Introduction to Artificial Intelligence Panikzettel

Philipp Schröer, Der Dude, Luca Oeljeklaus

April 5, 2018

Contents

1 Introduction

This Panikzettel is about the lecture Introduction to Artificial Intelligence by Prof. Lakemeyer held in the winter semester 2017/18.

This Panikzettel is Open Source. We appreciate comments and suggestions at
`https://git.rwth-aachen.de/philipp.schroer/panikzettel`.

2 Agent Architectures

2.1 Agent Types

An *agent* has *effectors* to influence its environment, based on *percepts* that are perceived through the agent's *sensors*. A *rational agent* acts according to some performance criteria. Rational action depends on the performance measure, the percept sequence, world knowledge and possible actions.

If you're the acronym type, you can remember this as *PAGE*: *percepts*, *actions*, *goals*, *environment*.

A *table-lookup agent* has a mapping indexed by percept sequences. Each action is just an entry in that table.

A *reflexive agent* has a set of condition-action rules that are pattern matched against a percept. Additionally, a *reflexive agent with an internal world model* may have a state machine as memory.

Goal-oriented agents try to fulfil (binary) goals by imagining the outcome of actions. *Utility-based agents* have a utility function that judges the expected outcomes of different actions somehow. This allows more flexibility, especially with uncertainty or conflicting goals.

Learning agents have a *learning element* that improves the system by changing and retrieving knowledge from the *performance element* (agent in the old sense, handles effectors). The *critic* judges sensor data and gives feedback to the learning element. Additionally, there's a *problem generator* that suggests actions to test performance of the performance element.

2.2 Environment Properties

- *accessible* (as opposed to *nonaccessible*):
  All relevant aspects of the world are available to the sensors.
- *deterministic* (as opposed to *nondeterministic/stochastic*):
  The next state depends completely on the current state and chosen action.
- *episodic* (as opposed to *nonepisodic*):
  The choice of an action depends only on the current state (not on the past).
- *static* (as opposed to *dynamic*):
  The world does not change while deciding on an action.
- *discrete* (as opposed to *continuous*):
  There are only finitely many world states in a range.

3 Search

3.1 Search Problems

This section is concerned with goal-oriented agents. Given an *initial state* of the world, an agent searches through a *state space*, where world states are connected by actions of the agent. The agent tries to find a sequence of actions leading to a world state that satisfies a goal (*goal test*).

The connections of states are given by the *operator*: It gives a description of which state is reached by an action from a given state. The *successor function* $S(x)$ returns the set of states reachable by any action from state x.

A sequence of actions is called a *path*. Paths have a *past cost*. A *solution* is a path from the initial state to a state that satisifies the goal test.

Finding a solution may have a *search cost*. The *total cost* also includes the path cost to the solution.

Definition: State space

- Set of states
- Set of operators
- Goal test function
- Path cost function
- Search cost function

Definition: Search problem

Starting at an *initial state*, search the state space to find a path to a goal state.

States are *expanded* during search to create successor states. This search induces a *search tree*.

There are varying degrees of difficulty for state space search problems.

	World knowledge	Action knowledge
Single-state problem	complete	complete
Multiple-state problem	incomplete	complete
Contingency problem	to be found at run-time	incomplete
Exploration problem	unknown	unknown

3.2 Search Strategies

Search strategies can be evaluated by a few metrics:

- *Completeness*: Always finds a solution if it exists.
- *Time complexity*: Worst case search time.
- *Space complexity*: Worst case memory usage.
- *Optimality*: Always finds the best solution, not just a solution.

An *uninformed (blind) search* does not use any information about the length or cost of a solution. An *informed (heuristic) search* does have some information.

In the following, b will be the maximum branching factor and d the depth of a solution.

3.2.1 Uninformed Search Strategies

Breadth-first search (BFS) expands nodes in the order they are generated.

Uniform cost search is a modified BFS that expands nodes with the least path cost first.

Depth-first search (DFS) always expands the node at the deepest level.

Depth-limited search is a DFS that only expands up to a specified depth.

Iterative deepening combines BFS and DFS. It executes depth-limited search at increasing depths until a solution is found.

If forward and backward searches are symmetric, then *bidirectional search* is possible.

Criterion	BFS	Uniform-Cost	DFS	Depth-Limited	Iterative Deepening	Bidirectional
Complete?	Yes	Yes	Yes	No	Yes	Yes
Time	$\mathcal{O}(b^{d+1})$	$\mathcal{O}(b^{1+\lfloor C^\star/\varepsilon \rfloor})$	$\mathcal{O}(b^m)$	$\mathcal{O}(b^\ell)$	$\mathcal{O}(b^d)$	$\mathcal{O}(b^{d/2})$
Space	$\mathcal{O}(b^{d+1})$	$\mathcal{O}(b^{1+\lfloor C^\star/\varepsilon \rfloor})$	$\mathcal{O}(bm)$	$\mathcal{O}(b\ell)$	$\mathcal{O}(bd)$	$\mathcal{O}(b^{d/2})$
Optimal?	No	Yes (pos. costs)	No	No	No	No

These properties hold with regard to finite graph assuming the algorithm tests for cycles and with b being the maximal branching factor, d the minimal depth of a goal state, m the maximal search depth, ℓ the depth restriction, $C^\star$ the cost of the optimal path and ε the minimum step cost.

3.2.2 Informed Search Strategies

Informed search strategies use an *evaluation function* f that returns the cost to the goal from a given node.

Best-first Search always expands the node with the best f-value.

Greedy search always chooses the node with the minimal (expected) cost to the goal state.

A^* search combines uniform cost search with greedy search.

- $g(n) =$ actual cost from initial state to n.
- $h(n) =$ estimated cost from n to the nearest goal.
- $f(n) = g(n) + h(n)$, the estimated cost of the cheapest path through n.

The heuristic h used for A^* must be *admissible*, that is $h(n) \leq h^*(n)$ must hold where $h^*(n)$ is the cost of an optimal path from n to the nearest goal.

To compare the quality of admissible heuristics we introduce the effective branching factor b^*. If an instance of A^* generates N nodes and the solution depth is d then b^* is the solution of:

$$N + 1 = 1 + (b^*)^1 + \cdots + (b^*)^d$$

E.g. a tree of depth d and branching factor b^* would contain the same number of nodes as the tree generated during the search.

Iterative Deepening A^* is a variant of Iterative Deepening search that explores branches up to a given threshold for the value of $f(n)$. If this threshold is passed and no solution was found the threshold is set to the minimal value of $f(n)$ for all found nodes n that exceeded the threshold.

Simplified Memory-Bounded A^* uses a bounded priority queue. New nodes are added to the queue. Each time a node is added, the algorithm checks whether all sibling nodes are in the queue, then the parent is removed. If the memory is full, the node with the highest cost is removed from the queue and its parent is added (if not already present).

Hill Climbing simply iteratively expands the highest-valued successor of the current node.

Simulated Annealing uses a *temperature* that decreases with time. In each iteration, a random successor is chosen. If the successor has a higher value, then the next iteration starts with this node, otherwise only with probability $e^{\Delta E/T}$ where ΔE is the value difference and T is the temperature (iteration count). Otherwise the iteration restarts with the old node.

4 Games

Games are special cases of search problems. States are usually accessible. *Actions* are possible moves by a player.

From a player's point of view, there may be uncertain outcomes of actions, so games are contingency problems.

We will look at 2-person games. The players are called MIN and MAX. MAX moves first. Operators are legal moves.

MAX needs to find a path which leads to a winning state for every possible reaction of MIN.

Algorithm: Minimax

1. Generate complete game tree.
2. Apply utility function to terminal states.
3. Recursively calculate values for parent nodes, starting from the terminal states:
   - If parent is at MIN level, assign minimum of values of children.
   - If parent is at MAX level, assign maximum of values of children.

Alpha-Beta search cuts off the search using two variables in each path it traverses.

α is the best score for MAX along the path, while β is the best score for MIN.

As an explanation, let us consider the example on the next page. We let circles stand for MAX, squares for MIN, and numbers fur cutoff levels.

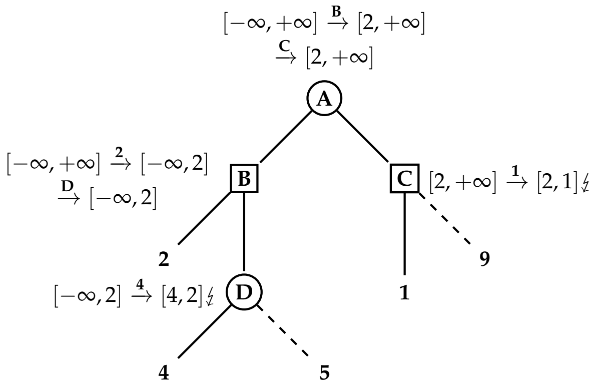
Let it be noted that we will be performing *left-to-right* Alpha Beta Search, and that dashed edges represent those that we do not visit.

Algorithm: Alpha Beta Search

```
Initially: alphaBeta(initialState, $-\infty$, $\infty$).

function alphaBeta(state, $\alpha$, $\beta$):
1. If this is a cutoff state, return this node.
2. For each successor $s$:
   • If at MAX level:
      a) Set $\alpha$ to max($\alpha$, alphaBeta($s$, $\alpha$, $\beta$)).
      b) If $\alpha \geq \beta$, return $\beta$.
   • If at MIN level:
      a) Set $\beta$ to min($\beta$, alphaBeta($s$, $\alpha$, $\beta$)).
      b) If $\beta \leq \alpha$, return $\alpha$.
3. Return $\alpha$ (MAX level) or $\beta$ (MIN level).
```

- We initialise $[\alpha, \beta] = [-\infty, +\infty]_A$ in **A**.
- We move to **B** and update to $[-\infty, 2]_B$, since $2 \leq +\infty$.
- We move to **D** and look at **4**. As $4 \geq 2$, going there is already worse than just going to **2** for **B**, so we go back up without even looking at **5**.
- From **B**, we move back to **A**. Since the best move **B** can make leads to **2**, we have a lower limit on how bad it can get for **A**. Thus, we set $[2, +\infty]_A$.



$[-\infty,+\infty] \xrightarrow{B} [2,+\infty]$
$\xrightarrow{C} [2,+\infty]$
A
$[-\infty,+\infty] \xrightarrow{2} [-\infty,2]$
$\xrightarrow{D} [-\infty,2]$
B
C
$[2,+\infty] \xrightarrow{1} [2,1]$ ↯
2
9
$[-\infty,2] \xrightarrow{4} [4,2]$ ↯
D
1
4
5

- We then move down to **C**, from where we look at **1**. As $1 \leq 2$, this is worse than going to **B**. Thus we don't visit **9**, as it doesn't matter how good or bad it is.

5 Knowledge Representation

In a rational agent the world must be explicitly represented as a *knowledge base* that contains sentences in a formal language.

Knowledge representation can be distinguished on three different levels:

- *Knowledge Level*: What is known by the knowledge base.
- *Symbolic Level*: The encoding of the knowledge base in a formal language.
- *Implementation Level*: Internal representation of sentences, like lists or strings of things.

5.1 First-order Logic

For more detail, refer to our Mathematische Logik Panikzettel (German).

5.1.1 Alphabet

Logical Symbols			**Nonlogical Symbols**	
Delimiter	*Operators*	*Variables*	*Predicate Symbols*	*Function Symbols*
$),($	$\neg, \wedge, \vee, \exists, \forall, =$	x, x_1, y, z	Friend, Enemy, etc.	bestFriendOf, etc.

5.1.2 Grammar

Terms	*Atomic wffs*	*Formulas*
• Every variable, • $f(t_1, \dots, t_n)$	• $P(t_1, \dots, t_n)$, • $t_1 = t_2$	• Every atomic wff, • x, $\neg\alpha$, $(\alpha \wedge \beta)$, $(\alpha \vee \beta)$, • $\exists x\ \alpha$, $\forall x\ \alpha$.

In the above α, β are wffs, and x is a variable.

Valid *expressions* consist of terms and (well-formed) formulas. *Propositional Logic* is a sub-language of First-order Logic: It does not contain terms, nor variables nor quantifiers, but only atomic wffs.

5.1.3 Notation

Parentheses can be omitted.

As abbreviations, we have:

- $\alpha \supset \beta$ ("α implies β") for $\neg\alpha \vee \beta$.
- $\alpha \equiv \beta$ ("α is equivalent to β") for $(\alpha \supset \beta) \wedge (\beta \supset \alpha)$.

A *substitution* $\alpha[x/t]$ is a replacement of all free occurrences of x by t in α.

5.1.4 Semantics

The semantics of a first-order formula is defined with respect to an *interpretation* $I = \langle D, \Phi \rangle$ where D is called the *domain/universe of discourse* and Φ is an interpretation function, i.e. for every predicate symbol P/function symbol f $\Phi(P)/\Phi(f)$ is the corresponding relation/function.

$I||t||$ denotes the element of the term t evaluated with respect to the interpretation I. For terms with free variables we write $I, \nu||t||$ where v is a function mapping variable names to elements of D.

Definition: First-order Logic semantics

Then the semantics of first-order logic is defined inductively by (for interpretation $I = \langle D, \Phi \rangle$):

- $I, \nu \models P(t_1, \ldots, t_n)$ iff $\langle d_1, \ldots, d_n \rangle \in \Phi(P)$ and $d_i = I, \nu||t_i||$.
- $I, \nu \models (t_1 = t_2)$ iff $I, \nu||t_1||$ equals $I, \nu||t_2||$.
- $I, \nu \models \neg\alpha$ iff $I, \nu \not\models \alpha$.
- $I, \nu \models (\alpha \wedge \beta)$ iff $I, \nu \models a$ and $I, \nu \models \beta$.
- $I, \nu \models (\alpha \vee \beta)$ iff $I, \nu \models a$ or $I, \nu \models \beta$.
- $I, \nu \models \exists x \alpha$ iff it exists one $d \in D$ such that $\nu_d^x \models \alpha$ where ν_d^x is like ν except that $\nu_d^x(x) = d$.

We also introduce the *logical consequence* which is similar to the implication but defined on the level of semantics and not syntax (S is a set of sentences and α a sentence):

$$S \models \alpha \text{ iff every interpretation satisfying every sentence in } S \text{ also satisfies } \alpha.$$

5.2 Knowledge-Based Systems

We start with a knowledge base (KB) that represents our knowledge of the world. The system needs to generate implicit knowledge that is a consequence of its KB. Therefore we need to be able to infer sentences using inference methods.

Deductive Inference is a process to compute the logical consequences of a KB, i.e. given a KB and a sentence α we want to compute if KB $\models \alpha$.

This process is *correct* if the computation is correct, e.g. if we compute KB $\models \alpha$ then KB $\models \alpha$.

This process is *complete* if for every sentence α such that KB $\models \alpha$ we can also compute that KB $\models \alpha$.

From Kurt Gödel we know that because there exists such a complete and correct process (resolution) for first-order logic, first-order logic is only semi-decidable.

6 Resolution

6.1 Clausal Form

A formula φ in conjunctive normal form (CNF) can be rewritten in clausal form by transforming every clause of φ into the set of literals it contains. The *clausal form* is the set of all those sets.

For readability we write [and] for set of literals (clauses) and $\{, \}$ for formulas. Furthermore we write $\sim$ for the negation.

Furthermore we define [] (the empty clause) to be false ($\vee$ monoid identity) and the empty formula $\{\}$ to be true ($\wedge$ monoid identity).

Definition: Clausal Form

Given a formula φ in conjunctive normal form, i.e.

$$\varphi = (\varphi_{1,1} \vee \ldots \vee \varphi_{1,n}) \wedge \ldots,$$

then φ is in *clausal form* if written as

$$\{\, [\varphi_{1,1}, \ldots, \varphi_{1,n}], \ldots \,\}\,.$$

$$\text{The clausal form of} \quad (a \vee b) \wedge (b \vee \neg c) \wedge d \quad \text{is} \quad \{[a,b],[b,{\sim}c],[d]\}.$$

6.2 CNF for First-order Logic

Algorithm: Transformation to CNF

Input: Propositional wff.

Output: Formula in CNF.

1. Rename variables such that all different variables have unique names.
2. Eliminate $\equiv$ and $\supset$.
3. Move $\forall$'s and $\exists$'s to the left.
4. Eliminate $\exists$'s using *skolemisation*:
   - For each $\exists x\ \varphi$, add new symbol a representing that x.
   - For each $\forall y\ z$, add new function f such that all y are mapped to the corresponding z.
5. Distribute $\vee$ over $\wedge$.
   (At this point we have a formula in *prenex normal form*. We then transform the quantifier-free part of the formula into CNF).
6. To obtain the clausal form we eliminate all $\forall$'s.

6.3 The Rules of Resolution

Definition: First-order Resolution Inference Rules

Propositional Resolution

From two clauses

$$\{p\} \cup C_1 \text{ and } \{\sim p\} \cup C_2,$$

we can infer the clause

$$C_1 \cup C_2$$

which is called the *resolvent* of the input clauses relative to p.

First-Order Resolution

To deal with variables and quantifiers we introduce a function θ (*unifier*) which substitutes variable names.

Then given two clauses

$$\{l_1\} \cup C_1 \text{ and } \{\sim l_2\} \cup C_2 \text{ with } l_1\theta = l_2\theta,$$

we can infer

$$(C_1 \cup C_2)\theta.$$

If the empty clause [] is derivable from a formula in the resolution calculus, the formula is unsatisfiable. This allows proving KB $\models \alpha$ by inferring the empty clause from $(\bigwedge KB) \wedge \neg\alpha$.

To extract the answer of a query $\exists x\ P(x)$ we can introduce an answer predicate $A(x)$ which just occurs in our query but not in the KB, i.e.

$$\exists x\ P(x) \quad \text{to} \quad \exists x\ [P(x) \wedge A(x)]$$

Then instead of inferring the empty clause we try to infer the clause $[A(y)]$ where y is the searched value for x.

Skolemisation is not equivalence preserving but satisfiability preserving.

6.4 Most General Unifier (MGU)

A problem of using resolution may be to use unifiers θ that are not general enough. The *Most General Unifier* (MGU) can be calculated using the following algorithm in exponential time. Every unifier θ' can be represented as a composition of the MGU and some other unifier.

Algorithm: Computing the MGU

Input: Set of literals $\{ l_1, \ldots, l_n \}$.

Output: MGU θ.

```
1. Initialise $\theta = \emptyset$.
2. If all literals are unified $\theta$ (all $l_i\theta$ are identical), then success. Return $\theta$.
3. Find a disagreement set $DS$ of (sub-)formulas which are not equal, e.g.
       $P(a, v)$
       $P(a, f(x))$   $\Rightarrow$   $DS = \{ v, f(x) \}$.
4. Find variable $v \in DS$ and term $t \in DS$ that does not contain $v$.
   If this is not possible, abort. Not unifiable.
5. Set $\theta = \theta \cup \{ v/t \}$.
6. Goto 2.
```

7 Planning

Planning can be summed up as follows: Given a set of actions, an initial and a goal state, find a plan to reach the initial state from the goal state. Such a plan will consist of an arrangement of (possibly only partially) ordered actions.

It is important to distinguish planning from searching. While searching considers search states abstractly without actually requiring any more information than successor states and edges to them, planning works based on more detailed information on nodes, e.g. multiple preconditions. This allows the planning algorithm to create plans more easily, without as much search state generation.

7.1 STRIPS Operator

For planning, we can use STRIPS operators to formalise a single step. Such an operator consists of an action, a set of positive literals as preconditions and another set of positive and negative literals as effects.

All preconditions have to be met in the previous state so that an action can be carried out, and the following state is defined by the previous state, modified by the effects of the action.

Definition: STRIPS Operator

STRIPS Operators are noted as follows:

```
Op(  Action:  Go(there),
     Precond: At(here)
          ∧ Path(here,there),
     Effect:  At(there)
          ∧ ¬At(here)  )
```

Further, there exist `Start` and `Finish` operators. The `Start` operator only has effects and no preconditions, thus defining an initial state. The `Finish` operator only has preconditions and no effects, defining the final state.

7.2 Plans

Definition: Complete Plan

A plan is complete if:

$\forall S_j, c \in \text{Precond}(S_j) : \exists S_i, S_i \prec S_j, c \in \text{Effects}(S_i)$

and if, for *every* linearization

$$\forall S_k, S_i \prec S_k \prec S_j, \neg c \notin \text{Effects}(S_k)$$

Definition: Consistent Plan

A plan is consistent if:

$$S_i \prec S_j \implies S_j \not\prec S_i$$

and, provided distinct A and B

$$x = A \implies x \neq B$$

A *complete plan* requires that for any step every precondition to be fulfilled by some predecessor and that no step between the fulfilment and the requirement of a condition undoes this. A *consistent plan* requires that no two actions take place at the same time. A plan that is both complete and consistent is called a *solution*.

The *initial plan* consists only of the problem description, i.e. the `Start` and `Finish` states.

Partially ordered plans are a preorder on steps which means it defines a must-happen-before-relationship ($\succ$). *Steps* are STRIPS operators with instantiated variables. This delays plan decisions as long as possible: We try not to commit to an order of two unrelated series of steps. Partially ordered plans can be *linearised* to an actual order on steps.

Algorithm: POP (Partial Order Planning)

Input: `Start` and `Finish` states, operators.

Output: Partial order plan *plan*.

Instructions may `fail`: You may need to *backtrack* (revert to earlier state and try something else).

```
1. Create the initial plan with Start $\succ$ Finish and no causal links.
2. While plan is not a solution (steps fulfil successors):
   a) Select a subgoal: A step $S_{need}$ with a precondition $c$ that has not been achieved.
   b) Choose an operator $S_{add}$ that has effect $c$.
      fail if this is not possible.
        i. Add a causal link $S_{add} \xrightarrow{c} S_{need}$.
       ii. Add ordering constraint $S_{add} \succ S_{need}$.
      iii. If $S_{add}$ is a new step in the plan, then also add Start $\succ S_{add} \succ$ Finish.
   c) Resolve threats. For each $S_{threat}$ that threatens $S_i \xrightarrow{c} S_j$ choose either:
        • Promotion: Add $S_{threat} \succ S_i$.
        • Demotion: Add $S_j \succ S_{threat}$.
      If the plan is not consistent, fail.
3. Return plan.
```

The *POP algorithm* calculates a partial order plan. It does so by inserting actions that fulfil at least one of an operators unfulfilled conditions. Sometimes steps *threaten* each other: Some step may destroy the causal link between to other steps. In this case the threat must be either *promoted* (put the threat after the link) or *demoted* (put the threat before the link). Since threat resolution and operator selection can fail, backtracking may be necessary. The POP algorithm is PSPACE-complete.

8 Uncertainty

8.1 Probability Theory

$\mathrm{P}(A)$ is the probability that A holds. A is a proposition of propositional logic where we allow contructs like $X = n$ for random variables X and n taken from a finite domain. For example $P(\text{roll} = 6) = \frac{1}{6}$ describing the probability to roll a 6 using a dice with random variable roll 6 taken out of the domain $\langle 1, \ldots, 6 \rangle$.

Definition: Axioms of Probability Theory

$$\mathrm{P} \in [0,1] \subseteq \mathbb{R} \qquad \mathrm{P}(\text{true}) = 1,\ \mathrm{P}(\text{false}) = 0 \qquad \mathrm{P}(A \vee B) = \mathrm{P}(A) + \mathrm{P}(B) - \mathrm{P}(A \wedge B)$$

Conditional Probabilities $\mathrm{P}(C \mid T)$ describe the probability the C holds if T holds.

For example:
$\mathrm{P}(\text{playingFootball} \mid \text{weather} = \text{sunny})$.

Definition: Conditional Probability

$$\mathrm{P}(A \mid B) := \frac{\mathrm{P}(A \wedge B)}{\mathrm{P}(B)}$$

Bayes rule allows diagnostic reasoning: Based on prior knowledge of a condition B for A ($\mathrm{P}(A \mid B)$), and knowledge about A ($\mathrm{P}(A)$) and B ($\mathrm{P}(B)$) independently, we can infer probabilities about B being true given A holds ($\mathrm{P}(B \mid A)$).

Put differently: Bayes Rule calculates the probability of a cause B given an effect A from a causal relationship $A \mid B$ and independent information.

Theorem: Bayes Rule

$$\mathrm{P}(B \mid A) = \frac{\mathrm{P}(A \mid B) \cdot \mathrm{P}(B)}{\mathrm{P}(A)}$$

$$\mathrm{P}(cause \mid effect) = \frac{\mathrm{P}(effect \mid cause) \cdot \mathrm{P}(cause)}{\mathrm{P}(effect)}$$

$1/\,\mathrm{P}(A)$ is not an "important" term, so Bayes Rule is often written with $\alpha = 1/\,\mathrm{P}(A)$:

$$\mathrm{P}(B \mid A) = \alpha \cdot \mathrm{P}(A \mid B) \cdot \mathrm{P}(B).$$

This is called *normalisation*.

8.2 Belief Networks

Belief networks use Bayes Rule with a conditional independence assumption for efficient calculation of conditional probabilities. These assumptions take the form of "if C is given, then A and B are independent of each other":

$$\mathrm{P}(A \mid C \wedge B) = \mathrm{P}(A \mid C) \text{ and } \mathrm{P}(B \mid C \wedge A) = \mathrm{P}(B \mid C).$$

Then we can calculate conditional probabilities as follows:

$$\mathrm{P}(X \mid Y \wedge Z) = \alpha \cdot \mathrm{P}(X) \cdot \mathrm{P}(Y \mid X) \cdot \mathrm{P}(Z \mid X)$$

A *belief network* represents causal relationships with independence assumptions. Each node is has edges to other nodes if there is a direct causal relationship between the corresponding variables.

The probability associations for each node are called *Conditional Probability Tables* (CPTs).

The network is a correct representation of a joint distribution if each node is conditionally independent of all its other (recursive) predecessors given its (direct) parents.

Definition: Belief Network

A *belief network/Bayesian network* is a directed acyclic graph $G = (V, E)$ where

- V represent random variables,
- each node $X \in V$ has conditional probabilities
$$\mathrm{P}(X \mid V_{\text{in}}(X))$$
with $V_{\text{in}}(X)$ being the set of nodes with incoming edges to X.

To compute joint distributions, we can now simply use the network's independence relations.

$$\mathrm{P}(X_1 \wedge \ldots \wedge X_n) = \prod_{i=1}^{n} P(X_i \mid V_{\text{in}}(X_i))$$

8.3 d-Separation

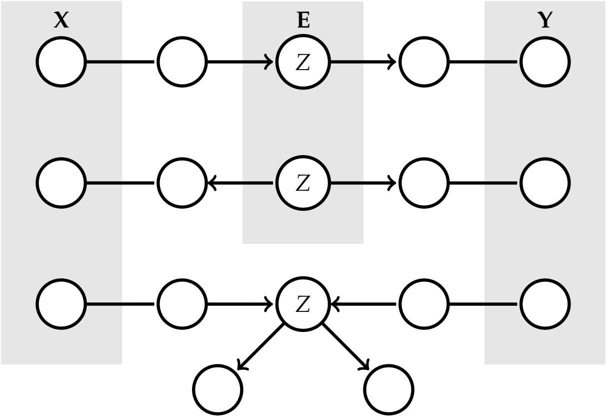
d-Separation (direction-dependent separation) allows computation of independence relationships in polynomial time. This is especially useful for belief network construction. The algorithm is incomplete (does not necessarily find all independence relationships), but is "good enough" for many applications.

$$\mathbf{E} \text{ d-separates } \mathbf{X} \text{ from } \mathbf{Y} \Rightarrow \mathbf{X} \text{ is independent of } \mathbf{Y} \text{ given } \mathbf{E}.$$

Definition: d-Separation

E d-separates **X** and **Y** if, on every undirected path from a node in **X** to a node in **Y**, there exists a node Z such that:

- $Z \in \mathbf{E}$ and the path makes use of one edge leading into Z and one leading away from Z according to their original direction.
- $Z \in \mathbf{E}$ and both edges used point away from Z.
- Neither Z nor any of its successors are contained in E and both edges lead into Z.



9 Learning

9.1 Kinds of Feedback during Learning

In *supervised learning*, the learner has access to input and correct output.

Reinforcement learning only gives feedback in terms of rewards and punishment, but not correct answers.

Unsupervised learning happens without any feedback to the learner. The learner must learn on its own.

9.2 Decision Trees

We can measure the average amount of information produced by some stochastic source of data with probabilities using the metric *(information) entropy*. High entropy means the source generates more information. For our purposes, low entropy also means the source can be predicted more easily.

Consider a fair coin flip with probabilities $\frac{1}{2}$ for both heads and tails. This source has entropy $\mathrm{I}(\frac{1}{2}, \frac{1}{2}) = 1$. However, a biased coin, e.g. one with probability $\frac{99}{100}$ has entropy $\mathrm{I}(\frac{99}{100}, \frac{1}{100}) = 0.08$.

Definition: Information Entropy

Let $v_1, \dots, v_n$ be the outcomes of a probabilistic experiment with probabilities $\mathrm{P}(v_1), \dots, \mathrm{P}(v_n)$.

$$\begin{aligned} &\mathrm{I}(\mathrm{P}(v_1), \dots, \mathrm{P}(v_n)) \\ &\qquad := -\sum_{i=1}^{n} \mathrm{P}(v_i) \cdot \log_2(\mathrm{P}(v_i)) \end{aligned}$$

We define $\log_2 0 := 0$.

Definition: Average Attribute Entropy

Let a be an attribute with n negative and p positive examples. Further, let p_i respectively n_i be the number of positive respectively negative occurrences of attribute value i.

$$\mathrm{AI}(a) = -\sum_{\text{value } i} \left(\frac{p_i + n_i}{p + n} \cdot \mathrm{I}\left(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i} \right) \right)$$

Decision tree-learning now builds decision trees by choosing attributes with the lowest average entropy until the attribute set only has `Yes` or `No` results.

The key idea is that, to build a small decision tree, and assuming the the simplest hypothesis is the most likely, lowest average entropy is a metric of how good an attribute is at predicting the final decision.

(Most definitions of decision tree-learning use *information gain* instead of average attribute entropy. Our definition simplifies the algorithm by omitting unnecessary constant terms.)

Algorithm: Decision Tree-Learning

Input: A set of examples with attributes.

Output: A decision tree.

1. Choose $a = \operatorname{argmin}_a \mathrm{AI}(a)$, the attribute with the lowest average entropy.
2. Create a node for a:
   - If a only has positive or negative examples, return a leaf with `Yes` or `No` accordingly.
   - Otherwise recursively build subtrees for each value with the remaining attributes.

9.3 Neural Networks

Neural networks are kind of similar to the human brain (but not in any detail). *Neural networks* consist of *units* (neurons) with *links* (synapses) between units with input and output edges. These edges have *weights*, usually real numbers. Units have an output value based on its weighted inputs called *activation level*. Special *input and output units* are connected to the external environment.

The output a_i of a unit i is usually calculated using some nonlinear function g.

$$a_i := g\left(\sum_j W_{j,i} \cdot a_j\right)$$

9.3.1 Network Topologies

Feed-forward networks are connected like a directed acyclic graph. Here, we call units that are not connected to the environment *hidden units*. Special cases of feed-forward networks are *perceptrons* which do not have hidden units.

Recurrent networks have arbitrarily complex connections.

Networks often have a *layer* structure where units of one layer are only connected to nodes of the next layer.

9.3.2 Neural Network Learning

The following simple learning algorithm for perceptrons will always converge to the correct output for representable functions. Unfortunately, perceptrons can only represent linearly separable functions.

Algorithm: Perceptron Learning

Input: Set of *examples* and learning rate α.

Output: A perceptron.

```
1. Initialise network with randomly assigned weights.
2. Repeat until examples are correctly predicted or stopping criterion is reached:
   • For each example $e \in examples$:
      a) Let $I$ be the input for $e$ and $T$ be the actual observed output for $e$.
      b) Calculate output $O$ using current network and input $e$.
      c) Update weights: $W_j = W_j + \alpha \cdot I_j \cdot (T - O)$.
3. Return network.
```

The learning algorithm is a bit more complex for multi-layer feed-forward networks. It is called *back-propagation*, because it propagates updates backwards through the network.

Algorithm: Back-Propagation

Input: Set of *examples* and learning rate α and base *network*.

Output: A multi-layer feed-forward network.

```
1. Repeat until examples are correctly predicted or stopping criterion is reached:
   • For each example $e \in examples$:
      a) Let $I$ be the input for $e$ and $T$ be the actual observed output for $e$.
      b) Calculate output $O$ using current network and input $e$.
      c) Compute error in output layer: $Err = T - O$.
      d) Update weights in output layer: $W_{j,i} = W_{j,i} + \alpha \cdot a_j \cdot (T_i - O_i) \cdot g'(in_i)$.
      e) For each subsequent layer:
            i. $\delta_j = g'(in_j) \cdot \sum_i W_{j,i}\delta_i$
           ii. $W_{k,j} = W_{k,j} + \alpha \cdot I_k \cdot (T_i - O_i) \cdot g'(in_i)$.
2. Return network.
```