1 Introduction to Artificial Intelligence

What is Artificial Intelligence?

- What intelligent human capability/job/skill that is currently unautomated do you think is potentially automatable?
- What would be an interesting thing for an AI system to be able to do?

Some well-known existing applications

- Expert systems (organic chemistry, medicine, geology, configuring computers)
- Speech recognition (TicketMaster, Speechworks, Bell)
- Handwriting recognition (US postal service zip code readers)
- Game playing (chess: Deep Blue, checkers — world champion)
- Robots (automated cars, ping pong player, MIT robot, Honda robot)
- Automated theorem proving (algebra, 4-color theorem, circuit testing)
- Web search engines
- Natural language understanding (Eliza, Ask Jeeves, spell checkers, grammar checkers, limited machine translation: AltaVista, Meteo)
- NASA probes (takes light ~1min to Mars, ~7mins to Jupiter)

Main intelligent capabilities

AI is about automating intelligent capabilities.

1. ***Reasoning***
    - mathematical reasoning
    - think logically
    - solve problems
    - figure out puzzles, games
    - reason from first principles

   E.g. Writing programs, designing software

   Hard to automate because:
    - large search space
    - hard to decompose (modularity essential)
    - hard to abstract (choosing data structures/objects critical)

2. ***Knowing***
   - embody knowledge about the world
   - be an expert in some specialized domain
   - have common sense about the way things work/are

   E.g. Medicine, auto-mechanics
   Hard to automate because:
   - vast knowledge
   - implicit non-expert knowledge

3. ***Interpreting***
   - interpret inputs/signals
   - perceive, understand
   - draw plausible conclusions
   - reason from incomplete/ambiguous/conflicting evidence
   - understand speech/language/vision

   E.g. Vision
   - Hard to automate because:
     - show slides (digit 5 and the dog picture)
     - image is just an array of numbers (pixel intensities)
     - no introspection
     - ambiguous, sketchy, conflicting evidence

   E.g. Understanding language
   - Hard to automate because:
     - no introspection
     - ambiguous, sketchy, conflicting evidence

4. ***Behaving***
   - achieve goals
   - act efficiently
   - demonstrate skill, coordination
   - make good decisions
   - trade off short term vs. long term reward
   - survive

   E.g. prisoner's dilemma
   Two prisoners: A and B. They can cooperate or not (they do not know each other's decision.) If they cooperate, both get 3, if one cooperates and not the other they get 0:5, if no one cooperates, they get 1:1. What is a good long term strategy?

E.g. Chess, Office politics
Hard to automate because:
- short term vs. long term tradeoffs
- uncertainty

E.g. Walking
Hard to automate because:
- coordinate many variables
- uncertainty
- system is inherently unstable
- real time (i.e. limited control computation time)

5. ***Learning***
   - adapt behavior
   - acquire knowledge
   - acquire skills
   - improve performance with experience

   E.g. Learning language, learning to walk
   Hard to automate because:
   - hidden prior constraints
   - no introspection

What is this course about?

Fundamental principles and techniques for
- Automated reasoning, problem solving
- Knowledge representation *(briefly)*
- Probability models and probabilistic inference for interpretation *(sketch of applications to vision, speech, language interpretation)*
- Optimal behavior *(decisions, games, sketch of robotics)*
- Learning

We will focus on algorithms and underlying mathematics
- *not* applications (this is primarily for the course project)

AI is still primarily about *research*

- experimental computer science
- frontiers of computer science

Course Work

Marking policy:

	4 Assignments	40%
	Project	30%
	Final exam	30%

Assignments:

A1: automated reasoning, problem solving, propositional logic, search

A2: automated planning, programming first-order logic

A3: probabilistic modelling, probabilistic inference, toy speech-vision problem

A4: optimal behavior, game playing, learning

Implementation in Java

Java vs. Scheme (Lisp) or Prolog:

- Java has garbage collection,
- easy string manipulation,
- high-level structures: e.g., vector, hash table
- easy for prototyping.
- Java does not have equivalent of lambda (Perl has, for example), but it is not frequently needed (not for this course, at least).
- Java does not have built-in backtracking and unification (as Prolog), but it is not needed in this course.
- Scheme and Prolog force certain kind of thinking and programming, which is frequently an obstacle for programmers used to the procedural languages.

Only standard Java packages can be used (e.g., `java.util`, `java.io`, `java.math`).

Project:

CS486: groups of three
CS686: each student alone

- choose a problem
- literature survey (10-15 references)
- methods: approach, rationale, implementation
- evaluation plan: hypotheses, experimental design
- experiments: results, critical evaluation
- conclusions

Project due dates:

Project statement October 2, Tuesday, in class

A single sheet of paper giving:

- Project title
- List of team members (name, number, e-mail) (3 or 1)
- Statement of problem
- List of possible approaches to tackling problem

Project report November 29, Thursday, in class

Ideas:

- game playing
- simple vision
- natural language processing
- learning
- automatic programming

What this course is *not* about

- Magic
- Hype

This course will focus on an engineering approach to AI, where we view AI simply as another application area of CS. That is, the specific problems studied are motivated by the desire to achieve the intelligent capabilities outlined above, but once identified, we take a standard CS approach to solving them: mathematical modeling, algorithm design and analysis, performance evaluation. The criterion for success is simply to achieve the desired capability, and nowhere in this endeavor do we invoke biological or psychological plausibility.

That is, this is not a course about

- cognitive/neurological/psychological modeling.

In fact, strictly speaking, this is not even a course about *science*, in the sense that we will not postulate models about nature (people, birds, insects) which we then go out and verify or refute by conducting physical experiments. Our goal is simply to achieve the desired capabilities in a computer, however we can achieve them (for now, let's assume a non-quantum computer).

Complementary to this viewpoint would be a course on "cognitive science" where one would investigate various models of human (and animal) intelligence, which would presumably then be testable in the world. We will not do that here. Disciplines usually represented in a cognitive science program are: psychology, linguistics, neuroscience, philosophy and computer science. (Waterloo, in fact, has a very interesting undergraduate program in cognitive science.) Clearly, cognitive science informs AI and vice versa, but I will not have time to teach much about cognitive science in this course. (Some excellent background references on cognitive science are listed below.)

Another thing this course is *not* about is

- novel software engineering paradigms.

From time to time, AI is presented to computer scientists as being primarily about creating new methodologies for developing complex software systems. Clearly, the resulting systems are often complex and push the limits of contemporary software engineering practice (in whatever time period). In fact, useful new programming languages and engineering strategies have indeed emerged along the way (*e.g.*, LISP and PROLOG, rule-based and knowledge-based systems). However, *every* area of CS pushes software engineering practice, all the time, and it is hard to argue that AI has been specially prolific in this regard. So rather than try to sell "new-age general-purpose programming remedies" I will instead focus on basic problems and how we can go about solving them using conventional CS methods.

A perspective I will adopt throughout this course is that:

– There is no magic bullet
– There is no free lunch
– There is no universal anything
(learning, intelligence, problem solver, you name it)
(and, yes, this includes human beings
[as much as it includes chimpanzees, dolphins, elephants, ants ...])
– General purpose X generally means generally weak

In reality, when we study AI, we are faced with hard problems, hard engineering, and few easy answers. But with that in mind, perhaps we can start to make progress. So let the work begin...

Textbook readings

Russell and Norvig, Chapters 1 and 2.
Dean, Allen and Aloimonos, Chapter 1.
Course notes on the Web (Intro)

Background readings on cognitive science

Pinker (97) *How the Mind Works.*
Gopnik, Meltzoff, Kuhl (99) *The Scientist in the Crib: What Early Learning Tells Us About the Mind.*
Pinker (94) *The Language Instinct: How the Mind Creates Language.*
Damasio (94) *Descartes's Error: Emotion, Reason and the Human Brain.*
Dehaene (97) *The Number Sense: How the Mind Creates Mathematics.*
Lakoff, Nuñez (00) *Where Mathematics Comes From: How the Embodied Mind Brings Mathematics into Being.*

Course outline

	Introduction	
	Lecture 1	Introduction to AI
Part 1	**Reasoning**	
	Lecture 2	Automating reasoning: formal inference
	Lecture 3	Correct and exhaustive reasoning
	Lecture 4	Constraint satisfaction search
	Lecture 5	Problem solving search
	Lecture 6	Automated planning
	Lecture 7	Planning algorithms
	Lecture 8	General first order representation
	Lecture 9	Planning in logic, First order inference
Part 2	**Knowing**	
		(Read Russell & Norvig, Chap. 8, Building a Knowledge Base)
Part 3	**Interpreting**	
	Lecture 10	Automating interpretation systems
	Lecture 11	Probability modelling
	Lecture 12	Structured probability models
	Lecture 13	Efficient probabilistic inference
	Lecture 14	Inference in complex models
	Lecture 15	Interpreting senses (perception)
	Lecture 16	Interpreting natural language
Part 4	**Behaving**	
	Lecture 17	Optimal behavior: Decision theory
	Lecture 18	Optimal sequential decision making
	Lecture 19	Optimal behavior: Game theory
	Lecture 20	Scaling up: Partial observability
		(Read Russell & Norvig, Chap. 25, Robotics and Control)
Part 5	**Learning**	
	Lecture 21	Types of learning problems
	Lecture 22	Function learning algorithms
	Lecture 23	Generalization theory / Overfitting
	Conclusion	
	Lecture 24	Course review

2 Automating reasoning: Formal inference

Related to course PMATH 330:Introduction to Mathematical Logic

Modelling *mathematical reasoning*

- Drawing *certain* conclusions from facts
- More facts $\rightarrow$ strictly more conclusions

(Note: not modelling plausible reasoning (yet)

- Drawing plausible conclusions from evidence
- More evidence $\rightarrow$ change conclusions)

First: Need a language to represent facts and conclusions

A simple first language: Language of propositions or Propositional language, or Propositional logic

Definition of a *proposition*:

- Primitive or atomic propositions $p, q, r, ...$, two special logical symbols: $\top$ and $\bot$
- Compound propositions
  - Logical symbols $\wedge, \vee, \neg, \rightarrow, \leftrightarrow, \bot, \top$
  - Composition: $\alpha \wedge \beta$, $\alpha \vee \beta$, $\neg\alpha$, $\alpha \rightarrow \beta$, $\alpha \leftrightarrow \beta$
    where α, β are propositions, either primitive or compound
  - parentheses, precedence: $\neg\ \wedge\ \vee\ \rightarrow \leftrightarrow$

Inference

Given a set of facts (propositions), what conclusions to draw? Let:
w = work, p = pass the exam, e = take exam, f = fail, u = understand, a = assignment

Given	Infer ?	
$\{w \to p,\ w\}$	p	yes
$\{e \to p \vee f,\ \neg f\}$	$e \to p$	yes
$\{\mathit{over5ft} \to \mathit{over6ft},\ \mathit{over6ft}\}$	$\mathit{over5ft}$	no
$\{w \to p,\ p\}$	w	no
$\{w \to p,\ \neg p\}$	$\neg w$	yes
$\{u \to (a \to p)\}$	$(u \to a) \to (u \to p)$	yes
$\{w \to p\}$	$(p \to g) \to (w \to g)$	yes
$\{p\}$	$\mathit{elvis\text{-}lives} \to p$	yes

Formal inference

Conclusions drawn depend only on *logical form* of propositions

Formal inference rules

from premises draw conclusions: premises $\vdash$ conclusion, or $\frac{\text{premises}}{\text{conclusion}}$
(Learn the difference between $\vdash$ and $\models$.)

E.g., Formal rule of inference: *Modus Ponens*

Given $\{\alpha,\ \alpha \to \beta\}$, infer β

(written $\{\alpha,\ \alpha \to \beta\} \vdash \beta$ or $\frac{\alpha,\ \alpha \to \beta}{\beta}$)

Formal inference rules
- are automatable
- "pattern match" rules that depend only on logical form
- antecedent variables match existing propositions
- consequent variables produces new propositions

Two components of mechanized reasoning

Inference rules	– encode domain independent rules of logical reasoning
Propositions	– encode domain specific facts

Derivation (proof)

Starting with a set of propositions $A = \{\alpha_1, ..., \alpha_n\}$, can add new propositions β to A by applying available rules of inference. If a proposition γ can be added to A after a finite number of rule applications, then we say that γ is *derivable* from A; denoted $A \vdash \gamma$. If no finite number of rule applications can add γ to A, then γ is not derivable from A; denoted $A \not\vdash \gamma$.

Note that the derivability relation $\vdash$ depends on which inference rules are available.

A derivation is a sequence: $p_1, p_2, \ldots, p_n$ such that for each $i \in \{1, \ldots, n\}$, p_i is derived using an inference rule, and a subset of $A \cup \{p_1, \ldots, p_{i-1}\}$ as premises.

E.g. application: automated question answering

Given domain facts $\{\alpha_1, ..., \alpha_n\} = A$, ask: is it the case that γ ?

If $A \vdash \gamma$ answer *yes*
If $A \vdash \neg\gamma$ answer *no*
If $A \not\vdash \gamma$ and $A \not\vdash \neg\gamma$ answer *I don't know*

E.g. car domain (l=lights on, b=battery ok, r=radio works, f=fuse ok)

Given $\{lights\_on \rightarrow battery\_ok,\ battery\_ok \rightarrow radio\_works,\ lights\_on\}$
is it the case that *radio_works* ?
is it the case that $\neg$*radio_works* ?

Given
$\{lights\_on \rightarrow battery\_ok,\ battery\_ok \wedge fuse\_ok \rightarrow radio\_works,\ lights\_on\}$
is it the case that *radio_works* ?

Given
$\{lights\_on \rightarrow battery\_ok,\ battery\_ok \wedge fuse\_ok \rightarrow radio\_works,\ lights\_on,\ fuse\_ok\}$
is it the case that *radio_works* ?

Given
$\{lights\_on \rightarrow battery\_ok,\ battery\_ok \wedge fuse\_ok \rightarrow radio\_works,\ lights\_on,\ \neg radio\_works\}$
is it the case that $\neg fuse\_ok$?

Given
$\{lights\_on \rightarrow battery\_ok,\ battery\_ok \wedge fuse\_ok \leftrightarrow radio\_works,\ lights\_on,\ \neg radio\_works\}$
is it the case that $\neg fuse\_ok$?

Is Modus Ponens adequate?

$\{a,\ a \rightarrow b\} \quad \vdash \quad b$

No! Cannot derive any of the following

$\{\neg b,\ a \rightarrow b\}$	$\vdash$	$\neg a$?	*Modus Tollens* $\dfrac{\alpha \rightarrow \beta,\ \neg\beta}{\neg\alpha}$
$\{a \wedge b \rightarrow c,\ a,\ b\}$	$\vdash$	c ?	*And Introduction* $\dfrac{\alpha,\ \beta}{\alpha \wedge \beta}$
$\{a \vee b \rightarrow c,\ a\}$	$\vdash$	c ?	*Or Introduction* $\dfrac{\alpha}{\alpha \vee \beta}$
$\{a \rightarrow b,\ \neg a \rightarrow c,\ b \rightarrow d,\ c \rightarrow d\}$	$\vdash$	d ?	*Reasoning by cases* $\dfrac{\alpha \rightarrow \beta,\ \neg\alpha \rightarrow \beta}{\beta}$
$\{\neg\neg a\}$	$\vdash$	a ?	*Double Negation* $\dfrac{\neg\neg\alpha}{\alpha}$

Formal inference *system*

Set of inference rules

(plus, possibly, a restriction on the language)

Examples of formal inference systems

E.g. 1: Modus Ponens

E.g. 2: Resolution

– Assumes propositions are in *implicative normal form*:

$p_1 \wedge p_2 \wedge \cdots \wedge p_k \to q_1 \vee q_2 \vee \cdots \vee q_\ell$

- Single rule of inference: Resolution rule
  $$\frac{\alpha \wedge p \to \beta,\ \gamma \to p \vee \delta}{\alpha \wedge \gamma \to \beta \vee \delta}$$
- Generalizes Modus Ponens
  $$\frac{p \to \beta,\ \top \to p}{\top \to \beta}$$
- CNF and DNF — conjunctive (big conjunct) and disjunctive normal form (big disjunct)
- *Strict implicative normal form*:
  - Let γ be a nonempty conjunction of primitive propositions (or $\top$)
  - Let δ be a nonempty disjunction of primitive propositions (or $\perp$)
  - Legal forms
    $\top \to \perp$
    $\top \to \delta$
    $\gamma \to \perp$
    $\gamma \to \delta$
  - Simplification rules
    $$\frac{\top \wedge \alpha \to \beta}{\alpha \to \beta} \quad \frac{\alpha \to \beta \vee \perp}{\alpha \to \beta} \quad \frac{p \wedge p \wedge \alpha \to \beta}{p \wedge \alpha \to \beta} \quad \frac{\alpha \to \beta \vee q \vee q}{\alpha \to \beta \vee q}$$
- Can reason by cases:
  E.g., Given $\{p \to q,\ \top \to p \vee r,\ q \to s,\ r \to s\}$, can derive $\top \to s$.

  $p \to q$ $\quad$ $\top \to p \vee r$

  $\top \to q \vee r$ $\quad$ $q \to s$

  $\top \to s \vee r$ $\quad$ $r \to s$

  $\top \to s \vee s$

  $\top \to s$
- However, still missing some "reasonable" inferences?
  e.g., $\{\} \not\vdash p \to p$ under resolution

E.g. **3: Natural deduction system**

Restrict propositions to any form using $\wedge, \vee, \rightarrow, \neg, \top, \bot$.

And elim $\dfrac{\alpha \wedge \beta}{\alpha,\ \beta}$ *And intro* $\dfrac{\alpha,\ \beta}{\alpha \wedge \beta}$

Or intro $\dfrac{\alpha}{\alpha \vee \beta}$

Impl elim $\dfrac{\alpha,\ \alpha \rightarrow \beta}{\beta}$ *Impl intro* $\dfrac{\text{If } A \cup \{\alpha\} \vdash \beta}{\alpha \rightarrow \beta}$

Reductio $\dfrac{\text{If } A \cup \{\alpha\} \vdash \bot}{\neg\alpha}$

Cases $\dfrac{\text{If } A \cup \{\alpha\} \vdash \beta \text{ and } A \cup \{\neg\alpha\} \vdash \beta}{\beta}$

Tautology $\dfrac{\alpha \vee \neg\alpha}{\top}$

Contradiction $\dfrac{\alpha,\ \neg\alpha}{\bot}$

E.g., given $\{p \rightarrow q,\ \neg p \rightarrow r,\ q \rightarrow s,\ r \rightarrow s\}$ can derive s.

1	$p \rightarrow q$		
2	$\neg p \rightarrow r$		
3	$q \rightarrow s$		
4	$r \rightarrow s$		
5.0		Assume p	
5.1		q	by Impl elim on 1 and 5.0
5.2		s	by Impl elim on 3 and 5.1
5	$p \rightarrow s$		by Impl intro
6.0		Assume $\neg p$	
6.1		r	by Impl elim on 2 and 6.0
6.2		s	by Impl elim on 4 and 6.1
6	$\neg p \rightarrow s$		by Impl intro

$7a.0$	Assume p	
$7a.1$	s	by Impl elim on 5 and $7a.0$
$7b.0$	Assume $\neg p$	
$7b.1$	s	by Impl elim on 6 and $7b.0$
7 s		by Cases

E.g., given $\{\}$ can derive $p \to p$

1.0	Assume p	
1.1	p	
1 $p \to p$		by Impl intro on 1.0 and 1.1

Characterizing inference systems

For a given inference system:

- Take a given set of propositions $A = \{\alpha_1, ..., \alpha_n\}$ and consider applying all available inference rules to A repeatedly:
- Get a monotonically growing set
  (Note: inference rules do not block each other, can always add conclusions in any order)

A set A is *closed* if no available inference rule can introduce any new propositions to A.
Closure

- The closure of a set A, $close(A)$, is called the *theory* of A.
- Monotonicity: $A \subset B$ implies that $close(A) \subset close(B)$
  (That is, adding new facts and new rules will only strictly increase the theory.)
- Monotonicity gives modularity: It is clear how new facts affect the theory. You never lose old conclusions. (This is a special feature of *logical* reasoning as opposed to *plausible* reasoning, which usually doesn't obey monotonicity.)

A proposition γ is called a *tautology* if $\{\} \vdash \gamma$. Such a γ is contained in every closure.
A set of propositions A is said to contain a *contradiction* if A contains any of $\bot$, $\top \to\bot$, or both α and $\neg\alpha$ for some α.
exhaustive, consistent system

Computational complexity and search

Sometimes, even give that form of logical reasoning can be automated in principle, it can still be computationally hard to reach the desired conclusions. A surprising example of this is trying to prove the *"pigeonhole principle"* (that $N+1$ pigeons cannot be placed solitarily in N pigeonholes) using resolution:

E.g., 3 pigeons, 2 holes

		pigeons		
		A	B	C
holes	1	$A1$	$B1$	$C1$
	2	$A2$	$B2$	$C2$

Constraints:

$$\begin{array}{ccc} A1 \vee A2 & B1 \vee B2 & C1 \vee C2 \\ \neg(A1 \wedge B1) & \neg(A1 \wedge C1) & \neg(B1 \wedge C1) \\ \neg(A2 \wedge B2) & \neg(A2 \wedge C2) & \neg(B2 \wedge C2) \end{array}$$

Re-expressed in implicative normal form (e.g., $A1 \vee A2$ is same as $\top \rightarrow A1 \vee A2$):

$$\begin{array}{ccc} A1 \vee A2 & B1 \vee B2 & C1 \vee C2 \\ A1 \wedge B1 \rightarrow \perp & A1 \wedge C1 \rightarrow \perp & B1 \wedge C1 \rightarrow \perp \\ A2 \wedge B2 \rightarrow \perp & A2 \wedge C2 \rightarrow \perp & B2 \wedge C2 \rightarrow \perp \end{array}$$

Exercise: derive $\top \rightarrow \perp$ from these facts using resolution.
Hint: it can be done, but it is surprisingly hard!

Another system: Horn clauses (sentences)

$p_1 \wedge p_2 \ldots \wedge p_n \rightarrow q$

Modus ponens is sufficient. Efficient closure procedure: apply Modus ponens until no new clauses can be added.

Readings

Burris, Chapter 1 and 2.
Dean, Allen, Aloimonos, Chapter 3.
Russell and Norvig, Section 6.4.
Course notes on the web (Logic 1)

3 Correct and exhaustive reasoning

Is an inference rule "correct"?
Is an inference system "exhaustive"?
Is an inference system "self consistent"?
What do the represented facts/conclusions "mean"?

Interpretation

Assign *values* to all *primitive* propositions
Specifies true "state of affairs"

E.g. truth theory

assign	$v(p) = t$	true
or	$v(p) = f$	false

for all primitive propositions p
(specifies complete state of affairs)

E.g. relevance theory

	$v(p) = \{t\}$	true
or	$v(p) = \{f\}$	false
or	$v(p) = \{t, f\}$	evidence for both
or	$v(p) = \{\}$	evidence for neither

E.g. 3-valued logic

	$v(p) = \{t\}$	true
or	$v(p) = \{f\}$	false
or	$v(p) = *$	undeterminable

All possible primitive assignments = all possible states of affairs

Interpreting compound propositions

Decompositional semantics (compositional semantics)

value of composite depends only on *values* of components
(value functional semantics)

E.g. for truth theory

$$v(\neg\alpha) = \begin{cases} t & \text{if } v(\alpha) = f \\ f & \text{if } v(\alpha) = t \end{cases}$$

$$v(\alpha \wedge \beta) = \begin{cases} t & \text{if } v(\alpha) = t \text{ and } v(\beta) = t \\ f & \text{otherwise} \end{cases}$$

$$v(\alpha \vee \beta) = \begin{cases} t & \text{if } v(\alpha) = t \text{ or } v(\beta) = t \\ f & \text{otherwise} \end{cases}$$

$$v(\alpha \to \beta) = \begin{cases} f & \text{if } v(\alpha) = t \text{ and } v(\beta) = f \\ t & \text{otherwise} \end{cases}$$

Recursively evaluate compound propositions
– bottoms out in values of primitive propositions

Given a complete assignment to primitive propositions
– can evaluate every compound proposition

Truth table:

α	β	$\neg\alpha$	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\alpha \to \beta$	$\alpha \leftrightarrow \beta$
t	t	f	t	t	t	t
t	f	f	f	t	f	f
f	t	t	f	t	t	f
f	f	t	f	f	t	t

What do propositions *mean* ?

A restriction on the state of possible affairs
Asserting α means that the state of affairs v makes α evaluate to t; i.e. $v(\alpha) = t$.
v is a state of affairs (truth assignment)

Terminology

- v *satisfies* α if $v(\alpha) = t$
- v *falsifies* α if $v(\alpha) = f$
- α is *satisfiable* if exists v such that $v(\alpha) = t$
- α is *falsifiable* if exists v such that $v(\alpha) = f$

- α is *unsatisfiable* (or *inconsistent*) if $v(\alpha) = f$ for all v
- α is *unfalsifiable* (or *valid*) if $v(\alpha) = t$ for all v
- α *entails* β if every v that makes α evaluate to t, makes β evaluate to t as well. Written $\alpha \models \beta$.

Do not confuse entailment with derivability!

Entailment $\alpha \models \beta$ and derivability $\alpha \vdash \beta$ are two completely different mathematical systems.
We now want to *relate* derivability and entailment
Assume truth value theory defines correctness
Want inference system to implement entailment

Correct inference

If $A \vdash \gamma$ then $A \models \gamma$ (also called *soundness*)

Exhaustive inference

If $A \models \gamma$ then $A \vdash \gamma$ (also called *completeness*)

Resolution rule is correct w.r.t. truth assignments

$\left(\dfrac{\alpha \wedge p \to \beta,\ \gamma \to p \vee \delta}{\alpha \wedge \gamma \to \beta \vee \delta}\right)$ Claim: $\{\alpha \wedge p \to \beta,\ \gamma \to p \vee \delta\} \models \alpha \wedge \gamma \to \beta \vee \delta$

Proof. One approach: construct a large truth table (32 rows). We use a shorter way:

Assume antecedent satisfied; that is, $v(\alpha \wedge p \to \beta) = t$ and $v(\gamma \to p \vee \delta) = t$. That means: $v(\neg\alpha \vee \neg p \vee \beta) = f$ and $v(\neg\gamma \vee p \vee \delta) = f$. There are two cases.

Case 1, if $v(p) = t$. $v(\neg\alpha \vee \beta) = f$, so $v(\neg\alpha \vee \beta \vee \neg\gamma \vee \delta) = t$. Then $v(\alpha \wedge \gamma \to \beta \vee \delta) = t$.

Case 2, if $v(p) = f$. From $v(\neg\gamma \vee p \vee \delta) = f$, we conclude $v(\neg\gamma \vee \delta) = f$, which implies $v(\neg\alpha \vee \beta \vee \neg\gamma \vee \delta) = t$, i.e., $v(\alpha \wedge \gamma \to \beta \vee \delta) = t$.

Therefore, in both cases $v(\alpha \wedge \gamma \rightarrow \beta \vee \delta) = t$. ■

Resolution rule is not exhaustive w.r.t. truth assignments

E.g. $\top \models p \rightarrow p$ (i.e. $p \rightarrow p$ is valid)
but $\top \not\vdash p \rightarrow p$ using resolution

Note: Natural deduction system *is* correct and exhaustive with respect to truth assignments. For a proof see: D. E. Cohen (1987) Computability and Logic. Ellis Horwood. (Chapter 11)

Resolution rule *is* exhaustive w.r.t. deriving contradictions (refutation exhaustive)

If A is unsatisfiable (inconsistent) then resolution can derive contradiction (that is $A \vdash \top \rightarrow \bot$)

Proof. Let $A = \{\alpha_1, ..., \alpha_n\}$ (α's in strict implicative normal form). Let K be the number of "excess" propositional letters. We define it in the following way: For any proposition α,

$$\text{excess}(\alpha) = \begin{cases} 0 & \text{if } \alpha \text{ has not propositional letters} \\ m-1 & \text{if } \alpha \text{ has } m \text{ propositional letters} \end{cases}$$

For example, $\text{excess}(\top \rightarrow \bot) = 0$, $\text{excess}(\top \rightarrow p) = 0$, $\text{excess}(p \rightarrow p) = 1$, and so on. For a set of formulae, the number of excess propositional letter is the sum of these numbers over all formulae, i.e.,

$$\text{excess}(A) = \sum_{i=1}^{n} \text{excess}(\alpha_i)$$

Proof is by induction on $K = \text{excess}(A)$.

<u>*Base case*</u>: $K = 0$. A contains rules of the form $\top \rightarrow \bot$, $\top \rightarrow p$, $p \rightarrow \bot$. (We can ignore other rules having the excess number of propositional letters 0, like $\bot \rightarrow \top$ and $p \rightarrow \top$, because they are valid (always true).)

Case 1: A contains $\top \rightarrow \bot$, done.

Case 2: A contains opposing pair $\top \to p$, $p \to \bot$. Then can derive $\top \to \bot$ by resolution rule, done.

Case 3: A contains no opposing pair nor $\top \to \bot$. Then one can satisfy all the rules in A by assigning $v(p) = t$ if $\top \to p$, $v(p) = f$ if $p \to \bot$. This makes every rule evaluate to t, which contradicts the assumption that A is unsatisfiable; done.

Induction hypothesis: Assume the theorem holds for sets of propositions that have 0, 1, ..., or $K - 1$ excess propositional letters, where $K > 0$.

Induction step: Assume A is unsatisfiable and has K excess propositional letters. Since $K > 0$ there must be at least one rule $\alpha \in A$, such that excess$(\alpha) > 0$. Without loss of generality, assume α is of the form $\beta \wedge p \to \gamma$ (the other case, $\beta \to \gamma \vee p$ is similar).

Let $\alpha' = \beta \to \gamma$. Note that $\alpha' \models \alpha$ and therefore $A - \{\alpha\} \cup \{\alpha'\} \models A$; hence $A - \{\alpha\} \cup \{\alpha'\}$ is unsatisfiable (since A was assumed to be unsatisfiable); hence $A - \{\alpha\} \cup \{\alpha'\} \vdash \top \to \bot$ by the induction hypothesis. Now consider two cases.

Case 1: $A - \{\alpha\} \vdash \top \to \bot$, done.

Case 2: $A - \{\alpha\} \nvdash \top \to \bot$. But here we know that $A - \{\alpha\} \cup \{\alpha'\} \vdash \top \to \bot$ by the induction hypothesis. Consider a derivation of $\top \to \bot$ from $A - \{\alpha\} \cup \{\alpha'\}$. Since the only difference between A and $A - \{\alpha\} \cup \{\alpha'\}$ is that the rule $\alpha = \beta \wedge p \to \gamma$ has been changed to $\alpha' = \beta \to \gamma$ we must have that $A \vdash p \to \bot$. (Just take the derivation of $\top \to \bot$ from $A - \{\alpha\} \cup \{\alpha'\}$ and add a 'p' to the left hand side of every resolution step that involves the rule $\alpha' = \beta \to \gamma$.)

Now note that $p \to \bot \models \alpha$ and therefore $A - \{\alpha\} \cup \{p \to \bot\} \models A$; hence $A - \{\alpha\} \cup \{p \to \bot\}$ is unsatisfiable; hence $A - \{\alpha\} \cup \{p \to \bot\} \vdash \top \to \bot$ by the induction hypothesis. (Note that α had at least two propositional letters and $p \to \bot$ only has one, so we have reduced the number of excess propositional letters.)

To finish Case 2, notice that the last two paragraphs have established respectively that $A \vdash p \to \bot$ and $A \cup \{p \to \bot\} \vdash \top \to \bot$, which allows us to conclude $A \vdash \top \to \bot$, done. ∎

Readings

Burris, Chapter 2.
Dean, Allen, Aloimonos, Chapter 3.
Russell and Norvig, Chapters 6 and 9*.
Genesereth and Nilsson, Chapters 3* and 4*.
* - ignoring material on first order variables and quantification

4 Constraint satisfaction search

(or Constraint Satisfaction Problem (CSP))

Applications of propositional logic: automated reasoning about simple facts

Question answering

How to answer $A \models \gamma$?

Could implement with resolution:

$$\begin{array}{lll} A \models \gamma & \text{iff} & A \cup \{\neg\gamma\} \text{ unsatisfiable} \\ & \text{iff} & A \cup \{\neg\gamma\} \vdash \top \rightarrow \bot \text{ using resolution} \end{array}$$

Could also implement with evaluations:

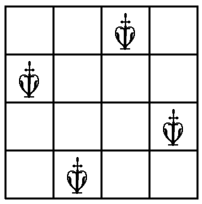
Search for a satisfying assignment for $A \cup \{\neg\gamma\}$

If none found, assert $A \models \gamma$

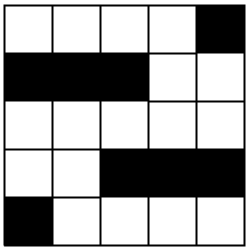
Can represent constraint satisfaction problems

Examples

1. Pigeonhole principle
2. N-queens problem



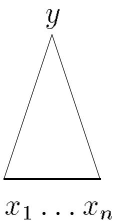
3. Designing crossword puzzles



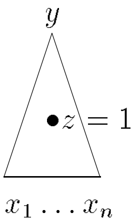
Given dictionary: aardvark
abdomen
⋮
zebra
zygote

4. Circuit testing

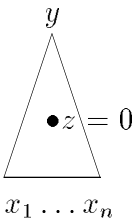
Boolean circuit ≡ propositional formula



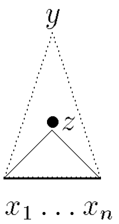
$F_{\mathbf{x}\to y}(\mathbf{x})$ original circuit



$F_{\mathbf{x}\to y|z=1}(\mathbf{x})$ z stuck at 1



$F_{\mathbf{x}\to y|z=0}(\mathbf{x})$ z stuck at 0



$F_{\mathbf{x}\to z}(\mathbf{x})$ internal circuit to z

Can use satisfiability search to design test inputs to identify faults:

– To test if z is stuck at 0

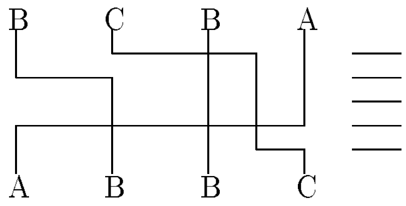
Need $\mathbf{x}$ such that $F_{\mathbf{x}\to z}(\mathbf{x}) = 1$ and $F_{\mathbf{x}\to y|z=1}(\mathbf{x}) \neq F_{\mathbf{x}\to y|z=0}(\mathbf{x})$
so find assignment $\mathbf{x}$ that satisfies proposition $F_{\mathbf{x}\to z} \wedge (F_{\mathbf{x}\to y|z=1} \oplus F_{\mathbf{x}\to y|z=0})$

– To test if z is stuck at 1

Need $\mathbf{x}$ such that $F_{\mathbf{x}\to z}(\mathbf{x}) = 0$ and $F_{\mathbf{x}\to y|z=1}(\mathbf{x}) \neq F_{\mathbf{x}\to y|z=0}(\mathbf{x})$
so find $\mathbf{x}$ that satisfies proposition $\neg F_{\mathbf{x}\to z} \wedge (F_{\mathbf{x}\to y|z=1} \oplus F_{\mathbf{x}\to y|z=0})$

5. Channel routing (VLSI design)

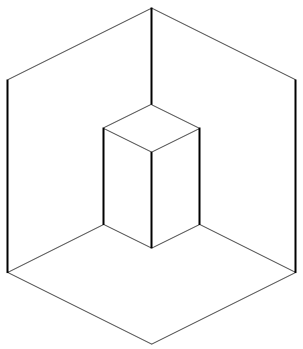
Connect labelled pins



Can only go along fixed set of channels

Must make perpendicular crossings

6. Polyhedral scene interpretation (R&N Sect 24.4)



label junction types (e.g. "innies" vs. "outies")

Implementing propositional reasoning

Search
– search space of resolution derivations
– search space of truth value assignments to primitive propositions

Constraint satisfaction search

Searching a finite product space

$$\begin{array}{llll} \text{variables} & p_1 & \dots & p_n \\ \text{values} & v_{11} & \dots & v_{n1} \\ & \vdots & & \vdots \\ & v_{1k_1} & \dots & v_{nk_n} \end{array}$$

Given a set of constraints $\alpha_1, \dots, \alpha_k$

Looking for assignment $p_1 = v_1 \dots p_n = v_n$ that satisfies all of the constraints

Systematic strategy 1: Enumerate assignments

Dumb

Exponential time 2^n

But *complete*
– if satisfying assignment exists, guaranteed to find it
– if none exists, then proves non-existence

Constraint satisfaction is NP-complete

Can we be clever about exponential time algorithms?

Systematic strategy 2: Backtrack search

Search partial assignments

$p_1 = v_1 \quad p_2 = v_2 \quad p_3 = ? \quad \dots \quad p_n = ?$

$\longrightarrow$

if any constraint becomes violated, backup and try alternative value

if all constraints satisfied, halt immediately

procedure <u>backtrack</u> $(p_j \ldots p_n)$
  for each value v_j of p_j
    $p_j := v_j$
    if no constraint violated
      <u>backtrack</u>$(p_{j+1} \ldots p_n)$
      if succeeds, **return** satisfying assignment
  if all fail, **return** fail

E.g. is $\{a \to b,\ b \to c,\ c \to \bot,\ a \to \bot\}$ satisfiable?

Enumerate	a	b	c	
	1	1	1	$\times$
	1	1	0	$\times$
	1	0	1	$\times$
	1	0	0	$\times$
	0	1	1	$\times$
	0	1	0	$\times$
	0	0	1	$\times$
	0	0	0	$\surd$

Backtrack	a	b	c	
	1	-	-	$\times$
	0	-	-	
	0	1	-	
	0	1	1	$\times$
	0	1	0	$\times$
	0	0	-	
	0	0	0	$\surd$

Systematic strategy 3: Constraint propagation: forward checking

Every time a variable is assigned, eliminate values from forward variables if possible

E.g. $\{a \to b,\ b \to c,\ c \to \bot,\ a \to \bot\}$

a	b	c		
1	1	1	(b, c forced)	$\times$
0	-	-		
0	1	1	(c forced)	$\times$
0	0	-		
0	0	0	$\surd$	

Systematic search improvements: Ordering heuristics

- Variable in most constraints
- Consistent variable/value assignments that do not affect other variables
- Value assignment that leaves constraints as unrestricting as possible

Good variable/value selection heuristics can lead to tremendous speedups

Unsystematic search

Unsystematic strategy 1: Random search

Global random: sample independent random assignments
Local random: start with a random assignment, then follow random walk by flipping single variable value

Actually works well if there is a large proportion of satisfying assignments!

But never halts if a solution does not exist

Unsystematic strategy 2: Heuristic local search

Use a heuristic like minimizing the # of violated constraints

Greedy descent:

```
while solution not found and not bored
    take random assignment
    while solution not found and walk length bound not exceeded
        evaluate neighboring assignments under heuristic
        step to best neighbor
```

Unsystematic strategy 3: Simulated annealing

Use a heuristic h

Annealing search:
    take random assignment
    while solution not found and not bored
        pick random neighbor
        if neighbor better than current assignment under h, move to neighbor
        else move to neighbor with probability $e^{(h(\text{curr})-h(\text{neigh}))/\text{temp}}$

where temp is a parameter that controls randomness vs. greediness

(high temp yields random search, zero temp avoids worse neighbors)

Unsystematic vs. complete search

Although unsystematic search runs forever if a solution doesn't exist, it can be astonishingly fast if a solution exists.

E.g. Hard random 3-SAT problems

4.3 times as many constraints as variables

Each constraint involves 3 variables

# variables	heuristic greedy	systematic
50	0.4 sec	1.4 sec
70	0.9 sec	15 sec
100	6 sec	2.8 min
120	14 sec	18 min
140	14 sec	4.7 hrs
$\vdots$		
500	1.6 hrs	–

Readings

Russell and Norvig, Sections 3.3, 3.7, 4.2, 4.4.
Dean, Allen, Aloimonos, Section 4.4.

Mitchell, Selman and Levesque, Hard and easy distributions of SAT problems, *Proceedings AAAI-92.*

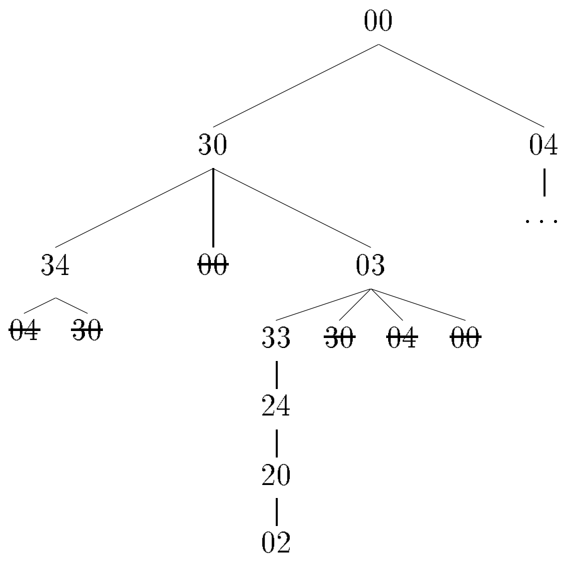
Selman, Levesque and Mitchell, A new method for solving hard satisfiability problems, *Proceedings AAAI-92.*

5 Problem solving search

E.g. Water jugs problems

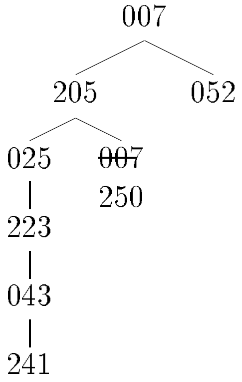
Problem 1:
Given a 3 gallon jug and a 4 gallon jug. Goal: achieve 2 gallons in the 4 gallon jug. The jugs are initially empty. Water can be poured from one jug to another, taken from outside source, or spilled out.

Solution: We can manually search for the solution using a tree:



Problem 2:
Given 2, 5, and 7 gallon jugs, achieve 1 gallon, don't spill water, and there is no external source of water. Initially, the 7 gallon jug is full, and two others are empty.

Solution: (Ignore repeated states, 007 is crossed out for the sake of illustration.)



State space search problem

Search space is generated by objects and operators on objects (implicit graph).

Given:

- initial state s_0
- set of possible actions for each state s: $a_1, \ldots, a_k$; $\quad a_i : s \mapsto s'$
- goal test

The goal is to find a sequence of actions that transforms initial state into a goal state.

Generalized CSP

- action = assign value to an unassigned variable
- initial state = empty assignment
- goal test = conjunction of constraints

But, now solutions may not have a bounded length.

E.g. Missionaries and cannibals

3 missionaries and 3 cannibals want to cross a river using a boat that holds 2 people. Cannibals should never outnumber missionaries. Empty boat cannot cross the river.

Initial state: MMMCCCB|
Goal state: |BMMMCCC

Action: take 1 or 2 people across the river

Solution:

MMMCCCB|

MMCC|BMC   MMMC|BCC

MMMCCB|C   MMMCB|C

and so on ...

Other examples

- towers of Hanoi
- Rubik's cube
- 8 puzzle

Uninformed Search Algorithms

(also called *Blind search algorithms*)

Automated problem solving search is graph search:
states = vertices
actions = labeled edges

We can apply general graph search strategies: DFS and BFS

Depth-First Search(DFS)

Algorithm 1 Depth-First Search(DFS)

```
1: list ← {s_0}
2: while list is not empty do
3:     s ← head(list)
4:     list ← rest(list)
5:     if s is a goal then
6:         return s
7:     else
8:         newstates ← apply actions to s
9:         list ← prepend(newstates, list)
10: return fail
```

Problem: graph may be infinite.

Breadth-first search (BFS)

Same as DFS, except:

```
9: list ← append(list, newstates)
```

DFS vs BFS

BFS:

- guaranteed to find solution (if one exists)
- not space efficient $|A|^{\text{solution depth}}$

DFS:

- space efficient
- not guaranteed

How to be space efficient and guaranteed?

Iterative deepening search (IDS)

Space efficient BFS.

Algorithm 2 Iterative deepening search (IDS)

```
1: for depth bound = 1, 2, ... do
2:     list ← {s_0}
3:     while list is not empty do
4:         s ← head(list)
5:         list ← rest(list)
6:         if s is a goal then
7:             return s
8:         else if depth(s) < depth bound then
9:             newstates ← apply actions to s
10:            list ← prepend(newstates, list)
```

(We have to keep depth of each node in the list recorded. In practice, the goal test (lines 6 and 7) can be done initially on s_0 and later only on newstates in line 9, when depth$(s) + 1 =$ depth bound, which could speed the algorithm up a bit.)

- same space as DFS
- guaranteed to find solution like BFS
- almost the same time as BFS: BFS running time $= A^d$, where A is the number of actions per state

$$\text{IDS running time} = 1 + A + A^2 + \ldots + A^d = \frac{A^{d+1} - 1}{A - 1} \approx A^d$$

Speedups

pruning:

- may determine that goal constraints are already violated
- don't revisit states!
  put states in a hash table; check if visited already

heuristics: - use heuristic function $\hat{h}(s)$ that estimates distance from s to goal

Informed Search Algorithms

(Also called *Heuristic search*)

We have some idea how close we are from the solution:
– heuristic function

Best-first search

Same as DFS, except that lines 3 and 4 are replaced by:
3: s ← extract_best_$\hat{h}$_value(list)

- allows big speedups: e.g., if $\hat{h}$ is perfect, walk straight to the goal
- problem: can be space inefficient

For DFS an appropriate data structure for list is stack, for BFS it is a FIFO queue, and for Best-first search it is a priority queue.

Finding shortest solution

Harder problem: find shortest solution: constrained optimization task

Definition: Algorithm that is guaranteed to find shortest solutions is called *admissible.*

BFS and IDS are admissible. DFS and Best-first are not admissible.

How to make Best-first admissible?

Let:

$g(s) =$ shortest distance from s_0 to s

$\hat{g}(s) =$ shortest distance from s_0 to s, that we know of (at certain moment during algorithm execution)

$h(s) =$ shortest distance from s to a goal

$\hat{h}(s) =$ our heuristic function, which approximates h

$d(s) =$ shortest distance from s_0 to a goal through s, i.e. $d(s) = g(s) + h(s)$

$\hat{d}(s) =$ our approximation of d, i.e., $\hat{d}(s) = \hat{g}(s) + \hat{h}(s)$

Admissible heuristic

is a heuristic function $\hat{h}(s)$ that underestimates $h(s)$: $\hat{h}(s) \leq h(s)$, and it does not lie about the goal, i.e., $\hat{h}(s) = 0$ iff s is goal, $\hat{h}(s) > 0$ otherwise.

It happens that such a heuristic is very useful. It is used in the A* search.

Admissible Best-first search (A*)

Uses an admissible heuristic, and it is same as Best-first search, except:

3: s $\leftarrow$ extract_min_$\hat{d}$_value(list)

Proof that A* finds shortest path

Proof. Let $s_1^*, \ldots, s_k^*$ be an optimal solution ($s_1^* = s_0$). Let us assume that the algorithm finds a non-optimal solution s_1, s_2, ..., s_l ($s_1 = s_0$), $l > k$.

The case $k = 1$ is trivial (s_0 is a solution), so we can assume $k > 1$. Before s_l is found, s_1^* is expanded and s_2^* is added to the list. (s_1^*, s_2^*) is a shortest path to s_2^*, so $\hat{g}(s_2^*) = g(s_2^*)$. If s_2^* is expanded before s_l is found, by the same argument, s_3^* is added to the list and $\hat{g}(s_3^*) = g(s_3^*)$, i.e., we know the length of a shortest path to s_3^*. We can repeat this argument for s_3^*, and so on.

However, we know that s_k^* is not expanded (otherwise, we would have found it before s_l), so the above reasoning has to stop, i.e., there has to be i, $1 < i \leq k$, such that s_{i-1}^* is expanded, s_i^* is not expanded, and $\hat{g}(s_i^*) = g(s_i^*)$. Since s_i^* is not expanded, it was already on the list when s_l was taken from the list to check the goal test. This means:

$$l = \hat{d}(s_l) \leq \hat{d}(s_i^*).$$

But, on the other hand:

$$\hat{d}(s_i^*) = \hat{g}(s_i^*) + \hat{h}(s_i^*) = g(s_i^*) + \hat{h}(s_i^*) \leq g(s_i^*) + h(s_i^*) = d(s_i^*) = k$$

Hence, we have $l \leq k$, which is a contradiction to the initial assumption $l > k$. ■

Generalized A*

Distance does not have to be expressed as the number of states that are expanded. We can associate a weight $w(a)$ (distance or cost) with each action a, and defined the distance as the sum of those weights along a path. ($w(a) = 1$ for the previous version of A*.) The problem is to find a shortest path in terms of this distance. The pseudo-algorithm is the same, and the optimality proof is almost the same, with a couple of details changed.

IDA*

Problem: A^* is space inefficient

IDA*: Iterative deepening on $\hat{d}$ bound.

- guaranteed
- admissible
- space efficient
- time efficient? (depends on $\hat{h}$)

Algorithm 3 Iterative deepening A* search (IDA*)

```
d̂_limit ← d̂(s_0)
while d̂_limit < ∞ do
    next_d̂_limit ← ∞
    list ← {s_0}
    while list is not empty do
        s ← head(list)
        list ← rest(list)
        if d̂(s) > d̂_limit then
            next_d̂_limit ← min(next_d̂_limit, d̂(s))
        else
            if s is a goal then
                return s
            newstates ← apply actions to s
            list ← prepend(newstates, list)
    d̂_limit ← next_d̂_limit
return fail
```

Incomplete search

- beam search
- genetic algorithm

Readings

Russell and Norvig: Sect. 3.4–3.6, 4.1–4.3
Dean, Allen, Aloimonos: Sect 4.1–4.3

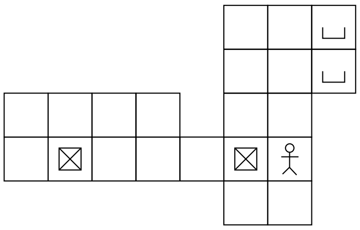
6 Automated planning

Automated planning is a complex form of problem solving:

– assume that states and actions have exploitable structure

Example 1: Sokoban (game)

Given a map with boxes, containers, and a person, as in the following snapshot:

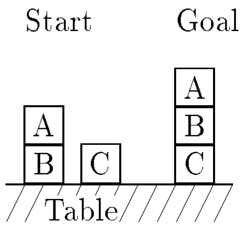


The actions are: up, down, left, and right, which move the person around.

– can go over containers
– can push one box from behind (and not more than one)
– goal: get all boxes into containers

This is a planning problem, and it is proven to be PSPACE complete.

Example 2: Blocks world



- actions: put box X on Y
- can move only one (top) box at the time
- can use table (large enough)
- initial state: full configuration
- goal: not necessarily full configuration, can be just partial description, e.g., A on B and C on D
- shortest plan is NP-hard

General propositional planning

PSPACE complete.

A kind of automated programming: choose a sequence of instructions, without conditionals, loops, recursion

State

State: Assignment of truth values to primitive propositions

We use as an example the blocks world:

We can choose primitive propositions to be: AonT (A on table), AonB, AonC, BonT, BonA, ...

If we use 1 to denote truth, and 0 to denote false, the above initial state can be represented as vector of 0s and 1s:

AonT	AonB	AonC	BonT	BonA	ConC	ConT	ConA	ConB
0	1	0	1	0	0	1	0	0

or

A T	A B	A C	B T	B A	C C	C T	C A	C B
0	1	0	1	0	0	1	0	0

Goal

The goal is a compound proposition γ; e.g., AonB $\wedge$ BonC

If γ is a conjunction of primitive propositions or negations of primitive propositions, it can be represented as a vector of 0s, 1s, and '*'s (don't cares).

For example the goal AonB $\wedge$ BonC can be represented as:

A T	A B	A C	B T	B A	C C	C T	C A	C B
$*$	1	$*$	$*$	$*$	1	$*$	$*$	$*$

Action

An action a is defined by a precondition π_a and a postcondition α_a. If we assume that π_a and α_a are conjunctions of primitive propositions or their negations, then preconditions and postconditions can be represented as vectors (patterns). So, an action can be represented as a pair of vectors.

For example, the action put A on table can be represented as:

	A T	A B	A C	B T	B A	C C	C T	C A	C B
precondition:	$*$	$*$	$*$	$*$	0	$*$	$*$	0	$*$
postcondition:	1	0	0	$*$	$*$	$*$	$*$	$*$	$*$

In order to apply an action, the preconditions have to be satisfied. The result of action is defined by the postcondition, and by the "frame assumption": Propositions that are not specified in the postcondition do not change their truth value.

Planning search

Any algorithm for general problem solving search can be applied (DFS, A*, etc). A state is expanded by applying any applicable actions to it; i.e., any actions for which the current state satisfies the precondition.

Forward search

In forward search we start with the initial state and apply the actions forward, i.e., we do *state progression.* Since the initial state is a complete configuration, the strongest postcondition is always obtained, i.e., the resulting state vector contains only 0s and 1s, i.e., it completely describes a configuration.

For example, if we use the initial state used previously, and apply action "put A on table" (also illustrated above), we have:

A	A	A	B	B	C	C	C	C
T	B	C	T	A	C	T	A	B
0	1	0	1	0	0	1	0	0
*	*	*	*	0	*	*	0	*
1	0	0	*	*	*	*	*	*
1	0	0	1	0	0	1	0	0

Backward search

In backward search, we start with a goal state and apply actions backward. The search is successfully finished when we achieve a sub-goal that matches the initial state. In this way, from a goal we obtain a new goal (which is hopefully easier to achieve). This is called *goal regression.*

In the blocks world, we apply "reversed" action to a vector which represents a set of goal configurations, and obtain a new sub-goal as the weakest precondition of the action. We can hope that search will be faster in this way, since a sub-goal typically encodes more than one complete configuration.

For example, the application of reversed action 'put A on B' on the goal $\gamma = \text{AonB} \wedge \text{BonC}$ can be represented as:

A	A	A	B	B	C	C	C	C
T	B	C	T	A	C	T	A	B
*	*	*	*	0	1	*	0	0
*	*	*	*	0	*	*	0	0
0	1	0	*	*	*	*	*	*
*	1	*	*	*	1	*	*	*

The new sub-goal is:

A	A	A	B	B	C	C	C	C
T	B	C	T	A	C	T	A	B
*	*	*	*	0	1	*	0	0

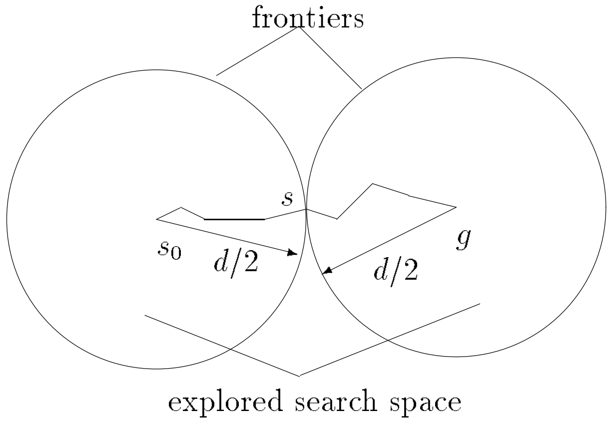
In order to see which approach, forward or backward search, is more promising, it is useful to analyze the branching factor. If the closest solution is at distance d from the initial state, and the branching factor is b, a typical search algorithm will spend $O(b^d)$ time to find the solution.

For blocks world, if there are k stacks in a configuration, then there are k^2 possible forward actions (we can remove the top block from any of the stacks and put it on one of the remaining $k-1$ stacks or on the table). For backward search, if we assume that a goal contains k positive atomic propositions and no negative propositions (vector of 1s and '*'s), then there are k actions that can be applied backward. For example, for goal AonB $\wedge$ ConD we can apply either 'put A on B' or 'put C on D' in backward direction. Hence, the branching factor is only k. This is not a strict analysis, but it does give us some intuition that the branching factor for backward search might be smaller than the branching factor for forward search, so backward search may be more efficient.

Bidirectional search

Forward and backward search can be done in parallel, which is called *bidirectional search.* In bidirectional search, two frontiers are expanded — one from the initial state, and the other from the goal state — until an intersecting state s of two frontiers is found. The final solution is constructed as the path $s_0 \to s \to g$, where s_0 is the initial state, and g is the goal state.

If we assume that the branching factor for forward search and for backward search is b, and the shortest distance to a solution is d, then both forward and backward search need $O(b^d)$ time to find the solution. However, the following figure illustrates that time needed for bidirectional search is $O(b^{d/2} + b^{d/2}) = O(b^{d/2})$ time.



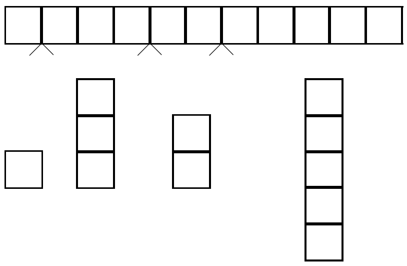
Hence, when the forward and backward branching factors are equal, bidirectional search may be a good approach.

A problem: In bidirectional search, at least one search frontier has to be in memory, so it consumes a lot of memory space.

Note: Number of states in blocks world

Assume there are N blocks. Let us first ignore block labels.

In order to count number of configurations having k stacks ($1 \leq k \leq N$), we have to choose $k-1$ out of $N-1$ places to "cut" a long stack of blocks into k stacks (as in the following picture):



We see that we can make $\binom{10}{3}$ choices to split 11 blocks into 4 stacks ($N = 11$, $k = 4$).

Stack permutations are not important, so the number of distinct unlabeled configurations of k stacks is $\frac{1}{k!}\binom{N-1}{k-1}$.

Now, we can add labels to blocks: there are $N!$ label permutations; so the size of the state space for the blocks world is:

$$N!\left[\binom{N-1}{0} + \frac{1}{2}\binom{N-1}{1} + \frac{1}{3!}\binom{N-1}{2} + \ldots + \frac{1}{N!}\binom{N-1}{N-1}\right]$$

One can also note that the upper bound on the shortest solution is $2N-2$: We can always flatten the configuration to stacks of height 1 (at most $N-1$ moves), and build any configuration we want in not more than additional $N-1$ moves. Hence, the size of the search tree is not larger than $(N^2)^{2N-2} = N^{4N-4}$.

Readings

Russell and Norvig: Chapter 11
Dean, Allen, Aloimonos: Chapter 7

7 Planning algorithms

Planning: Exploiting representation structure in problem solving search

Some approaches:

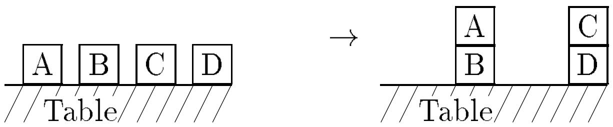
1. heuristics (examine representation)

   E.g., $\hat{h}(s)$ = Hamming distance from goal
   $\hat{g}(s)$ = Hamming distance from initial state

2. Approximate divide and conquer

   If actions only affect small parts of state, we can solve subgoal independently, and merge sub-plans.

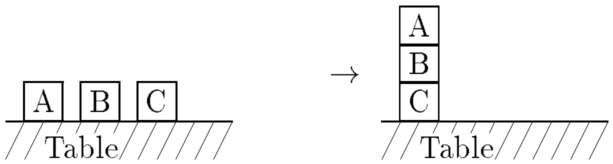
   E.g.

   We can solve the sub-goals 'A on B' and 'C on D' independently, and merged the resulting actions.

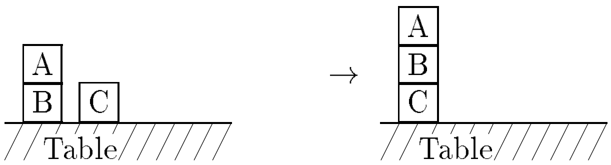
Problem: Sub-goals can interfere:

E.g.

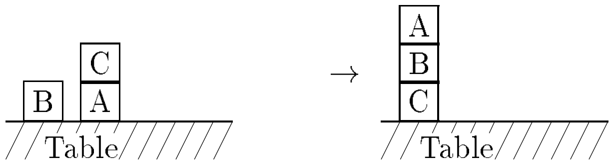


Getting A on B interferes with getting B on C.

We might even have to undo satisfied sub-goals, e.g.:

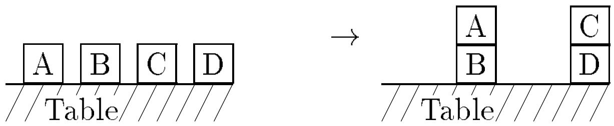


We may even have to avoid satisfying subgoals ("Sussman anomaly" due to Allen Brown):

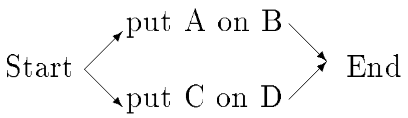


Partial order planning

For example:

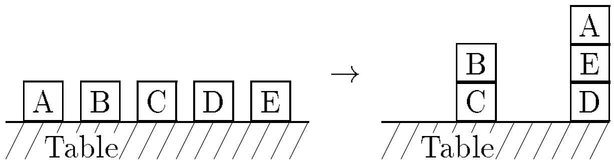


We can represent the plan as:

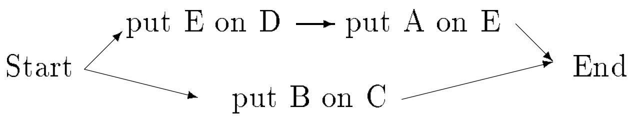


Any total ordering of the partial plan is a valid plan.

Second example:



A back-tracking algorithm may waste time back-tracking the action 'putBonC'. The partial ordering plan can be represented as:



Representing a partial order plan

- set of actions: $\{a_1, \ldots, a_k\}$
- set of ordering constraints between actions: $\{a_j \prec a_i\}$
- set of reasons for actions (links, causal links): $\{a_i \xrightarrow{l} a_j\}$
  a_i establishes l for a_j:
  - l is effect of a_i
  - l is precondition for a_j

Partial order planning

- start with artificial start and goal actions a_0 and a_∞ with effect of a_0 being s_0, and precondition of a_∞ being γ
- build a plan by adding actions where effects are desired preconditions: $a_i \xrightarrow{l} a_\infty$, where $l \in \gamma$; but add preconditions of a_i as new sub-goals.
- If action a_i *threatens* a link $a_1 \xrightarrow{l} a_2$, i.e., $\neg l$ is effect of a_i, then a_i must be ordered before a_1 or after a_2.
- "Least commitment planning"
  Do not commit to ordering until forced (avoids backtracking on bad decisions)

POP algorithm

Algorithm 4 POP_main

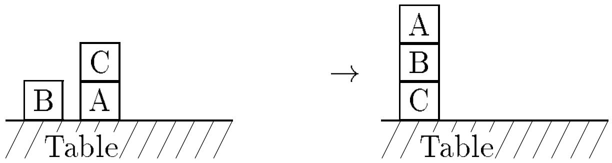
1: Create *start* and *end* actions a_0 and a_∞:
effect$(a_0) = s_0$ and precond$(a_\infty) = \gamma$
2: Initialize plan (actions, ordering constraints, links):
plan $\leftarrow (\{a_0, a_\infty\}, \{a_0 \prec a_\infty\}, \{\})$
3: sub-goal list $\leftarrow \{\gamma\}$
4: **return** POP(subgoal list, plan)

Algorithm 5 POP (subgoal list, plan)

1: **if** subgoal list is empty **then**
2:     **return** plan
3: Pick sub-goal l_{a_1} from sub-goal list
4: **for all** actions a_2 that establish l_{a_1} **do**
5:     plan' $\leftarrow$ plan $+ (\{a_2\}, \{a_0 \prec a_2, a_2 \prec a_1, a_2 \prec a_\infty\}, \{a_2 \xrightarrow{l_{a_1}} a_1\})$
6:     subgoal list' $\leftarrow$ subgoal list $\cup$ preconditions of a_2
7:     **for all** choices of additional order constraints in step 8 **do**
8:         for each action a threatening link $b \xrightarrow{l} c$ choose $a \prec b$ or $c \prec a$
9:         plan" $\leftarrow$ plan' + additional order constraints
10:         result $\leftarrow$ POP(subgoal list', plan")
11:         **if** POP successful **then**
12:             **return** result
13: **return** fail

- step 3 avoids backtracking (to some extent)
- with each sub-goal, have to keep track of the action requiring the sub-goal as precondition
- in step 4 we can choose an action from plan, or introduce a new action
- if there are no threats in steps 7–8, then loop 7–12 is iterated only once with an empty set of additional constraints.

Example: Sussman anomaly



Actions:

$$\text{start: } \frac{}{\text{AonT } \neg\text{AonB } \neg\text{AonC BonT } \neg\text{BonA } \neg\text{BonC } \neg\text{ConT ConA } \neg\text{ConB}}$$

$$\text{end: } \frac{\text{AonB BonC}}{}$$

$$\text{putConT: } \frac{\neg\text{AonC } \neg\text{BonC}}{\text{ConT } \neg\text{ConA } \neg\text{ConB}}$$

$$\text{putBonC: } \frac{\neg\text{AonB } \neg\text{ConB } \neg\text{AonC } \neg\text{BonC}}{\text{BonC } \neg\text{BonA } \neg\text{BonT}}$$

$$\text{putAonB: } \frac{\neg\text{AonB } \neg\text{ConB } \neg\text{BonA } \neg\text{ConA}}{\text{AonB } \neg\text{AonC } \neg\text{AonT}}$$

Algorithm:

$\boxed{\text{start}}$ Sub-goal list: AonB (end), BonC (end)

$\boxed{\text{end}}$

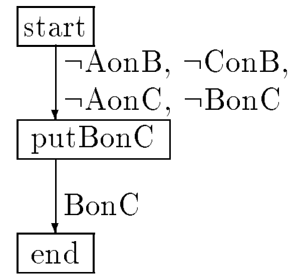
Pick sub-goal: BonC (end)

$\boxed{\text{start}}$

$$\boxed{\text{putBonC}} \xrightarrow{\text{BonC}} \boxed{\text{end}}$$

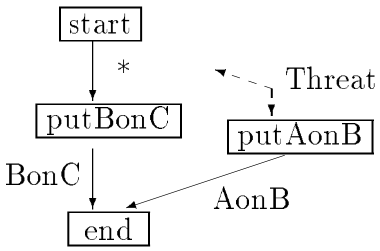
Sub-goal list: AonB (end), ¬AonB (putBonC), ¬ConB (putBonC), ¬AonC (putBonC), ¬BonC (putBonC)

For all sub-goals that are preconditions of putBonC, we can choose action start, and obtain:



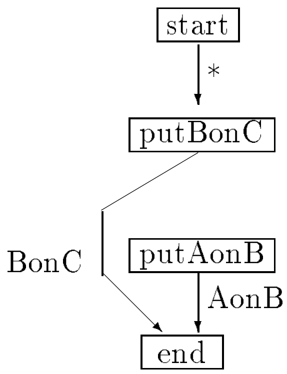
Sub-goal list: AonB (end)

Sub-goal AonB is picked, and action putAonB is chosen:



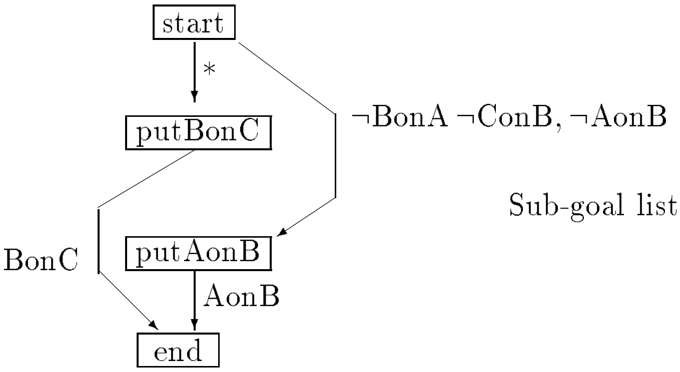
Sub-goal list: ¬BonA (putAonB), ¬ConA (putAonB), ¬AonB (putAonB), ¬ConB (putAonB),

There is a threat: action putAonB threatens the link start $\xrightarrow{\neg\mathrm{AonB}}$ putBonC. We have to put additional ordering constraints:



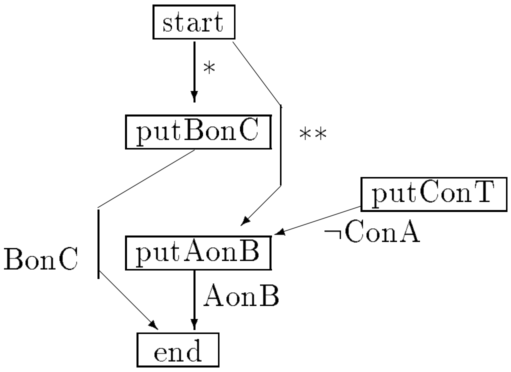
Sub-goal list: ¬BonA (putAonB), ¬ConA (putAonB), ¬AonB (putAonB), ¬ConB (putAonB),

All sub-goals except ¬ConA are post-conditions of the start action:



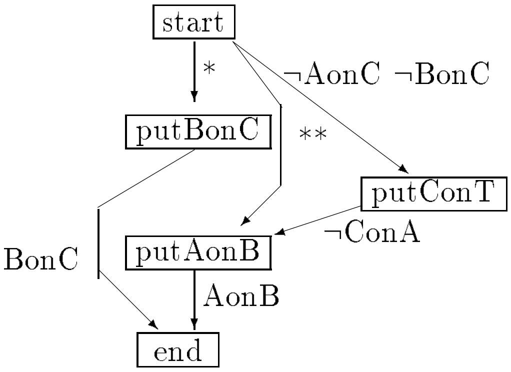
Sub-goal list: ¬ConA (putAonB)

We pick the subgoal ¬ConA and choose action putConT that has this post-condition:



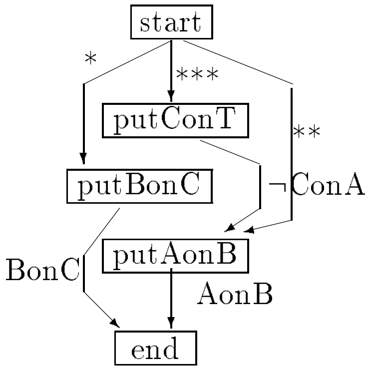
Sub-goal list: ¬AonC (putConT),
¬BonC (putConT)

The remaining sub-goals are post-conditions of the action start:



Sub-goal list:

The action putBonC threatens the link start $\xrightarrow{\neg \text{BonC}}$ putConT, so we have to reorder:



Sub-goal list:

Done! No backtracking!

Plain goal regression (backward search)

Let us see how the same problem could be solved with backward search:

end	AonB, $\underline{\text{BonC}}$
putBonC ↓ end	AonB, ¬AonB ¬ConB ¬AonC ¬BonC
	Stuck! (AonB and ¬AonB)
end	$\underline{\text{AonB}}$, BonC
putAonB ↓ end	BonC, ¬BonA ¬ConA ¬ConB $\underline{\neg\text{AonB}}$
start ↓ putAonB ↓ end	BonC ¬ConA

	Stuck!
putAonB $\downarrow$ end	<u>BonC</u>, $\neg$BonA $\neg$ConA $\neg$ConB $\neg$AonB
putBonC $\downarrow$ putAonB $\downarrow$ end	$\neg$ConA $\neg$ConB $\neg$AonB
	Pick start, stuck, backtrack
putConT $\downarrow$ putBonC $\downarrow$ putAonB $\downarrow$ end	$\neg$AonB
start $\downarrow$ putConT $\downarrow$ putBonC $\downarrow$ putAonB $\downarrow$ end	

Advantage of least commitment vs. plain backward search:

Smaller branching factor.

Backward search: branching factor = number of actions that can achieve some sub-goal

Least commitment: branching factor = number of actions that satisfy next sub-goal, does not backtrack through subgoal

Modern planning algorithms

- POP (1991)
- Graph Plan (1995)
- SAT plan (1996)
- forward search with heuristics (2000)

Readings

`ftp://ftp.cs.washington.edu/pub/ai/pi.ps` Weld, *AI Magazine 15(4)*

Also see:
`ftp://ftp.cs.washington.edu/pub/ai/pi2.ps` Recent advances in AI planning by Weld (`http://www.cs.washington.edu/homes/weld/pubs.html`) for a survey of more recent developments.

8 General first order representation

First-order predicate logic

Propositional core:

- **constants:** A, B, C, D
- **predicates:** $on(\cdot,\cdot)$
  - associated arity, e.g., arity of on is 2
- **atomic ground sentence** is equivalent to atomic proposition; it is created by applying predicate to constants; e.g., $on(A,B)$
- true and false symbols: $\top$, $\perp$
- **compound ground sentence:** built using operations: $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftrightarrow$; e.g., $on(A,B) \wedge \neg on(B,C)$

This is equivalent to propositional logic. For example, instead of writing $on(A,B)$, we used notation AonB. Instead of repeating the same rules for AonB, BonC, etc., we can use variables and use predicate $on(x,y)$:

- **variables:** x, y, z
- **atomic formula:** apply predicate on constants and variables; e.g., $on(A,x)$, $on(A,B)$, $on(x,y)$
- **compound formula:** use operators of propositional calculus; e.g., $on(A,X) \wedge \neg on(x,B)$
- special equality predicate =: instead of writing $=(x,B)$, write $x = B$
- **quantifiers** $\forall$ and $\exists$: If $\varphi(\ldots x \ldots)$ is a formula, then $\forall x \varphi(\ldots x \ldots)$ and $\exists x \varphi(\ldots x \ldots)$ are formulas.
  In such formula, x is called **bound** variable, and variables that are not bound are **free.**

Open formula is a formula in which all variables are free.

Closed formula is a formula with no free variables (all variables are bound).

Closed formula is also called a **sentence.**

Ground formulas (formulas with no variables) are sentences. If a formula contains free variables, we can add quantifiers and make it a sentence.

Sentences have truth values. (Open formulas do not.)

Further extension: functions

function symbols: $f(\cdot, \cdot)$

term: composition of functions, constants, and variables; e.g.

$$f(g(x), h(A, B))$$

Ground term is a term with no variables.

Ground terms can represent complex objects like strings, trees, expressions.

Open term contains variables (a term does not contain quantifiers)

Formal inference rules

Operate on sentences. Examples:

$\forall$ elimination:

$$\frac{\forall x \varphi(x)}{\varphi(t)} \quad \text{for any ground term } t$$

$\exists$ elimination:

$$\frac{\exists x \varphi(x)}{\varphi(A')} \quad \text{for a \textit{new} constant } A'$$

$\exists$ introduction:

$$\frac{\varphi(t)}{\exists x \varphi(x)} \quad t \text{ is a ground term}$$

A simple formal inference system

Three rules: resolution, substitution, simplification

Formulas represented in *implicative normal form.*

Implicative normal form:

$$\forall x_1 \ldots \forall x_n \, p_1(x) \wedge \ldots p_k(x) \rightarrow q_1(x) \vee \ldots \vee q_l(x)$$

Any formula can be transformed into implicative normal form:

- eliminate implications ($\rightarrow$) and equivalences ($\leftrightarrow$)
- move negations inwards
- standardize variables apart
- move $\forall$ and $\exists$ to left
- remove $\exists$ by "Skolemization":
  Go from outside and for each $\forall x_1 \forall x_2 \ldots \forall x_n \exists y$ substitute y with $f(x_1, x_2, \ldots, x_n)$, where f is a new function symbol (or constant symbol if $n = 0$)
- distribute $\wedge$ over $\vee$
- flatten, e.g., $(p \vee q) \vee r$ becomes $(p \vee q \vee r)$
- turn negations in conjuncts into implications, e.g., $(p \vee \neg q \vee r \vee \neg s)$ becomes $(q \wedge s \rightarrow p \vee r)$
- drop $\forall x$ and assume variables are universal

General conversion to implicative normal form can be found in Russell and Norvig p281–282.

Inference rules

Specialization (substitution):

$$\frac{\varphi}{[\varphi]_{x/t}}$$

replace x by term t

Resolution:

$$\frac{\alpha \rightarrow p(\underline{v}) \vee \beta \quad \gamma \wedge p(\underline{v}) \rightarrow \delta}{\alpha \wedge \gamma \rightarrow \beta \vee \delta}$$

Simplification:

$$\frac{p \wedge p \wedge \alpha \rightarrow \beta}{p \wedge \alpha \rightarrow \beta} \qquad \frac{\alpha \rightarrow \beta \vee p \vee p}{\alpha \rightarrow \beta \vee p}$$

(same as for propositional logic)

Example

1:	$on(x,y) \wedge on(y,z) \wedge on(z,x) \rightarrow \bot$	
2:	$\top \rightarrow on(A,A) \vee on(B,B)$	
3:	$on(w,w) \wedge on(w,w) \wedge (w,w) \rightarrow \bot$	apply substitution x/w y/w z/w on 1
4:	$on(w,w) \rightarrow \bot$	apply simplification on 3
5:	$on(A,A) \rightarrow \bot$	substitution w/A on 4
6:	$\top \rightarrow \bot \vee on(B,B)$	resolution on 2 and 5
7:	$on(B,B) \rightarrow \bot$	substitution w/B on 4
8:	$\top \rightarrow \bot \vee \bot$	resolution on 6 and 7
9:	$\top \rightarrow \bot$	simplification on 8
	Contradiction!	

Unification

Unification is matching formulas by substitution (specialization)

Example

1:	$\forall x P(A,x)$	
2:	$\forall y \forall z P(y,z) \rightarrow P(z,y)$	
3:	$\forall z P(A,z)$	substitution x/z on 1
4:	$\forall z P(A,z) \rightarrow P(z,A)$	substitution y/A on 2
	Two sub-formulas are unified, and we can apply resolution:	
5:	$\forall z P(z,A)$	

It is useful to have procedure UNIF(φ, χ) which returns either:

- a substitution (binding list) $s = x_1/y_1 \dots x_n/y_n$ such that $[\varphi]_s = [\chi]_s$, or
- fail, if none exists

s is a unifier of φ and χ

Example

$$
\begin{aligned}
\mathrm{UNIF}(\mathit{on}(A,x), \mathit{on}(A,B)) &= x/B \\
\mathrm{UNIF}(\mathit{on}(A,x), \mathit{on}(y,B)) &= x/B\ y/A \\
\mathrm{UNIF}(\mathit{on}(A,x), \mathit{on}(y,f(y))) &= y/A\ x/f(A) \\
\mathrm{UNIF}(\mathit{on}(x,y), \mathit{on}(y,f(y))) &= \text{fail} \\
\mathrm{UNIF}(\mathit{on}(A,x), \mathit{on}(x,B)) &= \text{fail}(A \neq B)
\end{aligned}
$$

Most general unifier

There can be more than one substitution, e.g.: UNIF($\mathit{on}(A,x), \mathit{on}(y,z)$) could return $\{y/A, x/z\}$, but also $\{y/A, z/x\}$, or $\{y/A, x/B, z/B\}$, or $\{y/A, x/A, z/A\}$, ...

We are interested in the *most general unifier*:

$$\mathrm{MGU}(\mathit{on}(A,x), \mathit{on}(y,z)) = \{y/A, x/z\}$$

Most General Unifier:

- makes least commitments
- exists and is unique up to variable renaming (if the terms are unifiable)
- can be turned into any other unifier by applying additional substitutions

If we know how to do unification, we can merge substitution and resolution in one step:

Resolution with unification

$$\frac{\alpha \wedge p(\underline{v}) \rightarrow \beta \quad \gamma \rightarrow q(\underline{u}) \vee \delta}{[\alpha \wedge \gamma \rightarrow \beta \vee \delta]_{\mathrm{MGU}(p(\underline{v}),q(\underline{u}))}}$$

Unification Algorithm

- Robinson 1965 exponential algorithm
- Huet's 1976 almost linear algorithm $O(n\alpha(n))$
- Paterson and Wegman 1976 improved Huet's algorithm: linear algorithm

A couple more examples:

$$\mathrm{UNIFY}(f(x, g(y, y), x), f(z, z, g(w, f(t)))) = ?$$

Answer: $x/g(f(t), f(t))$ $z/g(f(t), f(t))$ $y/f(t)$ $w/f(t)$

$$\mathrm{UNIFY}(f(x, h(A, t), g(x)), f(g(y), h(z, y), t)) = ?$$

Answer: fail

Robinson's algorithm

Algorithm 6 UNIFY(t_1, t_2)

```
current substitution ← empty substitution
if t_1 or t_2 is a variable then
    if t_1 = t_2 then
        return (true, empty substitution)
    let t_1 be variable
    if t_1 occurs in t_2 then
        return false
    return (true, t_1/t_2)
else
    if terms t_1 and t_2 cannot be matched then
        return false
    t_1 = f(a_1, ..., a_n), t_1 = f(b_1, ..., b_n)
    for all pairs (a_i, b_i) (i = 1...n) do
        make current substitution on a_i and b_i and call UNIFY on them
        if UNIFY successful then
            apply returned substitution to current substitution
        else
            return false
return (true, current substitution)
```

Readings

Russell and Norvig: Chapters 7 and 9
Genesereth and Nilsson: Chapter 4
Burris: 3.12
Knight, Kevin: Unification: a multidisciplinary survey, *ACM computing surveys*, March 1989

9 Planning in logic, First order inference

Similarly to propositional calculus, we ask questions:
Is a rule correct?
Are the rules exhaustive?

Same strategy as propositional logic: Come up with an independent evaluation scheme.

$A \vdash \gamma$ means γ can be syntactically derived from A.

$A \models \gamma$ means γ is entailed (semantically) by A.

Correctness (soundness): $A \vdash \gamma$ implies $A \models \gamma$

Exhaustiveness (completeness): $A \models \gamma$ implies $A \vdash \gamma$

Interpretation

$I = (D, C, F, R)$

D is a set.

C: constant symbols $\rightarrow D$

F: functions symbols $\rightarrow (D^n \rightarrow D)$

R: predicate symbols $\rightarrow S \subseteq D^n$

Given an interpretation, we can interpret any ground term as an element of D. E.g.,

$I(C) = C(C)$

$I(f(A, B)) = F(f)(C(A), C(B))$, which is an element of D.

Atomic ground formula is interpreted as true or false, e.g.:

$I(on(f(A), B)) = R(on)(F(f)(C(A)), C(B))$, which is true or false depending on whether the pair of objects $(F(f)(C(A)), C(B))$ is in relation $R(on)$ or not.

Using the same rules as in propositional calculus, any ground sentence is interpreted as true or false.
E.g., $I(\alpha \wedge \beta) = \text{true}$ if and only if $I(\alpha) = \text{true}$ and $I(\beta) = \text{true}$.

Interpretation of quantified sentences

Variable assignment: V: variables $\to D$

Given a variable assignment we can interpret open formulas, because assigned variables are treated as constants; e.g.

$$I_V(p(t_1, \ldots, t_n)) = \text{true} \;\text{ iff }\; (I_V(t_1), \ldots, I_V(t_n)) \in R(P)$$

Using the notion of assignment, we can interpret quantified formulas:

$I_V(\forall x \varphi(v)) = \text{true}$	iff for all variable assignments U that agree with V except possibly on x, $I_U(\varphi) = \text{true}$
$I_V(\exists x \varphi(v)) = \text{true}$	iff for some variable assignment U that agrees with V except possibly on x $I_U(\varphi) = \text{true}$
$I(\varphi(v)) = \text{true}$	(where φ is sentence) iff for some (all) variable assignment V: $I_V(\varphi) = \text{true}$

Given an interpretation I, we can evaluate any sentence to truth or false.

Terminology defined in the same way as in propositional calculus (α and β are sentences):

- I *satisfies* α if $I(\alpha) = \text{true}$
- I *falsifies* α if $I(\alpha) = \text{false}$
- α is *satisfiable* if exists I such that $I(\alpha) = \text{true}$
- α is *falsifiable* if exists I such that $I(\alpha) = \text{false}$
- α is *unsatisfiable* (or *inconsistent*) if $I(\alpha) = \text{false}$ for all I
- α is *unfalsifiable* (or *valid*) if $I(\alpha) = \text{true}$ for all I
- α *entails* β if every I that makes α evaluate to true, makes β evaluate to true as well. Written $\alpha \models \beta$.

Correctness of resolution and specialization

Resolution is correct:

Resolution:

$$\frac{\alpha \to p(\underline{v}) \vee \beta \quad \gamma \wedge p(\underline{v}) \to \delta}{\alpha \wedge \gamma \to \beta \vee \delta}$$

Correctness:

$$\alpha \to p(\underline{v}) \vee \beta, \gamma \wedge p(\underline{v}) \to \delta \models \alpha \wedge \gamma \to \beta \vee \delta$$

Proof. Same at with propositions: Assume $I(p(\underline{v})) =$ true, or assume $I(p(\underline{v})) =$ false.

(We have to keep in mind that universal quantifier is assumed in implicative normal form.) ∎

Specialization is correct:

Specialization:

$$\frac{\alpha}{[\alpha]_{x/t}}$$

Correctness:

$$\alpha \models [\alpha]_{x/t}$$

Proof. Let us assume that $I(\alpha) =$ true. Hence, $I_V(\alpha) =$ true for all V. Let us show that $I_U([\alpha]_{x/t}) =$ true for all U.

Let U be arbitrary assignment, and let $d = I_U(t)$. Let V' be an assignment that agrees with U on all variables except x, and $I_{V'}(x) = d$. Then $I_{V'}(\alpha) = I_U([\alpha]_{x/t})$. By assumption, $I_{V'}(\alpha) =$ true, so $I_U([\alpha]_{x/t}) =$ true. ∎

Exhaustiveness w.r.t. deriving contradictions

Resolution, specialization, and simplification are exhaustive w.r.t. deriving contradictions

If A is unsatisfiable then $A \vdash \top \to \bot$

Proof. (sketch) Let A be a set of sentences in INF (these sentences are also called *rules*).

Herbrand Universe = constructible ground terms from constants and function symbols, mentioned in A

Herbrand base = set of all ground sentences constructible by substituting ground terms from Herbrand universe into A

Lemma: (Herbrand's theorem) If A is unsatisfiable, then $HB(A)$ is unsatisfiable.

Lemma: (Compactness theorem) If B is unsatisfiable, then exists a finite subset $B' \subseteq B$ such that B' is unsatisfiable.

Thus, A unsatisfiable, implies exists finite subset H of $HB(A)$ that is unsatisfiable.

$A \vdash H$ by substitution. Finally, H unsatisfiable implies $H \vdash \top \to \bot$ using resolution (using same argument as in propositional logic). ∎

Planning in logic

Planning can be encoded as a special case of first order inference.

Situation calculus can be used.

Situation calculus

(Russell and Norvig 7.6)

- set of situations: $s_1, s_2, \ldots$, or "states"
- set of actions: $a_1, a_2, \ldots$, which map situations to situations

Language

- predicate fluents: $on(x, y, s)$
- constants: A, B, C, T
- initial situation s_0; e.g.: $on(C, A, s_0) \wedge on(A, T, s_0) \wedge on(B, T, s_0)$
- goal: $\exists s\, on(A, B, s)$
- parametrized actions: terms denote action objects: $\mathrm{move}(x, y)$, $\mathrm{move}(A, B)$
- special "do" function: maps situations $\times$ actions $\rightarrow$ situations
  $\mathrm{do}(\mathrm{move}(C, T), s_0)$ is a term: name for the state s that results from applying action $\mathrm{move}(C, T)$ on s_0
- unique names assumption: $\mathrm{move}(A, B) \neq \mathrm{move}(C, D)$,
  distinct terms denote distinct objects
- action preconditions represented by formulas; e.g., precondition of $\mathrm{move}(x, y)$ is

$$\forall x \forall y \forall s\, [\forall w\, \neg on(w, x, s) \wedge (\neg on(w, y, s) \vee y = \mathrm{T})]$$

- action effects; e.g., effect of of $\mathrm{move}(x, y)$ is

$$on(x, y, \mathrm{do}(\mathrm{move}(x, y), s))$$

For example, action 'put x on y' (or $\mathrm{move}(x, y)$) can be described by the sentence:

$$\forall x \forall y \forall s\, [\forall w\, \neg on(w, x, s) \wedge (\neg on(w, y, s) \vee y = \mathrm{T})] \rightarrow on(x, y, \mathrm{do}(\mathrm{put}(x, y), s))$$

This is called an *effect axiom.* It describes action 'put' that puts a block x on block y, and in this way we can move from situation s to situation $\mathrm{do}(\mathrm{put}(x, y), s)$. Function 'do' maps an action and a situation into the successor situation.

After introducing such axioms: effect axioms (EA), and *unique name assumption* (UNA, axioms that state that distinct terms denote distinct objects (e.g., $\mathrm{put}(A, B) \neq \mathrm{put}(C, D)$)), planning can be described as the task:

$$\mathrm{Th}(s_0) \cup \mathrm{EA} \cup \mathrm{UNA} \vdash \exists s \gamma(s)$$

$\mathrm{Th}(s_0)$ is the description of the initial situation, and $\exists s \gamma(s)$ is the goal, e.g, $\exists s\, on(A, B, s) \wedge on(B, C, s)$. Using resolution, we can attempt to establish:

$$\mathrm{Th}(s_0) \cup \mathrm{EA} \cup \mathrm{UNA} \cap \forall s \neg\gamma(s) \vdash \top \rightarrow \bot$$

If successful, during the inference the term s will be replaced with a nested 'do' term, which represents the plan.

Frame problem: If an action does not change some facts, then they remain the same in the next situation.

How do we express this?

A naive solution: frame axioms.

Readings

Russell and Norvig: Section 7.6, Chapter 9
Genesereth and Nilsson: Sections 2.3, 4.10
Burris: Chapter 4 (especially 4.10)

10 Automating interpretation systems

Part 3: Interpreting

Introduction to interpretation systems

Interpreting: Plausible inference of hidden semantic structure from inputs.

Examples

$$\begin{array}{rcl} \text{word sequence} & \rightarrow & \text{meaning} \\ \text{pixel matrix} & \rightarrow & \text{object, relations} \\ \text{speech signal} & \rightarrow & \text{phonemes, words} \end{array}$$

Combine ambiguous, incomplete, contradicting evidence, and draw reasonable conclusions

Distinction from logical reasoning:

- plausible inference:
  - non-monotonic: might change conclusions given more evidence
  - uncertain: conclusions not guaranteed to be correct (but still want to do as well as possible)
- logical inference:
  - monotonic: once conclusion drawn never retracted
  - certain: conclusions certain given assumptions

How to build an interpretation system?

$$\text{observables} \rightarrow \boxed{\ ?\ } \rightarrow \text{hidden semantic structure (conclusion)}$$

There are two problems:

1. need to represent facts about process that connects evidence to truth

2. need principles of evidence combination

In this course: We use probability theory.

(Not other theories or languages as: fuzzy logic, default logic, rule-based systems, Dempster-Shafer theory)

Using probability theory

1. represent knowledge with probabilistic models: forward generative models:

$$\begin{array}{ccc} \downarrow & \qquad & \text{world} \\ \boxed{\text{truth}} & & \downarrow \\ \downarrow & & \text{sensor} \\ \boxed{\text{evidence}} & & \downarrow \\ & & \text{observed measurement} \end{array}$$

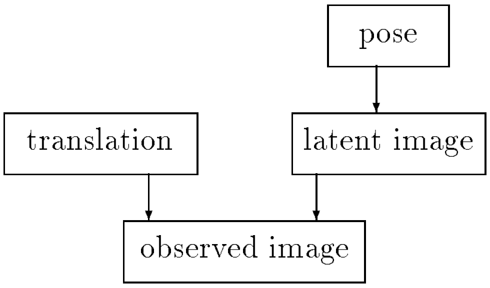
2. principle of evidence combination: Bayesian inference

$$\begin{aligned} \text{conclusion} &= \operatorname*{arg\ max}_{\text{possible truth}} P(\text{possible truth}|\text{evidence}) \\ &= \operatorname*{arg\ max}_{\text{possible truth}} \frac{P(\text{evidence}|\text{possible truth})P(\text{possible truth})}{P(\text{evidence})} \\ &= \operatorname*{arg\ max}_{\text{possible truth}} P(\text{evidence}|\text{possible truth})P(\text{possible truth}) \end{aligned}$$

Demos

Demo 1: Image normalization

Bayesian inference:



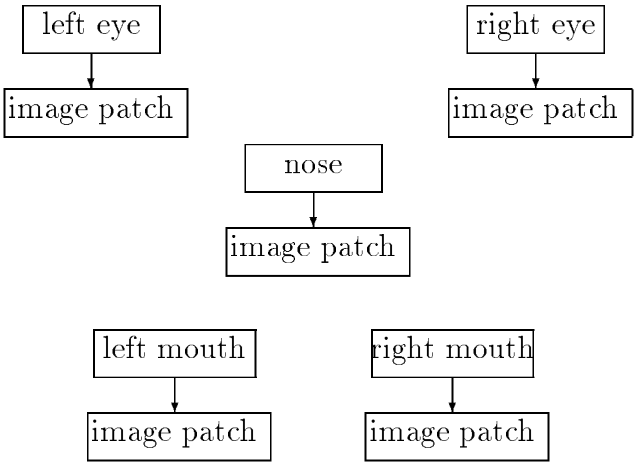
A time component is included to model image stabilization.

Demo 2: Independent object tracking

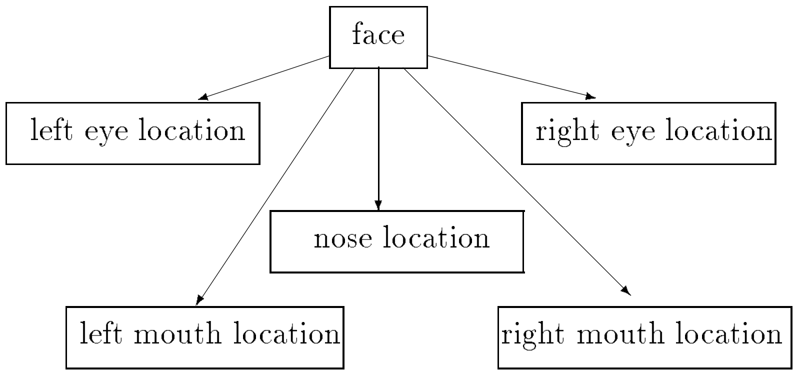
Demo 3: Independent object tracking and object removal

Demo 4: Face tracking

Two models combined: low level model:



High-level model:



Readings

Dean, Allen, Aloimonos: Section 3.7
Russell and Norvig: Section 15.6

11 Probability modelling

Random configuration

Imagine a bunch of "configurations" $\mathbf{x}^{(1)}, ..., \mathbf{x}^{(t)}$ drawn from some random source:

$$\begin{array}{rcl} \mathbf{x}^{(1)} & = & (x_{11} \quad x_{12} \quad \ldots \quad x_{1n}) \\ \mathbf{x}^{(2)} & = & (x_{21} \quad x_{22} \quad \ldots \quad x_{2n}) \\ & \vdots & \\ \mathbf{x}^{(t)} & = & (x_{t1} \quad x_{t2} \quad \ldots \quad x_{tn}) \end{array}$$

Here we assume a fixed number (n) of components in each configuration, and assume values x_{ij} are from a finite set, $x_{ij} \in \{1, 2, ..., V\}$.

Random variables

$$\mathbf{X} = X_1, X_2, ..., X_n$$

Joint distribution

$\mathrm{P}(X_1 = x_1, ..., X_n = x_n)$ gives probability of each complete configuration $\mathbf{x} = (x_1, ..., x_n)$. In general it takes V^n parameters (minus one constraint) to specify an arbitrary joint distribution on n random variables with V values. One could represent this by a lookup table $\theta_{\mathbf{x}^{(1)}}, \theta_{\mathbf{x}^{(2)}}, ..., \theta_{\mathbf{x}^{(V^n)}}$, where $\theta_{\mathbf{x}^{(\ell)}}$ gives the probability that the random variables jointly take on configuration $\mathbf{x}^{(\ell)}$. That is, $\theta_{\mathbf{x}^{(\ell)}} = \mathrm{P}(\mathbf{X} = \mathbf{x}^{(\ell)})$. These numbers are positive and satisfy the constraint that $\sum_{\ell=1}^{V^n} \theta_{\mathbf{x}^{(\ell)}} = 1$.

Probability models

Probability models are ways of representing/specifying *specialized* joint distributions.

Computational Tasks

Simulation generate random configurations

Evaluation compute probability of a complete configuration

Inference sub-tasks:

- **Marginalize** compute probability of a partial configuration
- **Conditioning** compute conditional probability of a completion given an observation
- **Completion** find the most probable completion, given an observation

Learning learn parameters (will be covered later)

Joint distribution probability model

Simulation

Draw a complete configuration $\mathbf{x}$ according to the joint distribution. Given the lookup table representation, one could just compute the cumulative value of the $\theta_{\mathbf{x}^{(\ell)}}$'s, draw a random number p between 0 and 1, and select the configuration $\mathbf{x}^{(\ell)}$ whose cumulative probability interval contains p.

Evaluation

Evaluate the probability of a complete configuration $\mathbf{x} = (x_1, ..., x_n)$. In the table lookup representation, one can just look up the answer:

$$\mathrm{P}(X_1 = x_1, ..., X_n = x_n) = \theta_{(x_1, x_2, ..., x_n)}$$

Inference

I1. *Marginalize.* Compute the probability of an *incomplete* configuration.

$$
\begin{aligned}
&\mathrm{P}(X_1 = x_1, ..., X_k = x_k) \\
&= \sum_{y_{k+1}} \cdots \sum_{y_n} \mathrm{P}(X_1 = x_1, ..., X_k = x_k, X_{k+1} = y_{k+1}, ..., X_n = y_n) \\
&= \sum_{y_{k+1}} \cdots \sum_{y_n} \theta_{(x_1,...,x_k,y_{k+1},...,y_n)}
\end{aligned}
$$

Need to be able to evaluate complete configurations and then sum over V^{n-k} possible completions.

I2. *Condition.* Compute the conditional probability of a possible completion $(y_{k+1}, ..., y_n)$ given an incomplete configuration $(x_1, ..., x_k)$.

$$
\begin{aligned}
&\mathrm{P}(X_{k+1} = y_{k+1}, ..., X_n = y_n | X_1 = x_1, ..., X_k = x_k) \\
&= \frac{\mathrm{P}(X_1 = x_1, ..., X_k = x_k, X_{k+1} = y_{k+1}, ..., X_n = y_n)}{\mathrm{P}(X_1 = x_1, ..., X_k = x_k)} \\
&= \frac{\theta_{(x_1,...,x_k,y_{k+1},...,y_n)}}{\sum_{z_{k+1}} \cdots \sum_{z_n} \theta_{(x_1,...,x_k,z_{k+1},...,z_n)}}
\end{aligned}
$$

Need to evaluate a complete configuration and then divide by a marginal sum.

I3. *Complete.* Find the most probable completion $(y^*_{k+1}, ..., y^*_n)$ given an incomplete configuration $(x_1, ..., x_k)$.

$$
\begin{aligned}
y^*_{k+1}, ..., y^*_n &= \arg\max_{y_{k+1},...,y_n} \mathrm{P}(X_{k+1} = y_{k+1}, ..., X_n = y_n | X_1 = x_1, ..., X_k = x_k) \\
&= \arg\max_{y_{k+1},...,y_n} \frac{\mathrm{P}(X_1 = x_1, ..., X_k = x_k, X_{k+1} = y_{k+1}, ..., X_n = y_n)}{\mathrm{P}(X_1 = x_1, ..., X_k = x_k)} \\
&= \arg\max_{y_{k+1},...,y_n} \mathrm{P}(X_1 = x_1, ..., X_k = x_k, X_{k+1} = y_{k+1}, ..., X_n = y_n) \\
&= \arg\max_{y_{k+1},...,y_n} \theta_{(x_1,...,x_k,y_{k+1},...,y_n)}
\end{aligned}
$$

Have to search through all V^{n-k} possible completions and evaluate each complete configuration to find the maximum.

Probability models are ways of specifying specialized joint distributions, in which we have:

- more compact representation, and
- more efficient computation.

Impose structure on joint distribution $P(X_1 = x_1, ..., X_n = x_n)$. One key tool for imposing structure is variable independence.

Definition. Random variables X_1 and X_2 are *independent* if $P(X_1 = x_1, X_2 = x_2) = P(X_1 = x_1)P(X_2 = x_2)$ for all x_1, x_2. Equivalently, if $P(X_1 = x_1 | X_2 = x_2) = P(X_1 = x_1)$ for all x_1, x_2.

Definition. Random variables X_1 and X_2 are *conditionally independent given* X_3 if $P(X_1 = x_1, X_2 = x_2 | X_3 = x_3) = P(X_1 = x_1 | X_3 = x_3)P(X_2 = x_2 | X_3 = x_3)$ for all x_1, x_2, x_3. Equivalently, if $P(X_1 = x_1 | X_2 = x_2, X_3 = x_3) = P(X_1 = x_1 | X_3 = x_3)$ for all x_1, x_2, x_3.

Example: Fully independent model

Assume $P(X_1 = x_1, ..., X_n = x_n) = P(X_1 = x_1) \cdots P(X_n = x_n)$. This yields a very restricted form of joint distribution where we can represent each component distribution separately. For a random variable X_j, one can represent $P(X_j = x)$ by a lookup table with V parameters (minus one constraint). Let $\theta_{j,x}$ denote the probability X_j takes on value x. That is, $\theta_{j,x} = P(X_j = x)$. These numbers are positive and satisfy the constraint $\sum_{x=1}^{V} \theta_{j,x} = 1$ for each j. Thus, the joint distribution over $X_1, ..., X_n$ can be represented by $n \times V$ positive numbers minus n constraints. The previous tasks (simulation, evaluation, and inference) now become almost trivial. Admittedly this is a silly model as far as real applications go, but it clearly demonstrates the benefits of structure (in its most extreme form).

Simulation

For $j = 1, ..., n$, independently draw x_j according to $P(X_j = x_j)$ (using the lookup table representation). Conjoin $(x_1, ..., x_n)$ to form a complete configuration.

Evaluation

Given a complete configuration $\mathbf{x} = (x_1, ..., x_n)$, look up the probability of each component x_j and then multiply the answers together.

$$\mathrm{P}(X_1 = x_1, ..., X_n = x_n) \quad = \quad \theta_{1,x_1} \times \cdots \times \theta_{n,x_n}$$

Inference

I1. *Marginalize.*

$$\begin{aligned}
&\mathrm{P}(X_1 = x_1, ..., X_k = x_k) \\
&= \sum_{y_{k+1}} \cdots \sum_{y_n} \mathrm{P}(X_1 = x_1, ..., X_k = x_k, X_{k+1} = y_{k+1}, ..., X_n = y_n) \\
&= \sum_{y_{k+1}} \cdots \sum_{y_n} \mathrm{P}(X_1 = x_1) \cdots \mathrm{P}(X_k = x_k) \mathrm{P}(X_{k+1} = y_{k+1}) \cdots \mathrm{P}(X_n = y_n) \\
&= \mathrm{P}(X_1 = x_1) \cdots \mathrm{P}(X_k = x_k) \left[\sum_{y_{k+1}} \mathrm{P}(X_{k+1} = y_{k+1}) \left[\sum_{y_{k+2}} \cdots \left[\sum_{y_n} \mathrm{P}(X_n = y_n) \right] \right] \right] \\
&= \mathrm{P}(X_1 = x_1) \cdots \mathrm{P}(X_k = x_k) \left[\sum_{y_{k+1}} \mathrm{P}(X_{k+1} = y_{k+1}) \right] \cdots \left[\sum_{y_n} \mathrm{P}(X_n = y_n) \right] \\
&= \mathrm{P}(X_1 = x_1) \cdots \mathrm{P}(X_k = x_k) \\
&= \theta_{1,x_1} \times \cdots \times \theta_{k,x_k}
\end{aligned}$$

Only have to lookup and multiply k numbers.

I2. *Condition.*

$$\begin{aligned}
&\mathrm{P}(X_{k+1} = y_{k+1}, ..., X_n = y_n | X_1 = x_1, ..., X_k = x_k) \\
&= \frac{\mathrm{P}(X_1 = x_1, ..., X_k = x_k, X_{k+1} = y_{k+1}, ..., X_n = y_n)}{\mathrm{P}(X_1 = x_1, ..., X_k = x_k)} \\
&= \frac{\mathrm{P}(X_1 = x_1) \cdots \mathrm{P}(X_k = x_k) \mathrm{P}(X_{k+1} = y_{k+1}) \cdots \mathrm{P}(X_n = y_n)}{\mathrm{P}(X_1 = x_1) \cdots \mathrm{P}(X_k = x_k)} \\
&= \mathrm{P}(X_{k+1} = y_{k+1}) \cdots \mathrm{P}(X_n = y_n) \\
&= \theta_{k+1,y_{k+1}} \times \cdots \times \theta_{n,y_n}
\end{aligned}$$

Only have to lookup and multiply $n - k$ numbers.

I3. *Complete.*

$$
\begin{aligned}
y^*_{k+1},\ldots,y^*_n &= \arg\max_{y_{k+1},\ldots,y_n} \mathrm{P}(X_{k+1}=y_{k+1},\ldots,X_n=y_n|X_1=x_1,\ldots,X_k=x_k) \\
&= \arg\max_{y_{k+1},\ldots,y_n} \mathrm{P}(X_{k+1}=y_{k+1})\cdots\mathrm{P}(X_n=y_n) \\
&= \arg\max_{y_{k+1}} \mathrm{P}(X_{k+1}=y_{k+1}) \left[\arg\max_{y_{k+2}} \cdots \left[\arg\max_{y_n} \mathrm{P}(X_n=y_n)\right]\right] \\
&\quad \text{(Since max and arg max distributes over product just like sum.} \\
&\quad \text{That is, } \max_i ax_i = a\max_i x_i \text{ (for } a, x_i \geq 0) \\
&\quad \text{just like } \textstyle\sum_i ax_i = a\sum_i x_i.) \\
&= \left[\arg\max_{y_{k+1}} \mathrm{P}(X_{k+1}=y_{k+1})\right]\cdots\left[\arg\max_{y_n} \mathrm{P}(X_n=y_n)\right] \\
&= \left[arg\max_{y_{k+1}} \theta_{k+1,y_{k+1}}\right]\cdots\left[\arg\max_{y_n} \theta_{n,y_n}\right]
\end{aligned}
$$

Only have to search through V possible completions for each of the $n-k$ variables separately.

Note

It is important to note a general rule which we used to separate summations in the above tasks of Marginalization and Completion: If a and b are two variables, and $f(a)$ and $g(b)$ are two functions, such that $f(a)$ does not depend on b and $g(b)$ does not depend on a, then:

$$
\begin{aligned}
\sum_a\sum_b f(a)g(b) &= \sum_a f(a)\left(\sum_b f(a)g(b)\right) \\
&\quad \text{(because } f(a) \text{ is a constant for summation over } b) \\
&= \left(\sum_b g(b)\right)\cdot\left(\sum_a f(a)\right) \\
&\quad \text{(because } \sum_b g(b) \text{ is a constant for sumation over } a) \\
&= \left(\sum_a f(a)\right)\cdot\left(\sum_b g(b)\right)
\end{aligned}
$$

If we assume that $f(a) \geq 0$ and $g(b) \geq 0$, the same rule applies for $\arg\max_a$ and $\arg\max_b$:

$$
\begin{aligned}
&\arg\max_a \arg\max_b f(a)g(b) = \\
&\quad = \arg\max_a f(a) \left(\arg\max_b f(a)g(b) \right) \\
&\qquad \text{(because } f(a) \text{ is a constant for maximization over } b\text{)} \\
&\quad = \left(\arg\max_b g(b) \right) \cdot \left(\arg\max_a f(a) \right) \\
&\qquad \text{(because } \arg\max_b g(b) \text{ is a constant for maximization over } a\text{)} \\
&\quad = \left(\arg\max_a f(a) \right) \cdot \left(\arg\max_b g(b) \right)
\end{aligned}
$$

Readings

Russell and Norvig: Chap 14
Dean, Allen, Aloimonos: Sect 8.2

12 Structured probability models

Bayesian networks

(belief networks, Bayesian belief networks, decision networks)

Examples:

- burglary, earthquake, alarm, John-calls, Mary-calls example
- Naive Bayes example with news classification

Bayesian networks are an important method for representing restricted forms of joint distributions that have certain conditional independence structures. To define a Bayesian network we will exploit the general fact that for *any* joint distribution we have

$$\begin{aligned}
&\mathrm{P}(X_1=x_1, ..., X_n=x_n) \\
&\quad= \mathrm{P}(X_1=x_1)\ \mathrm{P}(X_2=x_2|X_1=x_1)\ \mathrm{P}(X_3=x_3|X_2=x_2, X_1=x_1)\ \cdots \\
&\qquad \cdots \mathrm{P}(X_n=x_n|X_{n-1}=x_{n-1}, ..., X_2=x_2, X_1=x_1)
\end{aligned}$$

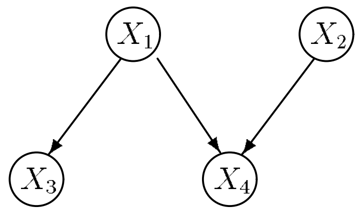
Definition. A *Bayesian network* is defined by a directed acyclic graph (DAG) and a collection of conditional probability tables, where nodes in the graph represent random variables and directed edges in the graph represent conditional independence assumptions. In particular, we assume the variables are ordered so that X_j's parents appear before X_j, and let $\pi(j)$ denote the indices of the parents of X_j in the graph. Then the conditional independence assumptions encoded by the graph are that any random variable X_j is independent of any other variable X_k given X_j's parents, $\mathbf{X}_{\pi(j)}$. That is,

$$\mathrm{P}(X_j=x_j|\mathbf{X}_{\pi(j)}=\mathbf{x}_{\pi(j)}, X_k=x_k) \quad = \quad \mathrm{P}(X_j=x_j|\mathbf{X}_{\pi(j)}=\mathbf{x}_{\pi(j)})$$

for any X_k. Thus, to represent a Bayesian network we first need to store the graph, and then store a lookup table for each variable X_j which represents the conditional probability of X_j given each possible configuration of its parents. Note that for a random variable X_j, we can represent $\mathrm{P}(X_j=x_j|\mathbf{X}_{\pi(j)}=\mathbf{x}_{\pi(j)})$ by a lookup table with $V \times |\pi(j)|$ positive numbers, minus one constraint (that $\sum_{x_j} \mathrm{P}(X_j = x_j | \mathbf{X}_{\pi(j)} = \mathbf{x}_{\pi(j)}) = 1$) for each configuration of the parents $\mathbf{X}_{\pi(j)}$. That is, let $\theta_{j,x,\mathbf{v}} = \mathrm{P}(X_j = x | \mathbf{X}_{\pi(j)} = \mathbf{v})$. These numbers are positive and satisfy the constraint $\sum_{x=1}^{V} \theta_{j,x,\mathbf{v}} = 1$ for each j and $\mathbf{v}$. Thus, the joint distribution over $X_1, ..., X_n$ can be represented by $\sum_{j=1}^{n} V \times V^{|\pi(j)|}$ positive numbers minus $\sum_{j=1}^{n} V^{|\pi(j)|}$ constraints.

If the maximum number of parents in the graph is bounded by k then this can be a severe restriction on the structure of the joint distribution, since the number of free parameters defining the distribution is reduced from $V^n - 1$ to $n(V-1)V^k$.

E.g.



$$
\begin{aligned}
&\mathrm{P}(X_1 = x_1, X_2 = x_2, X_3 = x_3, X_4 = x_4) \\
&= \mathrm{P}(X_1 = x_1)\, \mathrm{P}(X_2 = x_2)\, \mathrm{P}(X_3 = x_3 | X_1 = x_1)\, \mathrm{P}(X_4 = x_4 | X_1 = x_1, X_2 = x_2)
\end{aligned}
$$

How many parameters to represent?

$$
\begin{array}{ll}
V + V + V^2 + V^3 & \text{parameters} \\
-1 - 1 - V - V^2 & \text{constraints}
\end{array}
$$

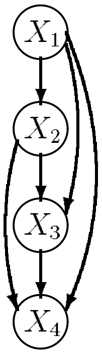
For each variable store a conditional probability table of size

$$V \cdot V^{\#\text{parents}} - V^{\#\text{parents}} \quad \text{(constraints)}$$

Representational power

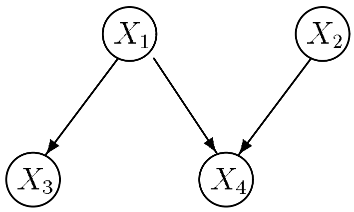
Using a Bayesian network representation one can represent: (1) arbitrary joint distribution, (2) fully independent distribution, and (3) distributions intermediate between these.

(1)



(2)

(3)



Number of parameters in each model:

(1) $(V-1)+(V^2-V)+(V^3-V^2)+(V^4-V^3)=V^4-1$

(2) $(V-1)+(V-1)+(V-1)+(V-1)=4V-4$

(3) solved above: V^3+V-2

Bayesian networks cannot represent all possible conditional independence structures, but they are still very useful.

Simulation

For $i=1,...,n$, draw x_j according to $\mathrm{P}(X_j=x_j|\mathbf{X}_{\pi(j)}=\mathbf{x}_{\pi(j)})$. Conjoin $(x_1,...,x_n)$ to form a complete configuration.

Evaluation

To compute the probability of a complete configuration, just multiply the local probabilities

$$\mathrm{P}(X_1=x_1,...,X_n=x_n) \quad = \quad \prod_{j=1}^{n} \mathrm{P}(X_j=x_j|\mathbf{X}_{\pi(j)}=\mathbf{x}_{\pi(j)})$$

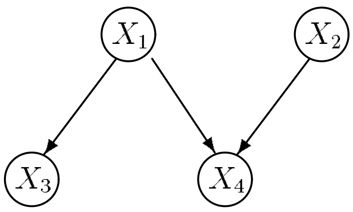
Inference

For some Bayesian networks inference must be hard (for example, inference with an arbitrary joint model like (1) above) because the size of the representation is exponentially large in the number of variables n (e.g., a size V^n lookup table in (1)). On the other hand, inference in trivial Bayesian networks is easy (such as the complete independent model (2)).

In general, inference (marginalization, conditioning, completion) is NP-hard for Bayesian networks, even if we restrict the graph to at most 2 parents per variable which forces a polynomial size representation. (This can be shown by taking arbitrary boolean function and turning it into a Bayesian network with probabilities of 0.5 for 0 and 1 at inputs.) If, however, graph is a *tree*, then efficient (polynomial time) inference algorithms can be derived as follows.

(The algorithm was re-invented in different disguises in Hidden Markov Models, Kalman filters, Error-correcting codes, Graphical belief models, Graphical probability models.)

Example



Compute $P(X_4 = x_4)$:

$$= \sum_{x_1}\sum_{x_2}\sum_{x_3} P(X_1 = x_1, X_2 = x_2, X_3 = x_3, X_4 = x_4)$$

$$= \sum_{x_1}\sum_{x_2}\sum_{x_3} P(X_1 = x_1)P(X_2 = x_2)P(X_3 = x_3|X_1 = x_1) P(X_4 = x_4|X_1 = x_1, X_2 = x_2)$$

V^3-1 additions and $3V^3$ multiplications, which is too expensive (as expensive as the joint distribution model).

Let us think of conditional probabilities as functions:

$$\begin{aligned}
f_1(x_1) &= P(X_1 = x_1) \\
f_2(x_2) &= P(X_2 = x_2) \\
f_3(x_1, x_3) &= P(X_3 = x_3 | X_1 = x_1) \\
f_4(x_1, x_2, x_4) &= P(X_4 = x_4 | X_1 = x_1, X_2 = x_2)
\end{aligned}$$

and let us try to get away with less additions and multiplications. (We use the fact that summations of factors which depend on variables x_2 and x_3 can be separated; see page 82.)

$$\begin{aligned}
&P(X_4 = x_4) = \\
&= \sum_{x_1}\sum_{x_2}\sum_{x_3} f_1(x_1) f_2(x_2) f_3(x_1, x_3) f_4(x_1, x_2, x_4) \\
&= \sum_{x_1} f_1(x_1) \sum_{x_3} f_3(x_1, x_3) \sum_{x_2} f_2(x_2) f_4(x_1, x_2, x_4) \\
&= \sum_{x_1} f_1(x_1) \left(\sum_{x_2} f_2(x_2) f_4(x_1, x_2, x_4) \right) \sum_{x_3} f_3(x_1, x_3)
\end{aligned}$$

First, for each value of x_1 we make V multiplications in sum over x_2, $V-1$ additions in sum over x_2, $V-1$ additions in sum over x_3, and two more multiplications of these two sums and f_1. That makes V^2+V multiplications and V^2-2V additions. After that, we make additional $V-1$ additions, which makes:
V^2+2V multiplications, and
$2V^2-V-1$ additions.

This is significantly better than $3V^3$ multiplications and V^3-1 additions.

Readings

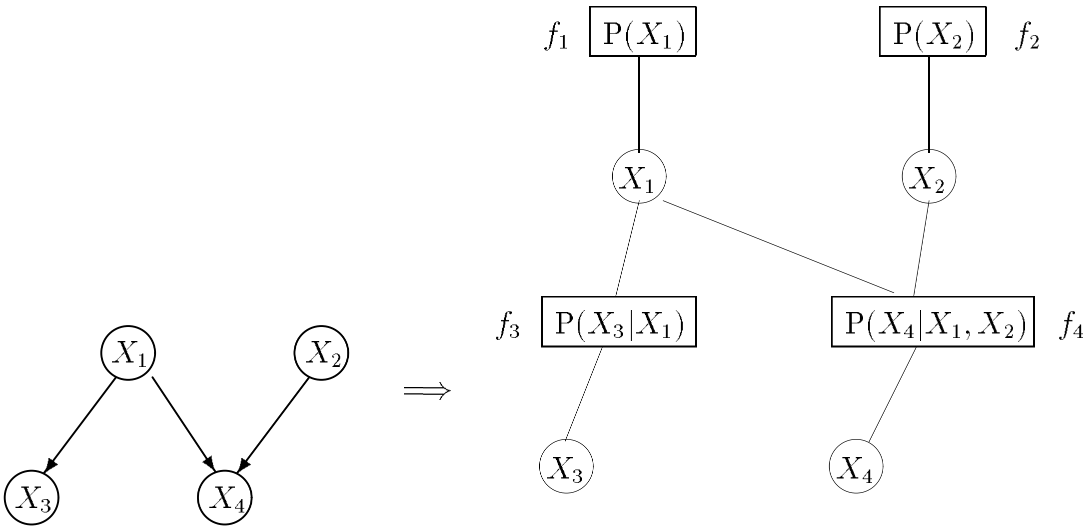
Russell and Norvig: Chapter 14
Dean, Allen, Aloimonos: Sect 8.3

13 Efficient probabilistic inference

Factor graph

The first step in Bayesian network inference is to convert the Bayesian network into a *factor graph*: an undirected graph that has one "round" node for each variable and one "square" node for each conditional probability function that is connected to each of the participating variables.

E.g.



An important property of this transformation is that if the original Bayes net is a tree then the resulting factor graph will also be a tree.

Given a factor graph with nodes $X_1, ..., X_n$ and functions $f_1, ..., f_k$, we can evaluate the probability of a complete configuration of the variables by

$$P(X_1 = x_1, ..., X_n = x_n) = \prod_{\ell=1}^{k} f_\ell(\mathbf{x}_\ell)$$

where $\mathbf{x}_\ell$ = the values of variables connected to f_ℓ.

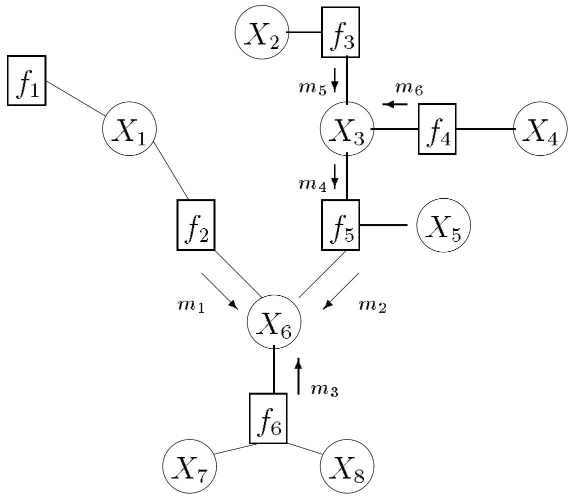
E.g.

$$P(X_1 = x_1, X_2 = x_2, X_3 = x_3, X_4 = x_4) = f_1(x_1)f_2(x_2)f_3(x_1, x_3)f_4(x_1, x_2, x_4)$$

Efficient marginalization for Bayesian networks becomes:

Efficient marginalization for tree structured factor graphs

Consider the following tree structured factor graph and consider computing the marginal probability $\mathrm{P}(X_6 = x_6)$.



The way to think about this computation is to first note that computing the marginal probability that $\mathrm{P}(X_6 = x_6)$ requires us to sum over all possible instantiations of the other variables, but that this sum can be factored into a product of sums over variables in each independent subtree, and each of these sums can then be recursively decomposed into a product of sums over independent subtrees, and so on. So, in particular, in this case we obtain

$$
\begin{aligned}
&\mathrm{P}(X_6 = x_6) = \\
&= \sum_{x_1,x_2,x_3,x_4,x_5,x_7,x_8} f_1(x_1) f_2(x_1,x_6) f_3(x_2,x_3) f_4(x_3,x_4) f_5(x_3,x_5,x_6) f_6(x_6,x_7,x_8) \\
&= \left[\sum_{x_1} f_2(x_1,x_6) f_1(x_1)\right] \left[\sum_{x_2,x_3,x_4,x_5} f_3(x_2,x_3) f_4(x_3,x_4) f_5(x_3,x_5,x_6)\right] \\
&\qquad \left[\sum_{x_7,x_8} f_6(x_6,x_7,x_8)\right] \\
&= m_1(x_6) \cdot m_2(x_6) \cdot m_3(x_6)
\end{aligned}
$$

Where now $m_2(x_6)$ further decomposes into

$$
\begin{aligned}
m_2(x_6) &= \sum_{x_2,x_3,x_4,x_5} f_3(x_2,x_3) f_4(x_3,x_4) f_5(x_3,x_5,x_6) \\
&= \sum_{x_3,x_5} f_5(x_3,x_5,x_6) \sum_{x_2,x_4} f_3(x_2,x_3) f_4(x_3,x_4) \\
&= \sum_{x_3,x_5} f_5(x_3,x_5,x_6) \cdot m_4(x_3)
\end{aligned}
$$

And $m_4(x_3)$ decomposes into

$$
\begin{aligned}
m_4(x_3) &= \sum_{x_2,x_4} f_3(x_2,x_3) f_4(x_3,x_4) \\
&= \left[\sum_{x_2} f_3(x_2,x_3)\right] \left[\sum_{x_4} f_4(x_3,x_4)\right] \\
&= m_5(x_3) \cdot m_6(x_3)
\end{aligned}
$$

And so on.

This factoring works because the sets of variables in each subtree are disjoint (in fact, this is precisely the reason why having a tree structure is significant). In general, we obtain an efficient marginalization procedure using the following message passing algorithm.

Message passing algorithm for marginalization

- Messages $m(x_j)$ are vectors of real numbers $(u_1, ..., u_V)$, where u_k is a number that summarizes the computation for the case $x_j = k$, and of course $k = 1, ..., V$.
- Messages are passed from variable nodes to function nodes, and from function nodes to variable nodes.
- A node can send a message to its neighbor only when it has received all of the messages from its other neighbors.
- Given a tree, the algorithm can start by sending messages from each of the leaves, and stops once every node has passed a message to every neighbor. (You should convince yourself that in a tree every node will eventually send a message to every neighbor, and therefore exactly two messages will be sent across every edge (one in each direction)).
- Function to variable messages $m_{f \to X}(x)$ are computed by

  $$m_{f \to X}(x) = \sum_{x_1, ..., x_k} f(x, x_1, ..., x_k) m_{X_1 \to f}(x_1) \cdots m_{X_k \to f}(x_k)$$

  over all *other* variables $X_1, ..., X_k$ (besides X) connected to f. If f contains only X, then $m_{f \to X}(x) = f(x)$.
- Variable to function messages $m_{X \to f}(x)$ are computed by

  $$m_{X \to f}(x) \quad = \quad \begin{cases} 1 & \text{if only } f \text{ contains } X \\ m_{f_1 \to X}(x) \cdots m_{f_k \to X}(x) & \text{otherwise} \end{cases}$$

  over all other functions $f_1, ..., f_k$ (besides f) containing X.
- Once all of the messages have been passed, then the final marginal for any variable X_j can be calculated by

  $$\mathrm{P}(X_j = x_j) \quad = \quad m_{f_1 \to X_j}(x_j) \cdots m_{f_k \to X_j}(x_j)$$

  over all $f_1, ..., f_k$ containing X_j.

This algorithm is efficient because there are $n - 1$ edges in an undirected tree containing n nodes (variables), $2(n - 1)$ messages get sent (one in each direction along each edge), each function to variable message can be computed in time $O(V^k)$ where k is the number of function neighbors, each variable to function message can be computed in time $O(Vk)$ where k is the number of variable neighbors, and the final marginal can be computed in time

$O(Vk)$. Thus, the total running time is bounded by $O(nV^k)$ where k is the maximum number of neighbors of any node in the graph. This is linear in n and polynomial in V (but exponential in k, so the maximum number of neighbors has to be bounded).

Computing the marginal of a collection of variables

Consider computing the marginal of one particular configuration $\mathrm{P}(X_1 = x_1, ..., X_k = x_k)$. For such a case, call the variables $X_1, ..., X_k$ *evidence variables* and call the instantiated values *observed evidence*. Then we can compute the desired probability by using the same message passing algorithm as above, except:

- An evidence-variable to function message is computed in the same way as before if $x = x_j$ (i.e., it is equal to observed evidence), otherwise it is 0. I.e.,

$$m_{X \to f}(x) = \begin{cases} 0 & \text{if } x \neq x_j \\ 1 & \text{if } x = x_j \text{ and only } f \text{ contains } X \\ m_{f_1 \to X}(x) \cdots m_{f_l \to X}(x) & \text{otherwise } (x = x_j) \end{cases}$$

  over all other functions $f_1, ..., f_l$ (besides f) containing X.
- Once all of the messages have been passed, then the final marginal can be determined by taking *any* evidence variable $X_j \in \{X_1, ..., X_k\}$ and computing

$$\mathrm{P}(X_1 = x_1, ..., X_k = x_k) = m_{f_1 \to X_j}(x_j) \cdots m_{f_l \to X_j}(x_j)$$

  over all $f_1, ..., f_l$ containing X_j.

Computing the conditional probability of a single variable

To compute $\mathrm{P}(X_{k+1} = y_{k+1} | X_1 = x_1, ..., X_k = x_k)$, we can use the same message passing algorithm as above, treating $X_1, ..., X_k$ *evidence variables*, except:

- Once all of the messages have been passed, then the final conditional probability can be determined by

$$\mathrm{P}(X_{k+1} = y_{k+1} | X_1 = x_1, ..., X_k = x_k) = \frac{m_{f_1 \to X_{k+1}}(y_{k+1}) \cdots m_{f_l \to X_{k+1}}(y_{k+1})}{Z}$$

where Z is a re-normalization constant over choices of y_{k+1}. That is,

$$Z = \sum_{y_{k+1}} m_{f_1 \to X_{k+1}}(y_{k+1}) \cdots m_{f_l \to X_{k+1}}(y_{k+1})$$

Computing arbitrary conditional probability

To compute arbitrary conditional probability $\mathrm{P}(\mathbf{X}_\alpha = \mathbf{y}_\alpha | \mathbf{X}_\beta = \mathbf{x}_\beta)$, where α and β are two disjoint sets of indices from $\{1, \ldots, n\}$, we can use formula:

$$\mathrm{P}(\mathbf{X}_\alpha = \mathbf{y}_\alpha | \mathbf{X}_\beta = \mathbf{x}_\beta) = \frac{\mathrm{P}(\mathbf{X}_\alpha = \mathbf{y}_\alpha, \mathbf{X}_\beta = \mathbf{x}_\beta)}{\mathrm{P}(\mathbf{X}_\beta = \mathbf{x}_\beta)},$$

where we know how to calculate marginal probabilities $\mathrm{P}(\mathbf{X}_\alpha = \mathbf{y}_\alpha, \mathbf{X}_\beta = \mathbf{x}_\beta)$ and $\mathrm{P}(\mathbf{X}_\beta = \mathbf{x}_\beta)$ using the message-passing algorithm.

Computing the most probable completion

To compute

$$y^*_{k+1}, ..., y^*_n = \arg\max_{y_{k+1},\ldots,y_n} \mathrm{P}(X_{k+1} = y_{k+1}, ..., X_n = y_n | X_1 = x_1, ..., X_k = x_k)$$

we can use the same message passing algorithm as the algorithm for calculating marginal probability $\mathrm{P}(X_1 = x_1, \ldots, X_k = x_k)$, except:

- Function to variable messages $m_{f\to X}(x)$ are computed by

$$m_{f\to X}(x) = \max_{x_1,\ldots,x_k} f(x, x_1, ..., x_k) m_{X_1 \to f}(x_1) \cdots m_{X_l \to f}(x_k)$$

  over all other variables $X_1, ..., X_l$ (besides X) connected to f.
- Once all of the messages have been passed, then the maximum probability completion for any free variable X_{k+j} can be calculated by

$$y^*_{k+j} = \arg\max_{y_{k+j}} \quad m_{f_1 \to X_{k+j}}(y_{k+j}) \cdots m_{f_\ell \to X_{k+j}}(y_{k+j})$$

  over all $f_1, ..., f_\ell$ containing X_{k+j}.

Readings

Frey: Section 2.1
Dean, Allen, Aloimonos: Section 8.3
Russell and Norvig: Section 15.3

14 Inference in complex models

What if graph is not a tree?

It is NP-hard even to approximate marginal and conditional probability.

Strategies:

exact methods

- cluster variables to make a tree
- find good variable ordering to minimize exhaustive summation (search)

heuristic methods

- repeated message passing, and hoping for convergence and that the answer is correct

estimation by random sampling

- use the standard procedure for generation (sampling) in Bayesian networks — *forward sampling*

Estimating marginals by random sampling

To estimate $\mathrm{P}(X_i = x_i)$, draw joint configurations:

$$\begin{array}{cccc} x_{11} & x_{12} & \dots & x_{1n} \\ x_{21} & x_{22} & \dots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{t1} & x_{t2} & \dots & x_{tn} \end{array}$$

Estimate:

$$\hat{\mathrm{P}}(X_i = x_i) = \frac{\text{count of occurences } X_i = x_i}{t}$$

Unbiased estimate.

This is Monte Carlo approximation for marginals.

Estimating conditionals by random sampling

Estimate: $\mathrm{P}(X_{k+1} = y_{k+1} | X_1 = x_1, \ldots, X_k = x_k)$

Draw joint configurations:

$$\begin{matrix} x_{11} & x_{12} & \ldots & x_{1n} \\ x_{21} & x_{22} & \ldots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{t1} & x_{t2} & \ldots & x_{tn} \end{matrix}$$

$$\hat{\mathrm{P}}(X_{k+1} = y_{k+1} | X_1 = x_1, \ldots, X_k = x_k) = \\ = \frac{\#\ \text{matches}(X_1 = x_1, \ldots, X_k = x_k, X_{k+1} = y_{k+1})}{\#\ \text{matches}(X_1 = x_1, \ldots, X_k = x_k)}$$

This is called *Logic sampling.*

It is a bad estimator if $(X_1 = x_1, \ldots, X_k = x_k, X_{k+1} = y_{k+1})$ is unlikely:

- small effective sample size

Importance sampling

Importance sampling is also called *likelihood weighting.*

Before describing importance sampling, let us define the notion of expected value of some function f. If x is a random variable, $P(x)$ probability distribution of x, and $f(x)$ a real-value function of x, then the expectation of $f(x)$ is defined by formula:

$$[f(x)]_{\mathrm{P}(x)} = \mathrm{E}_{\mathrm{P}(x)}(f(x)) = \sum_x f(x)\mathrm{P}(x)$$

Various problems (estimating conditionals among them) can be expressed as estimating the expected value of some function f, i.e., as estimating the value of $\mathrm{E}_{\mathrm{P}(x)} f(x)$.

A simple way to estimate $\mathrm{E}_{\mathrm{P}(x)} f(x)$ is the Monte Carlo method, which consists of the following:

- draw $x_1, x_2, \ldots, x_t$ from P, and
- estimate:

$$\hat{f} = \frac{1}{t} \sum_{i=1}^{t} f(x_i)$$

Problem: What if you cannot sample from P efficiently?

Let us assume that we can evaluate $\mathrm{P}(x)$ at parts of x.

Idea: Pick a proposed distribution Q that we can sample from.

Draw $x_1, x_2, \ldots, x_t$ from Q.

Weight points by $w(x_i) = \frac{\mathrm{P}(x_i)}{\mathrm{Q}(x_i)}$

Estimate: $\hat{f} = \frac{1}{t} \sum_{i=1}^{t} f(x_i) w(x_i)$

This sum gives an unbiased estimate:

$$\begin{aligned} \frac{1}{t} \sum_{i=1}^{t} f(x_i) w(x_i) \quad & \stackrel{t\to\infty}{\longrightarrow} \quad \mathrm{E}_{\mathrm{Q}(x)} f(x) w(x) \\ &= \quad \sum_x f(x) w(x) \mathrm{Q}(x) \\ &= \quad \sum_x f(x) \frac{\mathrm{P}(x)}{\mathrm{Q}(x)} Q(x) \\ &= \quad \sum_x f(x) \mathrm{P}(x) \\ &= \quad \mathrm{E}_{\mathrm{P}(x)} f(x). \end{aligned}$$

More realistically:

Assume that:

- cannot sample from $\mathrm{P}(x)$
- can sample from $\mathrm{Q}(x)$ and can evaluate $\mathrm{Q}(x)$
- cannot even evaluate $\mathrm{P}(x)$, but can evaluate $\mathrm{R}(x) = \beta \cdot \mathrm{P}(x)$ for unknown value of β.

Solution:

Draw $x_1, x_2, \ldots, x_t$ from Q.

Weight points by $u(x) = \frac{\mathrm{R}(x)}{\mathrm{Q}(x)}$

Estimate

$$\hat{f} = \frac{\sum_{i=1}^{t} f(x_i)u(x_i)}{\sum_{i=1}^{t} u(x_i)}$$

This estimate is biased, but it is asymptotically unbiased:

$$\frac{1}{t}\sum_{i=1}^{t} f(x_i)u(x_i) \stackrel{t\to\infty}{\longrightarrow} \sum_x f(x)u(x)\mathrm{Q}(x) = \sum_x f(x)\mathrm{R}(x) = \beta \sum_x f(x)\mathrm{P}(x)$$

$$\frac{1}{t}\sum_{i=1}^{t} u(x_i) \stackrel{t\to\infty}{\longrightarrow} \sum_x u(x)\mathrm{Q}(x) = \sum_x \mathrm{R}(x) = \beta \sum_x \mathrm{P}(x) = \beta$$

So,

$$\hat{f} \stackrel{t\to\infty}{\longrightarrow} \frac{\beta \sum_x f(x)\mathrm{P}(x)}{\beta} = \mathrm{E}_{\mathrm{P}(x)} f(x).$$

Estimating conditional probability using importance sampling

Let us see how the previous discussion applies to estimating conditional probabilities in Bayesian networks.

Let us estimate $\mathrm{P}(\mathbf{X}_\beta = \mathbf{y}_\beta | \mathbf{X}_\alpha = \mathbf{x}_\alpha)$, where α and β are sets of indices from $\{1, \ldots, n\}$ such that $\alpha \cap \beta = \emptyset$ and $\alpha \cup \beta = \{1, \ldots, n\}$.

This problem can be described in the following way:
If we fix variables $\mathbf{X}_\alpha$ to given values $\mathbf{x}_\alpha$, then the probability distribution

$$\mathrm{P}_1(\mathbf{X}_\beta) = \mathrm{P}(\mathbf{X}_\beta = \mathbf{y}_\beta | \mathbf{X}_\alpha = \mathbf{x}_\alpha)$$

over variables $\mathbf{X}_\beta$ is defined. If we additionally define

$$f(\mathbf{x}_\beta) = \begin{cases} 1 & \text{if } \mathbf{x}_\beta = \mathbf{y}_\beta \\ 0 & \text{otherwise} \end{cases}$$

then our target value $\mathrm{P}(\mathbf{X}_\beta = \mathbf{y}_\beta | \mathbf{X}_\alpha = \mathbf{x}_\alpha)$ can be expressed as expectation of the function f:

$$\mathrm{P}(\mathbf{X}_\beta = \mathbf{y}_\beta | \mathbf{X}_\alpha = \mathbf{x}_\alpha) = \mathrm{P}_1(\mathbf{X}_\beta = \mathbf{y}_\beta) = \mathrm{E}_{\mathrm{P}_1(\mathbf{x}_\beta)} f(\mathbf{x}_\beta)$$

This number could be estimated using the Monte Carlo method by sampling from distribution P_1: $\mathbf{x}_{\beta,1}, \ldots, \mathbf{x}_{\beta,t}$, and by estimating:

$$\hat{\mathrm{P}}_1(\mathbf{X}_\beta = \mathbf{y}_\beta) = \frac{1}{t} \sum_{i=1}^{t} f(\mathbf{x}_{\beta,i})$$

However, it is hard to sample from $\mathrm{P}_1(\mathbf{x}_\beta)$. We could do the sampling by sampling from P (the standard forward sampling in a Bayesian network) and by selecting only those configurations in which $\mathbf{X}_\alpha = \mathbf{x}_\alpha$; but if the probability of those configuration is very low then this sampling method is inefficient (small effective sample size).

We cannot efficiently evaluate $\mathrm{P}_1(\mathbf{x}_\beta)$ either.

On the other hand, it is easy to sample from another distribution $\mathrm{Q}(\mathbf{x}_\beta)$ in the following way: We fix all nodes $\mathbf{X}_\alpha$ in the Bayesian network to values $\mathbf{x}_\alpha$, and sample in the forward sampling fashion: The only difference is that for all nodes X_i from $\mathbf{X}_\alpha$ we sample by generating the fixed value x_i. This distribution Q is not the same as P_1, but, intuitively, it is a good representative of P_1.

We cannot evaluate $\mathrm{P}_1(\mathbf{x}_\beta) = \mathrm{P}(\mathbf{x}_\beta | \mathbf{x}_\alpha)$ efficiently, but can evaluate efficiently

$$\begin{aligned} \mathrm{R}(\mathbf{x}_\beta) &= \mathrm{P}(\mathbf{X}_\alpha = \mathbf{x}_\alpha, \mathbf{X}_\beta = \mathbf{x}_\beta) \\ &= \mathrm{P}(\mathbf{X}_\beta = \mathbf{x}_\beta | \mathbf{X}_\alpha = \mathbf{x}_\alpha) \cdot \mathrm{P}(\mathbf{X}_\alpha = \mathbf{x}_\alpha) \\ &= \mathrm{P}_1(\mathbf{x}_\beta) \cdot \beta \end{aligned}$$

where $\beta = \mathrm{P}(\mathbf{X}_\alpha = \mathbf{x}_\alpha)$ is a constant with respect to $\mathbf{x}_\beta$, and it is difficult to evaluate. We can evaluate $\mathrm{R}(\mathbf{x}_\beta)$ easily, since it is the probability of a complete configuration (Evaluation task for Bayesian networks).

Now, we can apply importance sampling:

1. Draw $\mathbf{x}_{\beta,1}, \ldots, \mathbf{x}_{\beta,t}$ from Q.

2. Calculate weights:

$$u(\mathbf{x}_{\beta,i}) = \frac{\mathrm{R}(\mathbf{x}_{\beta,i})}{\mathrm{Q}(\mathbf{x}_{\beta,i})}$$

3. And estimate

$$\hat{\mathrm{P}}_1(\mathbf{y}_\beta) = \frac{\sum_{i=1}^{t} f(\mathbf{x}_{\beta,i})u(\mathbf{x}_{\beta,i})}{\sum_{i=1}^{t} u(\mathbf{x}_{\beta,i})}$$

This method has larger effective sample size than logic sampling.

Works even if $P(\mathbf{X}_\alpha = \mathbf{x}_\alpha)$ is small, as long as Q is "representative" of P_1.

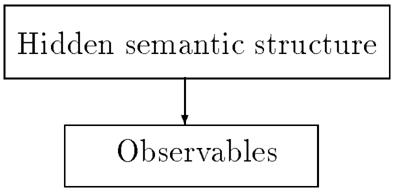
Readings

Russell and Norvig: Section 15.4
Dean, Allen, Aloimonos: Section 8.3

15 Interpreting senses (perception)

We use probabilistic models to build interpretation systems.

Generic model:



Interpretation:

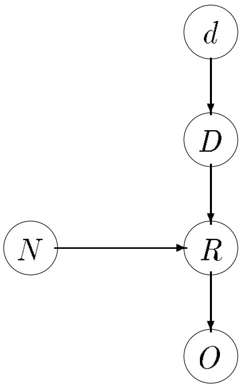
$$\text{interpretation} = \operatorname*{arg\,max}_{\text{possible truth}} \text{P}(\text{possible truth}|\text{observables})$$

Handwritten digit recognition

Given an image, recognize the digit.

Two generative probabilistic models are used.

Model 1



Pick digit: $d \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$ (equal probability for each digit)

Get image template:

$$D = \begin{bmatrix} d_{11} & \dots & d_{18} \\ \vdots & \ddots & \vdots \\ d_{81} & \dots & d_{88} \end{bmatrix} \in \{1, 2, 3, 4\}^{8\times 8}$$

Add noise:

$$N = \begin{bmatrix} n_{11} & \dots & n_{18} \\ \vdots & \ddots & \vdots \\ n_{81} & \dots & n_{88} \end{bmatrix} \in \{-3, -2, -1, 0, 1, 2, 3\}^{8\times 8}$$

Get raw image $r_{ij} = d_{ij} + n_{ij}$:

$$R = \begin{bmatrix} r_{11} & \dots & r_{18} \\ \vdots & \ddots & \vdots \\ r_{81} & \dots & r_{88} \end{bmatrix} \in \{-2,-1,0,1,2,3,4,5,6,7\}^{8\times 8}$$

Threshold: $o_{ij} = \begin{cases} 1 & \text{if } r_{ij} < 1 \\ r_{ij} & \text{if } 1 \le r_{ij} \le 4 \\ 4 & \text{if } r_{ij} > 4 \end{cases}$

$$O = \begin{bmatrix} o_{11} & \dots & o_{18} \\ \vdots & \ddots & \vdots \\ o_{81} & \dots & o_{88} \end{bmatrix} \in \{1,2,3,4\}^{8\times 8}$$

Each pixel is generated independently. Actually, a more accurate graph would include a node for each element of a matrix, e.g., n_{ij}, d_{ij}, etc., however it would not necessarily make the problem more clear.

Recognition

Given an image O, compute:

$$\begin{aligned} d^* &= \arg\max_d \mathrm{P}(d|O) \\ &= \arg\max_d \frac{\mathrm{P}(dO)}{\mathrm{P}(O)} \\ &= \arg\max_d \mathrm{P}(dO) \end{aligned}$$

Since $\mathrm{P}(O)$ is a positive constant with respect to d.

$\mathrm{P}(dO)$ is a marginal probability, which is calculated in this way:

$$\mathrm{P}(dO) = \sum_D \sum_N \sum_R \mathrm{P}(dDNRO) = \sum_D \sum_N \sum_R \mathrm{P}(D|d)\mathrm{P}(N)\mathrm{P}(R|DN)\mathrm{P}(O|R)$$

The template D is uniquely determined by d. $R = N + D$ always holds, so given N and D, $\mathrm{P}(R|DN) = 1$ if $R = N + D$, and $\mathrm{P}(R|DN) = 0$ otherwise. Hence we have:

$$\mathrm{P}(dO) = \sum_N \mathrm{P}(d)\mathrm{P}(N)\mathrm{P}(O|d,N)$$

Pixels are independent, and $\mathrm{P}(D)$ does not depend on N, so:

$$\mathrm{P}(dO) = \mathrm{P}(d) \sum_{n_{11}} \ldots \sum_{n_{88}} \prod_{ij} \mathrm{P}(n_{ij}) \mathrm{P}(o_{ij} | d_{ij}, n_{ij})$$

Using the "sum-separation rule" discussed on page 82, we get:

$$\mathrm{P}(dO) = \mathrm{P}(d) \prod_{i=1}^{8} \prod_{j=1}^{8} \sum_{n_{ij}=-3}^{3} \mathrm{P}(n_{ij}) \mathrm{P}(o_{ij} | d_{ij}, n_{ij})$$

Using this formula we can calculate:

$$\begin{aligned} d^* &= \arg\max_d P(dO) \\ &= \arg\max_d \mathrm{P}(d) \prod_{i=1}^{8} \prod_{j=1}^{8} \sum_{n_{ij}=-3}^{3} \mathrm{P}(n_{ij}) \mathrm{P}(o_{ij} | d_{ij}, n_{ij}) \end{aligned}$$

$\mathrm{P}(o_{ij} | d_{ij}, n_{ij})$ is 0 or 1. We add d_{ij} and n_{ij}, apply threshold on them, and if the result is equal to o_{ij} then the probability is 1, otherwise it is 0.

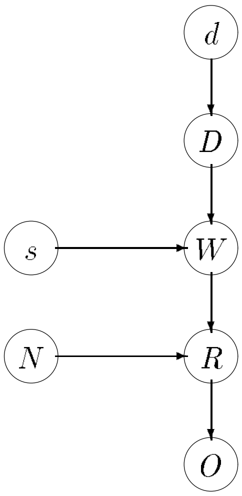
For each $d = 0, 1, 2, \ldots, 9$, we can calculate the conditional probability $P(d|O)$:

$$P(d|O) = \frac{P(dO)}{P(O)} = \frac{P(dO)}{\sum_{d=0}^{9} P(dO)}$$

Model 2

Add a global shear transformation over the whole image.

Generation:



template

warped image: W =shear(D, s)

raw image: $R = W + N$

thresholded image: O = threshold(R)

$s \in \{0, 1, 2, 3\}$ and the operation "shear" is defined by:

$$w_{ij} = \begin{cases} d_{i,j+\text{round}(s(5-i)/4)} & \text{if } 1 \leq j + \text{round}(s(5-i)/4) \leq 8 \\ 1 & \text{otherwise} \end{cases}$$

Recognition

As with Model 1, D, W, and R are "deterministic" nodes, i.e., given d, s, and N, they are uniquely defined, so:

$$\begin{aligned}
\mathrm{P}(dO) &= \sum_D \sum_s \sum_W \sum_N \sum_R \mathrm{P}(dDsWNRO) \\
&= \sum_s \sum_N \mathrm{P}(d)\mathrm{P}(s)\mathrm{P}(N)\mathrm{P}(O|d,s,N) \\
&= \mathrm{P}(d) \sum_{s=0}^{3} \mathrm{P}(s) \sum_{n_{11}} \ldots \sum_{n_{88}} \prod_{i=1}^{8} \prod_{j=1}^{8} \mathrm{P}(n_{ij})\mathrm{P}(o_{ij}|d_{i,t(i,j,s)}, n_{ij}) \\
&= \mathrm{P}(d) \sum_{s=0}^{3} \mathrm{P}(s) \prod_{i=1}^{8} \prod_{j=1}^{8} \sum_{n_{ij}=1}^{8} \mathrm{P}(n_{ij})\mathrm{P}(o_{ij}|d_{i,t(i,j,s)}, n_{ij})
\end{aligned}$$

Note that $d_{i,t(i,j,s)}$ is equivalent to w_{ij}. It is really $d_{i,t(i,j,s)}$, i.e., an entry in the matrix D, if $t(i,j,s)$ is in the range 1 to 8, so the entry exists. If $t(i,j,s)$ is not in that range, then the value is 1, as defined for w_{ij}.

$\mathrm{P}(o_{ij}|d_{i,t(i,j,s)}, n_{ij})$ is calculated by adding $d_{i,t(i,j,s)}$ and n_{ij}, thresholding the result, and comparing it to o_{ij}: if they are equal, the probability is 1, and it is 0 otherwise.

Now, digits can be recognized and conditional probabilities $P(d|O)$ can be calculated in the same way as in Model 1.

Readings

Russell and Norvig: Chapter 24
Dean, Allen, Aloimonos: Chapter 9

16 Interpreting natural language

Interpreting natural language is a part of Natural Language Processing (NLP). NLP can be divided into the following areas according to the "level" of processing:

phonetics: physical sounds of a language

phonology: sound system (phonemes) of a language

morphology: word structure

syntax: sentence structure

semantics: meaning at the sentence level

pragmatics: discourse, inter-sentence: meaning, references

For example, in speech recognition, we interpret sound and infer hidden structure, which is a string of words. In the next phase, or when interpreting written text, the hidden semantic structure is the meaning of the text.

Natural language content is typically too ambiguous, incomplete, imprecise, and error-ridden for exact computational approaches. Hence the need for uncertain, in our case probabilistic, methods.

N-gram models

An important task in probabilistic NLP is *language modeling:* Estimating the probability of arbitrary NL (natural language) sentence: P(sentence)

One application of this problem is in speech recognition. In speech recognition, we are interested in

$$\arg\max_{\text{sentence}} \mathrm{P}(\text{sentence}|\text{sound})$$

This is equal to:

$$\begin{aligned}
\arg\max_{\text{sentence}} \mathrm{P}(\text{sentence}|\text{sound}) &= \arg\max_{\text{sentence}} \frac{\mathrm{P}(\text{sentence}, \text{sound})}{\mathrm{P}(\text{sound})} \\
&= \arg\max_{\text{sentence}} \mathrm{P}(\text{sentence}, \text{sound}) \\
&= \arg\max_{\text{sentence}} \mathrm{P}(\text{sound}|\text{sentence})\mathrm{P}(\text{sentence})
\end{aligned}$$

It is easier to estimate P(sound|sentence) than P(sentence, sound), and it is done by an *acoustic model,* while P(sentence) is estimated by a *language model.*

N-gram model is very simple and it is among the most successful models for language modeling; trigram ($n = 3$) model in particular. In an n-gram model, we calculate joint distribution for all n-tuples of consecutive words (or characters). For example, in the trigram model, we count the number of occurrences of each triple of consecutive words from a corpus. Using this statistics, we can estimate the probability of arbitrary word w_3 following two given words w_1 and w_2: $\mathrm{P}(w_3|w_1w_2)$. It is useful to assign some small probability to unseen triples as well (using a technique called *smoothing*). If we use two "dummy" words '$\cdot$' at the beginning of each sentence, then the probability of arbitrary sentence can be calculated as:

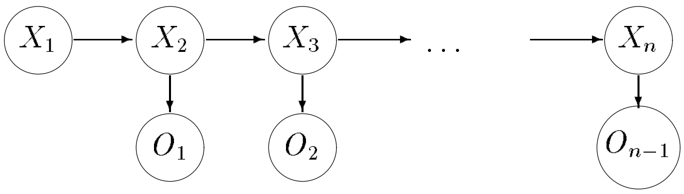
$$\mathrm{P}(w_1w_2\dots w_n) = \mathrm{P}(w_1|\cdot\cdot)\mathrm{P}(w_2|w_1\cdot)\mathrm{P}(w_3|w_2w_1)\dots\mathrm{P}(w_n|w_{n-1}w_{n-2})$$

Hidden Markov Model (HMM)

HMMs are successfully used for acoustic modeling in speech recognition.

An HMM is similar to a finite automaton: It includes a finite set of states. Instead of having an initial state, we have a probability $\pi(s)$ for each state s to be the initial state, so that $\sum_s \pi(s) = 1$. Instead of making transitions from state to state deterministically, for each pair of states s and t there is a probability of the transition from state s to t a_{st}, so that $\sum_t a_{st} = 1$ for each s. In each transition $s \to t$ and observable o is generated. The probability of generating observable o in transition $s \to t$ is given as parameter b_{sto}, so that for all s and t $\sum_o b_{sto} = 1$.

It is easy to sample from an HMM by generating an initial state, and by generating each transition and production of an observable. For evaluation and inference, it is sufficient to note that for a particular length of the sequence of observables n, the following Bayesian network is equivalent to the HMM:



The value of X_1 is the entered state of the HMM s_1, and the value of X_i for $2 \leq i \leq n$ is the pair of consecutive states (s_{i-1}, s_i). The value of each node O_i is the corresponding observable.

Probabilistic Context-Free Grammar (PCFG)

(Also known as Stochastic Context-Free Grammar (SCFG).)

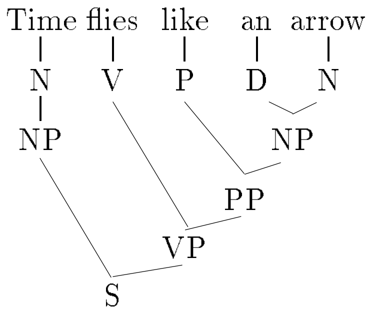
Both, n-gram model and HMM are linear models.

This does not always correspond to the structural nature of a NL; for example in sentence

The velocity of the seismic waves rises to ...

a linear model will give higher probability to the word "rise" after the plural "waves" than to the word "rises," which actually appears in the sentence.

Additionally, in order to reconstruct meaning of a sentence, we need to know the syntax structure, like the one represented by the tree:

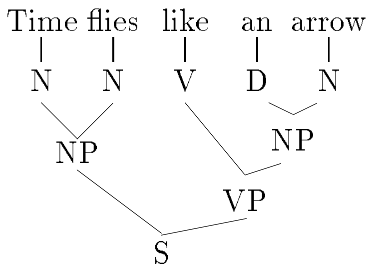


The task of determining word classes, e.g. from the example above:

Time/N flies/V like/P an/D arrow/N

is called *part-of-speech tagging* (POS tagging) and n-gram model and HMM are actually pretty good at this task.

For determining the rest of the structure, we use a grammar and a parser. We can use a context-free grammar, but unlike programming (or other formal) languages, a NL is ambiguous. For example, the above sentence has another possible parse:



Actually, symbolic approaches (i.e., non-probabilistic) to NL parsing use more complex grammars than CFG. Nonterminals are replaced by more complex feature structures, and additional consistency checks are made. However, even in these richer symbolic models the problem of ambiguity is not completely solved.

The above examples of parse trees would be produced by the following CFG:

S	$\rightarrow$	NP VP	VP	$\rightarrow$	V NP	N	$\rightarrow$	time	V	$\rightarrow$	like
NP	$\rightarrow$	N	VP	$\rightarrow$	V PP	N	$\rightarrow$	arrow	V	$\rightarrow$	flies
NP	$\rightarrow$	N N	PP	$\rightarrow$	P NP	N	$\rightarrow$	flies	P	$\rightarrow$	like
NP	$\rightarrow$	D N				D	$\rightarrow$	an			

We want to estimate which parse is more probable. This can be done by assigning a probability to each rule, and by assuming that each rule appears independently given its left-hand-side nonterminal. Hence, we calculate the probability of the tree by multiplying probabilities of all rules occurring in the tree:

$$\begin{aligned} \text{P(first tree)} &= \text{P(N} \rightarrow \text{time)P(V} \rightarrow \text{flies)P(P} \rightarrow \text{like)P(D} \rightarrow \text{an)} \\ &\quad \text{P(N} \rightarrow \text{arrow)P(NP} \rightarrow \text{N)P(NP} \rightarrow \text{D N)} \ldots \text{P(S} \rightarrow \text{NP VP)} \end{aligned}$$

If we assign the following probabilities to the rules:

S	$\rightarrow$	NP VP	/1	VP	$\rightarrow$	V NP	/.5	N	$\rightarrow$	time	/.5
NP	$\rightarrow$	N	/.4	VP	$\rightarrow$	V PP	/.5	N	$\rightarrow$	arrow	/.3
NP	$\rightarrow$	N N	/.2	PP	$\rightarrow$	P NP	/1	N	$\rightarrow$	flies	/.2
NP	$\rightarrow$	D N	/.4					D	$\rightarrow$	an	/1
V	$\rightarrow$	like	/.3								
V	$\rightarrow$	flies	/.7								
P	$\rightarrow$	like	/1								

then the probability of the first tree is 0.0084, and the probability of the second tree is 0.00036. We can conclude that the first tree is more likely, which should correspond to our intuition.

The probability assigned to a rule $N \to \zeta$ is the probability $\mathrm{P}(N \to \zeta | N)$, so if $N \to \zeta_1$, $N \to \zeta_2$, ... , $N \to \zeta_n$ are all rules with the nonterminal N on its left hand side, then

$$\sum_{i=1}^{n} \mathrm{P}(N \to \zeta_i) = 1$$

This model is a language model, since the sum of probabilities of all sentences in the language is 1. Actually, in order to be a language model, we also require that the grammar is *proper,* i.e., that all infinite trees have probability 0, which is not always the case.

We will not go into further details regarding this question here.

How do we solve basic computational tasks in this model?

Generation

We can generate (sample) sentences by starting with the initial nonterminal S, and by rewriting it using a rule $S \to \zeta$ according to the probabilities assigned to the rules that have S on their right hand side ($S \to \zeta_j$). We obtain a *sentential form*—a string of terminals and nonterminals. Take the first nonterminal in this form, and rewrite it in the same way; and so on. The loop stops when there are no nonterminals, i.e., when we obtain a sample sentence.

An interesting question is whether this process stops. If the grammar is *proper*, it stops with probability 1. If the grammar is not proper, we can stop after a large number of rewritings, and count the sample as an "infinite" one.

Evaluation

Given a tree t, what is $\mathrm{P}(t)$? As seen in the example, we simply multiply probabilities associated with each node of the tree.

Inference

Marginalization: P(sentence) =?

Conditioning: P(tree|sentence) =?

Completion: $\arg\max_{\text{tree}} \mathrm{P}(\text{tree}|\text{sentence})$

Marginalization

P(sentence) =?

If 'sentence' is the following sequence of words: $w_1 w_2 \dots w_n$, then P(sentence) is the following conditional probability:

$$\mathrm{P}(\text{sentence}) = \mathrm{P}(w_1 w_2 \dots w_n | S)$$

I.e., it is the probability of generating the sentence given that we start from S, i.e, it is $\mathrm{P}(S \Rightarrow^* w_1 \dots w_n)$.

Let us assume that the grammar is in CNF (Chomsky Normal Form).[1] In order to find $\mathrm{P}(w_1 w_2 \dots w_n | S)$, we can sum over all possible rules that can be applied to S, and for each of these rules $S \to XY$, we sum over all possible partitions of the string $w_1 \dots w_n$ into pieces $w_1 \dots w_k$ and $w_{k+1} \dots w_n$, so that $X \Rightarrow^* w_1 \dots w_k$ and $Y \Rightarrow^* w_{k+1} \dots w_n$. I.e.,

$$\mathrm{P}(w_1 w_2 \dots w_n | S) = \sum_{S \to XY} \sum_{k=1}^{n-1} \mathrm{P}(S \to XY) \mathrm{P}(w_1 \dots w_k | X) \mathrm{P}(w_{k+1} \dots w_n) | Y)$$

Now, we can recursively calculate $\mathrm{P}(w_1 \dots w_k | X)$ and $\mathrm{P}(w_{k+1} \dots w_n) | Y)$ using the same formula.

The problem is: This is an exponential algorithm. We unnecessarily recalculate some partial results over and over.

Using a dynamic programming approach, we get a much more efficient algorithm ($O(n^3)$, where n is the sentence length). It is a modified CYK (Cocke-Younger-Kasami) algorithm.[2]

[1] A Context-Free Grammar in CNF has only rules of the form $A \to B\,C$, where A, B, and C are nonterminals, and $A \to w$, where A is a nonterminal and w is a terminal. If a CFG is not in CNF, it can be converted into CNF.

[2] Most of the standard parsing algorithms for context-free grammars can be adapted for the inference task of probabilistic context-free grammars.

CYK can be applied only on a CNF grammar, so if the grammar is not already in CNF, we have to convert it.

Let all nonterminals be: $N^1, \ldots\ N^m$.

In the standard CYK algorithm, we have a two dimensional table V in which only the entries V_{ij}, $1 \leq i \leq i + j - 1 \leq n$, are used. Each entry V_{ij} contains a set of nonterminals that can produce substring $w_i \ldots w_{i+j-1}$ using the grammar rules, i.e., $V_{ij} = \{N | N \Rightarrow^* w_i \ldots w_{i+j-1}\}$.

If we enumerate all nonterminals: $N^1, N^2, \ldots, N^m$, then each set of nonterminals V_{ij} can be represented by extending V to be a 3-dimensional table V_{ijk}, in which $V_{ijk} = 1$ means that N^k can produce substring $w_i \ldots w_{i+j-1}$, and $V_{ijk} = 0$ that it cannot.

We adapt the algorithm by having entries in V to be real numbers between 0 and 1. Actually, we use the name β for the same table, and we call those entries *inside probabilities*. They represent the following probability:

$$\beta[i, j, k] = \mathrm{P}(w_i \ldots w_{i+j-1} | N^k)$$

So, $\beta[i, j, k]$ is the probability that the string $w_i \ldots w_{i+j-1}$ is generated from non-terminal N^k. Algorithm 7 is the probabilistic CYK algorithm for calculating P(sentence).

Algorithm 7 Probabilistic CYK for P(sentence)

Require: sentence $= w_1 \ldots w_n$, and a PCFG in CNF with nonterminals $N^1 \ldots N^m$, N^1 is the start symbol
Ensure: P(sentence) is returned

allocate $\beta \in \mathbb{R}^{n \times n \times m}$ and initialize all entries to 0
for $i \leftarrow 1$ to n **do**
  for all rules $N^k \rightarrow w_i$ **do**
    $\beta[i, 1, k] \leftarrow \mathrm{P}(N^k \rightarrow w_i)$
for $j \leftarrow 2$ to n **do**
  for $i \leftarrow 1$ to $n - j + 1$ **do**
    for $l \leftarrow 1$ to $j - 1$ **do**
      for all rules $N^k \rightarrow N^{k_1} N^{k_2}$ **do**
        $\beta[i, j, k] \leftarrow \beta[i, j, k] +$
          $\mathrm{P}(N^k \rightarrow N^{k_1} N^{k_2}) \cdot \beta[i, l, k_1] \cdot \beta[i + l, j - l, k_2]$
return $\beta[1, n, 1]$

Conditioning

For finding the conditional probability: P(tree|sentence), we use the following formula:

$$\mathrm{P}(\text{tree}|\text{sentence}) = \frac{\mathrm{P}(\text{tree}, \text{sentence})}{\mathrm{P}(\text{sentence})} = \frac{\mathrm{P}(\text{tree})}{\mathrm{P}(\text{sentence})}$$

since the sentence is a part of the tree. P(tree) is calculated by multiplying probabilities of all rules in the tree, and P(sentence) is calculated by Algorithm 7.

Completion

The most probable completion is computed in a similar way to marginalization (Algorithm 7). The difference is that the line 9 is replaced by the line

9: $\beta[i,j,k] \leftarrow \max(\beta[i,j,k], \mathrm{P}(N^k \to N^{k_1} N^{k_2}) \cdot \beta[i,l,k_1] \cdot \beta[i+l, j-l, k_2])$

Additionally in step 10, we are not just interested in $\beta[1,n,1]$, which is the probability of the most probable tree, but also in the actual tree. The tree can be reconstructed from the table using algorithm 8:

10: **return** Reconstruct$(1, n, 1, \beta)$

Note: What if the grammar is not in CNF

There is a standard algorithm for converting arbitrary CFG into CNF. The problem is: How to calculate probabilities of the rules in the new grammar? One way is to sample from the old grammar, and to estimate probabilities in the new grammar by parsing the sample sentences and counting. The probabilities can also be calculated directly, but it is tricky in some steps.

Here are the steps needed for conversion of an arbitrary CFG into CNF. This is just a partial sketch of the algorithm: calculating probabilities of the new rules in the first two steps is not trivial and it is not given.

Algorithm 8 Reconstruct(i, j, k, β)

Require: β — table from CYK, i — index of the first word, j — length of sub-string sentence, k — index of non-terminal

Ensure: a most probable tree with root N^k and leaves $w_i \dots w_{i+j-1}$ is returned

```
if j = 1 then
    return tree with root N^k and child w_i
for l ← 1 to j − 1 do
    for all rules N^k → N^{k1} N^{k2} do
        if β[i, j, k] = P(N^k → N^{k1} N^{k2}) · β[i, l, k1] · β[i + l, j − l, k2] then
            create a tree t with root N^k
            t.left_child ← Reconstruct(i, l, k1, β)
            t.right_child ← Reconstruct(i + l, j − l, k2, β)
            return t
```

Eliminate empty rules $N \to \epsilon$

Find all "nullable" nonterminals, i.e., terminals N such that $N \Rightarrow^* \epsilon$.

From each rule $A \to X_1 \dots X_n$ create new rules by striking out some nullable nonterminals. This is done for all combination of nonterminals in the rule, except for striking out all $X_1 \dots X_n$ if they are all nullable.

Remove empty rules.

If the start symbol is nullable, add $S \to \epsilon$, and treat that as a special case.

Eliminate unit rules $N \to M$

For any two variables $A \Rightarrow^* B$, for all non-unit rules $B \to \zeta$, we add $A \to \zeta$. Remove unit rules.

All possible derivations $A \Rightarrow^* B$ are easy to find since the empty rules are already eliminated.

Eliminate terminals in rules, except $A \to w$

For each terminal w that appears on the right hand side of some rule with some other symbols, we introduce a new nonterminal N_w, and a rule $N_w \to w$

with probability 1. Then we replace w in all other rules with N_w.

Eliminate rules $A \to B_1 B_2 \ldots B_n$ $(n > 2)$

For each rule $A \to B_1 B_2 \ldots B_n$ $(n > 2)$, we introduce $n-2$ new nonterminals $X_1, \ldots, X_{n-2}$, and replace this rule with the following rules: $A \to B_1 X_1$, $X_1 \to B_2 X_2$, $\ldots$ $X_{n-2} \to B_{n-1} B_n$, and assign the following probabilities to them: $\mathrm{P}(A \to B_1 X_1) = \mathrm{P}(A \to B_1 B_2 \ldots B_n)$, $\mathrm{P}(X_1 \to B_2 X_2) = 1$, $\ldots$ $\mathrm{P}(X_{n-2} \to B_{n-1} B_n) = 1$.

Readings

Russell and Norvig: Chapter 22
Dean, Allen, Aloimonos: Chapter 10
Hopcroft, Ullman: about CYK and CNF, *Introduction to Automata Theory, Languages, and Computation*

17 Optimal behavior: Decision theory

Part 4: Behaving

How to act optimally under uncertainty?

Like in planning, we have:

- set of states: S
- set of actions: A

State dynamics: taking a in s leads to s'

Unlike planning, there is uncertainty in state dynamics: by applying action a in state s we cannot be certain which resulting state s' will be next.

Instead of the goal states, we introduce a more general concept — reward function: $R(s)$. For each state s, the number $R(s)$ is the reward (a negative reward is a penalty). We want to choose such action in each state, so that for the resulting sequence of states s_1, s_2, ... , s_t, the total reward $\sum_{i=1}^{t} R(s_i)$ is maximal.

The reward function is a more general concept that goal states. Goal states can be expressed using a reward function:

$$R(s) = \begin{cases} 1 & \text{if } s \text{ is a goal state} \\ 0 & \text{otherwise} \end{cases}$$

and once we achieve a goal state, no further actions can be applied.

The optimal behaviour problem is:
Given state dynamics and a reward function, determine best actions to take.

We differentiate two kinds of uncertainty in state dynamics:

- world could be random
- world could be adversarial (hostile)

We may have to tradeoff short term vs. long term reward.

Easiest case: Planning

Actions are deterministic $s' = a(s)$

Given an initial state and the goal condition:

1. precompute an optimal action sequence
2. execute the sequence blindly

Slightly harder case: Conditional planning

Actions are non-deterministic:

$s' \in a(s)$, i.e., $a(s)$ is a set of possible next states that can happen when a is executed in s.

This problem can be tackled in two ways:

conditional planning: (or contingency planning) plan for multiple outcomes

execution monitoring: monitor execution of the plan and re-plan in the middle of execution if needed. (deferring to deal with alternatives until they actually happen)

General case

Plan an action for every possible state.

I.e., come up with a *total policy* (or *controller*), which is given by: $\pi : S \rightarrow A$

In order to determine an optimal behaviour, we precompute optimal policy.

There are two cases:

Decision theory: state dynamics are random: Living in an oblivious stochastic environment.

Game theory: state dynamics are adversarial: The world (or your opponents) is out to get you.

Optimal decision theory

The model includes:

- state space: S
- actions: A
- reward function $R : S \to \mathbb{R}$
- state transition model is random, i.e., we are given: $\mathrm{P}(s'|s,a)$

We assume that we can identify the current state, i.e., that the model is *accessible.*

This kind of model: stochastic model which is accessible and in which the transition model is known, i.e., $\mathrm{P}(s'|s,a)$ is known, is called *Markov decision problem.*

Our task is to find an optimal policy: $\pi^* : S \to A$

Simplest case: optimize immediate expected reward

We look only one step ahead. For a given state s and for any action a, the expected total reward in the next state is:

$$R(s) + \sum_{s'} P(s'|s,a)R(s')$$

So, the optimal action in short term is the action that maximizes this reward:

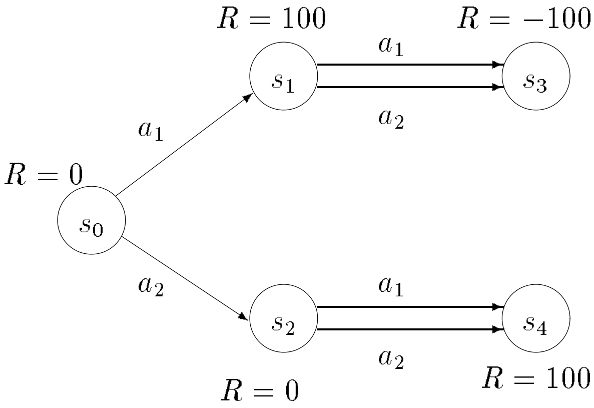
$$a^* = \arg\max_a \left(R(s) + \sum_{s'} \mathrm{P}(s'|s,a)R(s')\right) = \arg\max_a \sum_{s'} P(s'|s,a)R(s')$$

Harder case: Sequential decision problem

Sequential decision problem

In this case we have to choose several actions in sequence so that the cumulative reward is maximized.

The following example illustrates that the greedy approach of optimizing the immediate expected reward is not necessarily the best approach:



Optimal policy $\pi^*(s_0) = a_2$, even though the optimal action for one step is a_1.

Computing optimal policies: Acyclic case

Let us first consider an easier case: acyclic case.

We assume that the model is acyclic, i.e., after applying any (non-empty) action sequence to a state s, the resulting state will not be s.

The above assumption is sufficient for the following ideas to work, but for the clarity of representation, we additionally assume that:

- we start at state s^0 and finish at state s^t after t actions
- after applying first action to s^0 we go to one of the states:

  $$s_0^1, s_1^1, \dots, s_{k_1}^1$$

  then after the second action, we are in one of the states:

  $$s_0^2, s_1^2, \dots, s_{k_2}^2$$

  and so on

  $$\dots$$

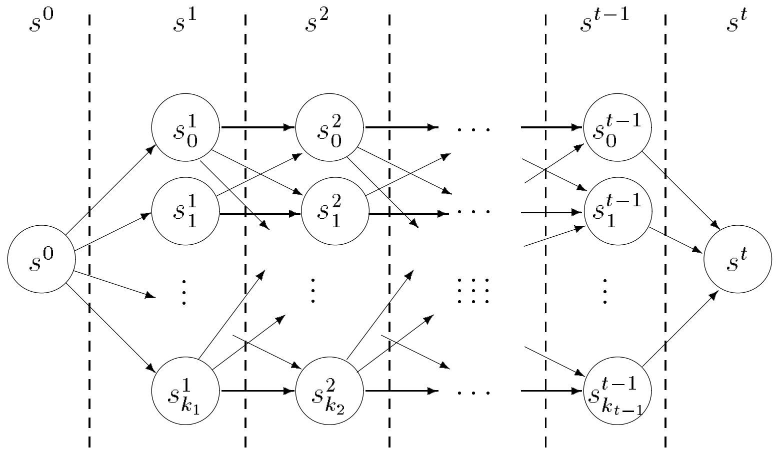
  after the $(t-1)$-th action, we are in one of the states:

  $$s_0^{t-1}, s_1^{t-1}, \dots, s_{k_{t-1}}^{t-1}$$

  and after the t-th action, we are in state:

  $$s^t$$

This is represented in the following figure:



We are given:

- the transition model or state dynamics as probabilities: $\mathrm{P}(s_n^{i+1}|s_m^i, a)$, for $0 \leq i < t-1$, $0 \leq n \leq k_{i+1}$, $0 \leq m \leq k_i$, and for all applicable actions a.
- the reward function $R(s) \in \mathbb{R}$ for all states s

Let A be the set of all actions, and S the set of all states.

The task is: Compute $\pi^* : S \to A$, a lookup table of length $|S|$, that maximizes expected future total reward for each state.

We define *utility function* or *value function* $U : S \times \Pi \to \mathbb{R}$, where Π is the set of all policies. For arbitrary state s and policy π:

$U(s, \pi) =$ expected total reward that will be obtained by policy π in state s.

If we know the utility of all states that can result from the action $\pi(s)$, then we can evaluate utility of the state s using the following formula:

$$U(s, \pi) = R(s) + \sum_{s'} U(s', \pi)\mathrm{P}(s'|s, \pi(s))$$

We have to find the policy π^* that maximizes $U(s, \pi)$ for all s.

Naïve algorithm

A naive algorithm would be:

1. enumerate all policies ($|A|^{|S|}$ of them)
2. evaluate each policy (with a smart evaluation by starting from s^t and going backwards, this could be done in $O(|A| \cdot |S|^2)$ time)
3. pick winner, i.e., the policy that gives highest expected total reward for each state (there always exists one)

However, this is too expensive.

Efficient algorithm: Dynamic programming

In our special case, the utility function can be expressed as:

$$U(s^i, \pi) = R(s^i) + \sum_{s^{i+1}} U(s^{i+1}, \pi)\mathrm{P}(s^{i+1}|s^i, \pi(s^i))$$

for $0 \leq i < t$ and $U(s^t, \pi) = R(s^t)$.

We optimize the policy for states s^{t-1} and then go backwards towards s^0 using formula:

$$\pi^*(s^i) = \arg\max_a \sum_{s^{i+1}} U(s^{i+1}, \pi^*)\mathrm{P}(s^{i+1}|s^i, a)$$

Algorithm 9 describes precisely how this is done.

Algorithm 9 Sequential decision problem: acyclic case

$U(s^t) \leftarrow R(s^t)$;
for $j \leftarrow 0$ to k_{t-1} **do**
  $\pi^*(s_j^{t-1}) \leftarrow$ the only available action, which leads to s^t
  $U(s_j^{t-1}) \leftarrow R(s_j^{t-1}) + U(s^t)$
for $i \leftarrow t-2$ down to 1 **do**
  for $j \leftarrow 0$ to k_i **do**
    $\pi^*(s_j^i) \leftarrow \arg\max_a \sum_{k=0}^{k_{i+1}} U(s_k^{i+1}, \pi^*) \mathrm{P}(s_k^{i+1} | s_j^i, a)$
    $U(s_j^i, \pi^*) \leftarrow R(s_j^i) + \sum_{k=0}^{k_{i+1}} U(s_k^{i+1}, \pi^*) \mathrm{P}(s_k^{i+1} | s_j^i, \pi^*(s_j^i))$
$\pi^*(s^0) \leftarrow \arg\max_a \sum_{k=0}^{k_1} U(s_k^1, \pi^*) \mathrm{P}(s_k^1 | s^0, a)$
$U(s^0, \pi^*) \leftarrow R(s^0) + \sum_{k=0}^{k_1} U(s_k^1, \pi^*) \mathrm{P}(s_k^1 | s^0, \pi^*(s^0))$

Readings

Russell and Norvig: Chapter 13, Section 16.1
Dean, Allen, Aloimonos: Section 8.4

18 Optimal sequential decision making

The problem is: What if the state space is acyclic?

Example: Cat and mouse

Cat and mouse in a 4×4 grid.

16 positions for cat $\times$ 16 positions for mouse = 256 states.

Cheese is in two corner squares: $(0,0)$ and $(3,3)$.

5 actions for cat and mouse: left, up, stay, down, right

$$\text{Reward function} = \begin{cases} 0 & \text{mouse alone on square} \\ 1 & \text{mouse + cheese} \\ \text{-3} & \text{mouse + cat} \\ \text{-2} & \text{mouse + cat + cheese} \end{cases}$$

There are two environments:

Environment R: Cat moves randomly

Environment A: Cat can see and is clairvoyant (can guess your move)

`Reward` function is given as a 256×1 matrix.

`Probs` is $256 \times 256 \times 5$ matrix with the following probabilities:

$$\text{Probs}(sg, sn, a) = \text{P}(sn|sg, a)$$

A discounting factor `Gamma` is given (default value 0.95).

Policy is a 256×1 vector with values from the set $\{1, 2, 3, 4, 5\}$.

Value function

How to calculate utility (value function) of policy π in a state s?

The following formula holds:

$$U(s, \pi) = R(s) + \sum_{s'} U(s', \pi) \mathrm{P}(s'|s, \pi(s))$$

How to evaluate this function?

If we try recursive call, we will get stuck in an infinite recursive descent due to loops in the state space.

Let us denote $V_\pi(s) = U(s, \pi)$. The previous formula becomes:

$$V_\pi(s) = R(s) + \sum_{s'} V_\pi(s') \mathrm{P}(s'|s, \pi(s))$$

If we write the above formula for each state s, then we get $|S|$ equalities with $|S|$ unknown values: $V_\pi(s)$

It is a system of linear equations and we can solve it using an exact method (e.g., Gaussian), or using an iterative method.

The problem is: The system can easily diverge.

For example, if we have a strategy which keeps us in a loop of states, and each time we go through the loop we expect to collect some positive reward, then the expected total reward will be ∞ since this can be repeated forever.

This is the problem of *infinite horizon.*

In this case we would not differentiate between a policy that earns reward 1 after each action, and a policy that earns 1 after every 5 actions. There are two ways to handle this:

- maximize asymptotic rate of reward
- maximize discounted reward

We use discounted reward.

The idea is similar to a situation in economy:
1 dollar today is worth more than 1 dollar tomorrow. It will be 1.01 tomorrow if we have (an astronomic) interest rate of 1% per day. In other words, $1 tomorrow is worth about 99 cents today. So, instead of calculating future rewards as:

$$R_0 + R_1 + R_2 + \ldots$$

we calculate

$$R_0 + \gamma R_1 + \gamma^2 R_2 + \ldots$$

with some *discount factor* γ, $0 < \gamma < 1$.

This approach is called *discounting.* It will prefer policies that earn rewards faster, and it is guaranteed to converge, i.e., $V_\pi(s) < \infty$, since the series $\sum_{i=0}^{\infty} \gamma^i$ converges. The sum of the series is $\frac{1}{1-\gamma}$, so $V_\pi(s) \leq \frac{1}{1-\gamma} \max_s(R(s))$ for all s.

If we use discounting, then the formula for the value function is:

$$V_\pi(s) = R(s) + \gamma \sum_{s'} V_\pi(s')\mathrm{P}(s'|s, \pi(s))$$

If we write the above equation for each s, then given a policy π, this becomes a system of linear equations with unknown variables $V_\pi(s)$, which can solved using some exact or iterative method.

Example: Calculating value function

Going back to the cat and mouse example, the formulas (for all s):

$$V_\pi(s) = R(s) + \gamma \sum_{s'} V_\pi(s')\mathrm{P}(s'|s, \pi(s))$$

can be expressed as the matrix equation:

$$V_\pi = R + \gamma P_\pi V_\pi$$

where matrix P_π can be calculated using Matlab code like:

```
for s=1 to 256,
  P_pi(s,:) = Probs(s,:,policy(s));
end
```

The above system of linear equations can be solved in two ways:

- by an exact method, or
- iteratively

For solving it using an exact method (e.g., Gaussian), we note that the solution is:

$$V_\pi = (I - \gamma P_\pi)^{-1} R$$

For example, in Matlab this can be solved using backslash operator.

For $\gamma \in (0, 1)$ the system has a solution, but for γ close to 1, it can be nearly singular. A more precise method can be the iterative method:

- start from arbitrary values for $V_\pi(s)$ and
- iterate using formula

$$V_\pi^{new} \leftarrow R + \gamma P_\pi V_\pi$$

- if V_π^{new} and V_π are sufficiently close, then V_π^{new} is the solution, otherwise $V_\pi \leftarrow V_\pi^{new}$ and we repeat iteration.

Policy iteration

Now we go back to our initial problem of finding the optimal policy. The first approach to solving this problem is *policy iteration:*

1: generate random policy π
2: calculate value of the policy V_π
3: generate new policy by optimizing old values:
   namely, for all states s
   $\pi^{new}(s) \leftarrow \arg\max_a \left(\sum_{s'} \mathrm{P}(s'|s,a)V_\pi(s')\right)$
4: **if** $\pi = \pi^{new}$ **then**
5:     **return** π^{new}
6: $\pi \leftarrow \pi^{new}$
7: **go to** 2

Example

When the algorithm for policy iteration is implemented, e.g., in Matlab, there are two issues:

- maximization in Matlab is more efficient on a matrix, then when done in a for-loop, and that is how it should be implemented

  - Due to rounding errors when evaluation the value function for γ close to 1, the loop may get stuck in an infinite flip-flopping of policies π and π^{new} between two equivalent policies. In order to handle this problem, two fixes are possible:
    - monitor value function, or
    - add an ϵ-preference (e.g., for $\epsilon = 1^{-10}$) for keeping the old policy in step 3 (This can be done by adding ϵ to $\sum_{s'} \mathrm{P}(s'|s, \pi(s))V_\pi(s')$ for each s.)

Value iteration

Second method for finding the optimal policy is to first find the optimal value function, without explicitly finding corresponding policies, and then find the policy that corresponds to this value function.

1: start with an arbitrary value function, e.g. $V(s) = \frac{1}{1-\gamma} \max_s R(s)$ is a reasonable choice.
2: **repeat**
3: $\quad V^{old} \leftarrow V$
4: $\quad$ for all $s \in S$, $V \leftarrow R(s) + \gamma \max_a \sum_{s'} V^{old}(s')\mathrm{P}(s'|s, a)$
5: **until** V^{old} and V are close enough for all s

After finding the optimal value function, we find the optimal policy; for all $s \in S$:

$$\pi(s) \leftarrow \arg\max_a R(s) + \gamma \sum_{s'} V(s')\mathrm{P}(s'|s, a)$$

$$= \arg\max_a \sum_{s'} V(s')\mathrm{P}(s'|s, a)$$

Readings

Russell and Norvig: Chapter 17

19 Optimal behavior: Game theory

In game theory, we have adversarial state dynamics: have to account for the worst case.

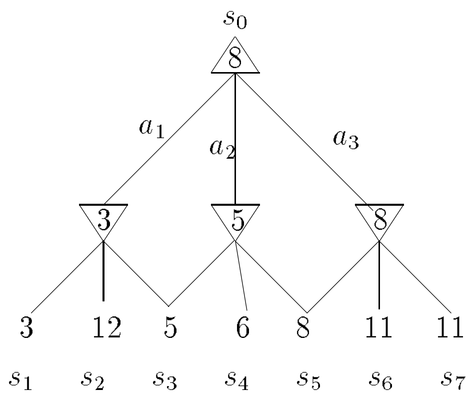
Compute policy (strategy) $\pi : S \to A$ that maximizes *minimal* (instead of expected) reward.

$$S'(a, s) = \{ \text{ set of possible states reachable after doing } a \text{ in } s \ \}$$

(We assume we can identify the current state.)

Single step case: maximin reward

Example:



We are in state s_0. After applying a_1, we can end up in one of the states s_1, s_2, or s_3, depending on the opponent's move (i.e., the "environment's" outcome). That is: $S'(a_1, s_0) = \{s_1, s_2, s_3\}$

Similarly: $S'(a_2, s_0) = \{s_3, s_4, s_5\}$ and $S'(a_3, s_0) = \{s_5, s_6, s_7\}$.

We anticipate that the outcome will always be the worst for us, so we choose the best of the worst, i.e., the maximin:

$$a^* = \arg\max_a \min_{s' \in S'(a, s_0)} S'(a, s_0)$$

which is a_3 in the above case.

Sequential acyclic case

Similarly to the sequential acyclic case in decision theory, assume:

- the model is acyclic, i.e., after applying any (non-empty) action sequence to a state s, the resulting state will not be s.
- we start at state s^0 and finish at state s^t after t actions
- after applying first action to s^0 we go to one of the states:

$$s^1_0, s^1_1, \dots, s^1_{k_1}$$

then after the second action, we are in one of the states:

$$s^2_0, s^2_1, \dots, s^2_{k_2}$$

and so on

$$\dots$$

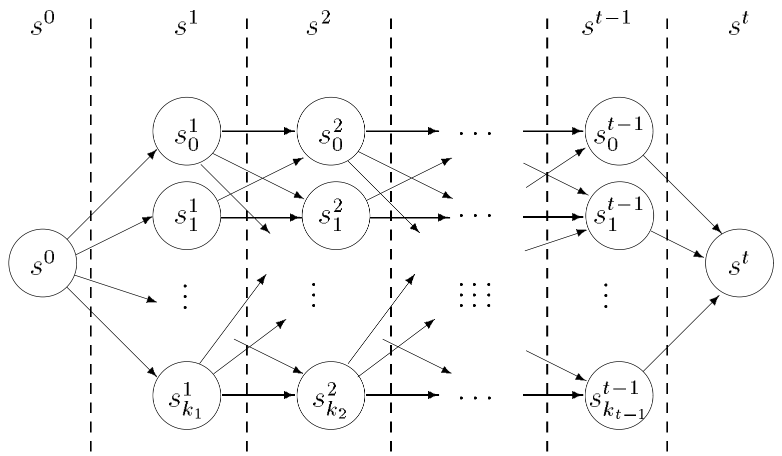
after the $(t-1)$-th action, we are in one of the states:

$$s^{t-1}_0, s^{t-1}_1, \dots, s^{t-1}_{k_{t-1}}$$

and after the t-th action, we are in state:

$$s^t$$

This is represented in the following figure:



We are given:

– the reward function $R(s) \in \mathbb{R}$ for all states s

Unlike decision theory, we are not given transition probabilities, but

– the sets: $S'(a, s^i_m) = \{\text{possible } s^{i+1}\}$

Let A be the set of all actions, and S the set of all states.

The task is: Compute $\pi^* : S \to A$, a lookup table of length $|S|$, that maximizes minimal future total reward for each state.

Utility function

We define *utility function* or *value function* $U : S \times \Pi \to \mathbb{R}$, where Π is the set of all policies. For arbitrary state s and policy π:

$U(s, \pi)$ = minimal total reward that will be obtained by policy π in state s.

If we know the utility of all states that can result from the action $\pi(s)$, then we can evaluate utility of the state s using the following formula:

$$U(s, \pi) = R(s) + \min_{s' \in S'(\pi(s), s)} U(s', \pi)$$

We have to find the policy π^* that maximizes $U(s, \pi)$ for all s.

The sequential acyclic case can be solved efficiently using dynamic programming.

We optimize the policy for states s^{t-1} and then go backwards towards s^0 using formulas:

$$\begin{aligned} \pi^*(s^i) &= \arg\max_a \min_{s^{i+1} \in S'(a, s^i)} U(s^{i+1}, \pi^*) \\ U(s^i, \pi^*) &= \min_{s^{i+1} \in S'(\pi^*(s^i), s^i)} U(s^{i+1}, \pi^*) \end{aligned}$$

An approach to cyclic case: Cat and mouse example

We illustrate an approach to the game theory using the previous cat and mouse example.

This time, we assume that cat does not move in random way, but always makes the worst move for the mouse. Hence, we have to apply maximin optimization.

Instead of the probability transitions, this time we are only interested in the possible cat moves, i.e., in the set of possible outcome states after we (the mouse) make certain action in a state. This function $S'(a,s)$, i.e., the set of possible next states after making action a and after the cat makes one move, is given by the matrix `Possibs`:

$$\texttt{Possibs}(s,s',a) = \begin{cases} 0 & \text{if } s' \notin S'(a,s) \\ 1 & \text{if } s' \in S'(a,s) \end{cases}$$

As in the random case, the problem is that the value function may have an infinite value, so we apply discounting as before.

The following formula holds:

$$U(s,\pi) = V_\pi(s) = R(s) + \gamma \min_{s' \in S'(\pi(s),s)} U(s',\pi)$$

How to evaluate this function?

This time, we cannot apply an exact method, but we can use the iterative method. However, while in the decision theory case the method was sound and guaranteed to converge, and to converge to the correct result, here we do not have that guarantees. Instead, this method, policy iteration, and value iteration that will be presented shortly are just heuristic methods that sometimes happen to work well.

The iterative method:

- start from arbitrary values for $V_\pi(s)$ and
- iterate using formula

$$V_\pi^{new}(s) = R(s) + \gamma \min_{s' \in S'(\pi(s),s)} V_\pi^{old}(s')$$

– if V_π^{new} and V_π are sufficiently close, then V_π^{new} is the solution, otherwise $V_\pi \leftarrow V_\pi^{new}$ and we repeat iteration.

Policy iteration

We go back to our initial problem of finding the optimal policy. The first approach to solving this problem is *policy iteration:*

1: generate random policy π
2: calculate value of the policy V_π
3: generate new policy by optimizing old values:
namely, for all states s
$\pi^{new}(s) \leftarrow \arg\max_a \left(\min_{s' \in S'(a,s)} V_\pi(s')\right)$
4: **if** $\pi = \pi^{new}$ **then**
5: **return** π^{new}
6: $\pi \leftarrow \pi^{new}$
7: **go to** 2

Note: When implementing these methods in Matlab, it is many times more efficient to use operations of `min` and `max` on matrices, than implementing them using for-loops.

Value iteration

Second approach to finding the optimal policy is to first find the optimal value function, without explicitly finding corresponding policies, and then find the policy that corresponds to this value function. Again, unlike in decision theory, this approach is not guaranteed to find an optimal policy in this case.

1: start with an arbitrary value function, e.g. $V(s) = \frac{1}{1-\gamma} \max_s R(s)$ is a reasonable choice.
2: **repeat**
3: $V^{old} \leftarrow V$
4: for all $s \in S$, $V \leftarrow R(s) + \gamma \max_a \min_{s' \in S'(a,s)} V^{old}(s')$
5: **until** V^{old} and V are close enough for all s

After finding the optimal value function, we find the optimal policy; for all $s \in S$:

$$\pi(s) \leftarrow \arg\max_a R(s) + \gamma \min_{s' \in S'(a,s)} V(s')$$

$$= \arg\max_a \min_{s' \in S'(a,s)} V(s')$$

Game playing

We consider a special case of a game:

- there are two players,
- reward $= 0$ except for leaf states, i.e., the states at which the game stops, and
- one player makes a move (action), then the other player plays, and so on
- at any leaf state one player's reward is r and the opponent's reward is $-r$, i.e. it is a *zero-sum* game

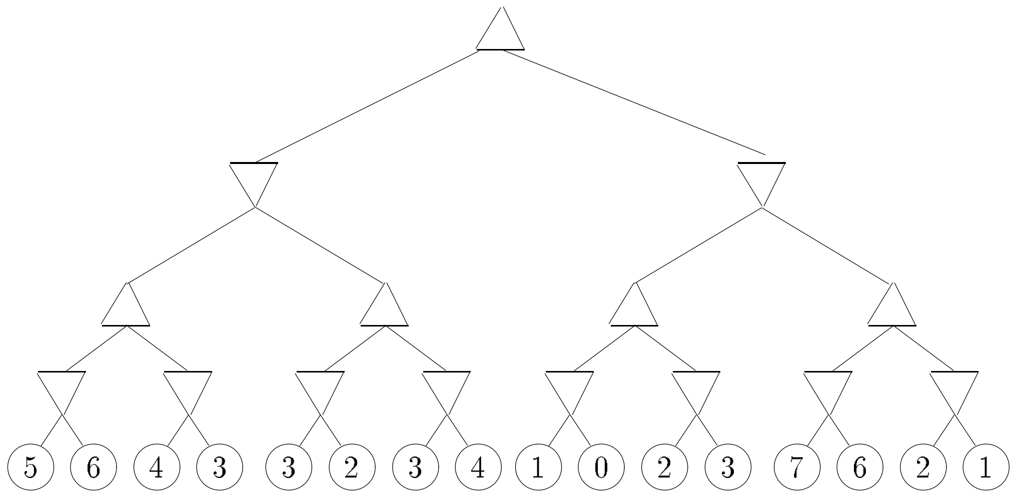
We assume that the state space is acyclic, but we will see that with some modification, it is possible to apply the following methods in cyclic games as well.

Minimax algorithm

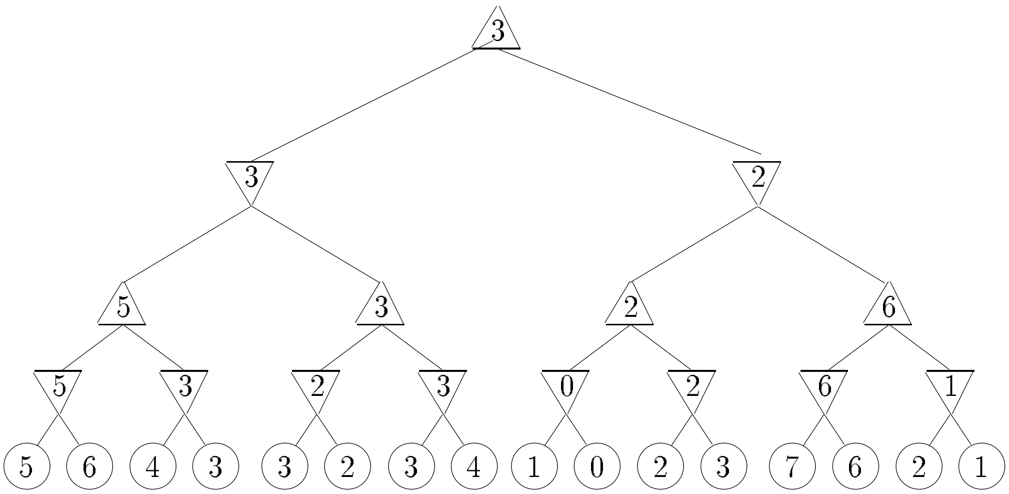
In a two person, zero-sum game, we can represent future possible states using a *game tree.* In a game tree, the root node represents the current position. Similarly to the search trees in problem solving, each possible action is represented with a branch to the next level of the tree, and the resulting situation is represented by the node at the other end of the branch, i.e., by a child node. The leaves of the tree represent the ending situations in the game, and they are labeled with our reward. Our goal is to achieve the highest reward, but the goal of our opponent is to reach the lowest reward for us (which is the highest reward for him/her).

The nodes in which we choose the move are called *max nodes* and they are depicted as $\triangle$, because we always choose the action, so that the maximal final reward is achieved. On the contrary, when our opponent chooses the move, the corresponding node is called a *min node* and it is depicted as $\bigtriangledown$, since the plan that the opponent will always choose the option that leads to the minimal reward for us (if she/he chooses some other option, it can just be better for us). If the game is acyclic, as we currently assume, it is always possible in principle to construct such a tree, although it may be too big to fit it in memory and it may take a long time to construct it.

An example of such a tree is the following:



If we start from bottom, i.e. from the leaves of the tree, and work our way up, we can easily predict what is the worst outcome of the game if we play the best strategy. For example, if opponent has to choose between rewards 5 and 6, the worst case for us is 5, and we mark 5. If we are choosing between two situations that can lead to the final reward 5 and 3 in the worst case, we always choose 5. Hence the above tree is labeled in the following way:



Using this tree, it is easy to come up with the best strategy. This is captured in the algorithms 10 and 11.

Algorithm 10 Minimax algorithm: $max\_value(s)$

Require: s is a max-node situation
Ensure: the max value and an optimal action is returned

```
if s is a leaf node then
    return (R(s), 'no action')
max ← −∞
for all possible actions a do
    (v, m) ← min_value(a(s))
    if v > max then
        max ← v
        opt_action ← a
return (max, opt_action)
```

When using these algorithms, we do not explicitly construct the tree. The game trees are typically very large, and in order to improve efficiency we can use various techniques, one of them being pruning.

Algorithm 11 Minimax algorithm: $min\_value(s)$

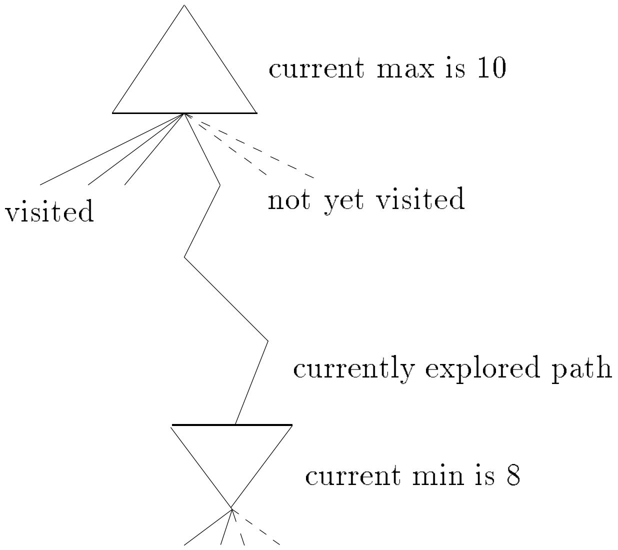
Require: s is a min-node situation
Ensure: the min value and an optimal action is returned

if s is a leaf node **then**
  return $(R(s),$ **'no action'**$)$
$min \leftarrow +\infty$
for all possible actions a **do**
  $(v, m) \leftarrow max\_value(a(s))$
  if $v < min$ **then**
    $min \leftarrow v$
    $opt\_action \leftarrow a$
return (min, opt_action)

α-β pruning

First note that in the minimax tree the values are propagated from leaves toward the root: a min child's value is chosen for a min node, and a max child's value for the max node.

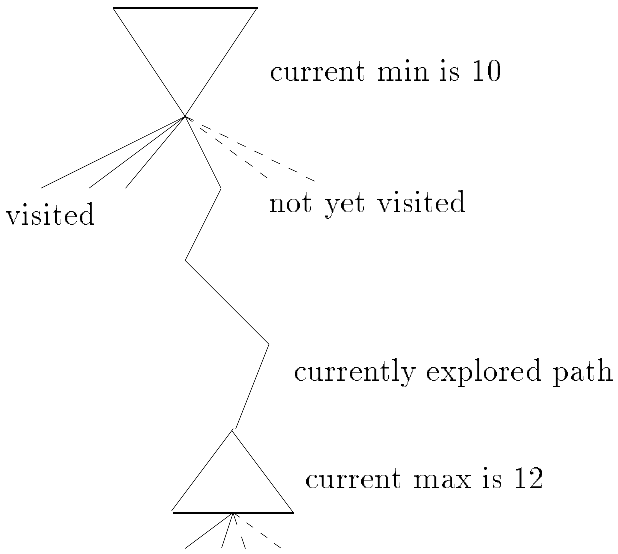
Consider the following situation:



Suppose that we are exploring a max node, we have explored some paths, and our current max is 10. Then, we are not interested in any rewards in the paths below this node that are less or equal 10 (which we denote α), since they cannot influence and make change in this node. For example, if, as in the above figure, we are exploring a min node, and the current min value is 8, then we should stop exploring other options from this node, since this node cannot possibly return a value greater than 8.

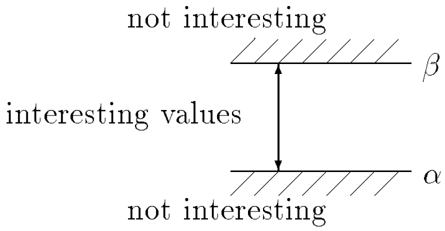
In other words, if we are exploring the best move for us in a situation, and we already have a move that guarantees us a reward of 10, then we should stop exploring any option which obviously cannot give us more than 10.

And similarly, if our opponent has an option in a min node that guarantees him/her the result 10 (denoted β), then it is redundant to explore optional paths that are obviously leading to a reward higher than 10. This is illustrated in the following figure:



In the above situation, we should stop searching for the max in the max node, since we already know that it will be greater than 10.

This method is called α-β pruning, and it is implemented by calculating the bounds α and β of "interesting" values:



Algorithms 12 and 13 describe operations in max and min nodes for α-β pruning.

Algorithm 12 α-β pruning algorithm: $alpha\_beta\_max(s, \alpha, \beta)$

Require: s is a max-node situation, α and β are boundaries
Ensure: the max value and an optimal action is returned, assuming given boundaries

```
if s is a leaf node then
    return (R(s), 'no action')
opt_action ← 'not important'
for all possible actions a do
    (v, m) ← alpha_beta_min(a(s), α, β)
    if v > α then
        α ← v
        opt_action ← a
        if α ≥ β then
            return (α, opt_action)
return (α, opt_action)
```

If we know the minimal and maximal value of the reward, i.e., $\min_s R(s)$ and $\max_s R(s)$, then those are the initial values for α and β; otherwise, $\alpha = -\infty$ and $\beta = +\infty$ initially.

The figure 1 illustrates what values of α and β are passed during the search, and which nodes are visited (circled ones), in α-β pruning (the initial values are $\alpha = -\infty$ and $\beta = +\infty$.

Improvements and variations

In some games, the sequence of states does not depend only on players' actions but also on come random result, like throwing dice. Backgammon is

Algorithm 13 α-β pruning algorithm: $alpha\_beta\_min(s, \alpha, \beta)$

Require: s is a min-node situation, α and β are boundaries

Ensure: the min value and an optimal action is returned, assuming given boundaries

```
if s is a leaf node then
    return (R(s), 'no action')
opt_action ← 'not important'
for all possible actions a do
    (v, m) ← alpha_beta_max(a(s), α, β)
    if v < β then
        β ← v
        opt_action ← a
        if α ≥ β then
            return (β, opt_action)
return (β, opt_action)
```

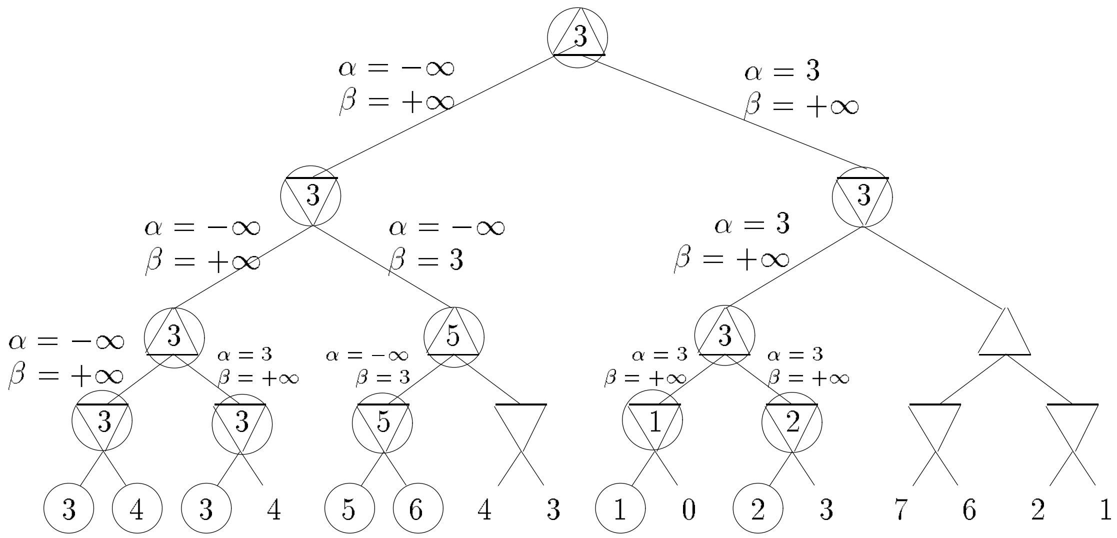


Figure 1: α-β pruning example

such a game. In this case, the game tree contains a third kind of nodes: chance nodes. The value in chance nodes in normally calculated as the expected value of the child-nodes' values, using the probability distribution of the random action. In this case, we use the terms *expecti-mini-max algorithm, expecti-mini-max policy,* and so on.

A problem with almost all practical games is that the search tree is too large. If the game is cyclic, than it is infinite. In any case, it is usually impossible in practice to search the whole tree. On improvement is to use *memoization,* i.e., keep a cache of calculated situations (positions). Additionally, memoization prevent us to end up in an infinite loop in cyclic state spaces.

The leaves of the search tree are usually to far away to be reached by our search algorithm, so we cannot get any values. This is the problem of *infinite horizon.* The problem can be tackled by limiting horizon: After getting at a certain depth we evaluate the position using some heuristic, and treat this value as the reward.

Readings

Russell and Norvig: Sections 5.1, 5.2, 5.5

20 Scaling up: Partial observability

What if the current state is uncertain?

The knowledge about the current state can be represented as a probability distribution over possible current states. This distribution is called *belief state*: $B(s)$

In this case, the policy is not defined as a function that maps a state to action, but a function that maps a belief state to action:

$$\pi : B(s) \mapsto a$$

The performance is better if we have a better idea what the current state is. We can use perceptions to get more input about the environment, in order to reduce uncertainty about the current state. Memory, i.e., history of previous belief states is also very useful in determining the belief state.

We distinguish three states in each moment:

- world state (true state)
- perceptual state (state perceived by senses)
- information state (perceptual + history)

Our decision is based in information state: information state $\mapsto$ action

In order to improve expected future rewards, we can act to

- directly achieve higher reward, or
- reduce uncertainty about current state, which can maximize future rewards in a long run.

Value of Information

Consider expected future 1-step reward, given the current state:

$$U(s, a) = \sum_{s'} \mathrm{P}(s'|s, a)R(s')$$

Expected future 1-step reward given a belief state is:

$$U(B(s), a) = \sum_{s} B(s)U(s, a) = \sum_{s} B(s) \sum_{s'} \mathrm{P}(s'|s, a)R(s')$$

In order to decide whether to take an action that will maximize the immediate reward, or to execute a "sensing" action, which will provide us with more information about the current state, we need to know what is the expected value of information that the sensing action can provide us. This is the *value of information.*

Typically, the sensing action will give us information whether the current state is in a set of states I or not, i.e., whether $s \in I$ or $s \in S \setminus I$. We want to calculate who much this information is worth.

The formula for value of information is:

$$\begin{aligned}\text{Value of information} &= \text{expected reward given information} - \\ &\quad\ \text{expected reward without information}\end{aligned}$$

The value of information whether $s \in I$ or not, can be calculated as follows:

$$\begin{aligned}\text{VOI} &= B(s \in I) \max_a U(B(s|s \in I), a) + \\ &\quad B(s \in S \setminus I) \max_a U(B(s|s \in S \setminus I), a) \\ &\quad - \max_a U(B(s), a)\end{aligned}$$

Using our current belief state B, we calculate expected reward if we know the information, and subtract the expected reward without information.

Example: coin

Suppose that we are making a \$1 bet on a coin toss.

First, assume that the coin is not biased, i.e., there is a 50% chance of getting head (H) and 50% chance of getting tail (T). The expected reward is the same, whether we bet on H or on T:

$$\text{expected reward} = 0.5 \cdot 1 + 0.5 \cdot (-1) = \$0$$

since there is the probability of 0.5 for winning \$1 and 0.5 for loosing \$1.

If someone can give us information about the outcome of the toss, then our expected reward is \$1, because we are winning for sure. The value of this information is:

$$\$1 - \$0 = \$1$$

On the other hand, if the coin is biased, giving H with probability 0.99, then by betting on H, our expected reward is:

$$0.99 \cdot 1 + 0.01 \cdot (-1) = \$0.98$$

The expected reward with information is \$1, so the value of the information in this case is:

$$\$1 - \$0.98 = 2 \text{ cents}$$

Example: 3 doors

Assume that we are playing a game: There are three doors and \$9999 reward is behind one door, with uniform probability distribution. There are two ways of playing the game:

1. we can simply pick a door, and we get the reward if the reward is behind this door, or
2. we can pick a door, then a person opens one of the two other doors, with no money behind it, and now we can stick with the previously chosen door or change to the other closed door.

An interesting question in the second scenario is whether we should stick with the previously chosen doors, or we should change our choice, or it does not matter. We are also interested in the value of information provided by opening the door in the second scenario.

The expected reward in the first scenario is:

$$\frac{1}{3} \cdot \$9999 + \frac{2}{3} \cdot 0 = \$3333$$

For the second scenario, let us denote doors A, B, and C, so that A is our first guess, and let us list all possible situations:

winning doors	change?	Reward
A	Yes	\$0
A	No	\$9999
B	Yes	\$9999
B	No	\$0
C	Yes	\$9999
C	No	\$0

Since the winning doors A, B, and C are equally likely, we see that the expected reward when changing the choice is:

$$\frac{1}{3} \cdot \$0 + \frac{1}{3} \cdot \$9999 + \frac{1}{3} \cdot \$9999 = \$6666$$

and the expected reward for not changing the selected door is:

$$\frac{1}{3} \cdot \$9999 + \frac{1}{3} \cdot \$0 + \frac{1}{3} \cdot \$0 = \$3333$$

We conclude that changing the choice is better. The value of information is:

$$\$6666 - \$3333 = \$3333$$

Readings

Russell and Norvig: Section 16.6
Dean, Allen, Aloimonos: Section 8.4

21 Types of learning problems

Part 5: Learning

Types of learning phenomena:

- gaining knowledge (machine thinking)
- discovering patterns (observing, experimenting)
- acquiring skills (practice, adjusting)
- development (maturation)
- evolution

Machine learning: Building systems that improve with experience.

There is a fundamental tradeoff between

- amount of experience
- amount of prior constraints

There are strong mathematical limits on these two parameters. (There is no such thing as "universal" learning.)

Example

We used an example of handwritten digit recognition using Bayesian networks. We did not go into details how the probability distributions for noise and shear distribution (`trans`) were determined. They can be learned from training data.

The Bayesian network that was used represented *prior constraints,* and the network parameters are learned from some training data, i.e., experience.

If prior constraints are reduced and experience is over-emphasized, then the model will fit the experience too well, and it will not be general enough to perform well on unseen data. This problem is known as *overfitting.*

If too much emphasis is put on prior constraints, then the problem may be that the model is too general and it does not fit well even the training data, and also the unseen data. This is the problem of *underfitting.*

Mathematical modelling

The tasks of mathematical modelling are:

- formalize learning environment: types of inputs/outputs to system
- formalize language for expressing constraints and assumptions
- formalize measures of learning performance
- design provably effective/efficient learning algorithms, or analyze capabilities of learning algorithms

Example: formalize learning environment

Data can be represented as a table with some columns representing input attributes, and some columns representing output attributes.

Example: language for expressing constraints

- learning function: linear, quadratic
- learning Bayesian network
- learning decision tree
- learning rules, version space algorithm
- learning neural networks

Example: formalize measures of learning performance

Measure:

- rate of improvement
- level of achievement

Evaluating a model:

- divide data into training and testing part (9:1)
- train model on training part
- test on the testing part
- held-out or validation data, cross-validation

Some Applications of Machine Learning

- data mining: using historical data to improve decisions; e.g., medical records, corporate databases
- software applications we cannot program by hand: speech recognition, driving
- self-customizing programs: spam-filter, news reader that learns user interests

Examples

1 Checkers player

Arthur Samuel wrote a program to learn to play checkers in 50s.

The program learned to evaluate board situation. The evaluation function used a weighted combination of board features. For example, some of the features can be:

- f_1 = number of black pieces
- f_2 = number of red pieces
- f_3 = number of black kings
- f_4 = number of red kings
- f_5 = number of threatened red pieces
- f_6 = number of threatened black pieces

If w_i are some weights, then the evaluation function is calculated as the linear combination $\sum_i f_i w_i$.

The program played against a previous version of self. The weights are updated along an optimal path. E.g., if an optimal minimax path (after searching up to a certain depth) leads from situation x to y, then the values of the evaluation function for x and y should be the same. If they are not the same, we treat the value in y as the correct value (y is further from the root), and update the weights a bit so that the value of x is close to the value of y.

Win and loss values at leaves are backed up.

Others:

Backgammon: Tesauro 80s 90s

Chess: Baxter 98

2 Neural network zip code reader

- in 80s and 90s
- learn to automatically read zip codes on envelopes
- neural network, trained on a large corpus of manually labeled examples:

$$\begin{array}{ll} \text{image}_1 & \text{label}_1 \\ \text{image}_2 & \text{label}_2 \\ \vdots & \vdots \end{array}$$

Others:

- cheque readers
- handwriting recognition
- face recognition
- tracking

3 Speeding up automated problem solvers

- 80s
- learn to speed up performance of working problem solver
- How? Learn search control:
  - how to rank alternatives
  - how to prune alternatives
  - learn macro steps

Others:

- transaction processing
- database query processing

4 Robot exploration

- autonomous exploration and mapping of an unknown environment
- How? Remember landmarks, topology, explore unknown regions.

5 Industrial plant optimization

- optimize operating parameters of a plant while in operation
- e.g. pulp mill:
  - roller pressure
  - conveyor rate
- How: very tricky, continuously explore parameter space

Fundamental types of learning problems

A learning system maps: experience $\mapsto$ a base performance system.

The set of base systems = set of hypotheses, or *hypothesis space.*

1 Batch vs online

- **batch:**
  - separate training then testing phase
  - pure exploration during training
- **on-line:**
  - no separate training phase
  - learn while doing
  - tradeoff between acting to do well vs. acting to gain information

2 Supervised vs. reinforcement vs. unsupervised learning

It is useful for learner to know how well it performs on a data. The available performance information is called *feedback.*

Feedback amount:

supervised learning: inputs and outputs are available

reinforcement learning: only *reinforcement* is available, i.e., some indicator of performance (reward or punishment)

unsupervised learning: no feedback

3 Complete vs. partial vs. point-wise feedback

complete:
- every item of experience evaluates every candidate base system
- e.g., base system is set of functions $f : X \to Y$, the training data (experience) is the set of observations (x, y), each observation gives feedback about each system, i.e. function

partial:
- items of experience only evaluate a subset of possible candidate base systems
- e.g.,
  - base system: controller $\pi : S \to A$
  - experience is a set of triples (s, a, r)
  - an experience item only evaluates controllers π that would have taken action a in state s

point-wise:
- each item of experience evaluates only one base system

4 Passive vs. active learning

passive learning: experience is given, we cannot chose the inputs; observation

active learning: we can choose the input for which the output will be provided; experimentation

5 Stationary vs. non-stationary environment

stationary: evaluation does not change over time

non-stationary: evaluation changes over time

6 Acausal vs. causal learning

acausal learning: actions do not affect future experience

causal learning: action do affect future experience

Readings

Dean, Allen, Aloimonos: Sections 5.1, 5.2
Russell and Norvig: Sections 18.1, 18.2, 18.6
Ripley: Chapter 1

22 Function learning algorithms

Case study: learning a function from examples

The set of hypotheses is a class of parametrized functions; e.g., linear functions.

An experience, i.e., training data is a set of pairs $(\mathbf{x}_1, y_1)$, $(\mathbf{x}_2, y_2)$, $\ldots$, $(\mathbf{x}_t, y_t)$.

$$\begin{matrix} \mathbf{x}_1, y_1 \\ \mathbf{x}_2, y_2 \\ \vdots \\ \mathbf{x}_t, y_t \end{matrix} \quad \Rightarrow \quad \boxed{\text{Learner}} \Rightarrow f : X \to Y$$

Our task is to extrapolate observed y's over all x's, and predict a value $\hat{y}$ for any x.

A prediction y is evaluated by the error function $\text{err}(\hat{y}, y)$, where y is the correct value.

The function err can be expressed in different ways. For example, for the task of *classification* we use:

$$\text{err}(\hat{y}, y) = \begin{cases} 0 & \hat{y} = y \\ 1 & \hat{y} \neq y \end{cases}$$

For regression, one can use:

$$\text{err}(\hat{y}, y) = (\hat{y} - y)^2$$

In any case, we want to minimize the prediction error $\text{err}(h(x), y)$ on test examples (x, y).

A generic algorithm for learning can be described as follows:

1. fix hypothesis space $H = \{h\}$
2. given $(x_1, y_1) \ldots (x_t, y_t)$ calculate $h^* \in H$ that minimizes

$$\frac{1}{t} \sum_{i=1}^{t} \text{err}(h(x_i), y_i)$$

the average training error.

Special case: real valued prediction

Learn $h : \mathbb{R}^n \to \mathbb{R}$ to minimize the prediction error:

$$\mathrm{err}(\hat{y}, y) = (\hat{y} - y)^2$$

We assume that the set of hypotheses H is

$$H = \{\text{linear functions}\} = \{h_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^n\}$$

where $h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^{n} w_i x_i$.
We are given the training data:

$$\begin{array}{cccc|c}
x_{11} & x_{12} & \dots & x_{1n} & y_1 \\
x_{21} & x_{22} & \dots & x_{2n} & y_2 \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
x_{t1} & x_{t2} & \dots & x_{tn} & y_t
\end{array}$$

We want to calculate $\mathbf{w}$ that minimizes SSE (sum of square errors):

$$\mathrm{SSE} = \sum_{i=1}^{t} (\mathbf{w} \cdot \mathbf{x}_i - y_i)^2$$

where $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{in})$.
This can be done by finding partial derivations $\frac{\partial}{\partial w_j}\mathrm{SSE}$ and by solving the system of equations $\frac{\partial}{\partial w_j}\mathrm{SSE} = 0$, for $j = 1 \dots n$:

$$\begin{aligned}
\frac{\partial}{\partial w_j}\mathrm{SSE} &= \sum_{i=1}^{t} 2(\mathbf{w} \cdot \mathbf{x}_i - y_i) x_{ij} \\
&= 2\left(\sum_{i=1}^{t} \sum_{k=1}^{n} w_k x_{ik} x_{ij} - \sum_{i=1}^{t} y_i x_{ij} \right)
\end{aligned}$$

If we use the notation $\nabla_{\mathbf{w}}\mathrm{SSE}$ for the vector of these partial derivations, X for the matrix whose rows are vectors $x_1, x_2, \dots, x_t$, and $\mathbf{y}$ for the vector of values y_i, then the above expression can be written in a matrix form in the following way:

$$
\begin{aligned}
\nabla_{\mathbf{w}}\mathrm{SSE} &= \begin{bmatrix} \frac{\partial}{\partial w_1}\mathrm{SSE} \\ \frac{\partial}{\partial w_2}\mathrm{SSE} \\ \vdots \\ \frac{\partial}{\partial w_n}\mathrm{SSE} \end{bmatrix} = \begin{bmatrix} 2\left(\sum_{i=1}^{t}\sum_{k=1}^{n} w_k x_{ik} x_{i1} - \sum_{i=1}^{t} y_i x_{i1}\right) \\ 2\left(\sum_{i=1}^{t}\sum_{k=1}^{n} w_k x_{ik} x_{i2} - \sum_{i=1}^{t} y_i x_{i2}\right) \\ \vdots \\ 2\left(\sum_{i=1}^{t}\sum_{k=1}^{n} w_k x_{ik} x_{in} - \sum_{i=1}^{t} y_i x_{in}\right) \end{bmatrix} \\
&= 2\begin{bmatrix} \sum_{i=1}^{t}\sum_{k=1}^{n} w_k x_{ik} x_{i1} \\ \sum_{i=1}^{t}\sum_{k=1}^{n} w_k x_{ik} x_{i2} \\ \vdots \\ \sum_{i=1}^{t}\sum_{k=1}^{n} w_k x_{ik} x_{in} \end{bmatrix} - 2\begin{bmatrix} \sum_{i=1}^{t} y_i x_{i1} \\ \sum_{i=1}^{t} y_i x_{i2} \\ \vdots \\ \sum_{i=1}^{t} y_i x_{in} \end{bmatrix} \\
&= 2X^T\begin{bmatrix} \sum_{k=1}^{n} w_k x_{1k} \\ \sum_{k=1}^{n} w_k x_{2k} \\ \vdots \\ \sum_{k=1}^{n} w_k x_{tk} \end{bmatrix} - 2X^T\mathbf{y} \\
&= 2X^T X\mathbf{w} - 2X^T\mathbf{y}
\end{aligned}
$$

Hence, we just have to solve the system of linear equations $2X^TX\mathbf{w} - 2X^T\mathbf{y} = 0$ for $\mathbf{w}$: $X^TX\mathbf{w} = X^T\mathbf{y}$, $\mathbf{w} = (X^TX)^{-1}X^T\mathbf{y}$, or in Matlab $\mathbf{w} = (X^TX)/(X^T\mathbf{y})$.

$\mathbf{w}$ is unique if the columns of X are independent.

Geometrical interpretation

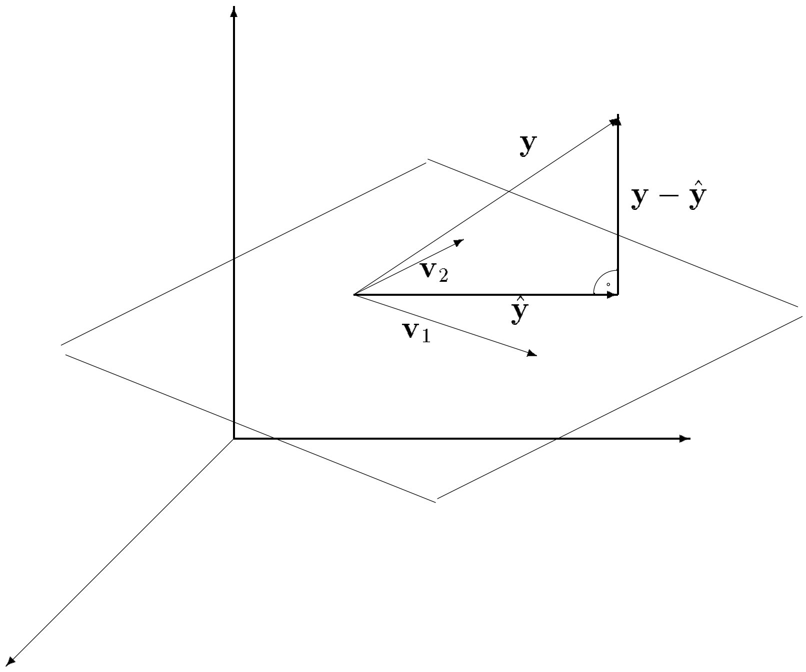
We can think of columns of X as vectors: $\mathbf{v}_1 \dots \mathbf{v}_n$

$$\left[\begin{bmatrix} \\ \mathbf{v}_1 \\ \\ \end{bmatrix} \begin{bmatrix} \\ \mathbf{v}_2 \\ \\ \end{bmatrix} \dots \begin{bmatrix} \\ \mathbf{v}_n \\ \\ \end{bmatrix} \right]$$

Looking for the best weights $\mathbf{w}$ can be interpreted as looking for linear combination of columns that best approximates $\mathbf{y}$:

$$\hat{y} = w_1\mathbf{v}_1 + w_2\mathbf{v}_2 + \dots + w_n\mathbf{v}_n$$

Any such $\hat{y}$ lies in span$(\mathbf{v}_1, \dots, \mathbf{v}_n)$. The element $\hat{\mathbf{y}}$ of span$(\mathbf{v}_1, \dots, \mathbf{v}_n)$ that minimizes $||\hat{\mathbf{y}}-\mathbf{y}||^2$ is given by orthogonal projection of $\mathbf{y}$ onto span$(\mathbf{v}_1, \dots, \mathbf{v}_n)$. This can be illustrated in a 3-dimensional space in the following way:



Since the vector $\mathbf{y} - \hat{\mathbf{y}}$ is orthogonal onto span$(\mathbf{v}_1, \dots, \mathbf{v}_n)$ (plane $\mathbf{v}_1, \mathbf{v}_2$ in figure), the inner products $\mathbf{v}_i(\mathbf{y} - \hat{\mathbf{y}})$ are equal to 0, i.e., $\mathbf{v}_i \cdot \hat{\mathbf{y}} = \mathbf{v}_i \cdot \mathbf{y}$, for all $i = 1 \dots n$. Vectors $\mathbf{v}_i$ a columns of matrix X, so these equalities can be written as $X^T\hat{\mathbf{y}} = X^T\mathbf{y}$. The estimate $\hat{\mathbf{y}}$ is linear combination $\sum_{i=1}^n \mathbf{v}_i w_i = X\mathbf{w}$, so we finally get $X^T X\mathbf{w} = X^T\mathbf{y}$.

Other error functions

Sum of square errors is not always the best; options:

$$
\begin{array}{rcll}
\mathrm{err}(\hat{\mathbf{y}}, \mathbf{y}) & = & \sum_i (\hat{y}_i - y_i)^2 & L_2 \\
& & \sum_i |\hat{y}_i - y_i| & L_1 \\
& & \sum_i |\hat{y}_i - y_i|^p & L_p \\
& & \max_i |\hat{y}_i - y_i| & L_\infty
\end{array}
$$

Example: Minimize maximal training error (L_∞)

- choose $\mathbf{w}$ to minimize $\max_i |\mathbf{w} \cdot \mathbf{x}_i - y_i|$
- linear programming: find $\mathbf{w}, \delta$ to minimize δ, subject to constraints

$$
\begin{array}{rcl}
\mathbf{w} \cdot \mathbf{x}_1 & \leq & y_1 + \delta \\
\mathbf{w} \cdot \mathbf{x}_1 & \geq & y_1 - \delta \\
\mathbf{w} \cdot \mathbf{x}_2 & \leq & y_2 + \delta \\
\mathbf{w} \cdot \mathbf{x}_2 & \geq & y_2 - \delta \\
\vdots & \vdots & \vdots \\
\mathbf{w} \cdot \mathbf{x}_t & \leq & y_t + \delta \\
\mathbf{w} \cdot \mathbf{x}_t & \geq & y_t - \delta
\end{array}
$$

- can be done in polynomial time

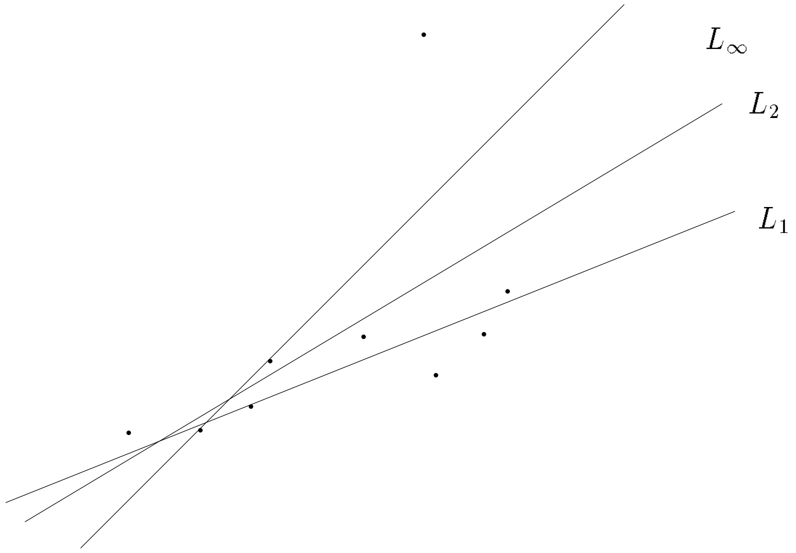
Example: Minimize sum of absolute error (L_1)

- choose w to minimize $\sum_{i=1}^t |\mathbf{w} \cdot \mathbf{x}_i - y_i|$
- linear programming: choose $\mathbf{w}, \boldsymbol{\delta}$ to minimize $\sum_{i=1}^t |\delta_i|$ subject to

$$
\begin{array}{rcl}
\mathbf{w} \cdot \mathbf{x}_1 + \delta_1 & = & y_1 \\
\mathbf{w} \cdot \mathbf{x}_2 + \delta_2 & = & y_2 \\
& \vdots & \\
\mathbf{w} \cdot \mathbf{x}_t + \delta_t & = & y_t
\end{array}
$$

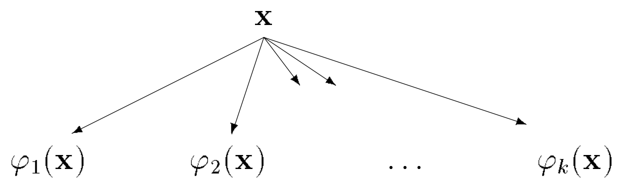
Which objective is the best?

- L_1 is the most robust against outliers
- L_2 is cheapest computationally
- L_∞ gives the best upper bound on errors in training set



Generalized linear functions

Linear functions may not be expressive enough. However, we can still use almost directly the above method by using a trick: *expanded representation:* $\mathbf{x} \mapsto \boldsymbol{\varphi}(\mathbf{x})$, i.e.,



We simply define new attributes, which are nonlinear functions of original attributes, and again learn the best linear combination of the new extended set of attributes.

The functions φ_1, φ_2, ... , φ_k are called *basis functions.*

The training set is expanded in the following way:

$$\left[\begin{array}{ccc|c} x_{11} & \dots & x_{1n} & y_1 \\ x_{21} & \dots & x_{2n} & y_2 \\ \vdots & \ddots & \vdots & \vdots \\ x_{t1} & \dots & x_{tn} & y_t \end{array}\right]$$

$$\Downarrow$$

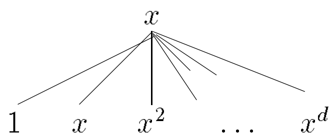
$$\begin{array}{c} \left[\begin{array}{ccc|c} \varphi_1(\mathbf{x}_1) & \dots & \varphi_k(\mathbf{x}_1) & y_1 \\ \varphi_1(\mathbf{x}_2) & \dots & \varphi_k(\mathbf{x}_2) & y_2 \\ \vdots & \ddots & \vdots & \vdots \\ \varphi_1(\mathbf{x}_t) & \dots & \varphi_k(\mathbf{x}_t) & y_t \end{array}\right] \\ \qquad\quad \mathbf{\Phi} \qquad\qquad\qquad \mathbf{Y} \end{array}$$

We learn a linear function over expanded representation, which is non-linear in the original representation.

For example, if we use L_2 error, we have to solve for $\mathbf{w}$: $\Phi^T\Phi\mathbf{w} = \Phi^T\mathbf{y}$.

Example: Polynomial regression

Let $X = \mathbb{R}$, and we use the expanded set of all powers of x, up to the power d:



$$\begin{array}{c} (x_1, y_1) \\ \vdots \\ (x_t, y_t) \end{array} \Rightarrow \left[\begin{array}{ccccc|c} 1 & x_1 & x_1^2 & \dots & x_1^d & y_1 \\ 1 & x_2 & x_2^2 & \dots & x_2^d & y_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 & x_t & x_t^2 & \dots & x_t^d & y_t \end{array}\right]$$

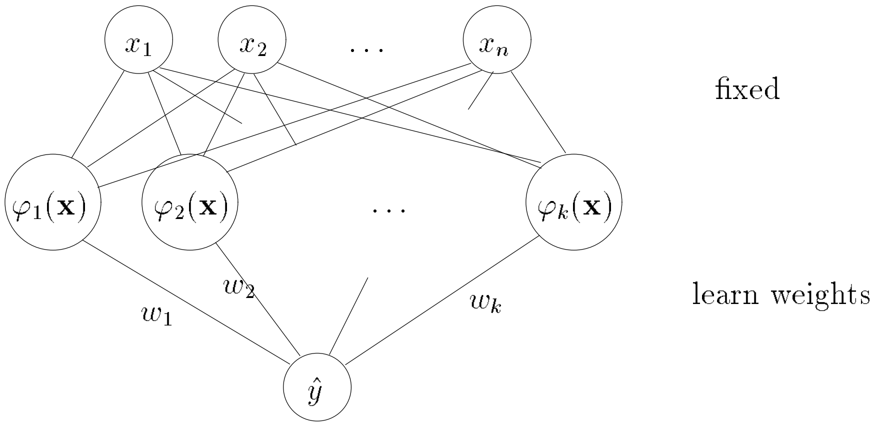
To find the best polynomial, we solve for $\mathbf{w}$ in $\Phi^T\Phi\mathbf{w} = \Phi^T\mathbf{y}$, and obtain $\mathbf{w}^*$, the coefficients of minimal squared error polynomial of degree d.

Other basis functions

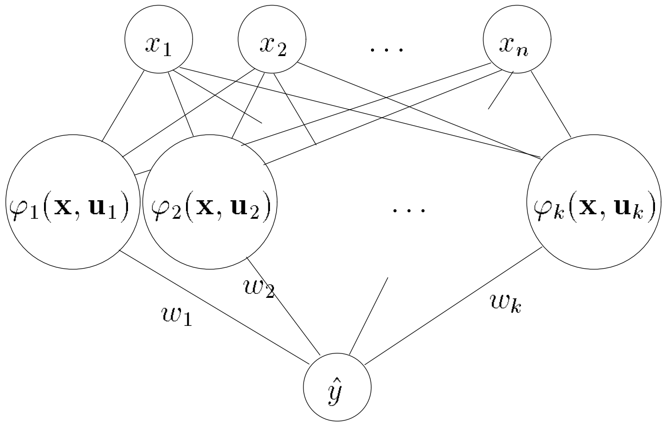
- spline fitting (basis splines)
- radial basis function
- Fourier analysis

Neural networks

The generalized linear regression can be depicted in the following way:



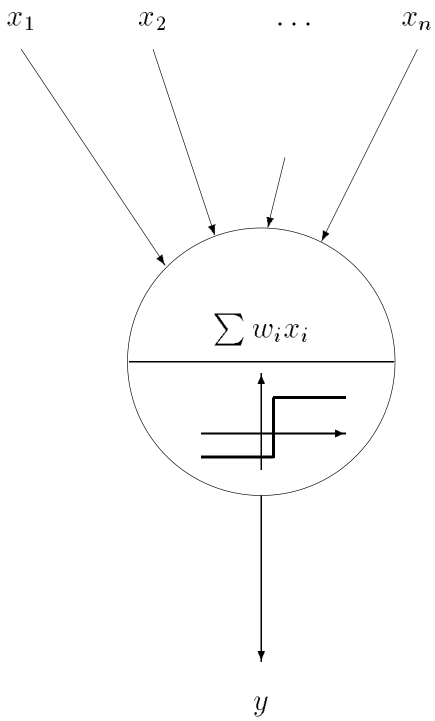
But beside final weights, we could also learn functions φ_i, if we parameterize them. The following figure corresponds to this case:



Now, we have to learn weights $\mathbf{w}$, $U = [\mathbf{u}_1; \ldots; \mathbf{u}_k]$ to minimize SSE. This problem is NP-hard: there is no efficient general procedure. We can use use gradient descent optimization in $\mathbf{w}, U$ (heuristic local minimum).

The actual function φ_i, i.e., the nodes in neural networks, are designed so that they loosely imitate the nerve cells or neurons in living organisms. A neuron has a number of fibers that act as inputs of signals, which are called dentrites, and a long "output" fiber called axon. If combined input on dentrites is higher than a threshold value, an axon signal is activated, which serves as an input to other neurons.

Following this simplistic principle, a node in a neural network is designed in the following way:

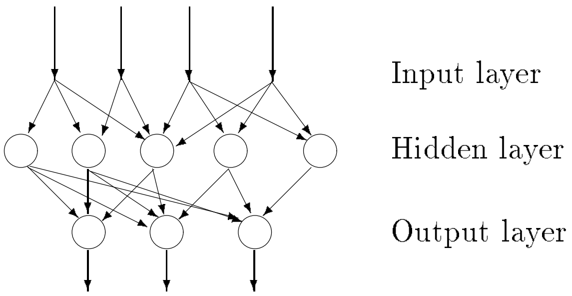


So, the output is calculated by taking a linear combination of inputs, and by applying a step activation function on this linear combination, i.e.,

$$y = f(\sum w_i x_i)$$

where f is an activation function.

A network is created by taking input from a set of variables, which are called the *input layer.* The input layer provides input for the second layer of nodes. The second layer is called the *output layer,* if their outputs are outputs of the network. However, we can have one or more layers of *hidden nodes,* before the signal reaches the output nodes, as in the following figure:



This kind of neural networks is called *feed-forward* networks because there are no cycles in node connectivity. The networks with cycles are called *recurrent neural networks.*

The activation function should be a *step* ($f = 0$ for $x < r$ and $f = 1$ otherwise), or a *sign* function ($f = -1$ for $x < r$ and $f = 1$ otherwise). One problem is whether the threshold value r is fixed, or it is also learned. Let us assume that it is fixed and equal 0.

The problem of learning the weights of a neural network is an optimization problem: Find the weights such that the sum of square errors is minimal. However, this problem is NP-hard, and a gradient descent optimization, which is not guaranteed to return a global optimal solution, is usually applied.

In order to apply a gradient descent optimization, the activation function had to be differentiable so the step or sign activation function is replaced by a function that approximates them: *sigmoid function:*

$$f(x) = \frac{1}{1 + e^{-x}}$$

A more general form is used:

$$f(x) = \frac{1}{1 + e^{-\lambda x}}$$

The sigmoid function has several good properties for using it in a neural node. One of them is that its derivative can be simply expressed using its value: $f'(x) = f(x)(1 - f(x))$.

The best-known algorithm for learning neural networks is a gradient descent algorithm called the *back-propagation algorithm.* It uses training examples of network inputs and outputs to update the network weights using the chain rule of differentiation.

The main idea with gradient descent (or climbing) is that when searching for a minimum or maximum of a function, we should move our current point in the direction of the function gradient (vector consisting of partial derivatives). The gradient direction is used for finding a maximum, and the opposite direction for finding a minimum.

Let us see how we can update a weight $w_{j\to i}$ of a link between nodes n_j and n_i. Let us assume that the node n_i is the output node. We use a training item to calculate the network output, and node n_j produces output $\hat{y}_j$ and node n_i produces $\hat{y}_i$. The correct output is y_i. Our goal is to adjust the weight $w_{j\to i}$ so that the error:

$$\sum_i (y_i - \hat{y}_i)^2$$

is minimal. We calculate partial derivative:

$$\frac{\partial}{\partial w_{j\to i}} \sum_i (y_i - f(\sum_j w_{j\to i}\hat{y}_j))^2 = -2(y_i - \hat{y}_i)\hat{y}'_i\hat{y}_j$$

where $\hat{y}'_i = f'(\sum_j w_{j\to i}\hat{y}_j)$. If we denote $\Delta_i = y_i - \hat{y}_i$, then using gradient descent we update the weight:

$$w_{j\to i} \leftarrow w_{j\to i} + \alpha\Delta_i\hat{y}'_i\hat{y}_j$$

The small positive constant α is the *learning rate.*

Let us see how we can update weights of hidden nodes. Let us use index i for output nodes, index j for preceding layer of hidden nodes, and index k for the next preceding layer. We want to calculate update of the weight $w_{k\to j}$:

$$\begin{aligned}
\frac{\partial}{\partial w_{k\to j}} \sum_i (y_i - f(\sum_j w_{j\to i}\hat{y}_j))^2 &= -2\sum_i (y_i - \hat{y}_i)\hat{y}'_i \frac{\partial \sum_j x_j w_{j\to i}}{\partial w_{k\to j}} \\
&\quad -2\sum_i (y_i - \hat{y}_i)\hat{y}'_i w_{j\to i}\hat{y}'_j y_k
\end{aligned}$$

If we denote

$$\Delta_j = \sum_i (y_i - \hat{y}_i)\hat{y}_i' w_{j\to i}$$

i.e.,

$$\Delta_j = \sum_i \Delta_i \hat{y}_i' w_{j\to i}$$

then the weight update has the same form as for the output nodes:

$$w_{k\to j} \leftarrow w_{k\to j} + \alpha \Delta_j \hat{y}_j' \hat{y}_k$$

The weights of other layers are updated in the same way. The value Δ_i is error the output nodes. The above formula shows how this error is calculated for other layers. The calculation of the error is done by starting from the output layer, and then moving towards the input layer—thus the name *error back propagation.*

The process is repeated for each training item.

Readings

Russell and Norvig: Chapter 19, Section 18.4
Dean, Allen, Aloimonos: Sections 5.5, 5.6, 5.8

23 Generalization theory / Overfitting

Generalization problem:

> How well does learned function predict on future test examples?

How do we choose hypothesis space H?

If H is too complex:

- overfitting problem
- small training error
- large test error
- very different functions have similar training error
- perturbing training data slightly yields very different optimal hypotheses

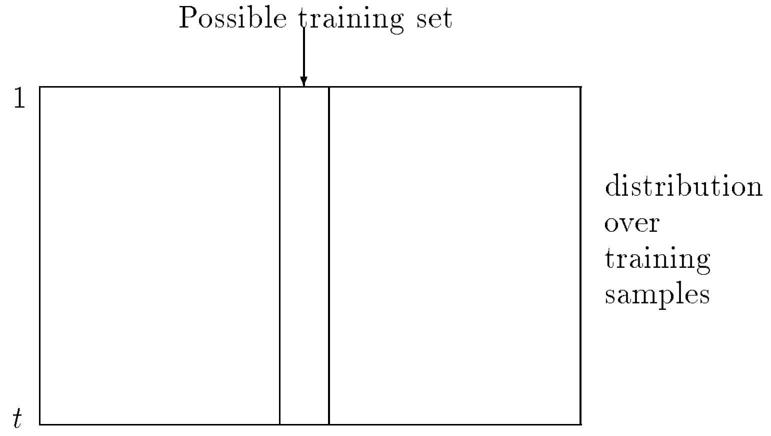
If H is too restricted:

- underfitting problem
- large training error
- large test error

Introduction to statistical generalization theory

Mathematical model: independent identically distributed (IID) random examples: Training examples (x_1, y_1), ... , (x_t, y_t), and testing example are independently drawn from a fixed joint distribution P_{xy} over $X \times Y$.

Learner maps the vector of training values $(x_1, y_1) \dots (x_t, y_t)$ to a hypothesis $h : X \to Y$.



$$\Downarrow$$
$$\boxed{\text{Learner}}$$
$$\Downarrow$$
$$h_{\mathbf{xy}} \qquad \text{distribution over hypotheses}$$
$$\Downarrow$$
$$E_{xy}\,\mathrm{err}(h_{\mathbf{xy}}(x), y) \qquad \text{expected test error}$$
$$\hat{E}_{x_iy_i}\mathrm{err}(h_{\mathbf{xy}}(x_i), y_i) \qquad \text{training error}$$

For squared prediction error:

$$\mathrm{err}(\hat{y}, y) = (\hat{y} - y)^2$$

we get expected hypothesis testing error, for a given hypothesis space H:

$$\begin{aligned} E_{\mathbf{xy}}\,E_{xy}(h_{\mathbf{xy}}(x) - y)^2 &= E_{\mathbf{xy}}\,\hat{E}_{x_iy_i}(h_{\mathbf{xy}}(x_i) - y_i)^2 + \\ & \quad E_{\mathbf{xy}}\,\hat{E}_{x_i}(h_{\mathbf{xy}}(x_i) - h^*(x_i))^2 + \\ & \quad E_{\mathbf{xy}}\,E_x(h_{\mathbf{xy}}(x) - h^*(x))^2 \end{aligned}$$

where $h^* = \arg\min E_{xy}(h(x) - y)^2$, i.e., it is the optimal hypothesis within the hypothesis space H.

The first term $E_{\mathbf{xy}}\hat{E}_{x_iy_i}(h_{\mathbf{xy}}(x_i) - y_i)^2$ is the expected *training error,* i.e., the error that we obtain when we compare the learned hypothesis with respect to the training set and calculate the average error with respect to the same set.

The second term $E_{\mathbf{xy}}\hat{E}_{x_i}(h_{\mathbf{xy}}(x_i) - h^*(x_i))^2$ is *training variance.* The sum of the first two terms

$$E_{\mathbf{xy}}\hat{E}_{x_i y_i}(h_{\mathbf{xy}}(x_i) - y_i)^2 + E_{\mathbf{xy}}\hat{E}_{x_i}(h_{\mathbf{xy}}(x_i) - h^*(x_i))^2$$

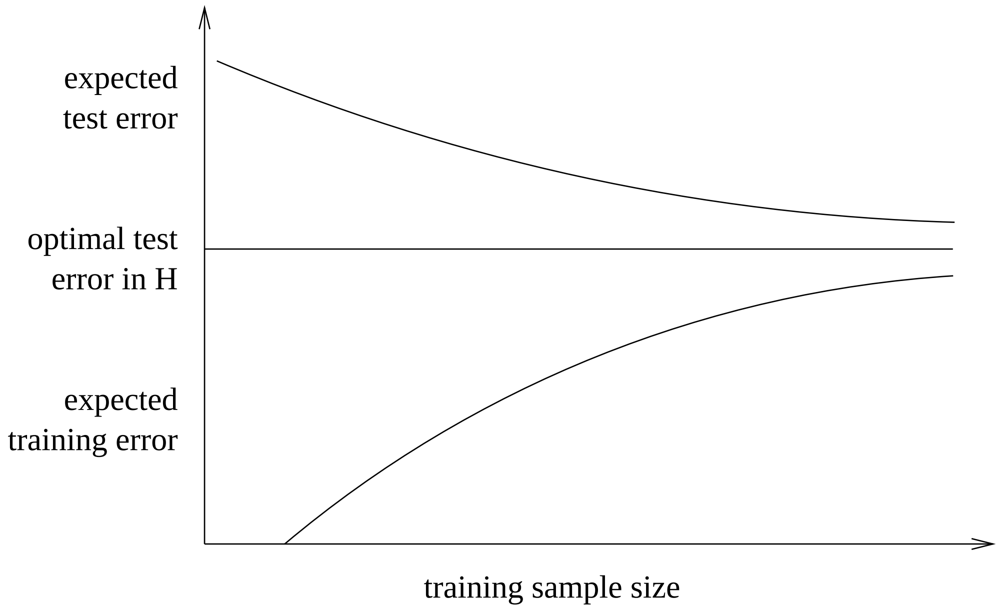
is the *optimal testing error in H.*

The last term $E_{\mathbf{xy}}E_x(h_{\mathbf{xy}}(x) - h^*(x))^2$ is *variance.* The sum of all three terms is the expected *hypothesis testing error.*

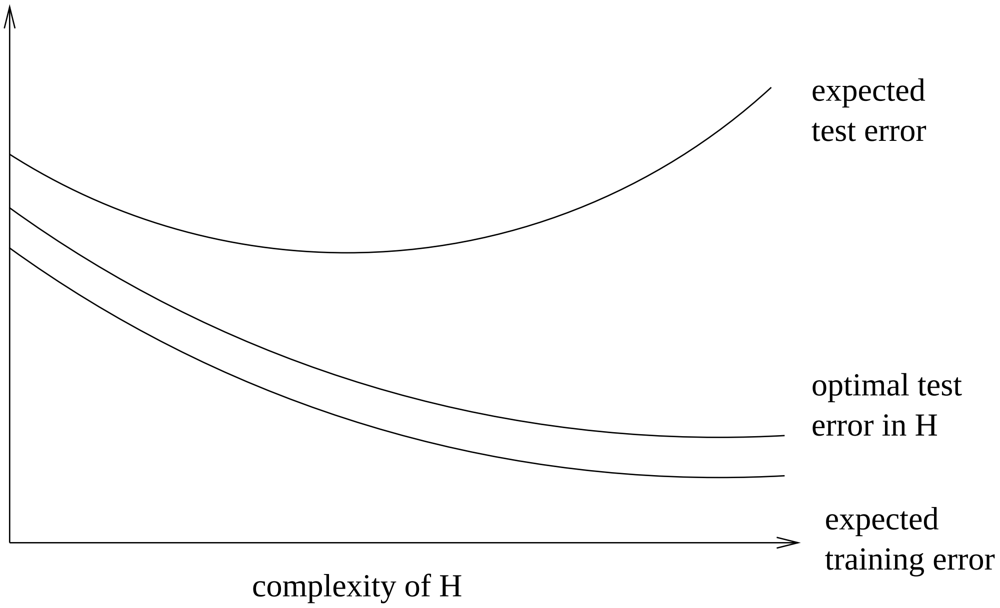
An immediate consequence is:

$$\begin{aligned} \text{expected hypothesis testing error} &\geq \text{optimal testing error in } H \\ &\geq \text{expected hypothesis training error} \end{aligned}$$

Learning curves

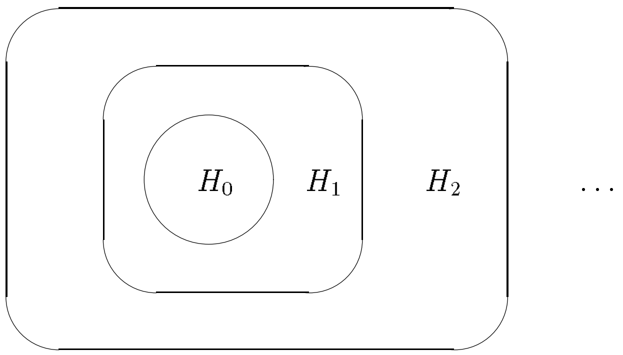


Overfitting curves



Automatic complexity control

Model selection:



How to choose the right complexity level?

With given data, for each hypothesis space H_0, H_1, ..., we find an optimal hypothesis h_0, h_1, h_2, ..., with decreasing training errors.

Which hypothesis to choose?

If we choose too early $\rightarrow$ underfitting, too late $\rightarrow$ overfitting.

Strategy 1: Complexity penalization

The difference between expected testing error and training error is in variance terms. We can try to guess those terms by introducing *penalty term.*

Penalty term = approximate variance at certain complexity.

For each hypothesis space H_i, we have

$$\text{training\_error}(i) + \text{penalty}(i)$$

Choose hypothesis space for which this expression is minimal.

Strategy 2: Hold out testing

Split the training data temporarily into a training part and testing part. The testing part is smaller than training part. This temporary testing part is called *held out* data.

For each hypothesis space, learn the hypothesis using temporary training data, and estimate testing error using held out data. This test gives an unbiased estimate of true test error.

Once we pick the hypothesis space with minimal estimated testing error, we use all training data to find the hypothesis.

Strategy 3: Metric space

Learning the best hypothesis can be described as estimating the conditional probability $\mathrm{P}(y|x)$.

Let us assume that we know P_x (it can be estimated from unlabeled data: $x_1, x_2, \ldots, x_t$).

Define metric on H:

$$d(h,g) = \sqrt{\int_x (h(x) - g(x))^2 d\mathrm{P}_x}$$

This metric gives an average distance between two hypotheses. We also define metric:

$$d(h,\mathrm{P}_{y|x}) = \sqrt{\int_x \int_y (h(x) - y)^2 d\mathrm{P}_{y|x} d\mathrm{P}_x}$$

Our goal is to minimize $d(h, \mathrm{P}_{y|x})$.

This metric satisfies metric space conditions, and triangular inequality among them. Hence, if h_i and h_j are two hypotheses, then

$$d(h_i,h_j) \le d(h_i,\mathrm{P}_{y|x}) + d(h_j,\mathrm{P}_{y|x})$$

The following simple procedure can be used to determine the optimal hypothesis space:

1. for $i = 0, 1, 2 \ldots$
2. calculate $d(h_i, h_{i+1})$ using known distribution P_x
3. estimate $d(h_i, \mathrm{P}_{y|x})$ and $d(h_{i+1}, \mathrm{P}_{y|x})$ as training errors for h_i and h_{i+1}
4. if triangular inequality

   $$d(h_i,h_{i+1}) \le d(h_i,\mathrm{P}_{y|x}) + d(h_{i+1},\mathrm{P}_{y|x})$$

   is not satisfied, stop and return h_i.

The method can be improved by adjusting $\hat{d}(h_i, P_{y|x})$ by multiplying it with $\max_{j<i} \frac{d(h_i,h_j)}{\hat{d}(h_i,h_j)}$.

Readings

Schuurmans, Dale. 1997. A new metric-based approach to model selection. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97).*

24 Course review

Part 1: Reasoning
Part 2: Knowledge (No)
Part 3: Interpretation
Part 4: Behaviour
Part 5: Learning

Part 1: Reasoning

2. Formal inference

- propositional logic
- inference system: inference rule + set of facts (axioms)
- $A \vdash \gamma$, $\gamma \in \text{close}(A)$
- implicative normal form
- resolution, simplification rules

3. Correct and exhaustive reasoning

- interpretation $A \models \gamma$
- truth table
- examples where $\alpha \vdash \beta$ but $\alpha \not\models \beta$ and vice versa
- resolution theorems

4. Constraint satisfaction search

- backtracking search
- pruning, constraint propagation
- heuristics
- systematic and unsystematic search
- random search

5. Problem solving search

- search space
- search tree, search graph
- uninformed search: DFS, BFS, IDS
- informed search: Best first search
- admissible heuristic
- A*, IDA*

6. Automated planning

- propositional planning
- representing actions: preconditions, postconditions
- progression, regression
- bidirectional search

7. Planning algorithms

- partial order plans
- POP algorithm

8. First order representation

- first order language:
    - constants, functions, variables, predicates, quantifiers
    - terms, formulas, sentences
- inference: specialization, resolution, unification

9. Planning in logic, First order inference

- interpretation
- correctness and exhaustiveness
- situation calculus

Part 2: Interpreting

10. Automating interpretation systems

- different from reasoning
- probability theory
- forward generic model
- demos

11. Probability modeling

- joint distribution
- computational tasks
- independence
- conditional independence
- fully independent model

12. Structured probability models

- Bayesian networks

13. Efficient probabilistic inference

- factor graph
- efficient marginalization
- message passing algorithms
  - for marginalization in one variable
  - general marginalization
  - conditional probability
  - completion

14. Inference in complex models

- exact, heuristic, and sampling strategy
- logic sampling
- importance sampling

15. Interpreting senses (perception)

- example: handwritten digit recognition

16. Interpreting natural language

- NLP in general
- N-gram model
- HMM
- PCFG

Part 3: Behaving

17. optimal behaviour: decision theory

- state dynamics
- reward policy
- utility (value) function
- optimizing immediate expected reward
- sequential acyclic decision problem

18. Optimal sequential decision making

- cat and mouse example
- calculating value function
- discounting
- policy iteration
- value iteration

19. Optimal behavior: Game theory

- single step: maximin
- sequential acyclic case
- utility function
- can and mouse example
- discounting, policy and value iteration
- game playing
- minimax algorithm
- α-β pruning

20. Scaling up: Partial observability

- belief state
- value of information
- demo

Part 4: Learning

21. Types of learning problems

- machine learning
- mathematical modelling
- examples
- fundamental types of learning

22. Function learning algorithms

- error function
- classification and regression
- linear regression
- geometrical representation of linear regression
- learning using different error functions: L_1, L_2, L_p, L_∞
- generalized linear functions
- polynomial regression
- neural networks

23. Generalization theory / Overfitting

- introduction to statistical generalization theory
- learning curves, overfitting curves
- automatic complexity control
- complexity penalization
- hold out testing
- metric space approach

24. Course review