no supervision.
  - Use cases: Community detection in social media, user profiling, market analysis ...
- **Reinforcement Learning:** An **Agent** operates in an environment e, which in response to action a (given by the agent) in the state s returns the next state and a reward r (which can be positive, negative or neutral). The goal of the Agent is to maximize a reward function.
  - Use cases: Robotics, Games, ...



- **Other Learning Strategies:**
  - Active Learning
  - Online Learning, Incremental & Continual Learning
  - Weak Supervised Learning
  - Self-supervised Learning
  - Deep Learning and Representation Learning
  - Federated Learning

17.2 Fundamental Ingredients

- **Input/Instance space $\mathbf{x} \in X$:** Representation of model's input (e.g. you can choose a Vector as a representation for your input). It contains all the possible inputs for a model. Suppose the model takes in a vector, $input = [x1, x2], x1, x2 \in [1, 10]$, then we have 10^2 possible inputs.
- **Output space $y \in Y$:** In supervised learning we want to perform a prediction based on the input. This prediction can be in the form of:
  - Binary Classification $y \equiv \{-1, +1\}$
  - Multi-Class Classification $y \equiv \{1, ..., m\}$
  - Regression $y \equiv \mathbb{R}$
- **Oracle/Nature:** It determines how examples are generated. We can have two cases of Oracle
  - Target function $f : X \to Y$ It's deterministic and given an object of the input space returns an object of the output space. This function is ideal and **unknown**.
  - Probability distribution $P(\mathbf{x}), P(y \mid \mathbf{x})$ The *selection* of y occurs from a probability distribution. This distribution is still unknown
- **Training set:** Set of pairs $\{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$ where each pair is composed by an instance of the input space and it's corresponding label.

x	y
000	0
001	1
010	1
.	.
.	.
.	.

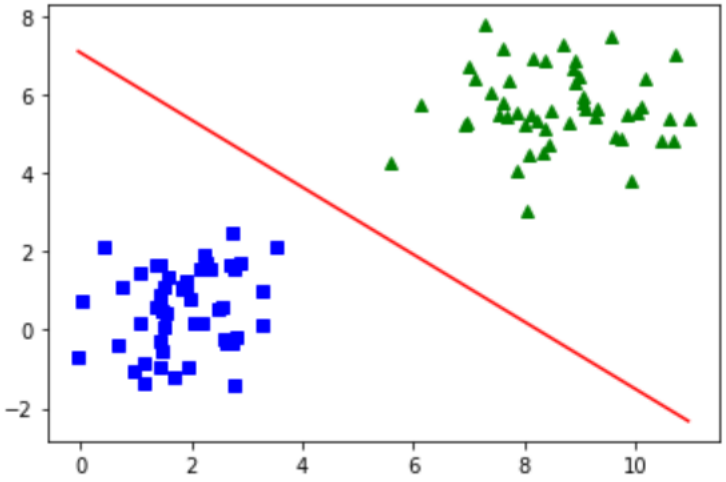
- **Hypothesis space:** A **predefined** set of hypothesis/functions $H \equiv \{h \mid h : X \to Y\}$. It constitutes the set of functions which can be implemented by the machine learning system. It is assumed that the function to be learned f may be approximated by a hypothesis $h \in H$. Note that H cannot coincide with the set of all possible functions and the search (into H) to be exhaustive, otherwise there would be infinitely many functions that fit the training data and h would just memorize the training data.

  For this reason we must make assumptions about the type of the unknown target function. The **inductive bias** consists of:
  - The hypothesis space: how H is defined

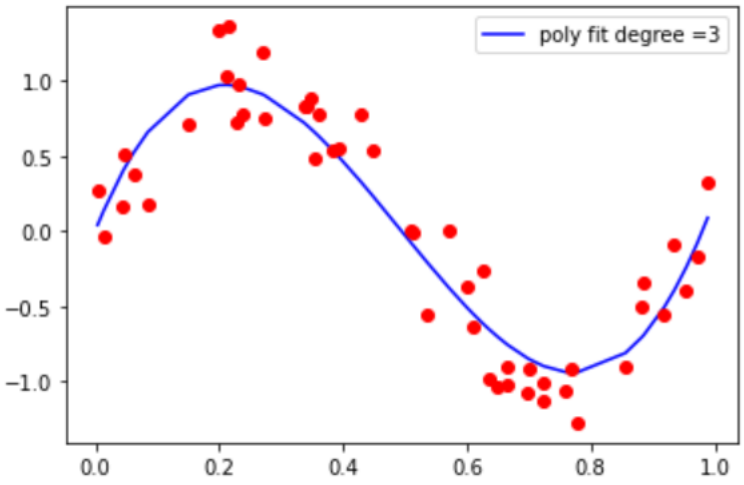
- The learning algorithm: how H is explored

Let's see some examples of hypothesis space.

- **Hyperplanes in $\mathbb{R}^2$**: We chose as input space points in the plane $X = \{y \mid y \in \mathbb{R}^2\}$, and as hypothesis space the dichotomies induced by hyperplanes in $\mathbb{R}^2$, that is, $H = \{f_{w,b}(y) = sign(\mathbf{w} \cdot y + b), \mathbf{w} \in \mathbb{R}^2, b \in \mathbb{R}\}$.



- **Polynomial functions**: Given a training set $S = \{(x_1, y_1), ..., (x_n, y_n)\}, x \in \mathbb{R}, y \in \mathbb{R}$, the hypothesis space is the one containing functions of type: $h_w(x) = w_0 + w_1 x + w_2 x^2 + ... + w_p x^p, p \in \mathbb{N}$.



17.3 Example of Learning Algorithm: Perceptron

Consider the space of hyperplanes in $\mathbb{R}^n$, where n is the dimension of the input.

$$H = \{f_{(\mathbf{w},b)}(\mathbf{x}) = sign(\mathbf{w} \cdot \mathbf{x} + b) : \mathbf{w},\mathbf{x} \in \mathbb{R}^n, b \in \mathbb{R}\}$$

where $\mathbf{w}$ is a vector of weights and b is the **bias** term.

We can redefine H as:

$$H = \{f_{\mathbf{w}'}(\mathbf{x}') = sign(\mathbf{w}' \cdot \mathbf{x}') : \mathbf{w}', \mathbf{x}' \in \mathbb{R}^{n+1}\}$$

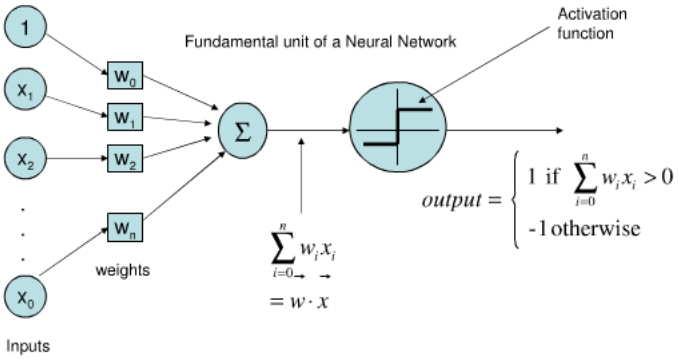
after the following change of variables:

$$\mathbf{w}' = [b, \mathbf{w}], \quad \mathbf{x}' = [1, \mathbf{x}]$$

it follows that:

$$\mathbf{w}' \cdot \mathbf{x}' = b + \sum_{i=1}^{n} \mathbf{w}_i \mathbf{x}_i = \mathbf{w} \cdot \mathbf{x} + b$$

Basically, we add a dimension to $\mathbf{w}$ and $\mathbf{x}$ just to simplify the notation of H.



The model described in the image above is called **Perceptron**. It first computes the dot-product between the weights $\mathbf{w}$ and the input $\mathbf{x}$. The result of this computation is usually called *net*

$$net = \sum_{i=0}^{n} w_i x_i$$

The final output is obtained by applying the **step function** to the *net*.

$$o = \sigma(net) = sign(net)$$

We will refer to this neuron (and associated learning algorithm) as Perceptron.

Since the hypothesis space of the Perceptron is defined as the hyperplanes in $\mathbb{R}^n$, it converges only if the examples in $\mathbb{R}^n$ are **linearly separable**. Otherwise, it will never *find* a hyperplane that separates them.

17.3.1 Perceptron: learning algorithm

Assume to have training examples in $\mathbb{R}^n$ that are linearly separable:

Input: Training set $S = \{(\mathbf{x}, t), \mathbf{x} \in \mathbb{R}^{n+1}, t \in \{-1, +1\}, \eta \geq 0\}$

1. Initialize the value of the weights $\mathbf{w}$ randomly;
2. Repeat (N epochs)
   (a) Select (randomly) one of the training examples $(\mathbf{x}, t)$
   (b) if $o = sign(\mathbf{w} \cdot \mathbf{x}) \neq t$ then

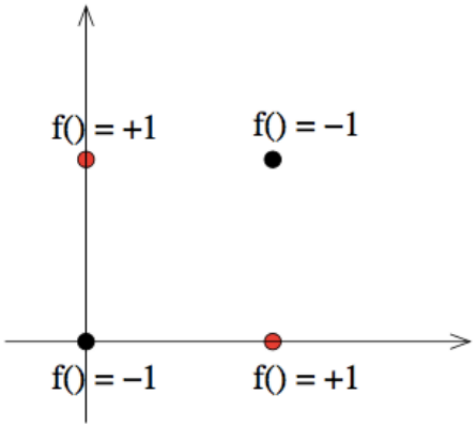
$$\mathbf{w} \leftarrow \mathbf{w} + \eta(t - o)\mathbf{x}$$

A small value of the learning rate η can make the learning process slow but more stable, that is, it prevents sharp changes in the weights vector. If the training set is linearly separable, it can be shown that the Perceptron training algorithm terminates in a finite number of steps.

Let's try to define a linear model that predicts the XOR function:

$$Tr = \{([0,0],0),([0,1],1),([1,0],1),([1,1],0)$$

The XOR function **cannot** be learned by any linear classifier. In order to solve this problem, we would need a more complex hypothesis space (e.g. Neural network).



17.4 Empirical/True Error

The **Empirical Error** $error_{Tr}(h)$ of hypothesis h with respect to Tr is the number of examples that h misclassifies.

The **True Error** $error_D(h)$ of hypothesis h with respect to target concept c and distribution D is the probability that h will misclassify an instance drawn according to D. This can only be estimated. One way to estimate this quantity is testing the model over new examples that are not in the training set (test set).

$$error_D(h) \equiv Pr_{x \in D}(c(x) \neq h(x))$$

$h \in H$ **overfits** Tr if $\exists h' \in H$ such that $error_{Tr}(h) < error_{Tr}(h')$, but $error_D(h) > error_D(h')$.

17.5 Hypothesis Space Complexity

Let's consider a plane with 4 points. If we choose the hyperplanes in $\mathbb{R}^2$ as hypothesis space H we can divide the points with a line and classify them with two colors (red and green). These partitions are called dichotomies. Note that this particular H **can't** implement all possible dichotomies.

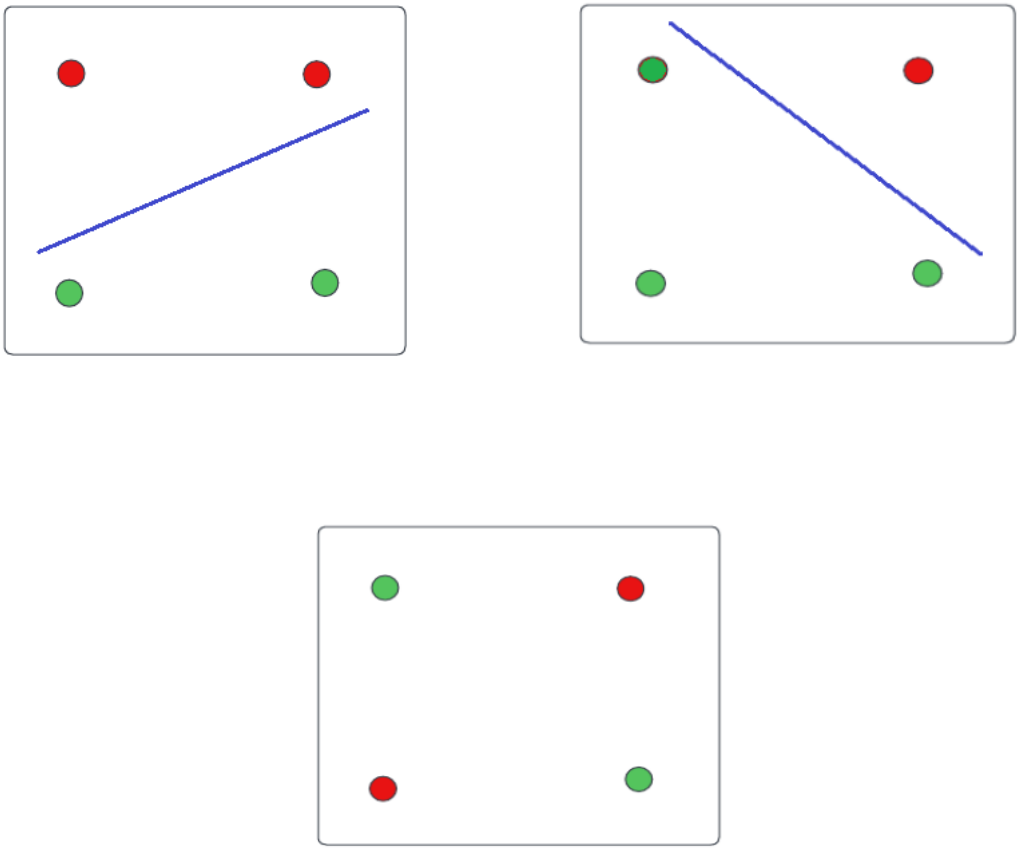


Figure 17.1: This classification can't be implemented by any hyperplane in H

- **Shattering:** Given $S \subset X, S$ is shattered by the hypothesis space H iff

$$\forall S^{'} \subseteq S, \exists h \in H, s.t. \forall x \in S, h(x) = 1 \iff x \in S^{'}$$

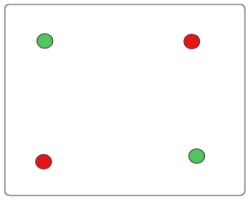
  That means that H is able to implement all possible dichotomies of S.

- **VC-dimension:** The VC-dimension of an hypothesis space H defined over an instance space X is the size of the largest finite subset of X shattered by H:

$$VC(H) = \max_{S \subseteq X} |S|$$

  such that S is shattered by H.

If we consider $H_1 = \{f_{w,b}(y) | f_{w,b}(y) = sign(w \cdot y + b), w \in \mathbb{R}^2, b \in \mathbb{R}\}$ as hypothesis space (hyperplanes in $\mathbb{R}^2$), $VC(H_1) = 3$ because it doesn't exist **any** configuration of 4 points that can be shattered by H_1. If we consider just three points, there are $2^3 = 8$ possible dichotomies, and there exists **at least** one configuration which is shattered by H_1.



In general, VC-dimension of hyperplanes in $\mathbb{R}^n = n + 1$: in 2 dimensions $VC(H) = 3$, in 3 dimensions $VC(H) = 4$ and so on.

Chapter 18

Lec 18 - Machine Learning II

18.1 VC bound and SRM

Consider a binary classification learning problem with:

- Training set $S = \{(\mathbf{x}_1, y_1), ..., (\mathbf{x}_n, y_n)\}$.
- Hypothesis space $H = \{h_\theta(\mathbf{x})\}$.
- Learning algorithm L, returning the hypothesis $g = h^*_\theta$ minimizing the empirical error on S, that is $g = argmin_{h \in H} error_S(h)$.

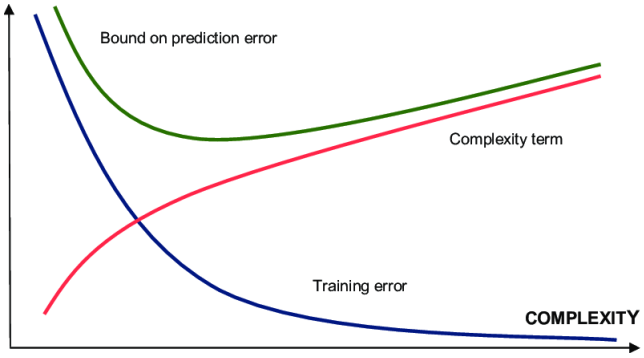
It is possible to derive [1] an upper bound of the ideal error which is valid with probability $(1 - \delta)$, δ being arbitrarily small, of the form:

$$error(g) \leq error_S(g) + F(\frac{VC(H)}{n}, \sigma)$$

where $error(g)$ is the ideal error. The term $F(\frac{VC(H)}{n}, \sigma)$ is called **VC-confidence** and it depends on:

- The training set size n (inversely).
- The VC-dimension of H, $VC(H)$ (proportionally).
- The confidence σ (inversely).

As the VC-dimension grows, you usually observe that the empirical error decreases and that the VC confidence increases. So you can use the inductive principle **Structural Risk Minimization (SRM)** in order to minimize the right side of the confidence bound to get a tradeoff between the empirical error and the VC confidence.



[1] We will not see the derivation here.

18.2 In Practice: how to use data

The machine learning pipeline can be divided in two parts: the training phase and the testing phase. The first part involves:

1. Analysis of the problem (Classification, Regression, ...)
2. Collection, analysis and cleaning data
3. Pre-processing and managing missing values
4. Study of correlation between variables
5. Feature selection/weighting/learning
6. Choice of the predictor and model selection

Once the model is trained, we need to **test it** on new unseen data (not used during training).

18.2.1 Model selection

Model selection is the process used to compare different models and select the optimal one. In particular, model selection can be performed with respect to different values of the hyper-parameters of a fixed model.

There are several methods used to implement it. A first example can be the so called Hold-out procedure. The idea is to obtain a validation set (or hold-out set) Va by splitting the training set Tr. Then, the fixed model is trained using examples in $Tr - Va$, trying different values of the hyper-parameters, and tested against the validation set. This procedure allows you to get an estimate of the error of the model on new unseen data.

Another approach for model selection (and evaluation) is the K-fold cross-validation:

1. The training set is partitioned in k disjointed validation sets $Va_1, ..., Va_k$. For each classifier $h_1, ..., h_k$, we apply the hold-out method on the k-th pair, that is, we train h_i using examples in $Tr - Va_i$ and we test it against V_i.
2. Final error is obtained by individually computing the errors of $h_1, ..., h_k$ on the corresponding validation set and averaging the results.

The above procedure is repeated for different values of the hyper-parameters and the predictor with the smallest final error is selected. The special case where $k = |Tr|$ (the validation sets are made of only one example) is called **leave-one-out** cross-validation.

18.3 Artificial Neural Networks

An Artificial Neural Network (ANN) is a system consisting of interconnected units that compute nonlinear (numerical) functions. The non linearity of the model is given by the activation functions. In fact, without them (linear activation) the result of the model, even if it's very complex, would still be linear. ANNs consist of:

- **input units**, which represent input variables;
- **output units**, which represent output variables;

- **hidden units** (if present), which represent internal variables that codify (after learning) correlations among input and desired output variables. Usually the output of hidden units is passed through a non-linear function called **activation function**.
- adjustable **weights** which are associated to connections among units.

Thanks to the non-linear activation function, ANNs are able to exploit a more complex decision surface to solve tasks that a linear model would not be able to solve (e.g. XOR problem). Note that, in order to exploit gradient-based optimization, the activation function must be derivable.

18.4 Gradient Descent for Feed-forward Networks

The basic idea of the learning algorithm for ANNs consists in two phases:

- **Forward phase:** for each example in the training set, present it to the network and compute the output.
- **Backward phase:** Back-propagate the committed error by computing the gradients of the cost function with respect to the network's weights.

Backpropagation is the algorithm used to **compute the gradient** of the loss function with respect to each parameter. The information from the cost function flows backward to compute these gradients. Then, another algorithm performs learning using the gradients (e.g. stochastic gradient descent).

Let's define some terminology:

- d **input units**: d input data size $\mathbf{x} \equiv (x_1, ..., x_d)$, $d+1$ when including the bias in the weight vector $\mathbf{x}^{'} \equiv (x_0, x_1, ..., x_d)$
- n_H **hidden units** (with output $\mathbf{y} \equiv (y_1, ..., y_{n_H})$)
- c **output units**: $\mathbf{z} \equiv (z_1, ..., z_c)$. The number of desired output is $\mathbf{t} \equiv (t_1, ..., t_c)$
- w_{ji} weight from the i-th input unit to the j-th hidden unit (w_j is the weight vector of the j-th hidden unit)
- w_{kj} weight from the j-th hidden unit to the k-th output unit (w_k is the weight vector of the k-th output unit)

Let's consider, for example, a Neural Network with just one hidden layer and **sigmoid activation functions**. We need to minimize an error function $E[\mathbf{w}]$ that can be defined as follows:

$$E[\mathbf{w}] = \frac{1}{2cN} \sum_{s=1}^{N} \sum_{k=1}^{c} (t_k^{(s)} - z_k^{(s)})^2$$

We need to first compute the gradient for the weights of both output and hidden units.

- **Gradient of the weights of an output unit:**

$$\frac{\partial E}{\partial w_{\hat{k}\hat{j}}} = -\frac{1}{cN} \sum_{s=1}^{N} (t_{\hat{k}}^{(s)} - z_{\hat{k}}^{(s)}) z_{\hat{k}} (1 - z_{\hat{k}}) y_{\hat{j}}^{(s)}$$

  where $y_{\hat{j}}^{(s)}$ is the output of the $\hat{j}$-th hidden unit.

- **Gradient of the weights of a hidden unit:**

$$\frac{\partial E}{\partial w_{\hat{j}\hat{i}}} = -\frac{1}{cN}\sum_{s=1}^{N} y_{\hat{j}}^{(s)}(1-y_{\hat{j}}^{(s)})x_{\hat{i}}^{(s)}\sum_{k=1}^{c}(t_{\hat{k}}^{(s)}-z_{\hat{k}}^{(s)})z_{\hat{k}}(1-z_{\hat{k}})w_{k\hat{j}}$$

Back-propagation (stochastic)

1. Initialize all weights with small random values (e.g. between -0.5 and +0.5)
2. Until the termination condition is satisfied:
   (a) For each $(\mathbf{x}, \mathbf{t}) \in S$:
      i. Present $\mathbf{x}$ to the net and compute the vectors $\mathbf{y}$ and $\mathbf{z}$
      ii. For each output unit k:

$$\delta_k = z_k(1-z_k)(t_k - z_k)$$

$$\Delta w_{kj} = \delta_k y_j$$

      iii. For each hidden unit j:

$$\delta_j = y_j(1-y_j)\sum_{k=1}^{c} w_{kj}\delta_k$$

$$\Delta w_{ji} = \delta_j x_i$$

      iv. Update all weights:

$$w_{sq} \leftarrow w_{sq} + \eta \Delta w_{sq}$$

Note that this algorithm works only for a network with one hidden layer, but can be easily extended. In fact, the algorithm computes the error term δ for each unit of each layer, and then it multiplies this error term for the input of the unit.

18.5 Deep Learning

Deep neural networks are ANNs with several hidden layers between input and output units. It is convenient to use more than one hidden layer in a neural network because a multi-layer neural network can learn high-level non-linear representations of the input data which may improve the effectiveness of the model. Basically, instead of providing to the model hand-designed features, we let the model to automatically learn abstract representations of data and use them to perform a specific task. Furthermore, deep networks seem to generalize better.

Chapter 19

Lec 19 - Reinforcement Learning

19.1 Reinforcement Learning

An **Agent** operates in an environment e, which in response to action a (given by the agent) in the state s returns the next state and a reward r (which can be positive, negative or neutral). The goal of the Agent is to maximize a reward function.

In many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels. For example, in game playing, it is very hard for a human to provide accurate and consistent evaluations of large numbers of positions, which would be needed to train an evaluation function directly from examples. Instead, the program can be told when it has won or lost, and it can use this information to learn an evaluation function that gives reasonably accurate estimates of the probability of winning from any given position.

We will assume that the agent does not know how the environment works or what its actions do, and we will allow for probabilistic action outcomes. Thus, the agent faces an unknown Markov decision process. It consists of the set of all the states S, the action set A, the transition function δ, and the reward function R. A Markov decision process relies on the following assumption: the probability of future state s_{t+1} only depends on the current state and action s_t, a_t, and doesn't depend on any of the previous states and actions.

At each discrete time t:

- the agent observes state $s_t \in S$;
- it chooses action $a_t \in A$ (among the possible actions in state s_t);
- it receives immediate reward r_t, that can be positive, negative or neutral.
- the state changes to s_{t+1}

As we said before, we assume that r_t and s_{t+1} only depend on current state and action. Note that δ and r may be nondeterministic and not necessarily known to agent.

More formally, the agent's goal is to learn an action policy $\pi : S \to A$ that maximizes the expected sum of (discounted) rewards obtained if policy π is followed:

$$E = \sum_{t=0}^{\infty} \gamma^t R(s_t)$$

where γ is called the **discount factor**. Note that with discounted rewards, the utility of an infinite sequence is finite. Furthermore, The closer γ is to 0, the more the agent will try to optimize the current reward r_t. The closer γ is to 1, the more the agent will aim to optimize future rewards.

19.2 What to Learn

To begin, consider **deterministic** environments: for each possible policy π the agent might adopt, we can define an evaluation function over states:

$$V^{\pi}(s) = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

where $r_t, r_{t+1}, ...$ are generated executing policy π starting at state s. Then the choice of the best actions to play becomes an optimization problem. Indeed, it comes down to finding the optimal policy π^* that maximizes the evaluation function:

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

So, how can we find the optimal policy π^*?

We might try to have agent learn the evaluation function V^{π^*} (which we write as V^*).

$$\pi^*(s) = argmax_a[r(s,a) + \gamma V^*(\delta(s,a))] \tag{19.1}$$

Unfortunately, learning V^* is a useful way to learn the optimal policy only when the agent has perfect knowledge of δ and r. This requires that it be able to perfectly predict the immediate result (i.e., the immediate reward and immediate successor) for every possible state-action transition. In many practical problems, it is impossible for the agent or its human programmer to predict in advance the exact outcome of applying an arbitrary action to an arbitrary state.

19.2.1 Q-learning

Let us define the evaluation function $Q(s,a)$ so that its value is the maximum discounted cumulative reward that can be achieved starting from state s and applying action a as the first action. In other words, the value of Q is the reward received immediately upon executing action a from state s, plus the value (discounted by γ) of following the optimal policy thereafter.

$$Q(s,a) = r(s,a) + \gamma V^*(\delta(s,a))$$

Then, we can rewrite (1) as:

$$\pi^*(s) = argmax_a Q(s,a)$$

Why is this rewrite important? Because it shows that if the agent learns the Q function instead of the V^* function, it will be able to select optimal actions even when it has no knowledge of the functions r and δ.

19.2.2 An Algorithm for Learning Q

The key problem is finding a reliable way to estimate training values for Q, given only a sequence of immediate rewards r spread out over time. This can be accomplished through iterative approximation. To see how, notice the close relationship between Q and V^*:

$$V^*(s) = max_{a'} Q(s,a')$$

Which allows us to write Q **recursively** as:

$$\begin{aligned} Q(s_t, a_t) &= r(s_t, a_t) + \gamma\, V^*(\delta(s_t, a_t)) \\ &= r(s_t, a_t) + \gamma\, max_{a'} Q(s_{t+1}, a') \end{aligned}$$

This recursive definition of Q provides the basis for algorithms that iteratively approximate Q.

Let $\hat{Q}$ denote learner's current approximation to Q. The agent repeatedly observes its current state s, chooses some action a, executes this action, then observes the resulting reward $r = r(s, a)$ and the new state $s' = \delta(s, a)$. Then, it updates $\hat{Q}$ as follows:

$$\hat{Q}(s, a) = r + \gamma\, max_{a'} \hat{Q}(s', a')$$

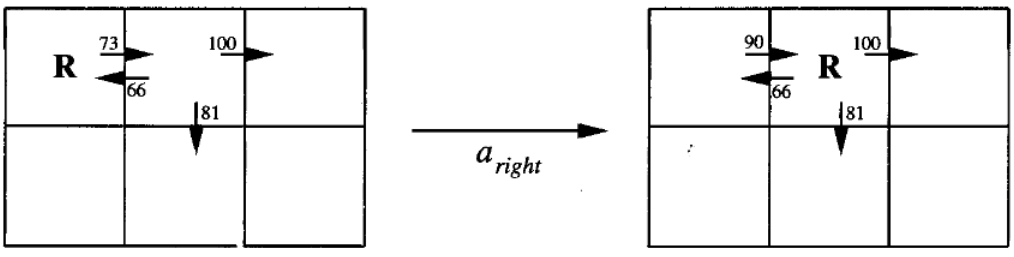
The above Q learning algorithm for **deterministic** Markov decision processes is described more precisely as follows:

1. for each s, a initialize table entry $\hat{Q}(s, a) \leftarrow 0$
2. observe current state s
3. do forever
   1. select an action a and execute it
   2. receive immediate reward r
   3. observe the new state s'
   4. update the table entry for $\hat{Q}(s, a)$ as follows:

$$\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

   5. $s \leftarrow s'$

Notice the algorithm does not specify how actions are chosen by the agent. One obvious strategy would be for the agent in state s to select the action a that maximizes $\hat{Q}(s, a)$, thereby **exploiting** its current approximation $\hat{Q}$. However, with this strategy the agent runs the risk that it will overcommit to actions that are found during early training to have high $\hat{Q}$ values, while failing to explore other actions that have even higher values. For this reason, it is common in Q learning to use a probabilistic approach to selecting actions. Actions with higher $\hat{Q}$ values are assigned higher probabilities, but every action is assigned a nonzero probability. Usually the agent favors **exploration** during early stages of learning, then gradually shifts toward a strategy of exploitation.



$$\begin{aligned} \hat{Q}(s_1, a_{right}) &\leftarrow r + \gamma \max_{a'} \hat{Q}(s_2, a') \\ &\leftarrow 0 + 0.9 \ \max\{66, 81, 100\} \\ &\leftarrow 90 \end{aligned}$$

Notice if rewards are non-negative, then:

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s, a) \geq \hat{Q}_n(s, a)$$

where n denotes the n-th iteration. A second general property that holds is that through-out the training process every $\hat{Q}$ value will remain in the interval between zero and its true Q value:

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s, a) \leq Q(s, a)$$

19.2.3 Convergence

Will the algorithm above converge toward a $\hat{Q}$ equal to the true Q function? The answer is yes, under certain conditions. First, we must assume the system is a deterministic MDP. Second, we must assume the immediate reward values are bounded. Third, we assume the agent selects actions in such a fashion that it visits every possible state-action pair infinitely often.

The key idea underlying the proof of convergence is that the table entry $\hat{Q}(s, a)$ with the largest error must have its error reduced by a factor of γ whenever it is updated.

Let $\hat{Q}_n$ be the table storing the values for each (s, a) after n updates, and Δ_n be the maximum error in $\hat{Q}_n$, i.e.

$$\Delta_n = max_{s,a} | \hat{Q}_n(s, a) - Q(s, a) |$$

For any table entry $\hat{Q}_n(s, a)$ updated on iteration $n + 1$, the error in the revised estimate $\hat{Q}_{n+1}(s, a)$ is:

$$\begin{aligned}
|\hat{Q}_{n+1}(s, a) - Q(s, a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s', a')) \\
&\quad -(r + \gamma \max_{a'} Q(s', a'))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s', a') - \max_{a'} Q(s', a')| \\
&\leq \gamma \max_{a'} |\hat{Q}_n(s', a') - Q(s', a')| \\
&\leq \gamma \max_{s'', a'} |\hat{Q}_n(s'', a') - Q(s'', a')| \\
|\hat{Q}_{n+1}(s, a) - Q(s, a)| &\leq \gamma \Delta_n
\end{aligned}$$

Note we used general fact that:

$$| max_a f_1(a) - max_a f_2(a) | \leq max_a | f_1(a) - f_2(a) |$$

19.2.4 Nondeterministic Case

Above we considered Q learning in deterministic environments. Here we consider the nondeterministic case, in which the reward function $r(s, a)$ and action transition function $\delta(s, a)$ may have probabilistic outcomes.

In this section we extend the Q learning algorithm for the deterministic case to handle nondeterministic MDPs. In the nondeterministic case we must first restate the objective of the learner to take into account the fact that outcomes of actions are no longer deterministic. The obvious generalization is to redefine the value V^π of a policy π to be the expected value (over these nondeterministic outcomes) of the discounted cumulative reward received by applying this policy.

$$V^\pi(s_t) = E \left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \right]$$

As before, we define the optimal policy π^* to be the policy π that maximizes $V^\pi(s)$ for all states s. Next we generalize our earlier definition of Q by taking its expected value.

$$Q(s,a) = E[r(s,a) + \gamma V^*(\delta(s,a))]$$

To learn, alter training rule to:

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \gamma\, max_{a'}\hat{Q}_{n-1}(s',a')] \tag{19.2}$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

where $visits_n(s,a)$ is the total number of times this state-action pair has been visited up to and including the n-th iteration. The key idea in this revised rule is that revisions to $\hat{Q}$ are made more gradually than in the deterministic case. Notice if we were to set α_n to 1 we would have exactly the training rule for the deterministic case. Under specific conditions, can still prove convergence of $\hat{Q}$ to Q.

19.2.5 Temporal Difference Learning (TD-lambda)

The Q learning algorithm learns by iteratively **reducing the discrepancy** between Q value estimates for adjacent state. In this sense, Q learning is a special case of a general class of temporal difference algorithms that learn by reducing discrepancies between estimates made by the agent at different times.

Whereas the training rule of Equation (2) reduces the difference between the estimated $\hat{Q}$ values of a state and its immediate successor, we could just as well design an algorithm that reduces discrepancies between this state and more distant descendants or ancestors.

$$Q^{(2)}(s_t,a_t) = r_t + \gamma r_{t+1} + \gamma^2 max_a \hat{Q}(s_{t+2},a)$$

or, in general, for n steps

$$Q^{(n)}(s_t,a_t) = r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n-1} + \gamma^n max_a \hat{Q}(s_{t+n},a)$$

Sutton (1988) introduces a general method for blending these alternative training estimates, called $TD(\lambda)$. The idea is to use a constant $0 \leq \lambda \leq 1$ to combine the estimates obtained from various lookahead distances in the following fashion:

$$Q^\lambda(s_t,a_t) = (1-\lambda)\left[Q^{(1)}(s_t,a_t) + \lambda Q^{(2)}(s_t,a_t) + \lambda^2 Q^{(3)}(s_t,a_t) + ...\right]$$

Note if we choose $\lambda = 0$ we have our original training estimate $Q^{(1)}$, which considers only one-step discrepancies in the $\hat{Q}$ estimates. As λ is increased, the algorithm places increasing emphasis on discrepancies based on more distant lookaheads.

Many improvements/extensions are possible, for example, we can replace $\hat{Q}$ table with a (deep) neural net!

Chapter 20

Lec 20 - Constraint Satisfaction Problems

20.1 Introduction

When we talked about problem solving agents we explored the idea that problems can be solved by searching in a space of **states**. These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states. From the point of view of the search algorithm, however, each state is atomic, or indivisible, a black box with no internal structure.

This chapter describes a way to solve a wide variety of problems more efficiently. We use 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-purpose rather than problem-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.

20.2 Constraint satisfaction problems

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

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

To solve a CSP, we need to define a state space and the notion of a solution. Each state in a CSP is defined by an **assignment** of values to some or all of the variables. 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, and a **solution** to a CSP is a consistent, complete assignment. A **partial assignment** is one that assigns values to only some of the variables.

20.2.1 Example problem: Map coloring

We are given the task of coloring each region of Australia either red, green, or blue in such a way that no neighboring regions have the same color. To formulate this as a CSP, we define the variables to be the regions:

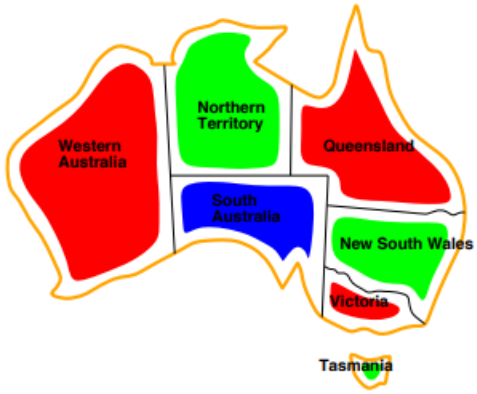
$$X = \{WA, NT, Q, NSW, V, SA, T\}$$

The domain of each variable is the set $D_i = \{red, green, blue\}$. The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints:

$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

There are many possible solutions to this problem, such as:

$$\{WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green\}.$$



It can be helpful to visualize a CSP as a **constraint graph**, as shown in the figure below.

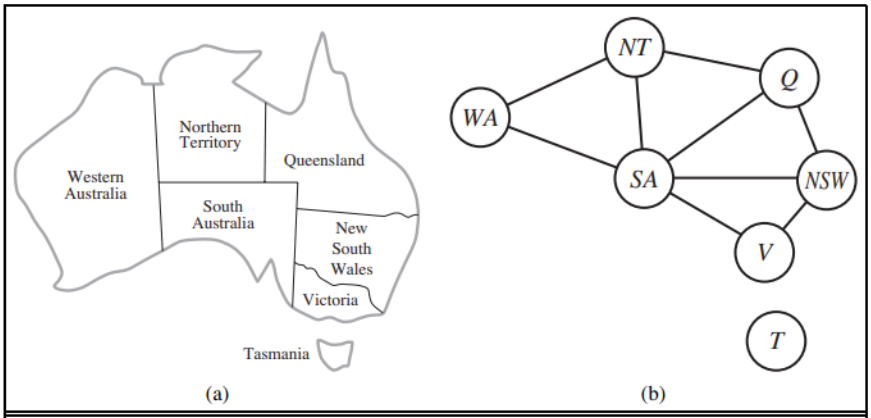


Figure 6.1 (a) The principal states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem (CSP). The goal is to assign colors to each region so that no neighboring regions have the same color. (b) The map-coloring problem represented as a constraint graph.

The nodes of the graph correspond to variables of the problem, and a link connects any two variables that participate in a constraint. Note that this problem is a Binary CSP, that is, each constraint relates at most two variables.

CSP solvers can be faster than state-space searchers because the CSP solver can quickly eliminate large swatches of the search space. For example, once we have chosen $\{SA = blue\}$ in the Australia problem, we can conclude that none of the five neighboring variables can take on the value blue.

20.3 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. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values. Instead, a **constraint language** must be used.

Special solution algorithms (which we do not discuss here) exist for **linear constraints** on integer variables, that is, constraints in which each variable appears only in linear form. It can be shown that no algorithm exists for solving general **nonlinear constraints** on integer variables.

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. Linear programming problems can be solved in time polynomial in the number of variables.

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. For example, in the map-coloring problem it could be $SA \neq green$.
- A **binary constraint** relates two variables. For example, $SA \neq NSW$ is a binary constraint.
- We can also describe **higher-order constraints**, which involve 3 or more variables, such as asserting that the value of Y is between X and Z. An example of higher-order constraints problem is provided by **cryptarithmetic** puzzles.

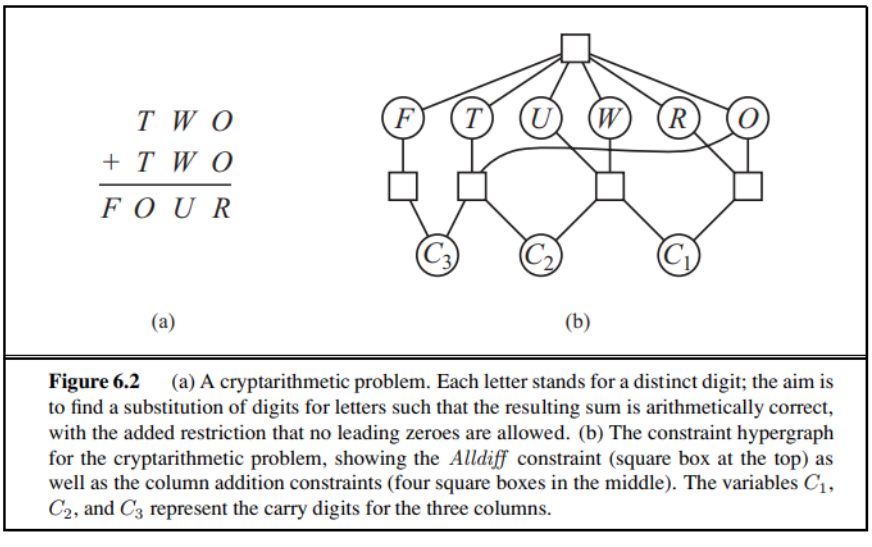


Figure 6.2 (a) A cryptarithmetic problem. Each letter stands for a distinct digit; the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct, with the added restriction that no leading zeroes are allowed. (b) The constraint hypergraph for the cryptarithmetic problem, showing the *Alldiff* constraint (square box at the top) as well as the column addition constraints (four square boxes in the middle). The variables C_1, C_2, and C_3 represent the carry digits for the three columns.

- Many real-world CSPs include **preference constraints** indicating which solutions are preferred, e.g., red is better than green, often representable by a cost for each variable assignment. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a **constraint optimization problem**, or COP. Linear programming problems do this kind of optimization.

Examples of Real-world CSPs:

- Assignment problems e.g., who teaches what class
- Timetabling problems
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling
- Floorplanning

Notice that many real-world problems involve real-valued variables.

20.4 Backtracking search

In this section we look at **backtracking search** algorithms that work on partial assignments. Let's start with the straightforward, dumb approach, then fix it. We could apply a standard depth-limited search. A state would be a partial assignment, and an action would be assign a value to an unassigned variable that does not conflict with current assignment. The goal test would be checking if the current assignment is complete. But for a CSP with n variables of domain size d, we quickly notice something terrible: the branching factor at the top level is nd because any of d values can be assigned to any of n variables. At the next level, the branching factor is $(n-1)d$, and so on (branching factor at depth l is $(n-l)d$). Therefore, we would generate a tree with $n!{\cdot}d^n$ leaves, even though there are only d^n possible complete assignments!

Our naive formulation ignores crucial property common to all CSPs: **commutativity**. CSPs are commutative because when assigning values to variables, we reach the same partial assignment regardless of order. Therefore, we need only to consider assignments to a **single variable** at each node in the search tree. For example, at the root node of a search tree for coloring the map of Australia, we might make a choice between $SA = red, SA = green$, and $SA = blue$, but we would never choose between $SA = red$ **and** $WA = blue$. With this restriction, the number of leaves is d^n, as we would hope. Depth-first search for CSPs with single-variable assignments is called **backtracking search**. Backtracking search is the basic uninformed algorithm for CSPs. It can solve n-queens for $n \approx 25$

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

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

Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k-consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

The algorithm above implements backtracking search. It repeatedly chooses an unassigned variable, and then tries all values in the domain of that variable in turn, trying to find a solution. If an inconsistency is detected, then BACKTRACK returns failure, causing the previous call to try another value.

In the previous chapters we improved the poor performance of uninformed search algorithms by supplying them with domain-specific heuristic functions derived from our knowledge of the problem. It turns out that we can solve CSPs efficiently without such domain-specific knowledge. Instead, we can add some sophistication to the unspecified functions in the algorithm presented above using them to address the following questions:

1. Which variable should be assigned next?

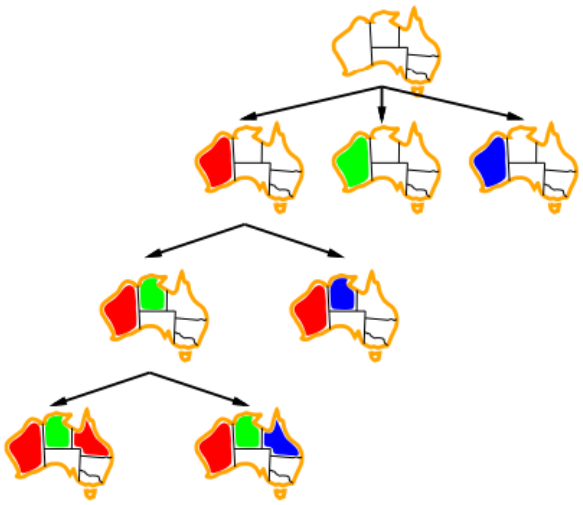
2. In what order should its values be tried?

3. When the search arrives at an assignment that violates a constraint, can the search avoid repeating this failure?

4. Can we take advantage of problem structure?

20.4.1 Variable and value ordering

The simplest strategy for SELECT-UNASSIGNED-VARIABLE is to choose the variable with the fewest "legal" values. This technique is usually called the **most constrained variable** heuristic.



For example, in the figure above, after the assignments for $WA = red$ and $NT = green$ (level 2) there is only one possible value for SA, so it makes sense to assign $SA = blue$ next rather than assigning Q.

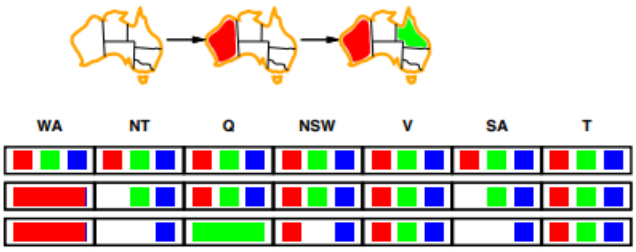
If some variable X has no legal values left, the heuristic will select X and failure will be detected immediately, avoiding pointless searches through other variables (pruning the search tree). It usually performs better than a random or static ordering, sometimes by a factor of 1,000 or more, although the results vary widely depending on the problem.

Once a variable has been selected, the algorithm must decide on the order in which to examine its values (ORDER-DOMAIN-VALUES). For this, the **least-constraining-value** heuristic can be effective in some cases. It prefers the value that rules out the fewest choices for the neighboring variables in the constraint graph. For example, suppose that we have generated the partial assignment with $WA = red$ and $NT = green$ and that our next choice is for Q. Blue would be a bad choice because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining-value heuristic therefore prefers red to blue.

Why should variable selection be fail-first, but value selection be fail-last? It turns out that, for a wide variety of problems, a variable ordering that chooses a variable with the minimum number of remaining values helps minimize the number of nodes in the search tree by pruning larger parts of the tree earlier. 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.

20.4.2 Forward checking

What inferences should be performed at each step in the search of the backtracking algorithm? The main idea behind **forward checking** is to keep track of remaining legal values for unassigned variables, terminating the search when any variable has no legal values. Whenever a variable X is assigned, for each unassigned variable Y that is connected to X by a constraint, delete from Y 's domain any value that is inconsistent with the value chosen for X. By doing this, it eliminates branching on those values, making the search more efficient. Once forward checking detects that the partial assignment is inconsistent with the constraints of the problem, it backtracks immediately. Basically, it makes the backtracking algorithm more efficient preventing it from continuing the search when a partial assignment is inconsistent with the constraints of the problem.



The figure above shows the progress of backtracking search on the Australia CSP with forward checking. There are two important points to notice about this example. First, notice that after $WA = red$ and $Q = green$ are assigned, the domains of NT and SA are reduced to a single value; we have eliminated branching on these variables altogether by propagating information from WA and Q. A second point to notice is that after $V = blue$, the domain of SA is empty. Hence, forward checking has detected that the partial assignment $\{WA = red, Q = green, V = blue\}$ is inconsistent with the constraints of the problem, and the algorithm will therefore backtrack immediately.

20.4.3 Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures. To solve this problem we 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 key idea is **local consistency**. There are different types of local consistency, that we do not have time to discuss...

20.4.4 Problem structure

In this section, we examine ways in which the structure of the problem, as represented by the constraint graph, can be used to find solutions quickly.

Looking again at the constraint graph for Australia, one fact stands out: Tasmania is not connected to the mainland. Intuitively, it is obvious that coloring Tasmania and coloring the mainland are **independent sub-problems**. Independence can be ascertained simply by finding **connected components** of the constraint graph. Each component corresponds to a sub-problem.

Suppose each sub-problem has c variables out of n total, where c is a constant. Then there are n/c sub-problems, each of which takes at most d^c work to solve, where d is the size of the domain. Hence, the total work is $O(d^c n/c)$, which is linear in n. Without the decomposition, the total work is $O(d^n)$, which is exponential in n. Let's make this more concrete: dividing a Boolean CSP with 80 variables into four sub-problems reduces the worst-case solution time from the lifetime of the universe down to less than a second.

Completely independent subproblems are delicious, then, but rare. Fortunately, some other graph structures are also easy to solve. For example, a constraint graph is a tree when any two variables are connected by only one path.

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time.

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

  n ← number of variables in X
  assignment ← an empty assignment
  root ← any variable in X
  X ← 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] ← any consistent value from D_i
    if there is no consistent value then return failure
  return assignment
```

Figure 6.11 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.

20.5 Local search for CSPs

Local search algorithms turn out to be effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time. The point of local search is to eliminate the violated constraints.

The next variable for an assignment is randomly selected within any conflicted variable. In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables, the **min-conflicts** heuristic. Min-conflicts is surprisingly effective for many CSPs. Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the run time of min-conflicts is roughly independent of problem size. It solves even the million-queens problem in an average of 50 steps (after the initial assignment).

Chapter 21

Lec 21 - Natural Language Processing

21.1 Introduction

There are over a trillion pages of information on the Web, almost all of it in natural language. An agent that wants to do knowledge acquisition needs to understand (at least partially) the ambiguous, messy languages that humans use. We examine the problem from the point of view of specific information-seeking tasks: text classification, information retrieval, and information extraction. One common factor in addressing these tasks is the use of **language models**: models that predict the probability distribution of language expressions.

21.2 Language models

Formal languages, such as the programming languages Java or Python, have precisely defined language models. A language can be defined as a set of strings; "print(2 + 2)" is a legal program in the language Python, whereas "2)+(2 print" is not. They are specified by a set of rules called a grammar. Formal languages also have rules that define the meaning or semantics of a program; for example, the rules say that the "meaning" of "2+2" is 4.

Natural languages, such as English or Spanish, cannot be characterized as a definitive set of sentences. Everyone agrees that "Not to be invited is sad" is a sentence of English, but people disagree on the grammaticality of "To be not invited is sad." Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set.

Natural languages are also ambiguous. "He saw her duck" can mean either that he saw a waterfowl belonging to her, or that he saw her move to evade something. Thus, again, we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings.

Finally, natural languages are difficult to deal with because they are very large, and constantly changing. Thus, our language models are, at best, an approximation.

21.3 *N*-gram models

Ultimately, a written text is composed of characters, letters, digits, punctuation, and spaces in English (and more exotic characters in some other languages). Thus, one of the simplest language models is a probability distribution over sequences of characters. We write $P(c_{1:N})$ for the probability of a sequence of N characters, c_1 through c_N. A sequence of written symbols of length n is called an n-gram. with special case "unigram"

for 1-gram, "bigram" for 2-gram, and "trigram" for 3-gram. A model of the probability distribution of n-letter sequences is thus called an **n-gram model**.

An n-gram model is defined as a **Markov chain** of order $n-1$. In a Markov chain the probability of character c_i depends only on the immediately preceding characters, not on any other characters. So in a trigram model (Markov chain of order 2) we have

$$P(c_i|c_{1:i-1}) = P(c_i|c_{i-2:i-1})$$

We can define the probability of a sequence of characters $P(c_{1:N})$ under the trigram model by first factoring with the chain rule and then using the Markov assumption:

$$P(c_{1:N}) = \prod_{i=1}^{N} P(c_i|c_{1:i-1}) = \prod_{i=1}^{N} P(c_i|c_{i-2:i-1})$$

For a trigram character model in a language with 100 characters, $\mathbf{P}(c_i|c_{i-2:i-1})$ has a million entries (100^3), and can be accurately estimated by counting character sequences in a body of text of10 million characters or more. We call a body of text a **corpus** (plural corpora), from the Latin word for body.

What can we do with n-gram character models? One task for which they are well suited is language identification: given a text, determine what natural language it is written in. One approach to language identification is to first build a trigram character model of each candidate language, $P(c_i|c_{i-2:i-1}, l)$, where the variable l ranges over languages. For each l the model is built by counting trigrams in a corpus of that language (About 100,000 characters of each language are needed). That gives us a model of $\mathbf{P}(Text|Language)$, but we want to select the most probable language given the text, so we apply Bayes' rule followed by the Markov assumption to get the most probable language:

$$\begin{aligned} l^* &= argmax_l P(l|c_{1:N}) \\ &= argmax_l P(l)P(c_{1:N}|l) \\ &= argmax_l P(l) \prod_{i=1}^{N} P(c_i|c_{i-2:i-1}, l) \end{aligned}$$

The trigram model can be learned from a corpus, but what about the prior probability $P(l)$? We may have some estimate of these values; for example, if we are selecting a random Web page we know that English is the most likely language and that the probability of Macedonian will be less than 1%. The exact number we select for these priors is not critical because the trigram model usually selects one language that is several orders of magnitude more probable than any other.

Other tasks for character models include spelling correction, genre classification, and named-entity recognition.

21.3.1 Smoothing *n*-gram models

The major complication of n-gram models is that the training corpus provides only an estimate of the true probability distribution. For common character sequences such as "_th" any English corpus will give a good estimate. On the other hand, "_ht" is very uncommon, no dictionary words start with ht. It is likely that the sequence would have a count of zero in a training corpus of standard English. Does that mean we should assign $P("ht") = 0$? If we did, then the text "The program issues an http request" would have an English probability of zero, which seems wrong. We have a problem in generalization: we want our language models to generalize well to texts they haven't seen yet.

The process od adjusting the probability of low-frequency counts is called **smoothing**. The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for $P(X = true)$ should be $1/(n+2)$. That is, he assumes that with two more trials, one might be true and one false. This technique is called **Laplace smoothing**.

A better approach is a **backoff model**, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to $(n-1)$-grams. **Linear interpolation** smoothing is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as

$$\hat{P}(c_i|c_{i-2:i-1}) = \lambda_3 P(c_i|c_{i-2:i-1}) + \lambda_2 P(c_i|c_{i-1}) + \lambda_1 P(c_i)$$

where $\lambda_3 + \lambda_2 + \lambda_1 = 1$.

21.4 Model evaluation

How do we know what model to choose? We can evaluate a model with cross-validation. The evaluation can be a task-specific metric, such as measuring accuracy on language identification, but we would like to define a task-independent language quality model. A different way of describing the probability of a sequence is with a measure called **perplexity**, defined as

$$Perplexity(c_{1:N}) = P(c_{1:N})^{-\frac{1}{N}}$$

Perplexity can be thought of as the reciprocal of probability, normalized by sequence length. Suppose there are 100 characters in our language, and our model says they are all equally likely. Then for a sequence of any length, the perplexity will be 100. If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

21.5 *N*-gram word models

Now we turn to n-gram models over words rather than characters. All the same mechanism applies equally to word and character models. The main difference is that the **vocabulary**, the set of symbols that make up the corpus and the model, is larger. There are only about 100 characters in most languages, but with word models we have at least tens of thousands of symbols, and sometimes millions.

Word n-gram models need to deal with **out of vocabulary** words. With character models, we didn't have to worry about someone inventing a new letter of the alphabet, but with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model. This can be done by adding just one new word to the vocabulary: $< UNK >$, standing for the unknown word. We can estimate n-gram counts for $< UNK >$ by this trick: go through the training corpus, and the first time any individual word appears it is previously unknown, so replace it with the symbol $< UNK >$. Then compute n-gram counts for the corpus as usual, treating $< UNK >$ just like any other word.

To get a feeling for what word models can do, we built unigram, bigram, and trigram models over the words in this book and then randomly sampled sequences of words from the models. The results are:

Unigram: logical are as are confusion a may right tries agent goal the was ...
Bigram: systems are very similar computational approach would be represented ...
Trigram: planning and scheduling are integrated the success of naive bayes model is ...

Even with this small sample, it should be clear that the unigram model is a poor approximation of either English or the content of an AI textbook, and that the bigram and trigram models are much better.

Chapter 22

Lec 22 - Natural Language Processing II

22.1 Chain rule for n-grams

The probability of a sentence W (sequence of words), is given by:

$$P(w_1, ..., w_n)$$

while a language model computes the probability

$$P(w_n|w_{1:n-1})$$

Then, Chain rule can be used to calculate the joint probability of words:

$$P(w_1, ..., w_n) = \prod_i P(w_i|w_{1:i-1})$$

For example, the probability of the sentence "Today is hot" is given by:

$$P(\text{Today is hot}) = P(\text{Today})P(\text{is}|\text{Today})P(\text{hot}|\text{Today is})$$

How to estimate these probabilities? We cannot estimate them by considering all the combinations of words in a corpus, since we will never see enough data to estimate such probabilities.

Then, as we did for n-grams models for characters, we can simplify by making a Markov assumption:

$$P(w_1, ..., w_n) = \prod_i P(w_i|w_{1:i-1}) = \prod_i P(w_i|w_{i-k:i-1})$$

where k determines which n-gram we are considering ($k = 1$ bigram, $k = 2$ trigram, etc.). Then, we can estimate the probability of $P(w_i|w_{i-1})$ directly from the corpus as follows:

$$P(w_i|w_{i-1}) = \frac{Count(w_i, w_{i-1})}{Count(w_{i-1})}$$

22.2 Text classification

We now consider the task of **text classification**: given a text of some kind, decide which of a predefined set of classes it belongs to. We can treat spam detection as a problem in supervised learning. A training set

is readily available: the positive (spam) examples are in my spam folder, the negative (ham) examples are in my inbox.

We can have two different ways to approach the classification of the text:

- Use of hand-coded classification rules
- Use of linguistic modeling and machine learning

If the rules are carefully refined by an expert, the accuracy can be high, but building such rules is very expensive and not always possible.

In the language-modeling approach, we define one n-gram language model for $\mathbf{P}(Message|spam)$, by training on the spam folder, and one model for $\mathbf{P}(Message|ham)$ by training on the inbox. Then we can classify a new message with an application of Bayes' rule:

$$argmax_{c\in\{spam,ham\}}P(c|message) = argmax_{c\in\{spam,ham\}}P(message|c)P(c)$$

where $P(c)$ is estimated just by counting the total number of spam and ham messages. Assume that each message is made of n tokens x_i. Then, the Bayes' rule can be rewritten as:

$$argmax_{c\in\{spam,ham\}}P(c|message) = argmax_{c\in\{spam,ham\}}P(x_1,...,x_n|c)P(c)$$

Estimate $P(x_1,...,x_n|c)$ is usually unfeasible since we don't have enough data to do it. Then, we need to rely on some approximation. In particular, we can exploit conditional independence. Basically, we assume that all the tokens are conditionally independent, given c, i.e., $P(x_1,...,x_n|c) = \prod_i P(x_i|c)$, which leads to the definition of Multinomial Naive Bayes Classifier:

$$c_{NB} = argmax_{c\in\{spam,ham\}}P(c)\prod_i P(x_i|c)$$

22.3 Text representation

In the machine-learning approach we represent the message as a set of feature/value pairs and apply a classification algorithm h to the feature vector $\mathbf{X}$. How can we represent a message?

22.3.1 Bag-of-Words

Bag-of-Words is a way to convert text data into a numerical format that machine learning algorithms can work with. The term "bag of words" implies that the order of words is disregarded, and the focus is solely on the presence or absence of words in a document.

The idea is to think of the n-grams as features. This is easiest to see with a unigram model. The features are the words in the vocabulary, and the values are the number of times each word appears in the message. That makes the feature vector large and sparse. This unigram representation has been called the bag of words model.

The notion of order of the words is lost; a unigram model gives the same probability to any permutation of a text. . Higher-order n-gram models maintain some local notion of word order, but the computational complexity increases with the order.

Representing words as discrete symbols has a big problem: there is no notion of similarity between any encoding vector, even if the sentences are semantically similar. We have 2 possible solutions:

- "Knowledge representation way": can rely on a list of synonyms or taxonomies of words (for example WordNet) to obtain similarity.

- "Machine learning way": learning to code the similarity in the vectors themselves.

22.3.2 Word Embedding - Word2Vec

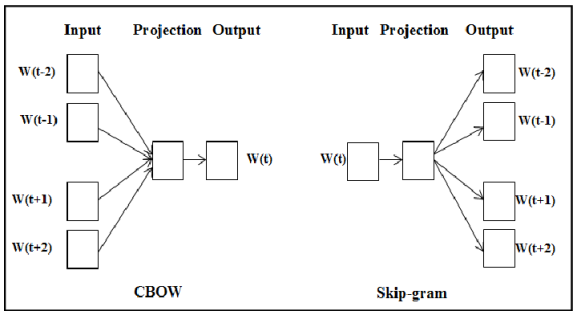
(Neural) **Word embedding** belongs to the text pre-processing phase. The goal is to transform a text (set of words) into a vector of numbers such that similar words produce similar vectors. The word2vec technique is based on a feed-forward, fully connected architecture. It is similar to an autoencoder, but rather than performing input reconstruction, word2vec trains words according to other words that are neighbors in the input corpus (called **context**).

Word2vec aims at computing a **vector** representation of words able to capture semantic and syntactic word similarity.

By using this concept of context, word2vec can learn word embedding in two different ways:

- **CBOW** (Continuous Bag of Words) in which the neural network uses the context to predict a target word.

- **Skip-gram** in which it uses the target word to predict a target context.



Word2Vec Skip-gram model

The input of the model is the one-hot encoding of the word according to a vocabulary. The output is a distribution over the words of being in the context of the target word. The formal method is defined as follows:

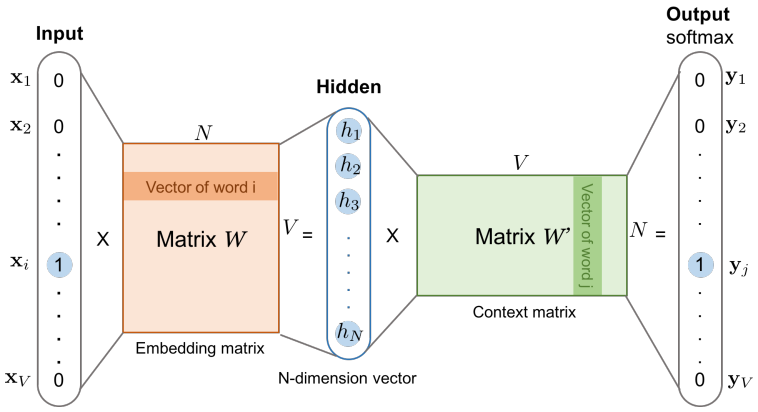
$$\mathbf{h} = \mathbf{W}^T\mathbf{x}$$

$$\mathbf{u} = \mathbf{W}^{'T}\mathbf{h} = \mathbf{W}^{'T}\mathbf{W}^T\mathbf{x}$$

$$\mathbf{y} = \sigma(\mathbf{u})$$

where $\sigma(\cdot)$ is the **softmax** function.

The goal of skip-gram model is to maximize the **average log probability**.



Word2Vec - Semantic and syntactic relations

Word2vec captures different degrees of similarity between words. For example, patterns such as *Man is to Woman as Brother is to Sister* can be generated through algebraic operations on the vectors representations of the words.

$$Brother - Man + Woman \approx Sister$$

22.4 Syntactic Parsing

Parsing is the process of analyzing a string of words to uncover its phrase structure, according to the rules of a grammar.

22.4.1 Dependency Parsing

Which words depend on other words in our text data? What word is the "head" word in charge, so to speak? These are the questions answered by dependency parsing. This is the task of extracting a grammatical structure that clearly defines the relationships between "head" words and all other words. Thinking of it like a graph, the words are nodes and the dependencies between them are edges. Some specifically call this a Dependency Structure. In this structure, there is a defined root word. All other words can be reached (are dependent to) through this word as a starting point of the text data.

22.4.2 Constituency Parsing

Where Dependency Parsing is based on dependency grammar, Constituency Parsing is based on context-free grammar. This type of parsing deals with the types of phrases in the text data. Constituency Parsing breaks text into sub-phrases, constituents, based on a grammar category. They are basically their own unit of grammar. Some categories for the phrase units of grammar are noun phrase (NP), verb phrase (VP) or sometimes prepositional phrase (PP).

Chapter 23

Lec 23 - Modern NLP by Deep Learning

23.1 Subword Models

We have already seen how the problem of unknown words can be fixed using the $< UNK >$ symbol. However, novel words may be variations, misspellings, new words, mostly derived from already existing words in the dictionary. Furthermore, In some languages the morphology of the words is very complex (100s of variations for the same word). It would be useful, then, to have a model capable of dealing with new words without oversimplifying the task using the $< UNK >$ approach.

This kind of model is called **subword model**. The main idea is to learn a vocabulary of parts of words (subword tokens) where each word is split into a sequence of known subwords. A general algorithm to do so is the following:

1. Initialize the vocabulary with only character and an "end-of-word" symbol.

2. Given a corpus of text, find the most common pair of adjacent characters x, y. Add the subword xy to the vocabulary.

3. Replace all instances of the character pair with the new subword. Repeat until the desired vocabulary size is reached.

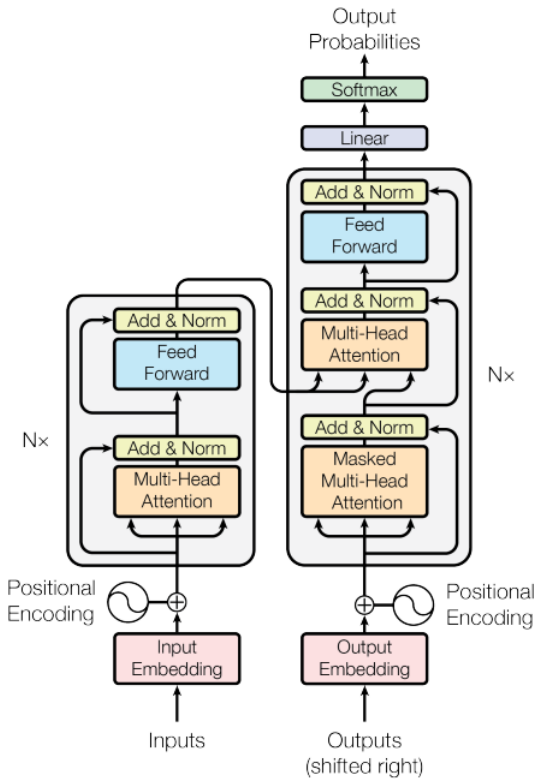
The result of this procedure is that common words are typically part of the subword vocabulary, while rarer words are split into components.

23.2 Transformers

The vanilla Transformer is a sequence-to-sequence model typically used for Machine translation and consists of an encoder and a decoder, each of which is a stack of N identical blocks. The encoder maps an input sequence of symbol representations $(x_1, ..., x_n)$ to a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$. Given $\mathbf{z}$, the decoder then generates an output sequence $(y_1, ..., y_m)$ of symbols one element at a time. At each step the model is auto-regressive, consuming the previously generated symbols as additional input when generating the next. Basically, the encoder first encodes the full sentence, then the decoder decodes one word at time.

- **Encoder:** The encoder is composed of a stack of N[1] identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, positionwise fully connected feed-forward network. For building a deeper model, a residual connection is employed around each module, followed by Layer Normalization module.
- **Decoder**: Compared to the encoder blocks, decoder blocks additionally insert cross-attention modules over the output of the encoder stack between the multi-head self-attention modules and the position-wise FFNs. Furthermore, the self-attention modules in the decoder are adapted to prevent each position from attending to subsequent positions.

Transformer is a model that uses **attention** to boost the speed. More specifically, it uses self-attention. Transformer allows for significantly more parallelization.



23.2.1 Input Embedding

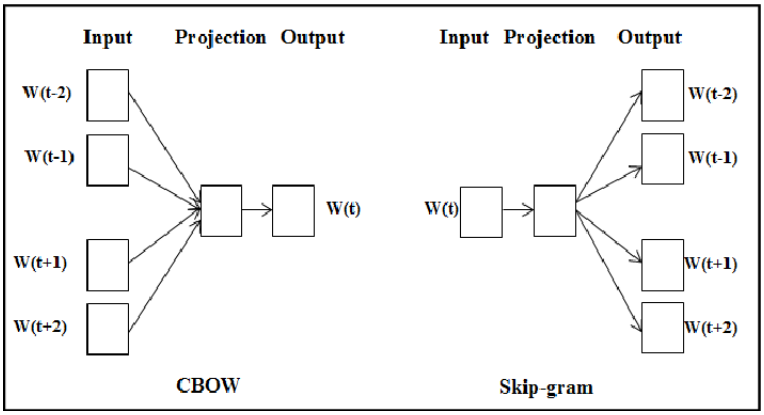
An embedding is a vector that **semantically** represents an object/input. In the context of NLP, the goal is to transform a text (set of words) into a vector of numbers such that similar words produce similar vectors. The **word2vec** technique is based on a feed-forward, fully connected architecture. It is similar to an autoencoder, but rather than performing input reconstruction, word2vec trains words according to other words that are neighbors in the input corpus.

Word2vec can learn word embedding in two different ways:

- **CBOW** (Continuous Bag of Words) in which the neural network uses the context to predict a target word.

[1] $N = 6$ in the original paper

- **Skip-gram** in which it uses the target word to predict a target context.



23.2.2 Positional Encoding

Position and order of words are the essential parts of any language. They define the grammar and thus the actual semantics of a sentence. Recurrent Neural Networks (RNNs) inherently take the order of word into account. They parse a sentence word by word in a sequential manner. This will integrate the words' order in the backbone of RNNs.

Since the model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence.

One possible solution to give the model some sense of order is the **positional encoding**.

Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique representation. There are many reasons why a single number, such as the index value, is not used to represent an item's position in transformer models. For long sequences, the indices can grow large in magnitude. If you normalize the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be normalized differently.

Transformers use a smart positional encoding scheme, where each position/index is mapped to a vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information.

Suppose you have an input sequence of length L. The positional encoding is given by sine and cosine functions of varying frequencies:

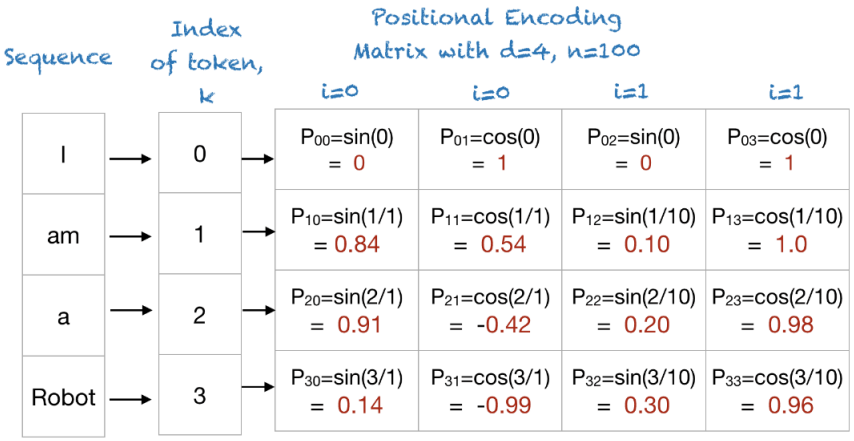
$$PE_{(k,2i)} = sin(k/n^{2i/d})$$
$$PE_{(k,2i+1)} = cos(k/n^{2i/d})$$

where:

- k is the position of an object in the input sequence.
- d is the dimension of the output embedding space.
- $PE_{(k,j)}$ is the position function for mapping a position k in the input sequence to index j of the positional matrix.

- n is a user-defined scalar, set to 10000 by the authors of the original Transformer's paper.
- i is used for mapping to column indices $0 \leq i < d/2$, with a single value of i maps to **both** sine and cosine functions.

You can also imagine the positional embedding as a vector containing pairs of sines and cosines for each frequency.

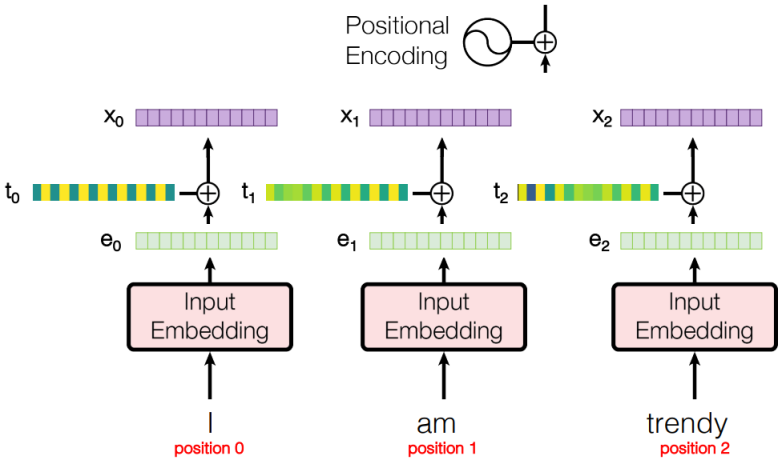


Positional Encoding Matrix for the sequence 'I am a robot'

The scheme for positional encoding has a number of advantages:

- The sine and cosine functions have values in $[-1, 1]$, which keeps the values of the positional encoding matrix in a normalized range.
- As the sinusoid for each position is different, you have a unique way of encoding each position.
- You have a way to add positional information to words embedding in order to encode the relative positions of words (e.g. the words 'brother' and 'sister' will probably have similar embedding representations, but if one word is used in a very different position than the other, they may not be correlated. In order to get different representations, we add positional information).

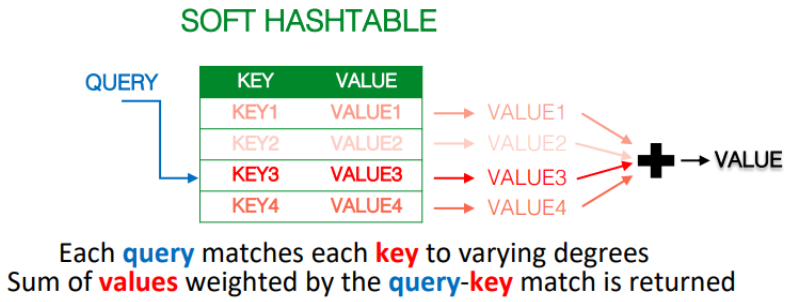
The positional encoding layer sums the positional vector with corresponding word embedding vector.



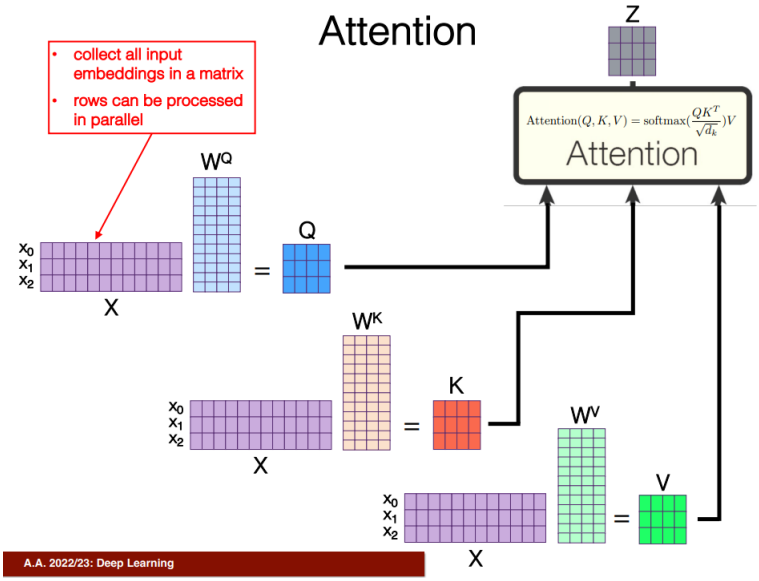
23.2.3 Attention

In order for the decoding to be precise, it needs to take into account every word of the input, using attention.

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.



The first step in calculating attention is to create three vectors from each of the encoder's input vectors. So for each word, we create a **Query vector**, a **Key vector**, and a **Value vector**. These vectors are created by multiplying the embedding by three matrices ($\mathbf{W^Q}, \mathbf{W^K}, \mathbf{W^V}$) that we trained during the training process.



If we collect all the query, keys and value vectors in three matrices ($\mathbf{Q}, \mathbf{K}, \mathbf{V}$), we can define the attention function as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

By taking the dot product between the query matrix and the key matrix, we compute a *score* for each word of the input sentence against the others. The score determines how relevant is a word with respect to the others. The dot-products of queries and keys are divided by $\sqrt{d_k}$ (dimensionality of the embedding vector) to alleviate gradient vanishing problem of the softmax function. The softmax normalizes the scores so they're all positive and add up to 1. $\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$ is often called **attention matrix**.

Then, we compute the dot product between the attention matrix and the value matrix $\mathbf{V}$. The intuition here is to keep intact the values of the relevant word(s), and drown-out irrelevant words (by multiplying them by tiny numbers in the attention matrix).

23.2.4 Multi-Head Attention

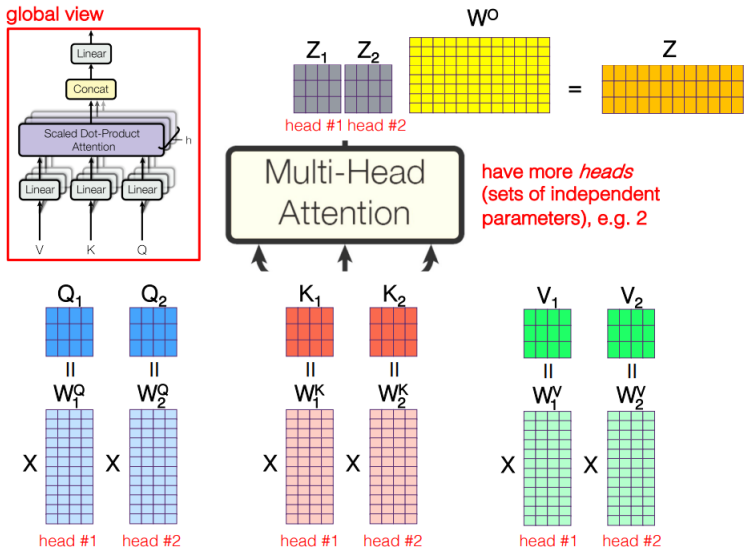
Instead of simply applying a single attention function, Transformer uses **multi-head attention**, where queries keys and values are linearly projected h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional output values. These are concatenated and once again projected, resulting in the final values.

Multi-head attention allows the model to jointly attend to information from different representation subspaces at different positions.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(head_1, ..., head_h)\mathbf{W^O}$$

where

$$head_i = Attention(\mathbf{QW}_i^\mathbf{Q}, \mathbf{KW}_i^\mathbf{K}, \mathbf{VW}_i^\mathbf{V})$$



23.2.5 Applications of Attention in the Model

The Transformer uses multi-head attention in three different ways:

- In "encoder-decoder attention" layers (cross-attention), the queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence.

- The encoder contains **self-attention** layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder stack. we set $\mathbf{Q} = \mathbf{K} = \mathbf{V} = \mathbf{Z}$, where $\mathbf{Z}$ is the output of the previous encoder layer.

- Masked Self-attention: We need to prevent leftward information flow in the decoder to preserve the auto-regressive property. Therefore, in the Transformer decoder, the self-attention is restricted such that queries at each position can only attend to all key-value pairs up to and including that position. This is implemented inside of scaled dot-product attention by masking out (setting to $-\infty$) all values in the input of the softmax which correspond to illegal connections.

23.2.6 Add & Norm and Feed Forward Network

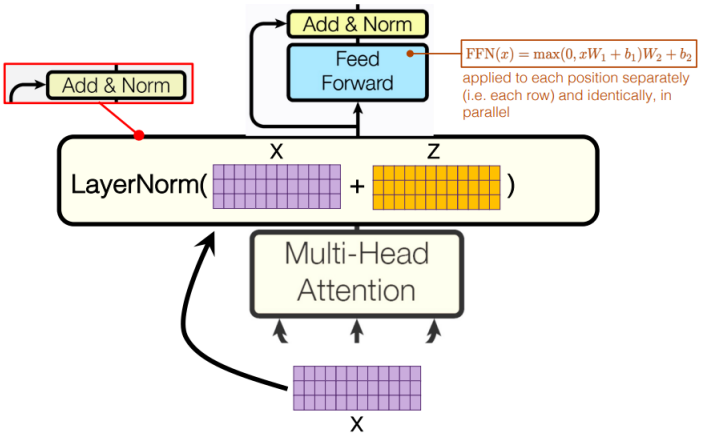
In addition to attention sub-layers, each of the layers in the encoder and decoder contains a fully connected feed-forward network, which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

$$FFN(h) = max(0, hW_1 + b_1)W_2 + b_2$$

where h is the output of the previous layer.

In order to build a deep model, Transformer employs a residual connection around each module, followed by Layer Normalization. For instance, each Transformer encoder block may be written as

$$\mathbf{H'} = \text{LayerNorm}(\text{SelfAttention}(\mathbf{X}) + \mathbf{X})$$
$$\mathbf{H} = \text{LayerNorm}(FFN(\mathbf{H'}) + \mathbf{H'})$$



23.3 Model Pre-training

Recent studies suggest that Transformer models that are pre-trained on large corpora can learn universal language representations that are beneficial for downstream tasks. The models are pre-trained using various self-supervised objectives:

- **Decoder only.**
- **Encoder only.**
- **Encoder-Decoder.**

After pre-training a model, one can simply fine-tune it on downstream datasets, instead of training a model from scratch.

23.3.1 Language model via a Decoder

The idea is to train a decoder to predict the next word/token in the sentence. Then, we fine-tune the resulting model to perform the learning task.

Generative Pre-Trained Transformer (GPT)

Given a corpus of unsupervised tokens $U = \{u_1, ..., u_n\}$, we first train a multi-layer transformer Decoder to maximize:

$$L_1(U) = \sum_i log\, P(u_i | u_{i-k}, ..., u_{i-1}, \Theta)$$

with i number of layers

$$h_0 = UW_e + W_p$$
$$h_l = \texttt{transformer\_block}(h_{l-1}) \forall i \in [1, n]$$
$$P(u) = \texttt{softmax}(h_n W_e^T)$$

Then, after the pre-training phase, we add on top of the decoder a softmax linear classifier and fine-tune the whole model (decoder + linear classifier) on the main learning task.

The first version of GPT was a Transformer decoder with 12 layers, 768-dimensional hidden states, 3072-dimensional feed-forward hidden layers; Byte-pair encoding with 40,000 merges; Trained on BooksCorpus: over 7000 unique books.

GPT-2 is a Larger version of GPT (1.5 billion parameters) trained on more data, was shown to produce relatively convincing samples of natural language.

GPT-3 is a Huge model: 175 billion parameters.

23.3.2 Masked Language Model via encoder

Decoders only take left context, while encoders are bidirectional (both left and right context), so how to train an encoder to perform Language Modeling ? We can perform **Mask Language Modeling** (MLM). Basically, we replace some fraction of words in the input with a special [MASK] token, and the self-supervised learning task is to predict these masked words.

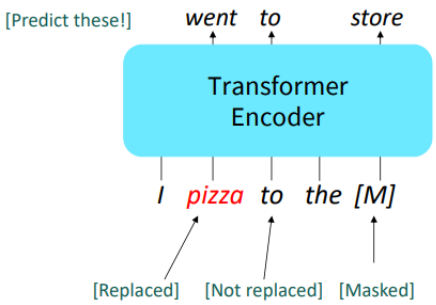
For example:

- original sentence: I went to the store.
- masked sentence: I [M] to the [M].

BERT: Bidirectional Encoder Representations from Transformers

A model based on MLM is BERT. The idea is to predict a random 15% of (sub)word tokens. In particular, the words are *masked* in 3 different ways:

- Replace the input word with [MASK] 80% of the time;
- Replace the input word with a random token 10% of the time;
- Leave the input word unchanged 10% of the time (but still predict it!)



This is done in order to make the model able to learn strong representations of non-masked words too.

Two models were released:

- BERT-base: 12 layers, 768-dim hidden states, 12 attention heads, 110 million params.
- BERT-large: 24 layers, 1024-dim hidden states, 16 attention heads, 340 million params.

Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architectures are used in both pre-training and fine-tuning.

23.3.3 Language Model via encoder-decoder

Encoders don't naturally lead to effective autoregressive (1-word-at-a-time) generation methods as decoders do. The idea is to use an encoder-decoder architecture in order to make the model able to perform both natural language understanding and generation.

In order to do this, we can replace different-length spans from the input with unique placeholders, and decode out the spans that were removed.

Chapter 24

Lec 24 - Basics of computer vision

24.1 Image representation

An **image** is a spatial distribution (two- or three-dimensional) of a physical entity that contains information related to the object (scene) that the image represents. That distribution can be represented as a continuous function that associates, to each point in the plane/space, the intensity of the physical entity at that point $f : \mathbb{R}^2 \to \mathbb{R}$(or $f : \mathbb{R}^3 \to \mathbb{R}$ for three-dimensional images).
Example:
Let's consider a monochrome image expressible by a continuous function with two independent variables:

$$f(n, m), f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

- n, m are the independent spatial coordinates of the image plane
- $f(n, m)$ gives the intensities of the measured physical quantity (usually light) at position (n, m).

24.1.1 Digitization

For digital processing is required a discrete representation of the images. This discretization process is composed by:

- **sampling:** can be expressed as a partition of the image plane in a grid of cells.
- **quantisation:** conversion from the continuous range of values of image intensity to a discrete and finite set of grey values [0-255].

24.1.2 Digital image representation

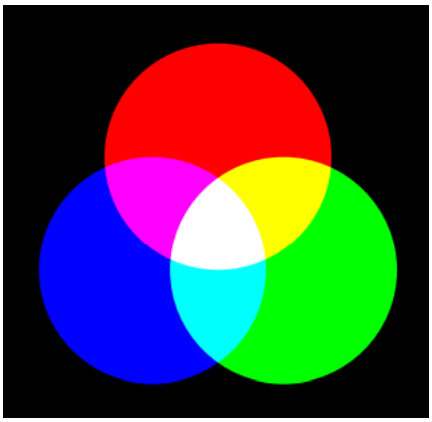
Images can be seen as a Cartesian coordinates system.

$$f[n, m] = \begin{bmatrix} \ddots & \vdots & \vdots & \vdots & \\ \dots & f[-1, 1] & f[0, 1] & f[1, 1] & \dots \\ \dots & f[-1, 0] & f[0, 0] & f[1, 0] & \dots \\ \dots & f[-1, -1] & f[0, -1] & f[1, -1] & \dots \\ & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

where $f[n, m]$ is a **discrete** function and n, m correspond to rows and columns of a matrix. Each element of the matrix is a **pixel** and each pixel can have a value, in case of a grey scale image, between 0-255. By convention, $f[0, 0]$ is the center of the image.

24.2 Colors

24.2.1 RGB representation



In the RGB (Red, Green, Blue) system, the primary colors are:

- red
- green
- blue

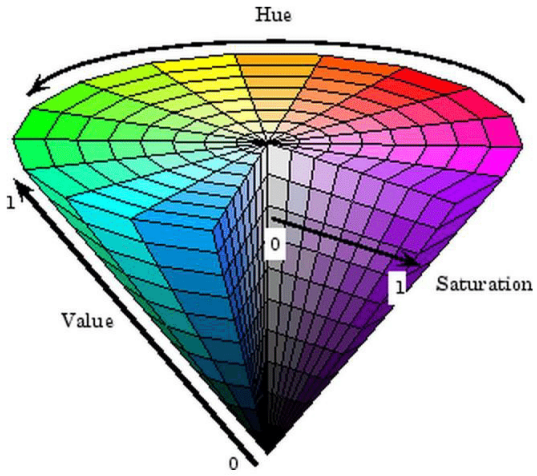
By composing two primary colors, you obtain secondary colors:

- cyan = green + blue
- magenta = red + blue
- yellow = red + green

By composing all the primary colors, you obtain white.
An RGB-encoded image consists of three channels, one for each component, where each color is obtained mixing red, green and blue values. If you use 8 bit to represent each component, the number of distinct colors that can be represented in the image is $(2^8)^3 = 16777219$.
The function that describes an RGB-image is the following:

$$f[n,m] = \begin{bmatrix} r[n,m] \\ g[n,m] \\ b[n,m] \end{bmatrix}$$

24.2.2 HSV representation



The HSV representation specifies colors in term of:

- **hue:** dominant color
- **saturation:** it measures how far you are from the fully saturated color (in which you don't have any grey).
- **lightness:** color brightness. While saturation measures the "dilution" of hue with white, brightness indicates dilution with black.

HSV representation is closer to how people perceive colors.

24.3 Filters

Filtering: Forming a new image whose pixel values are transformed from the original ones.
The goals of filtering are:

- **extract** useful information
- **transform** images into another domain where we can modify/enhance image properties.

24.3.1 Linear Filters

We define a filter as a unit that converts an input function $f[n, m]$ into an output function $g[n, m]$, where (n, m) are the independent variables.
Linear filters work by moving a sliding window (kernel) through the image, pixel by pixel. The window contains coefficients which characterize the transformation. At each position, the result of the filter is calculated by combining the values of the image subtended to the window with the coefficients of the window itself. In order to compute the new value of the central pixel, the coefficients are:

- multiplied by the values of the original image subtended to the window
- added

These filters are defined by the following mathematical operator called **convolution**:

$$(f * h)[n, m] = \sum_{k,l} f[k, l]h[n - k, m - l]$$

where h is the *kernel*.

Borders:
Given an $n \times n$ kernel, its external row/column coincides with the border of the image when the center of this mask is at distance $\frac{n-1}{2}$ from the edge. If you move further out, part of the window *leaves* the image. This situation can be managed in three different ways:

- limit the movement of the mask, keeping it at a minimum distance of $\frac{n-1}{2}$ from the edges.
- duplicate the external rows/columns of the image
- enlarge the image with rows/columns of zeros

Solution 1 gives reliable results, but produces a different size image from the original. Solutions 2 and 3, on the other hand, give results that are not exactly authentic near the edges, but are often convenient because they allow you to obtain an output image with the same size as the input one.

24.3.2 Smoothing filters

Smoothing filters are low-pass filters: they emphasize low frequencies and attenuate the high frequencies. They cause image blur, which is helpful to:

- remove small details
- reduce noise in the image

Examples:

- **Moving average filter** (*box filter*) in which all the kernel coefficients are equal to 1.

$$\frac{1}{9} \cdot \begin{array}{|c|c|c|}\hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} = \begin{array}{|c|c|c|}\hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \hline \end{array}$$

  It is defined by the following formula:

$$g[n, m] = \frac{1}{9} \sum_{k=-1}^{1} \sum_{l=-1}^{1} f[n - k, m - l]$$

  Basically, it replaces each pixel with an average of its neighbors. This filter can be applied in order to remove small details, to obtain blurring effect (more evident as the size of the window increases) but it is not useful to remove the so called *salt and pepper noise*, because corrupted pixels affect the computation of the average, so the noise points can even tend to dilate.

24.3.3 Sharpening filters

A sharpening filter emphasizes image details. They are high pass filters that emphasize high frequencies, and penalize low frequencies.
The sharpening effect is obtained in two steps:

- Apply an operator that extracts the details of the image (produces an image with light values in the transition areas and dark values in the uniform areas)
- This result is added to the original image

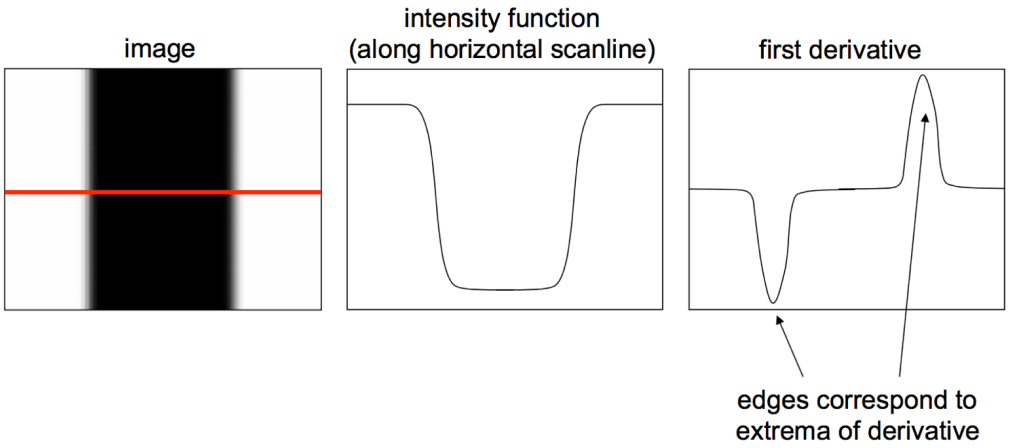
One way to obtain the details is by applying a smoothing filter to an image and computing a pixel by pixel difference between the original image and the smoothed one. Then you add the resulting image to the original one to obtain the sharpening effect.

24.4 Edge detection

An edge is a set of connected pixels between two regions in which there is a sudden change in intensity. These discontinuities can be caused by:

- Illumination discontinuity: cast shadows
- Change in surface orientation: shape
- Depth discontinuity: object boundary
- Surface color discontinuity

The goal of edge detection is to detect easily and automatically these regions.
Basically, a simple edge detection algorithm computes the first order derivative of the intensity function of an image and detects as edges the extremes of the derivative function.



24.4.1 Discrete approximation of the derivatives

In order to apply a derivative operator in a digital environment, we need a discrete approximation of derivatives.

- Derivative in 1D:

$$\frac{df}{dx} = lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x} = f^{'}(x)$$

- Discrete derivative in 1D:

$$\frac{df}{dx} = lim_{\Delta x \to 0} \frac{f(x) - f(x - \Delta x)}{\Delta x} = f^{'}(x)$$

$$= \frac{f(x) - f(x-1)}{1} = f^{'}(x)$$

$$= f(x) - f(x-1) = f^{'}(x)$$

Operators based on the first order derivative must satisfy the following properties:

- have zero value in the homogeneous sections of the image (no response in constant regions)
- have a non-zero value along the transition areas.

Formulations satisfying these properties can be defined in terms of differences between pixel values.

According to this, a 2D derivative filter could be defined by the following kernels (central derivative):

1	1	1
0	0	0
-1	-1	-1

Figure 24.1: *
Detects horizontal edges

1	0	-1
1	0	-1
1	0	-1

Figure 24.2: *
Detects vertical edges

24.4.2 Image gradient

The operators based on the first order derivative are formulated starting from **gradient**. Given a function $f(x, y)$ (2D image), the gradient vector is defined as follows:

$$\nabla f = \begin{bmatrix} G_x \\ G_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \\ \frac{\partial f}{\partial y} \end{bmatrix}$$

The image gradient is obtained by applying two-dimensional derivative filters and it provides the following information:

- The partial derivatives with respect to both horizontal and vertical directions.
- The gradient magnitude, which indicates the intensity of the discontinuity.

$$||\nabla f|| = \sqrt{G_x^2 + G_y^2}$$

- The gradient direction, given by:

$$\theta = tan^{-1}\left(\frac{G_y}{G_x}\right)$$

- It points in the direction of most rapid increase in intensity.

24.5 Key-points

Edge detection is useful to extract information, recognize objects, recover geometry and viewpoint, but edges are not very robust against a number of transformations. So, it's not a good idea to use them, for example, for object detection.

Edges are an example of **key-point**. A key-point is a region of the image in which you have specific properties.

- **flat region:** no change in all direction.
- **edge:** no change along the edge direction.
- **corner:** significant change in all directions.

Corners are **local features** that are more informative and more robust to different image transformations than edges.

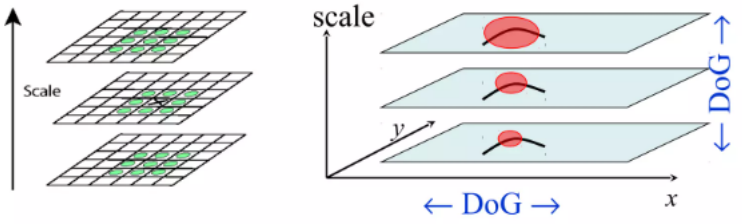
Scale invariant detection goal: given different images of the same scene with large scale differences between them, find the same key-points independently in each image.

24.6 SIFT algorithm

SIFT is a scale invariant feature detection algorithm that applies *DoG* both in space and over different scales. Thanks to this, it is computationally more efficient than the Harris-Laplacian algorithm. Following are the major stages of computation used to generate the set of image features:

1. **Scale-space extrema detection:** The first stage of computation searches for local features over all scales and image locations using a difference of Gaussian function.

2. **Key-point localization:** At each candidate location, a detailed model is fit to determine location and scale. Key-points are selected based on measures of their stability (**It was not covered in class**).
3. **Orientation assignment:** One or more orientations are assigned to each key-point location based on local image gradient directions.
4. **Key-point descriptor:** Describing the key-points as a high dimensional vector such that it is highly distinctive and invariant as possible to variations such as changes in viewpoint, illumination, translation, rotation and scale.



24.7 Bag of Visual Words

In order to implement image classification, we need two major components:

- A way to describe images. Not just local features descriptors, but a **global** representation of the image.
- A procedure to **compare** different images and learn a statistical model of a specific class.

Bag Of Visual Words is a technique to describe images. The approach has its origin in text retrieval and it is an extension of the Bag of Words algorithm. In Bag Of Words, we scan through the entire document and keep a count of each word appearing in the document. Then, we create a histogram of frequencies of words and use it to describe the text document. In Bag Of Visual Words, our input are images and we use **visual words** to describe them.

The pipeline follows these steps:

1. Extract local features (e.g. using SIFT) from training images.
2. Quantize the feature space (build a visual dictionary or codebook). Make this operation via clustering algorithms such as K-means. The center points, that we get from the clustering algorithm, are our visual words.
3. For each feature of each training image, find the closest visual word in the visual dictionary and build frequency histograms (one for each training image).
4. Compute histograms of visual words of **test** images (following the same procedure) and predict their class using the histograms of training images (e.g. using k-NN).

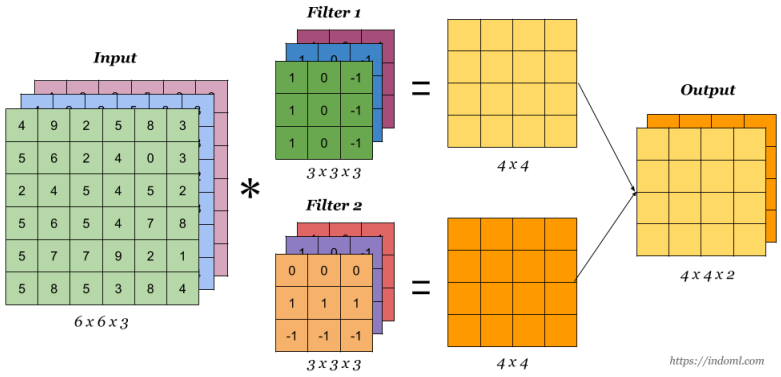
24.8 Convolutional Neural Networks (CNN)

Convolutional Neural Networks are a specialized kind of NN for processing data that has a **grid-like topology** (e.g. images). The name CNN indicates that the network employs a mathematical operation called **convolution**.

CNN learns different levels of abstraction of the input, e.g. for images:

- in the first few hidden layers, the CNN usually detects general pattern, like edges.
- the deeper we go into the CNN, these learned abstractions become more specific, like textures, patterns and parts of objects.

For what concerning convolution of multi-channel images, we perform the convolution for each channel and we sum up the results.

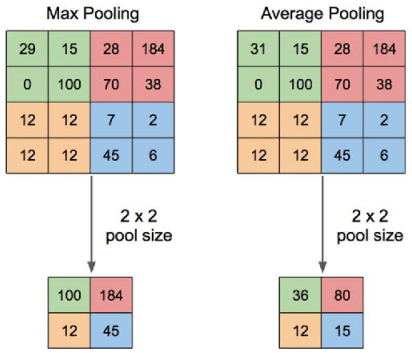


24.8.1 Pooling

Another commonly used technique in CNN is **Pooling**.

Pooling layer is used to reduce the size of the representations and to speed up calculations, as well as to make some features it detects a bit more robust.

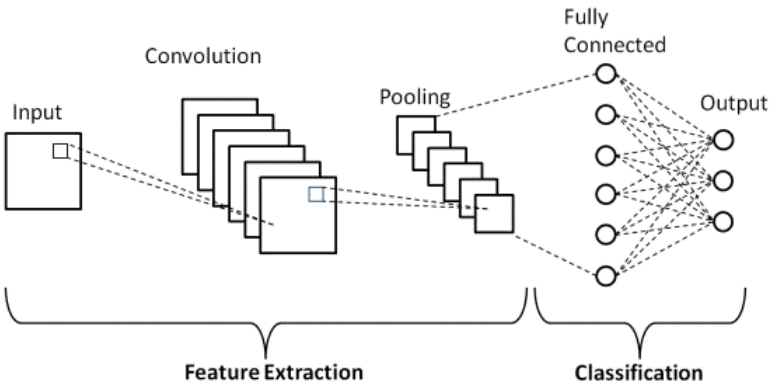
- **Max pooling:** It gets the maximum value of the pixels for each section of the image.
- **Average pooling:** It gets the average value of the pixels for each section of the image.



24.8.2 Convolution layer

The Convolution layer applies a set of filters to the input data (performing convolution) in order to extract features or representations from the data. The result is usually passed through an activation function (ReLu) in order to achieve non linearity.

Basically, a CNN is a sequence of convolution layers and sub-sampling layers with a fully-connected layer at the end.


Fully
Connected
Convolution
Pooling
Input
Output
Feature Extraction
Classification