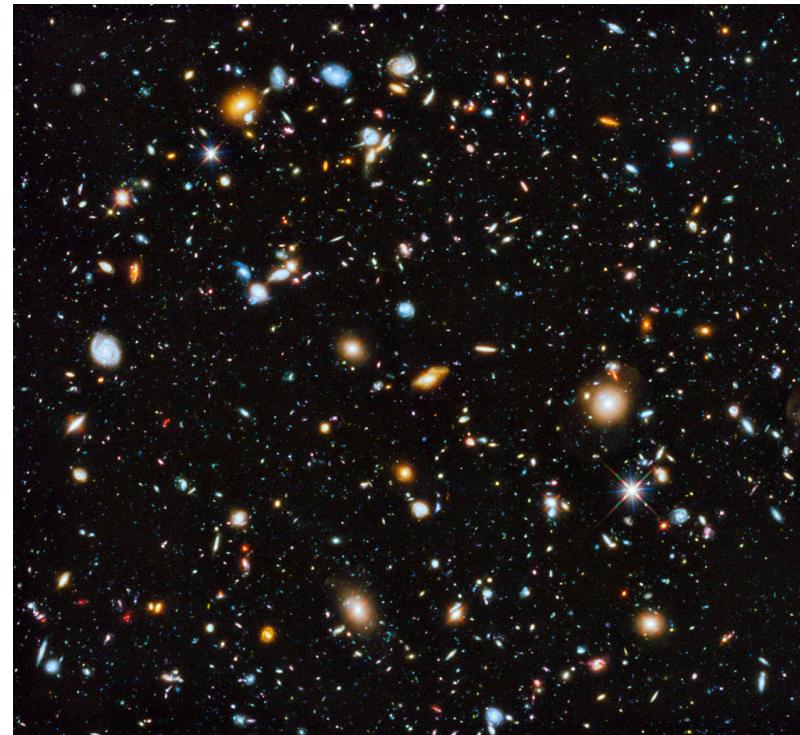




# Lecture 1: Overview





Percy Liang



Dorsa Sadigh



Reid Pryzant (Head CA)



Susanna Maria Baby



Di Bai



Zach Barnes



Hancheng Cao



Horace Chu



Will Deaderick



Haoshen Hong



Cindy Jiang



Chuma Kabaghe



Dhruv Kedia



Jon Kotker



Richard Diehl Martinez



Marcus Pålsson



Chuanbo Pan



Jerry Qu



Andrew Tan



Sharman Tan



Christopher Waites



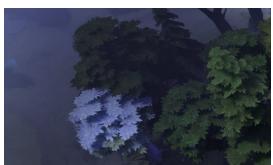
# Announcements

- Section: this Thursday, overview of foundations
- Homework foundations is out, due next Tuesday 11pm
- Gradescope code will be posted on Piazza



Microsoft creates AI that can read a document and answer questions about it as well as a person

January 15, 2018 | Allison Linn



June 24, 2014

Microsoft researchers achieve new conversational speech recognition milestone

August 20, 2017 | By Xuedong



## DeepFace: Closing the Performance Gap in Face Recognition

Conference on Computer Vision and Pattern Recognition (CVPR)

By: Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, Lior Wolf

### Abstract

In modern face recognition, the conventional pipeline consists of alignment, feature extraction, and classification. We revisit both the alignment step and the representation modeling in order to apply a piecewise affine transformation to each facial region. This deep network involves multiple locally connected layers without weight sharing, rather than a single fully connected layer. We trained it on the largest facial dataset to-date, an identity dataset containing more than 4,000 identities.

If you think AI will never replace radiologists—you may want to think again

May 14, 2018 | Michael Walter | Artificial Intelligence

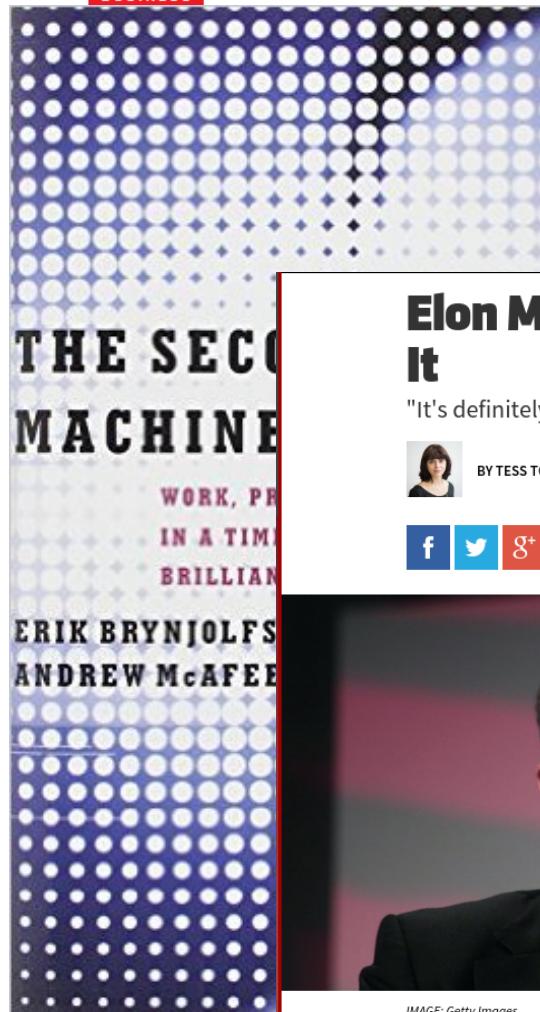


It's one of the most frequently discussed questions in radiology today: What kind of long-term impact will artificial intelligence (AI) have on radiologists?

Robert Schier, MD, a radiologist for RadNet, shared his own thoughts on the topic in a [new commentary](#) published by the *Journal of the American College of Radiology*—and he's not quite as optimistic as some of his colleagues throughout the industry.

- It is hard these days to escape hearing about AI — in the news, on social media, in cafe conversations. A lot of the triumphs of superhuman performance have been in **games**, such as Jeopardy! (IBM Watson, 2011), Go (DeepMind's AlphaGo, 2016), Dota 2 (OpenAI, 2019), Poker (CMU and Facebook, 2019).
- On non-game tasks, we have also have systems that achieve or surpass human-level performance on reading comprehension, speech recognition, face recognition, and medical imaging **benchmarks**.
- Unlike games, however, where the game is the full problem, good performance on a benchmark does not necessarily translate to good performance on the actual task in the wild. Just because you ace an exam doesn't necessarily mean you have perfect understanding or know how to apply that knowledge to real problems.

BUSINESS



## Elon Musk

"It's definitely



BY TESS TAYLOR



IMAGE: Getty Images

The advances we'

humanoid robots, speech recognition and systems like *Jeopardy!*-champion computers—are not the

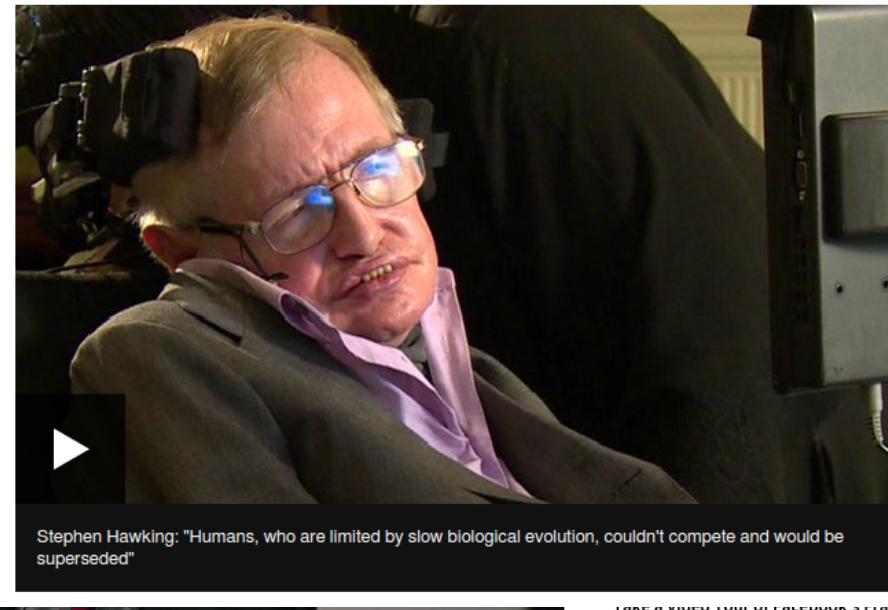
Elon Musk has emerged as a leading voice in speaking out on the potential [dangers](#) of artificial intelligence, going so far as to call it the "biggest existential threat" to

Technology

# Stephen Hawking warns artificial intelligence could end mankind

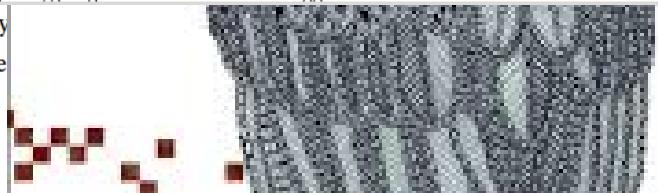
By Rory Cellan-Jones  
Technology correspondent

2 December 2014 | Technology | [Comment](#)



Take a video tour of Facebook's Frank Gehry-Designed New York City Office

HIT THE ROAD



- We also see speculation about the future: that it will bring about sweeping societal change due to automation, resulting in massive job loss, not unlike the industrial revolution, or that AI could even surpass human-level intelligence and seek to take control.
- While AI is likely to be transformational, what kind of transformation the future will hold, no one knows.

1956

# Birth of AI

1956: Workshop at Dartmouth College; attendees: John McCarthy, Marvin Minsky, Claude Shannon, etc.

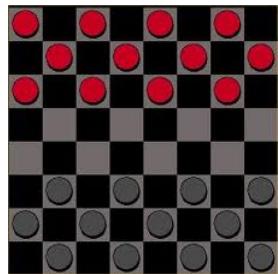


Aim for **general principles**:

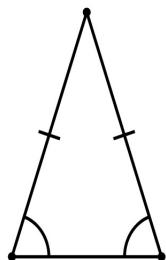
*Every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it.*

- How did we get here? The name **artifical intelligence** goes back to a summer in 1956. John McCarthy, who was then at MIT but later founded the Stanford AI lab, organized a workshop at Dartmouth College with the leading thinkers of the time, and set out a very bold proposal...to build a system that could do it **all**.

# Birth of AI, early successes



**Checkers (1952)**: Samuel's program learned weights and played at strong amateur level



**Problem solving (1955)**: Newell & Simon's Logic Theorist: prove theorems in Principia Mathematica using search + heuristics; later, General Problem Solver (GPS)

- While they did not solve it all, there were a lot of **interesting programs** that were created: programs that could play checkers at a strong amateur level, programs that could prove theorems.
- For one theorem Newell and Simon's Logical Theorist actually found a proof that was more elegant than what a human came up with. They actually tried to publish a paper on it but it got rejected because it was not a new theorem; perhaps they failed to realize that the third author was a computer program.
- From the beginning, people like John McCarthy sought **generality**, thinking of how commonsense reasoning could be encoded in logic. Newell and Simon's General Problem Solver promised to could solve any problem (which could be suitably encoded in logic).

# Overwhelming optimism...

*Machines will be capable, within twenty years, of doing any work a man can do.* —Herbert Simon

*Within 10 years the problems of artificial intelligence will be substantially solved.* —Marvin Minsky

*I visualize a time when we will be to robots what dogs are to humans, and I'm rooting for the machines.* —Claude Shannon

- It was a time of high optimism, with all the leaders of the field, all impressive thinkers, predicting that AI would be "solved" in a matter of years.

# ...underwhelming results

Example: machine translation

*The spirit is willing but the flesh is weak.*



(Russian)



*The vodka is good but the meat is rotten.*

1966: ALPAC report cut off government funding for MT, first AI winter

- Despite some successes, certain tasks such as machine translation were complete failures, which lead to the cutting of funding and the first AI winter.

# Implications of early era

## Problems:

- **Limited computation**: search space grew exponentially, outpacing hardware ( $100! \approx 10^{157} > 10^{80}$ )
- **Limited information**: complexity of AI problems (number of words, objects, concepts in the world)

## Contributions:

- Lisp, garbage collection, time-sharing (John McCarthy)
- Key paradigm: separate **modeling** and **inference**

- What went wrong? It turns out that the real world is very complex and most AI problems require a lot of **compute** and **data**.
- The hardware at the time was simply too limited both compared to both the human brain and computers available now. Also, casting problems as general logical reasoning meant that the approaches fell prey to the exponential search space, which no possible amount of compute could really fix.
- Even if you had infinite compute, AI would not be solved. There are simply too many words, objects, and concepts in the world, that this information has to be somehow encoded in the AI system.
- Though AI was not solved, a few generally useful technologies came out of the effort, such as Lisp (still the world's most advanced programming language in a sense).
- One particularly powerful paradigm is the separation between what you want to compute (modeling) and how to compute it (inference).

# Knowledge-based systems (70-80s)

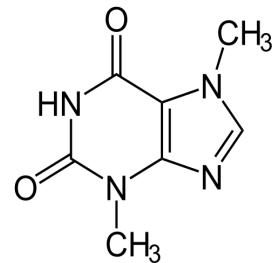


**Expert systems:** elicit specific domain knowledge from experts in form of rules:

if [premises] then [conclusion]

- In the seventies and eighties, AI researchers looked to knowledge as a way to combat both the limited computation and information problems. If we could only figure out a way to encode prior knowledge in these systems, then they will have the necessary information and also have to do less compute.

# Knowledge-based systems (70-80s)



DENDRAL: infer molecular structure from mass spectrometry



MYCIN: diagnose blood infections, recommend antibiotics



XCON: convert customer orders into parts specification;  
save DEC \$40 million a year by 1986

- Instead of the solve-it-all optimism from the 1950s, researchers focused on building narrow practical systems in targeted domains. These became known as **expert systems**.

# Knowledge-based systems

## Contributions:

- First **real application** that impacted industry
- Knowledge helped curb the exponential growth

## Problems:

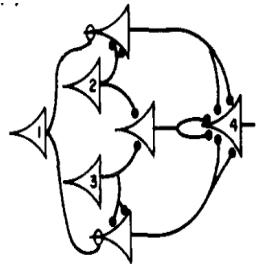
- Knowledge is not deterministic rules, need to model **uncertainty**
- Requires considerable **manual effort** to create rules, hard to maintain

1987: Collapse of Lisp machines and second AI winter

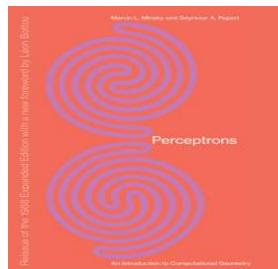
- This was the first time AI had a measurable impact on industry. However, the technology ran into limitations and failed to scale up to more complex problems. Due to plenty of overpromising and underdelivering, the field collapsed again.
- We know that this is not the end of the AI story, but actually it is not the beginning. There is another thread for which we need to go back to 1943.

1943

# Artificial neural networks



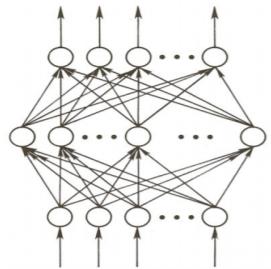
1943: introduced artificial neural networks, connect neural circuitry and logic (McCulloch/Pitts)



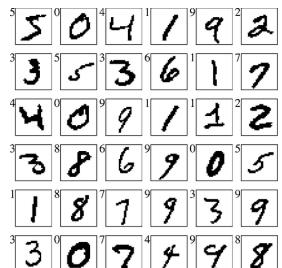
1969: Perceptrons book showed that linear models could not solve XOR, killed neural nets research (Minsky/Papert)

- Much of AI's history was dominated by the logical tradition, but there was another smaller camp, grounded in neural networks inspired by the brain.
- (Artificial) neural networks were introduced by a famous paper by McCulloch and Pitts, who devised a simple mathematical model and showed how it could be used to compute arbitrary logical functions.
- Much of the early work was on understanding the mathematical properties of these networks, since computers were too weak to do anything interesting.
- In 1969, a book was published that explored many mathematical properties of Perceptrons (linear models) and showed that they could not solve some simple problems such as XOR. Even though this result says nothing about the capabilities of deeper networks, the book is largely credited with the demise of neural networks research, and the continued rise of logical AI.

# Training networks



1986: popularization of backpropagation for training multi-layer networks (Rumelhardt, Hinton, Williams)



1989: applied convolutional neural networks to recognizing handwritten digits for USPS (LeCun)

- In the 1980s, there was a renewed interest in neural networks. Backpropagation was rediscovered and popularized as a way to actually train deep neural networks, and Yann LeCun built a system based on convolutional neural networks to recognize handwritten digits, one of first successful use of neural networks that became used by the USPS to recognize zip codes.

# Deep learning



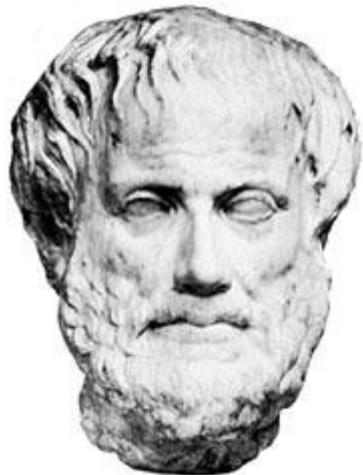
AlexNet (2012): huge gains in object recognition; transformed computer vision community overnight



AlphaGo (2016): deep reinforcement learning, defeat world champion Lee Sedol

- The real break for neural networks came in the 2010s. With the rise of compute (notably GPUs) and large datasets such as ImageNet (2009), the time was ripe for the world to take note of neural networks.
- AlexNet was a pivotal system that showed the promise of deep convolutional networks on ImageNet, the benchmark created by the computer vision community who was at the time still skeptical of deep learning. Many other success stories in speech recognition and machine translation followed.

# Two intellectual traditions



- AI has always swung back and forth between the two
- Deep philosophical differences, but deeper connections (McCulloch/Pitts, AlphaGo)?

- Reflecting back on the past of AI, there have been two intellectual traditions that have dominated the scene: one rooted in logic and one rooted in neuroscience (at least initially). This debate is paralleled in cognitive science with connectionism and computationalism.
- While there are deep philosophical differences, perhaps there are deeper connections.
- For example, McCulloch and Pitts work from 1943 can be viewed as the root of deep learning, but that paper is mostly about how to implement logical operations.
- The game of Go (and indeed, many games) can perfectly characterized by a set of simple logic rules. At the same time, the most successful systems (AlphaGo) do not tackle the problem directly using logic, but appeal to the fuzzier world of artificial neural networks.

# A melting pot

- Bayes rule (Bayes, 1763) from **probability**
- Least squares regression (Gauss, 1795) from **astronomy**
- First-order logic (Frege, 1893) from **logic**
- Maximum likelihood (Fisher, 1922) from **statistics**
- Artificial neural networks (McCulloch/Pitts, 1943) from **neuro-science**
- Minimax games (von Neumann, 1944) from **economics**
- Stochastic gradient descent (Robbins/Monro, 1951) from **optimization**
- Uniform cost search (Dijkstra, 1956) from **algorithms**
- Value iteration (Bellman, 1957) from **control theory**

- Of course, any story is incomplete.
- In fact, for much of the 1990s and 2000s, neural networks were not popular in the machine learning community, and the field was dominated more by techniques such as Support Vector Machines (SVMs) inspired by statistical theory.
- The fuller picture that the modern world of AI is more like a New York City—it is a melting pot that has drawn from many different fields ranging from statistics, algorithms, economics, etc.
- And often it is the new connections between these fields that are made and their application to important real-world problems that makes working on AI so rewarding.



# Roadmap

A brief history

**Two views**

Course overview

Course logistics

Optimization

# Two views of AI



AI agents: how can we create intelligence?



AI tools: how can we benefit society?

- There are two ways to look at AI philosophically.
- The first is what one would normally associate with the AI: the science and engineering of building "intelligent" agents. The inspiration of what constitutes intelligence comes from the types of capabilities that humans possess: the ability to perceive a very complex world and make enough sense of it to be able to manipulate it.
- The second views AI as a set of tools. We are simply trying to solve problems in the world, and techniques developed by the AI community happen to be useful for that, but these problems are not ones that humans necessarily do well on natively.
- However, both views boil down to many of the same day-to-day activities (e.g., collecting data and optimizing a training objective), the philosophical differences do change the way AI researchers approach and talk about their work. Moreover, the conflation of these two views can generate a lot of confusion.



*AI agents...*

# An intelligent agent

Perception

Robotics

Language



Knowledge

Reasoning

Learning

- The starting point for the agent-based view is ourselves.
- As humans, we have to be able to perceive the world (computer vision), perform actions in it (robotics), and communicate with other agents (language).
- We also have knowledge about the world (from procedural knowledge like how to ride a bike to declarative knowledge like remembering the capital of France), and using this knowledge we can draw inferences and make decisions (reasoning).
- Finally, we learn and adapt over time. We are born with none of the skills that we possess as adults, but rather the capacity to acquire them. Indeed machine learning has become the primary driver of many of the AI applications we see today.

# Are we there yet?



Machines: narrow tasks, millions of examples

Humans: diverse tasks, very few examples

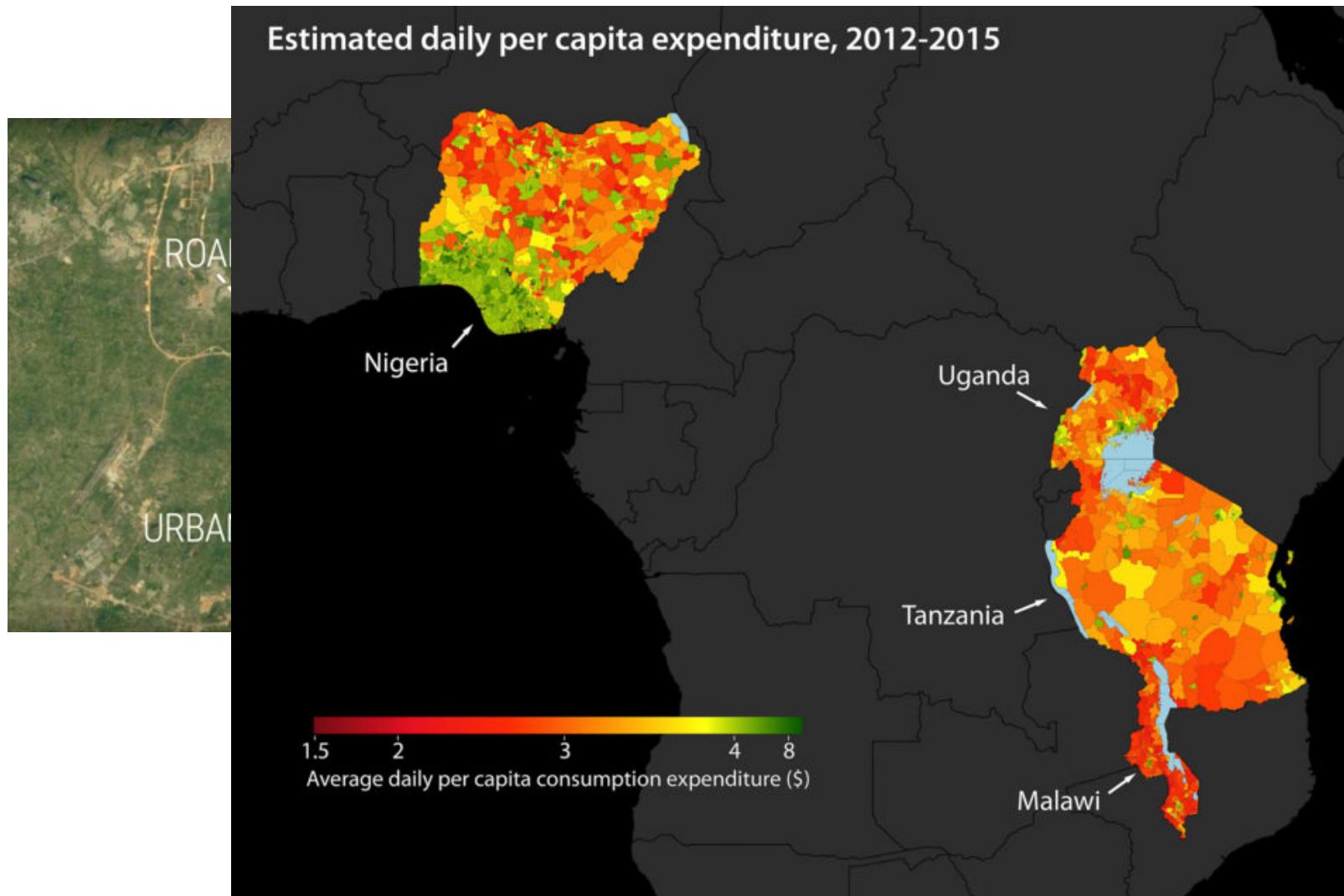
- The AI agents view is an inspiring quest to undercover the mysteries of intelligence and tackle the tasks that humans are good at. While there has been a lot of progress, we still have a long way to go along some dimensions: for example, the ability to learn quickly from few examples or the ability to perform commonsense reasoning.
- There is still a huge gap between the regimes that humans and machines operate in. For example, AlphaGo learned from 19.6 million games, but can only do one thing: play Go. Humans on the other hand, learn from a much wider set of experiences, and can do many things.



*AI tools...*

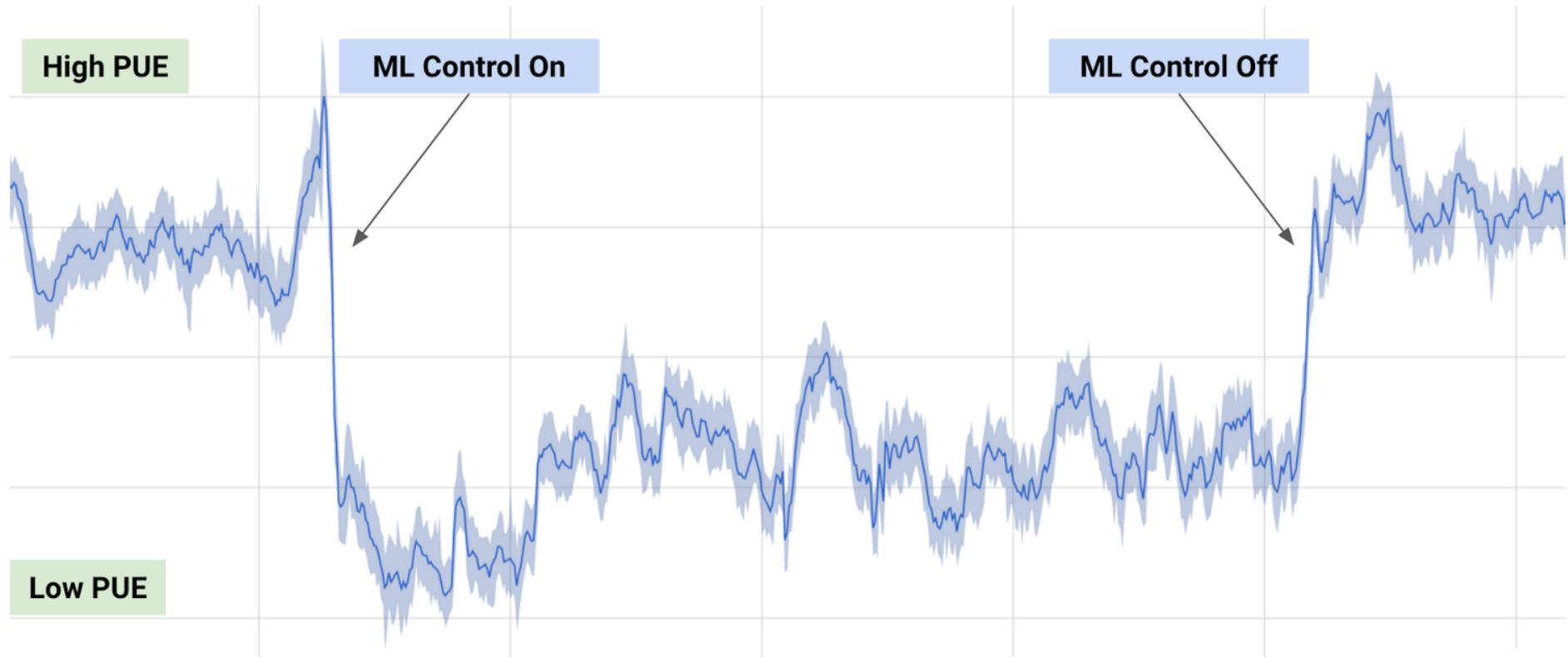
- The other view of AI is less about re-creating the capabilities that humans have, and more about how to benefit humans.
- Even the current level of technology is already being deployed widely in practice, and many of these settings are often not particularly human-like (targeted advertising, news or product recommendation, web search, supply chain management, etc.)

# Predicting poverty

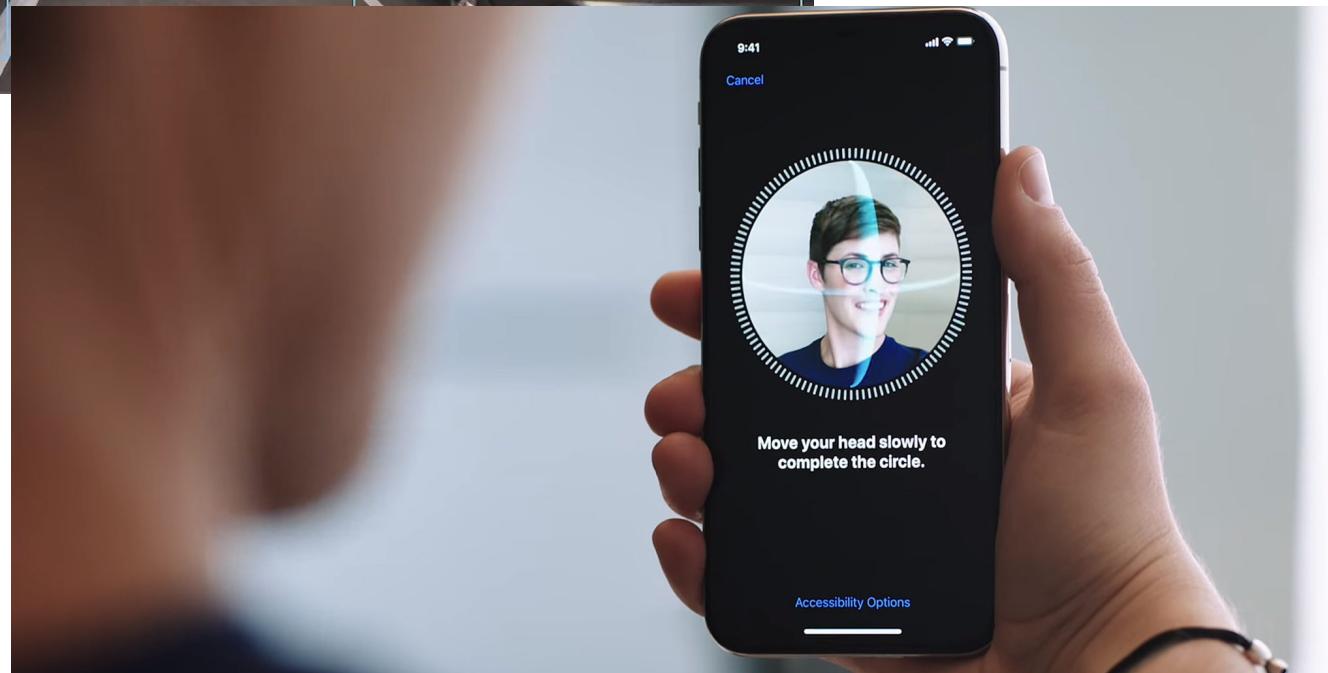
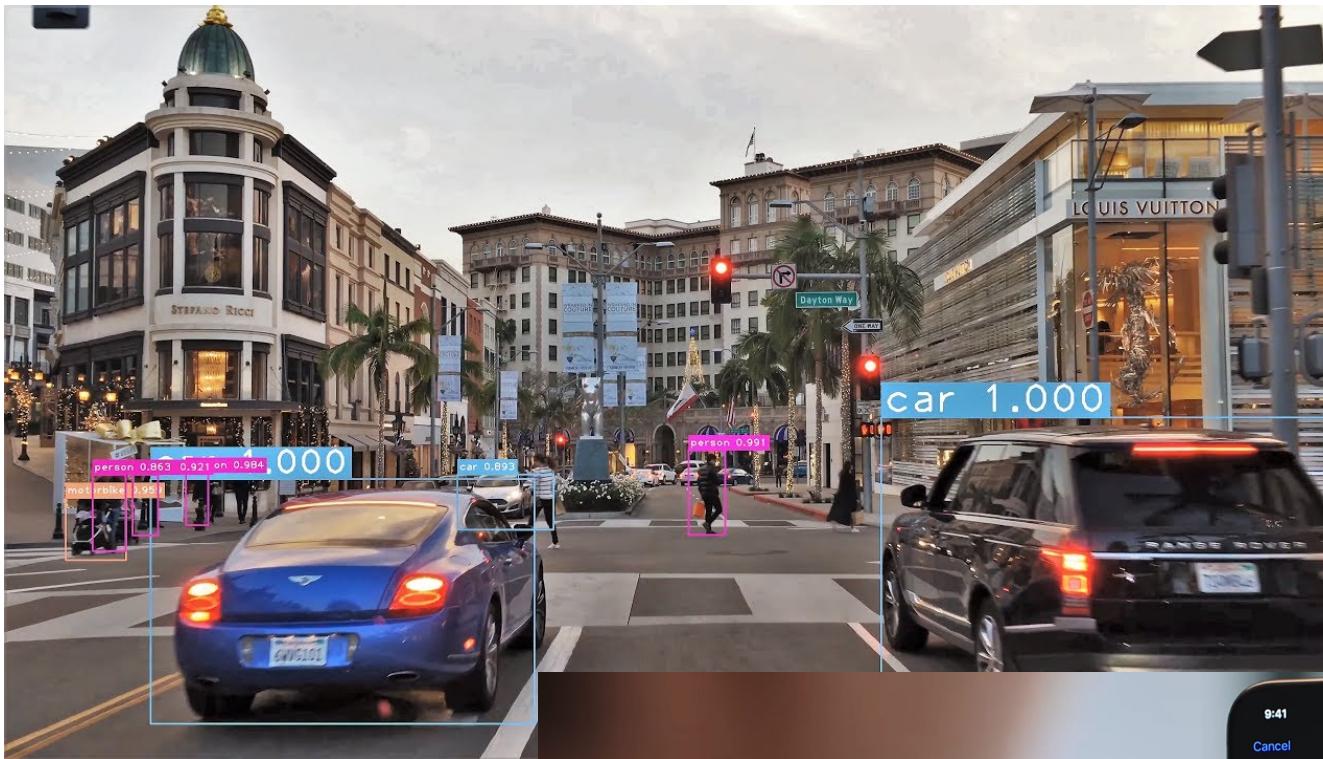


- The computer vision techniques, used to recognize objects, can also be used to tackle social problems. Poverty is a huge problem, and even identifying the areas of need is difficult due to the difficulty in getting reliable survey data. Recent work has shown that one can take satellite images (which are readily available) and predict various poverty indicators.

# Saving energy by cooling datacenters



- Machine learning can also be used to optimize the energy efficiency of datacenters, which given the hunger for compute these days makes a big difference. Some recent work from DeepMind shows how to significantly reduce Google's energy footprint by using machine learning to predict the power usage effectiveness from sensor measurements such as pump speeds, and using that to drive recommendations.

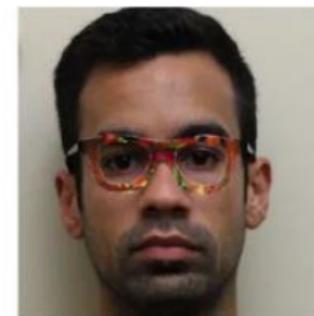


# Security

[Evtimov+ 2017]



[Sharif+ 2016]



- Other applications such as self-driving cars and authentication have high-stakes, where errors could be much more damaging than getting the wrong movie recommendation. These applications present a set of security concerns.
- One can generate so-called **adversarial examples**, where by putting stickers on a stop sign, one can trick a computer vision system to mis-classify it as a speed limit sign. You can also purchase special glasses that fool a system to thinking that you're a celebrity.

# Bias in machine translation

The screenshot shows a machine translation interface with two columns. The left column is for Malay input, and the right column is for English output. Both columns have dropdown menus for language selection and icons for microphone, refresh, and copy/paste.

Malay - detected	English
Dia bekerja sebagai jururawat.	She works as a nurse.
Dia bekerja sebagai pengaturcara. <small>Edit</small>	He works as a programmer.

society  $\Rightarrow$  data  $\Rightarrow$  predictions

# Fairness in criminal risk assessment

- Northpointe: COMPAS predicts criminal risk score (1-10)
- ProPublica: given that an individual did not reoffend, blacks 2x likely to be (wrongly) classified 5 or above
- Northpointe: given a risk score of 7, 60% of whites reoffended, 60% of blacks reoffended

**California just replaced cash bail with algorithms**

By [Dave Gershman](#) • September 4, 2018



- A more subtle case is the issue of bias. One might naively think that since machine learning algorithms are based on mathematical principles, they are somehow objective. However, machine learning predictions come from the training data, and the training data comes from society, so any biases in society are reflected in the data and propagated to predictions. The issue of bias is a real concern when machine learning is used to decide whether an individual should receive a loan or get a job.
- Unfortunately, the problem of fairness and bias is as much of a philosophical one as it is a technical one. There is no obvious "right thing to do", and it has even been shown mathematically it is impossible for a classifier to satisfy three reasonable fairness criteria (Kleinberg et al., 2016).



# Summary so far

- **AI agents:** achieving human-level intelligence, still very far (e.g., generalize from few examples)



- **AI tools:** need to think carefully about real-world consequences (e.g., security, biases)





# Roadmap

A brief history

Two views

**Course overview**

Course logistics

Optimization

# How do we solve AI tasks?



```
# Data structures for supporting uniform cost search.
class PriorityQueue:
    def __init__(self):
        self.DONE = -100000
        self.heap = []
        self.stateToPriority = {} # Map from state to priority

    # Insert (state, prio) into the heap with priority (newPriority). If
    # insert fails (i.e. if newPriority is smaller than the existing
    # priority), update the priority queue and return True.
    def update(self, state, newPriority):
        oldPriority = self.stateToPriority.get(state)
        if oldPriority < newPriority:
            self.stateToPriority[state] = newPriority
            heapq.heappush(self.heap, (newPriority, state))
            return True
        else:
            return False

    # Return (state with minimum priority, priority).
    def ar(self):
        if len(self.heap) == 0:
            return (None, None)
        priority, state = heapq.heappop(self.heap)
        self.stateToPriority.pop(state)
        return (state, priority)

# Simple examples of search problems to test your code for Problem 1.
# A simple search problem: the number 10.
# From state 1, you can move up 1 or down 1 to move down, 2 to move up.
class NumberSearchProblem:
    def __init__(self):
        self.stateToActions = {}
        self.stateToCosts = {}
        self.stateToParents = {}
        self.stateToGoals = {}

    def addState(self, state):
        self.stateToActions[state] = []
        self.stateToCosts[state] = 0
        self.stateToParents[state] = None
        self.stateToGoals[state] = None

    def addAction(self, state, action, cost):
        self.stateToActions[state].append(action)
        self.stateToCosts[state] += cost
        self.stateToParents[action] = state
        self.stateToGoals[action] = None

    def addGoal(self, state):
        self.stateToGoals[state] = True

    def actions(self, state):
        return self.stateToActions[state]

    def cost(self, state, action):
        return self.stateToCosts[action]

    def parent(self, state):
        return self.stateToParents[state]

    def goal(self, state):
        return self.stateToGoals[state]
```

- How should we actually solve AI tasks? The real world is complicated. At the end of the day, we need to write some code (and possibly build some hardware too). But there is a huge chasm.

# Paradigm

Modeling

Inference

Learning

- In this class, we will adopt the **modeling-inference-learning** paradigm to help us navigate the solution space. In reality, the lines are blurry, but this paradigm serves as an ideal and a useful guiding principle.

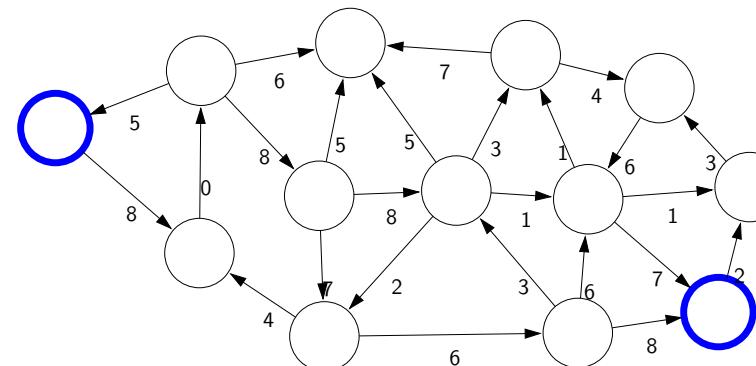
# Paradigm: modeling

Real world



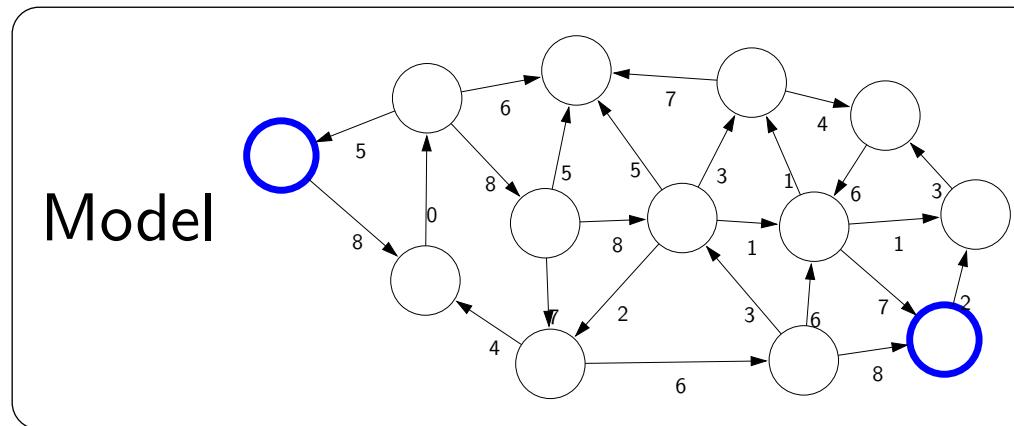
Modeling

Model

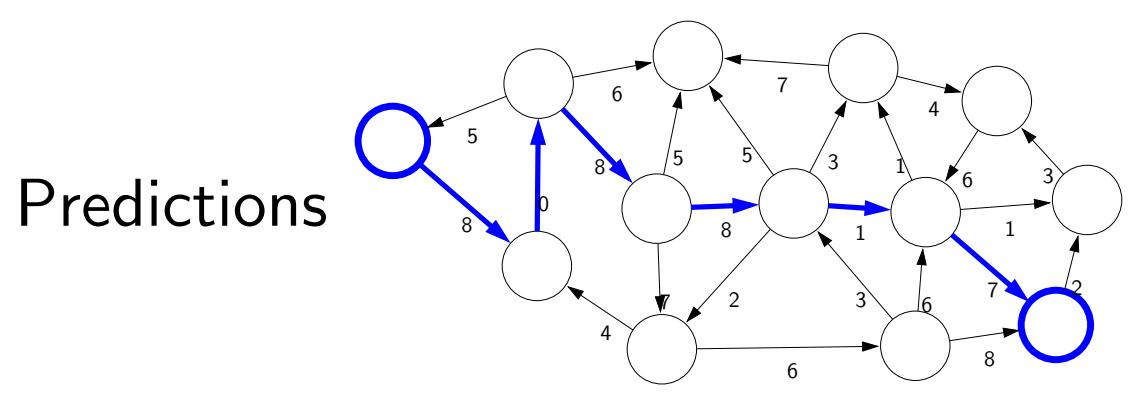


- The first pillar is modeling. Modeling takes messy real world problems and packages them into neat formal mathematical objects called **models**, which can be subject to rigorous analysis and that computers can operate on. However, modeling is lossy: not all of the richness of the real world can be captured, and therefore there is an art of modeling: what does one keep versus what does one ignore? (An exception to this is games such as Chess or Go or Sodoku, where the real world is identical to the model.)
- As an example, suppose we're trying to have an AI that can navigate through a busy city. We might formulate this as a graph where nodes represent points in the city, and edges represent the roads and cost of an edge represents traffic on that road.

# Paradigm: inference



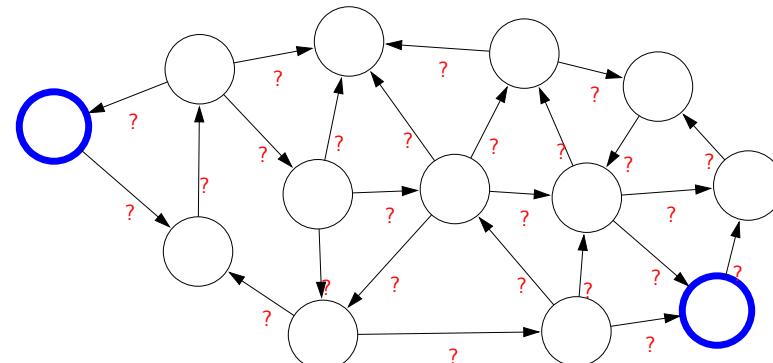
Inference



- The second pillar is inference. Given a model, the task of **inference** is to answer questions with respect to the model. For example, given the model of the city, one could ask questions such as: what is the shortest path? what is the cheapest path?
- The focus of inference is usually on efficient algorithms that can answer these questions. For some models, computational complexity can be a concern (games such as Go), and usually approximations are needed.

# Paradigm: learning

Model without parameters

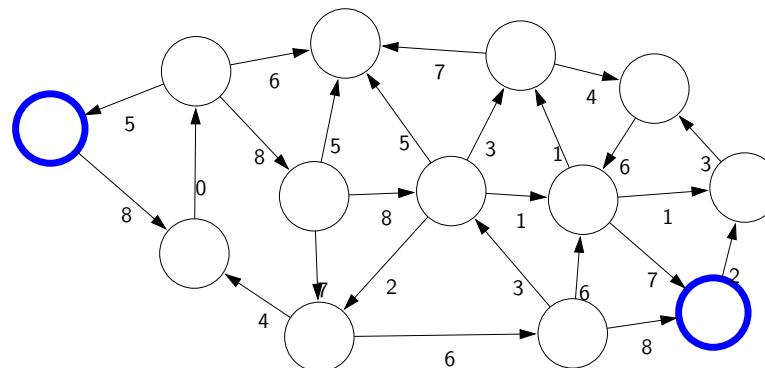


+data

I  
Learning



Model with parameters



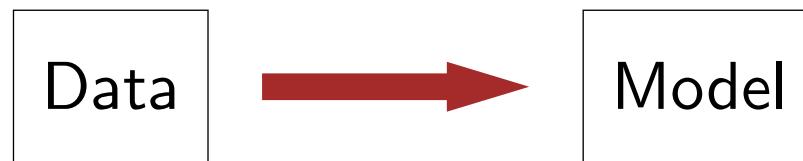
- But where does the model come from? Remember that the real world is rich, so if the model is to be faithful, the model has to be rich as well. But we can't possibly write down such a rich model manually.
- The idea behind (machine) **learning** is to instead get it from data. Instead of constructing a model, one constructs a skeleton of a model (more precisely, a model family), which is a model without parameters. And then if we have the right type of data, we can run a machine learning algorithm to tune the parameters of the model.
- Note that learning here is not tied to a particular approach (e.g., neural networks), but more of a philosophy. This general paradigm will allow us to bridge the gap between logic-based AI and statistical AI.

# Course plan



- We now embark on our tour of the topics in this course. The topics correspond to types of models that we can use to represent real-world tasks. The topics will in a sense advance from low-level intelligence to high-level intelligence, evolving from models that simply make a reflex decision to models that are based on logical reasoning.

# Machine learning



- The main driver of recent successes in AI
- Move from "code" to "data" to manage the information complexity
- Requires a leap of faith: **generalization**

- Supporting all of these models is **machine learning**, which has been arguably the most crucial ingredient powering recent successes in AI. From an systems engineering perspective, machine learning allows us to shift the information complexity of the model from code to data, which is much easier to obtain (either naturally occurring or via crowdsourcing).
- The main conceptually magical part of learning is that if done properly, the trained model will be able to produce good predictions beyond the set of training examples. This leap of faith is called **generalization**, and is, explicitly or implicitly, at the heart of any machine learning algorithm. This can even be formalized using tools from probability and statistical learning theory.

# Course plan

Reflex

"Low-level intelligence"

"High-level intelligence"

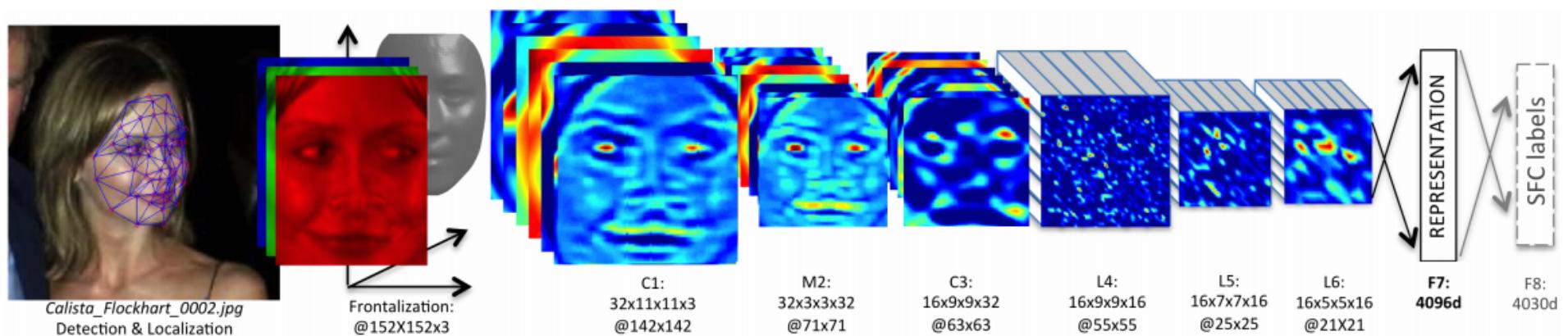
Machine learning

# What is this animal?



# Reflex-based models

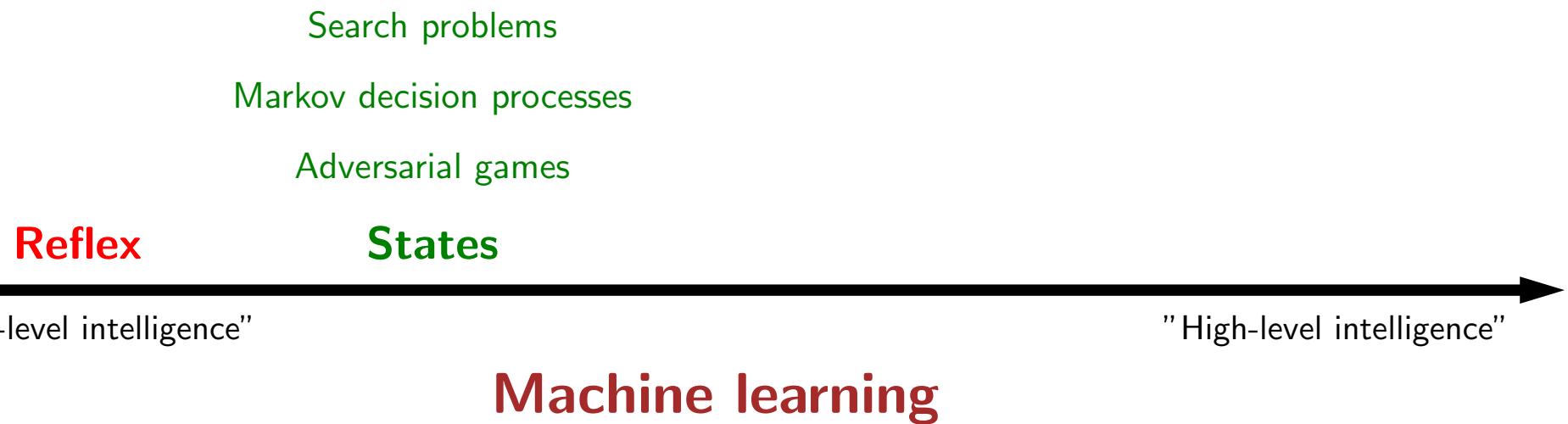
- Examples: linear classifiers, deep neural networks



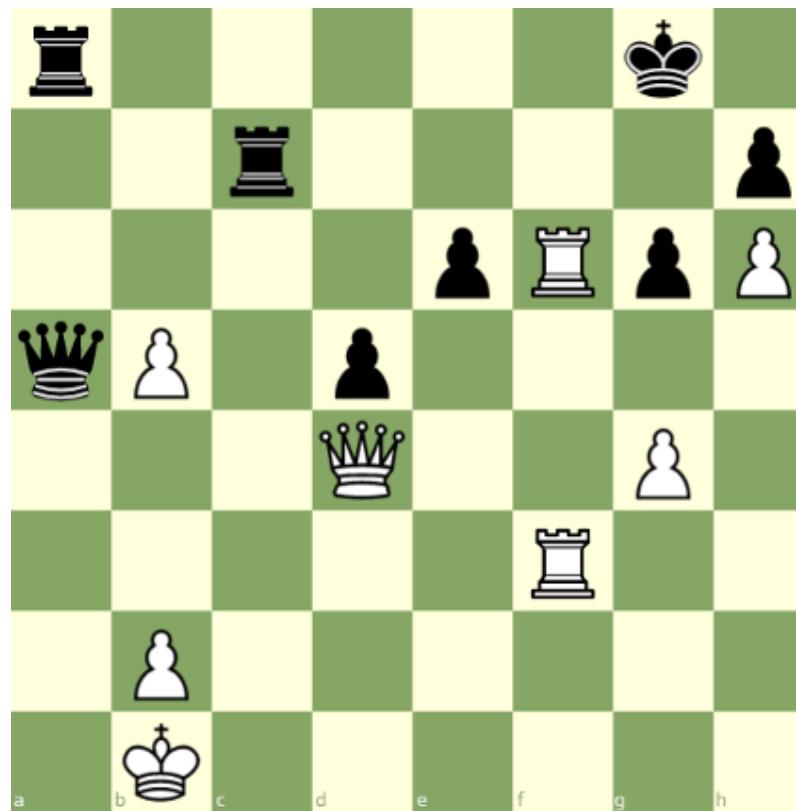
- Most common models in machine learning
- Fully feed-forward (no backtracking)

- A reflex-based model simply performs a fixed sequence of computations on a given input. Examples include most models found in machine learning from simple linear classifiers to deep neural networks. The main characteristic of reflex-based models is that their computations are feed-forward; one doesn't backtrack and consider alternative computations. Inference is trivial in these models because it is just running the fixed computations, which makes these models appealing.

# Course plan

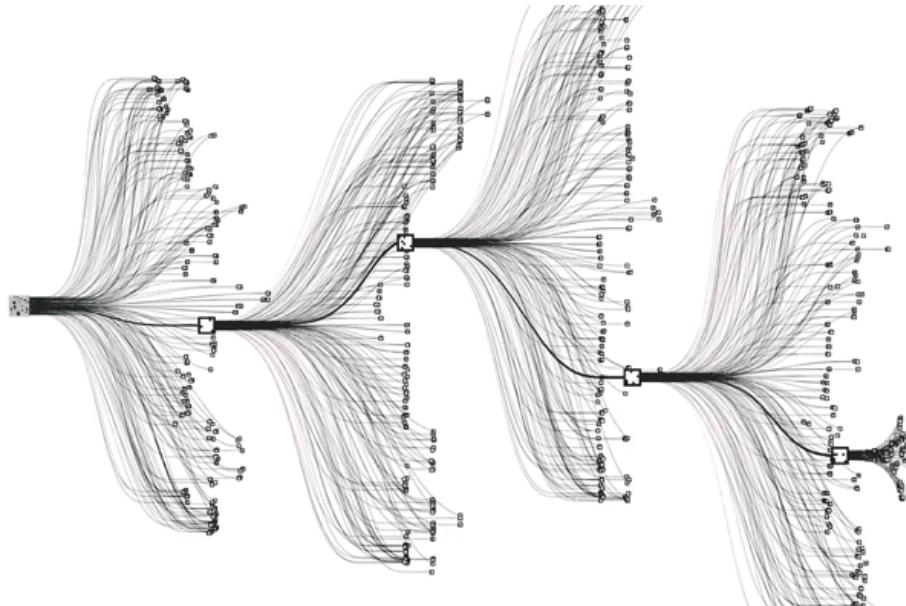


# State-based models



White to move

# State-based models



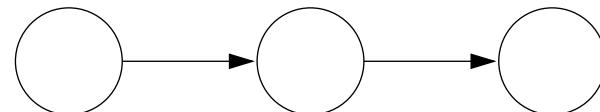
## Applications:

- Games: Chess, Go, Pac-Man, Starcraft, etc.
- Robotics: motion planning
- Natural language generation: machine translation, image captioning

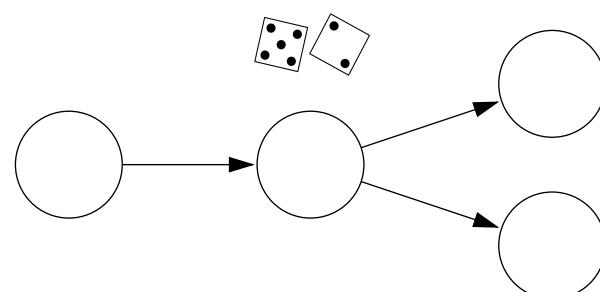
- Reflex-based models are too simple for tasks that require more forethought (e.g., in playing chess or planning a big trip). State-based models overcome this limitation.
- The key idea is, at a high-level, to model the **state** of a world and transitions between states which are triggered by actions. Concretely, one can think of states as nodes in a graph and transitions as edges. This reduction is useful because we understand graphs well and have a lot of efficient algorithms for operating on graphs.

# State-based models

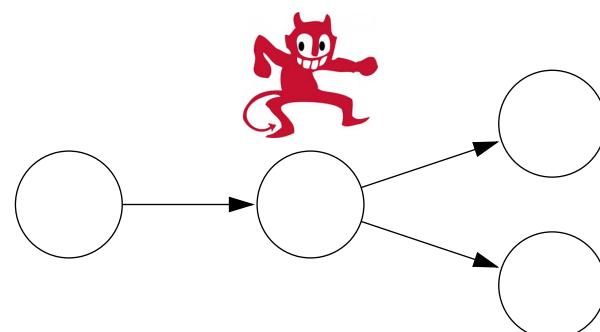
Search problems: you control everything



Markov decision processes: against nature (e.g., Blackjack)

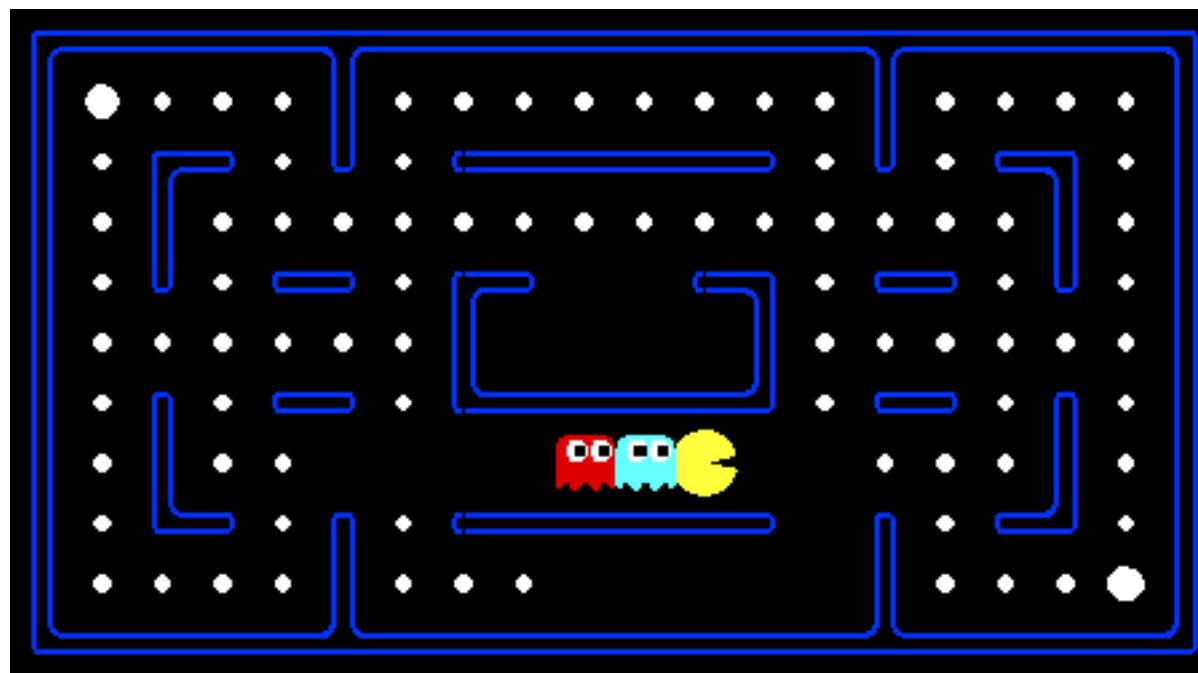


Adversarial games: against opponent (e.g., chess)



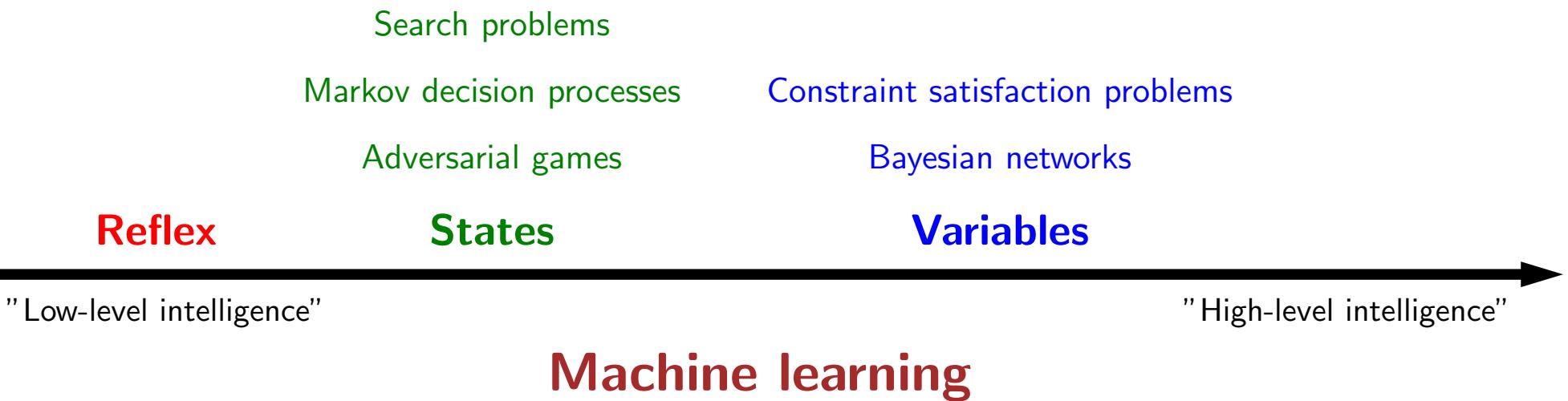
- Search problems are adequate models when you are operating in environment that has no uncertainty. However, in many realistic settings, there are other forces at play.
- **Markov decision processes** handle tasks with an element of chance (e.g., Blackjack), where the distribution of randomness is known (reinforcement learning can be employed if it is not).
- **Adversarial games**, as the name suggests, handle tasks where there is an opponent who is working against you (e.g., chess).

# Pac-Man



[demo]

# Course plan



# Sudoku

5	3		7					
6			1	9	5			
	9	8				6		
8			6				3	
4		8	3			1		
7			2			6		
	6			2	8			
		4	1	9			5	
		8		7	9			



5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

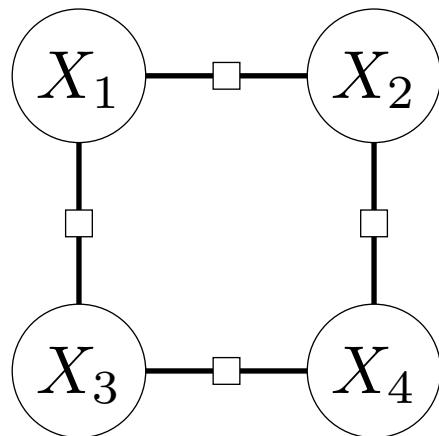
**Goal:** put digits in blank squares so each row, column, and 3x3 sub-block has digits 1–9

**Note:** order of filling squares doesn't matter in the evaluation criteria!

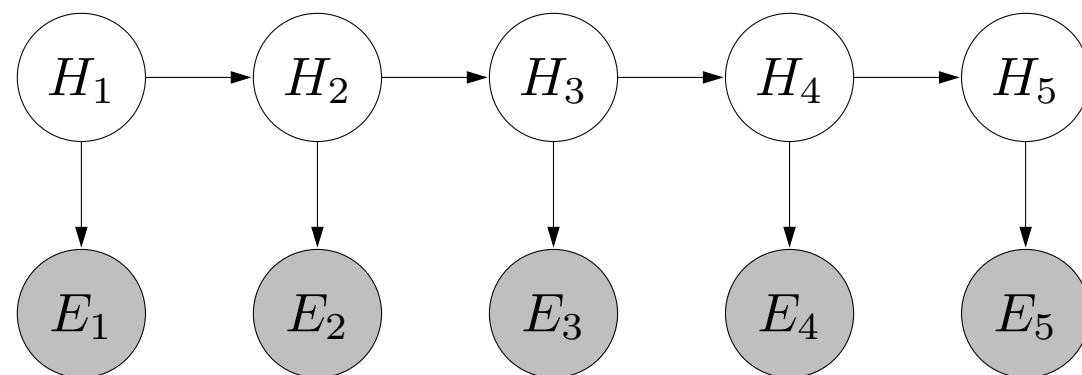
- In state-based models, solutions are procedural: they specify step by step instructions on how to go from A to B. In many applications, the order in which things are done isn't important.

# Variable-based models

Constraint satisfaction problems: hard constraints (e.g., Sudoku, scheduling)

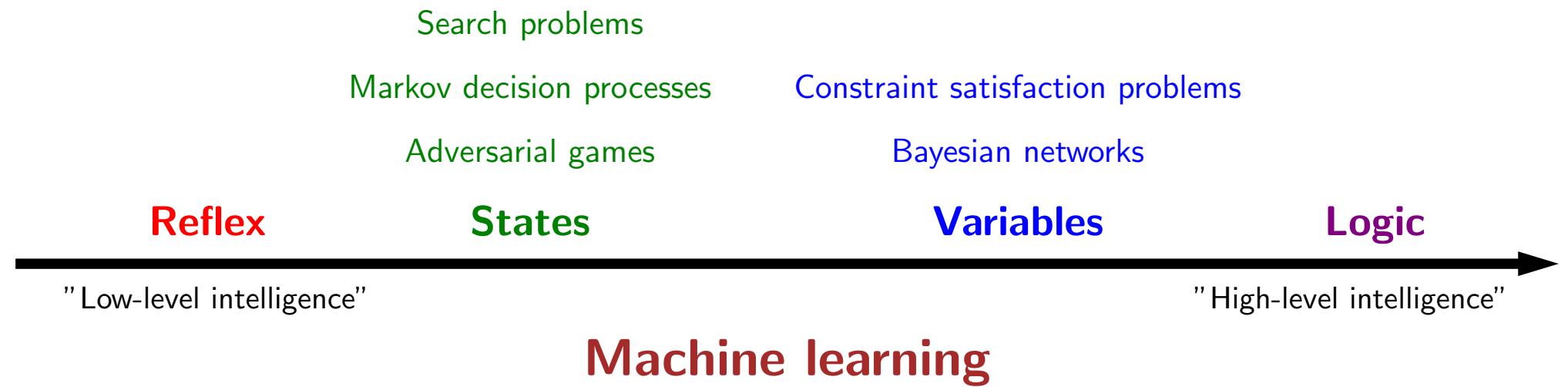


Bayesian networks: soft dependencies (e.g., tracking cars from sensors)



- **Constraint satisfaction problems** are variable-based models where we only have hard constraints. For example, in scheduling, we can't have two people in the same place at the same time.
- **Bayesian networks** are variable-based models where variables are random variables which are dependent on each other. For example, the true location of an airplane  $H_t$  and its radar reading  $E_t$  are related, as are the location  $H_t$  and the location at the last time step  $H_{t-1}$ . The exact dependency structure is given by the graph structure and it formally defines a joint probability distribution over all the variables. This topic is studied thoroughly in probabilistic graphical models (CS228).

# Course plan



# Motivation: virtual assistant

Tell information



Ask questions



**Use natural language!**

[demo]

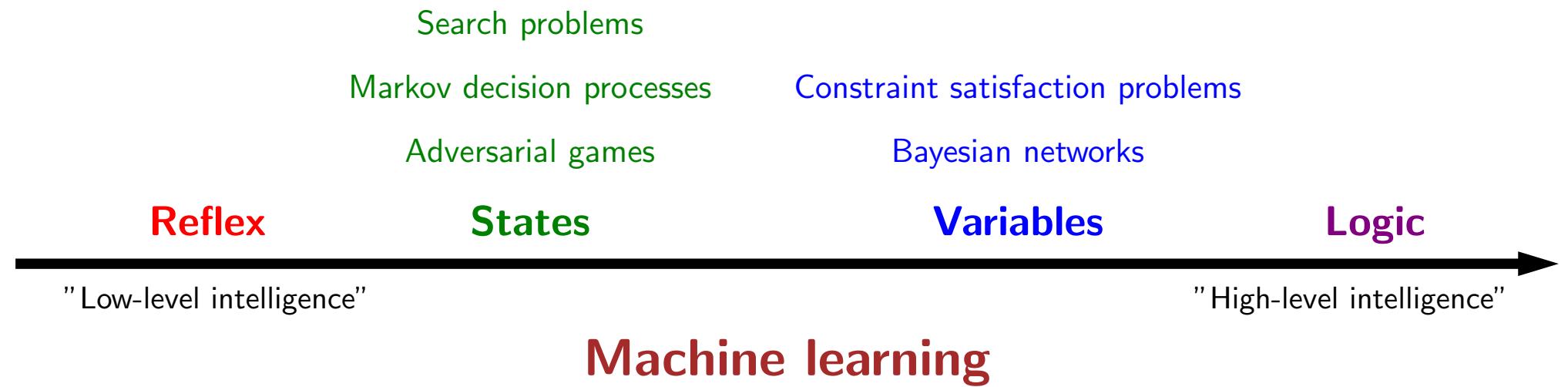
Need to:

- Digest **heterogenous** information
- Reason **deeply** with that information

- Our last stop on the tour is **logic**. Even more so than variable-based models, logic provides a compact language for modeling, which gives us more expressivity.
- It is interesting that historically, logic was one of the first things that AI researchers started with in the 1950s. While logical approaches were in a way quite sophisticated, they did not work well on complex real-world tasks with noise and uncertainty. On the other hand, methods based on probability and machine learning naturally handle noise and uncertainty, which is why they presently dominate the AI landscape. However, they are yet to be applied successfully to tasks that require really sophisticated reasoning.
- In this course, we will appreciate the two as not contradictory, but simply tackling different aspects of AI — in fact, in our course plan, logic is a class of models which can be supported by machine learning. An active area of research is to combine the richness of logic with the robustness and agility of machine learning.

- One motivation for logic is a virtual assistant. At an abstract level, one fundamental thing a good personal assistant should be able to do is to take in information from people and be able to answer questions that require drawing inferences from these facts.
- In some sense, telling the system information is like machine learning, but it feels like a very different form of learning than seeing 10M images and their labels or 10M sentences and their translations. The type of information we get here is both more heterogenous, more abstract, and the expectation is that we process it more deeply (we don't want to tell our personal assistant 100 times that we prefer morning meetings).
- And how do we interact with our personal assistants? Let's use natural language, the very tool that was built for communication!

# Course plan





# Roadmap

A brief history

Two views

Course overview

**Course logistics**

Optimization

# Course objectives

Before you take the class, you should know... —

- Programming (CS 106A, CS 106B, CS 107)
- Discrete math, mathematical rigor (CS 103)
- Probability (CS 109)

At the end of this course, you should... —

- Be able to tackle real-world tasks with the appropriate techniques
- Be more proficient at math and programming



# Coursework

- Homeworks (60%)
- Exam (20%)
- Project (20%)

# Homeworks

- 8 homeworks, mix of written and programming problems, each centers on an application

<b>Introduction</b>	foundations
<b>Machine learning</b>	sentiment classification
<b>Search</b>	text reconstruction
<b>MDPs</b>	blackjack
<b>Games</b>	Pac-Man
<b>CSPs</b>	course scheduling
<b>Bayesian networks</b>	car tracking
<b>Logic</b>	language and logic

- Pac-Man competition for extra credit
- When you submit, programming parts will be run on all test cases, but only get feedback on a subset

# Exam

- Goal: test your ability to use knowledge to solve new problems, not know facts
- All written problems (look at past exam problems for style)
- Closed book except one page of notes
- Covers all material up to and including preceding week

# Project

- Goal: choose any task you care about and apply techniques from class
- Work in groups of up to 3; find a group early, your responsibility to be in a good group
- Milestones: proposal, progress report, poster session, final report
- Task is completely open, but must follow well-defined steps: task definition, implement baselines/oracles, evaluate on dataset, literature review, error analysis (read website)
- Help: assigned a CA mentor, come to any office hours

# Policies

[Gradescope](#): submit all assignments there

[Late days](#): 7 total late days, max two per assignment

[Piazza](#): ask questions on Piazza, don't email us directly

[Piazza](#): extra credit for students who help answer questions

**All details are on the course website**

# THE HONOR CODE

- Do collaborate and discuss together, but write up and code independently.
- Do not look at anyone else's writeup or code.
- Do not show anyone else your writeup or code or post it online (e.g., GitHub).
- When debugging, only look at input-output behavior.
- We will run MOSS periodically to detect plagiarism.



# Roadmap

A brief history

Two views

Course overview

Course logistics

**Optimization**

# Optimization

Discrete optimization: find the best discrete object

$$\min_{p \in \text{Paths}} \text{Cost}(p)$$

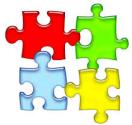
**Algorithmic** tool: dynamic programming

Continuous optimization: find the best vector of real numbers

$$\min_{\mathbf{w} \in \mathbb{R}^d} \text{TrainingError}(\mathbf{w})$$

**Algorithmic** tool: gradient descent

- We are now done with the high-level motivation for the class. Let us now dive into some technical details. Let us focus on the inference and the learning aspect of the **modeling-inference-learning** paradigm.
- We will approach inference and learning from an **optimization** perspective, which allows us to decouple the mathematical specification of **what** we want to compute from the algorithms for **how** to compute it.
- In total generality, optimization problems ask that you find the  $x$  that lives in a constraint set  $C$  that makes the function  $F(x)$  as small as possible.
- There are two types of optimization problems we'll consider: discrete optimization problems (mostly for inference) and continuous optimization problems (mostly for learning). Both are backed by a rich research field and are interesting topics in their own right. For this course, we will use the most basic tools from these topics: **dynamic programming** and **gradient descent**.
- Let us do two practice problems to illustrate each tool. For now, we are assuming that the model (optimization problem) is given and only focus on **algorithms**.



## Problem: computing edit distance

**Input:** two strings,  $s$  and  $t$

**Output:** minimum number of character insertions, deletions, and substitutions it takes to change  $s$  into  $t$

Examples:

$$\text{"cat"}, \text{"cat"} \Rightarrow 0$$

$$\text{"cat"}, \text{"dog"} \Rightarrow 3$$

$$\text{"cat"}, \text{"at"} \Rightarrow 1$$

$$\text{"cat"}, \text{"cats"} \Rightarrow 1$$

$$\text{"a cat!"}, \text{"the cats!"} \Rightarrow 4$$

[live solution]

- Let's consider the formal task of computing the edit distance (or more precisely the Levenshtein distance) between two strings. These measures of dissimilarity have applications in spelling correction, computational biology (applied to DNA sequences).
- As a first step, you should think of breaking down the problem into subproblems. Observation 1: inserting into  $s$  is equivalent to deleting a letter from  $t$  (ensures subproblems get smaller). Observation 2: perform edits at the end of strings (might as well start there).
- Consider the last letter of  $s$  and  $t$ . If these are the same, then we don't need to edit these letters, and we can proceed to the second-to-last letters. If they are different, then we have three choices. (i) We can substitute the last letter of  $s$  with the last letter of  $t$ . (ii) We can delete the last letter of  $s$ . (iii) We can insert the last letter of  $t$  at the end of  $s$ .
- In each of those cases, we can reduce the problem into a smaller problem, but which one? We simply try all of them and take the one that yields the minimum cost!
- We can express this more formally with a mathematical recurrence. These types of recurrences will show up throughout the course, so it's a good idea to be comfortable with them. Before writing down the actual recurrence, the first step is to express the quantity that we wish to compute. In this case: let  $d(m, n)$  be the edit distance between the first  $m$  letters of  $s$  and the first  $n$  letters of  $t$ . Then we have

$$d(m, n) = \begin{cases} m & \text{if } n = 0 \\ n & \text{if } m = 0 \\ d(m - 1, n - 1) & \text{if } s_m = t_n \\ 1 + \min\{d(m - 1, n - 1), d(m - 1, n), d(m, n - 1)\} & \text{otherwise.} \end{cases}$$

- Once you have the recurrence, you can code it up. The straightforward implementation will take exponential time, but you can **memoize** the results to make it  $O(n^2)$  time. The end result is the dynamic programming solution: recurrence + memoization.



## Problem: finding the least squares line

**Input:** set of pairs  $\{(x_1, y_1), \dots, (x_n, y_n)\}$

**Output:**  $w \in \mathbb{R}$  that minimizes the squared error

$$F(w) = \sum_{i=1}^n (x_i w - y_i)^2$$

**Examples:**

$$\{(2, 4)\} \Rightarrow 2$$

$$\{(2, 4), (4, 2)\} \Rightarrow ?$$

[live solution]

- The formal task is this: given a set of  $n$  two-dimensional points  $(x_i, y_i)$  which defines  $F(w)$ , compute the  $w$  that minimizes  $F(w)$ .
- **Linear regression** is an important problem in machine learning, which we will come to later. Here's a motivation for the problem: suppose you're trying to understand how your exam score ( $y$ ) depends on the number of hours you study ( $x$ ). Let's posit a linear relationship  $y = wx$  (not exactly true in practice, but maybe good enough). Now we get a set of training examples, each of which is a  $(x_i, y_i)$  pair. The goal is to find the slope  $w$  that best fits the data.
- Back to algorithms for this formal task. We would like an algorithm for optimizing general types of  $F(w)$ . So let's **abstract away from the details**. Start at a guess of  $w$  (say  $w = 0$ ), and then iteratively update  $w$  based on the derivative (gradient if  $w$  is a vector) of  $F(w)$ . The algorithm we will use is called **gradient descent**.
- If the derivative  $F'(w) < 0$ , then increase  $w$ ; if  $F'(w) > 0$ , decrease  $w$ ; otherwise, keep  $w$  still. This motivates the following update rule, which we perform over and over again:  $w \leftarrow w - \eta F'(w)$ , where  $\eta > 0$  is a **step size** that controls how aggressively we change  $w$ .
- If  $\eta$  is too big, then  $w$  might bounce around and not converge. If  $\eta$  is too small, then  $w$  might not move very far to the optimum. Choosing the right value of  $\eta$  can be rather tricky. Theory can give rough guidance, but this is outside the scope of this class. Empirically, we will just try a few values and see which one works best. This will help us develop some intuition in the process.
- Now to specialize to our function, we just need to compute the derivative, which is an elementary calculus exercise:  $F'(w) = \sum_{i=1}^n 2(x_i w - y_i)x_i$ .



# Summary

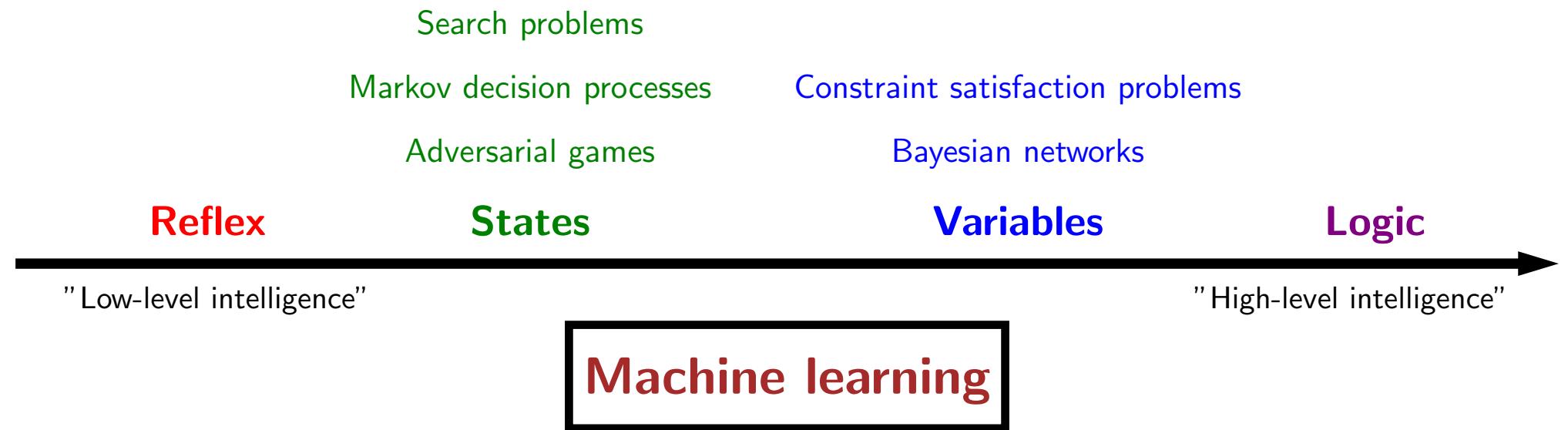
- History: roots from logic, neuroscience, statistics—melting pot!
- Modeling [reflex, states, variables, logic] + inference + learning paradigm
- AI has high societal impact, how to steer it positively?



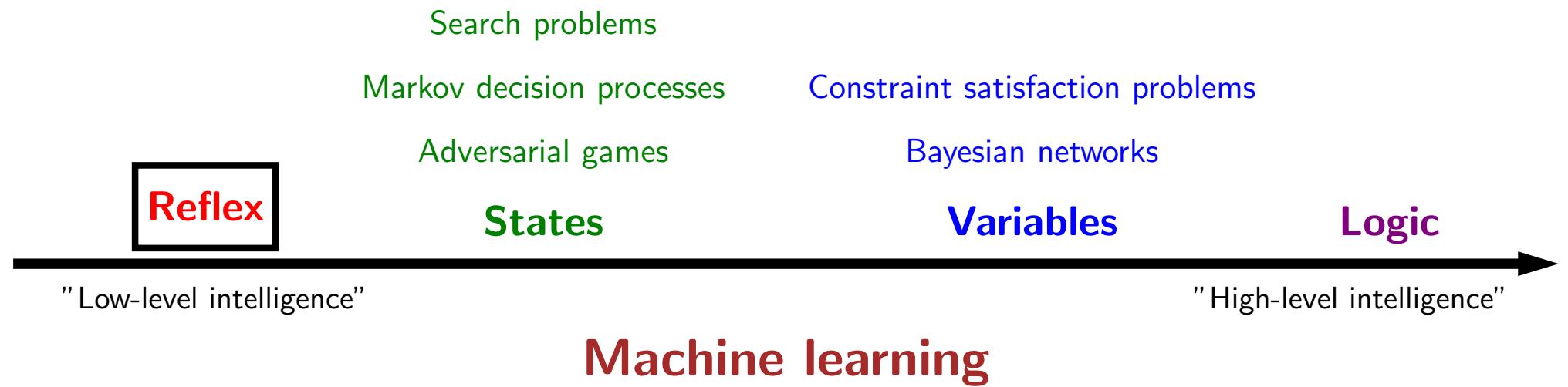
# Lecture 2: Machine learning I



# Course plan



# Course plan





# Roadmap

**Linear predictors**

Loss minimization

Stochastic gradient descent

- We now embark on our journey into machine learning with the simplest yet most practical tool: **linear predictors**, which cover both classification and regression and are examples of reflex models.
- After getting some geometric intuition for linear predictors, we will turn to learning the weights of a linear predictor by formulating an optimization problem based on the **loss minimization** framework.
- Finally, we will discuss **stochastic gradient descent**, an efficient algorithm for optimizing (that is, minimizing) the loss that's tailored for machine learning which is much faster than gradient descent.

# Application: spam classification

Input:  $x$  = email message

From: pliang@cs.stanford.edu  
Date: September 25, 2019  
Subject: CS221 announcement

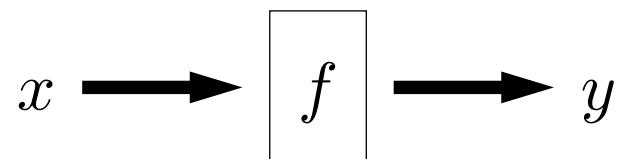
Hello students,  
Welcome to CS221! Here's what...

From: a9k62n@hotmail.com  
Date: September 25, 2019  
Subject: URGENT

Dear Sir or maDam:  
my friend left sum of 10m dollars...

Output:  $y \in \{\text{spam}, \text{not-spam}\}$

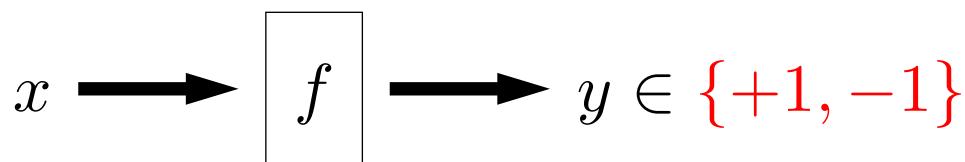
Objective: obtain a **predictor**  $f$



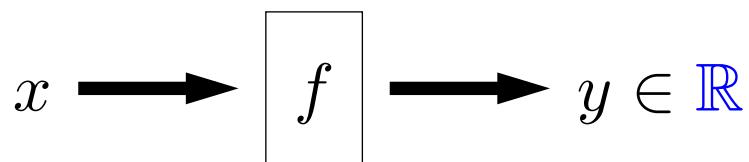
- First, some terminology. A **predictor** is a function  $f$  that maps an **input**  $x$  to an **output**  $y$ . In statistics,  $y$  is known as a response, and when  $x$  is a real vector, it is known as the covariate.

# Types of prediction tasks

Binary classification (e.g., email  $\Rightarrow$  spam/not spam):



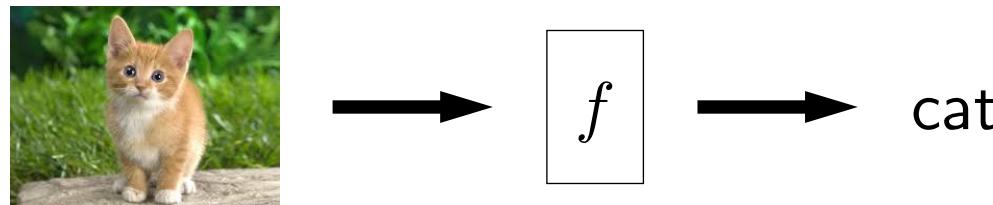
Regression (e.g., location, year  $\Rightarrow$  housing price):



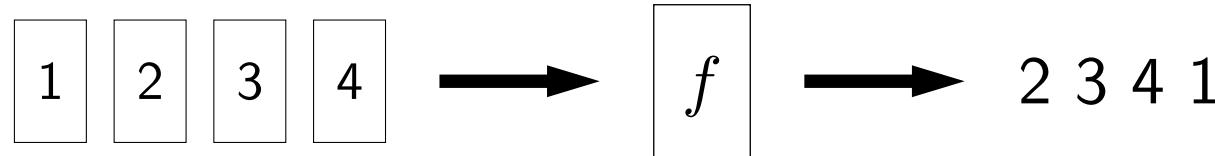
- In the context of classification tasks,  $f$  is called a **classifier** and  $y$  is called a **label** (sometimes class, category, or tag). The key distinction between binary classification and regression is that the former has **discrete** outputs (e.g., "yes" or "no"), whereas the latter has **continuous** outputs.
- Note that the dichotomy of prediction tasks are not meant to be formal definitions, but rather to provide intuition.
- For instance, binary classification could technically be seen as a regression problem if the labels are  $-1$  and  $+1$ . And structured prediction generally refers to tasks where the possible set of outputs  $y$  is huge (generally, exponential in the size of the input), but where each individual  $y$  has some structure. For example, in machine translation, the output is a sequence of words.

# Types of prediction tasks

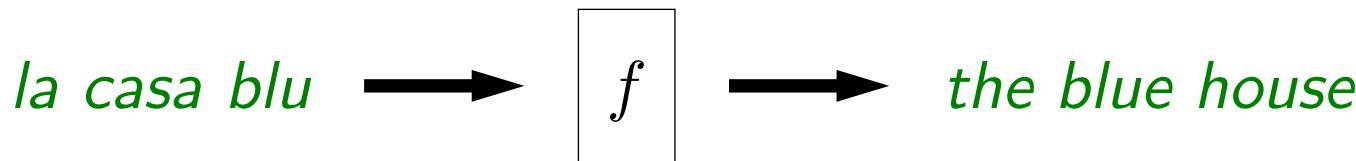
Multiclass classification:  $y$  is a category



Ranking:  $y$  is a permutation



Structured prediction:  $y$  is an object which is built from parts



# Data

Example: specifies that  $y$  is the ground-truth output for  $x$

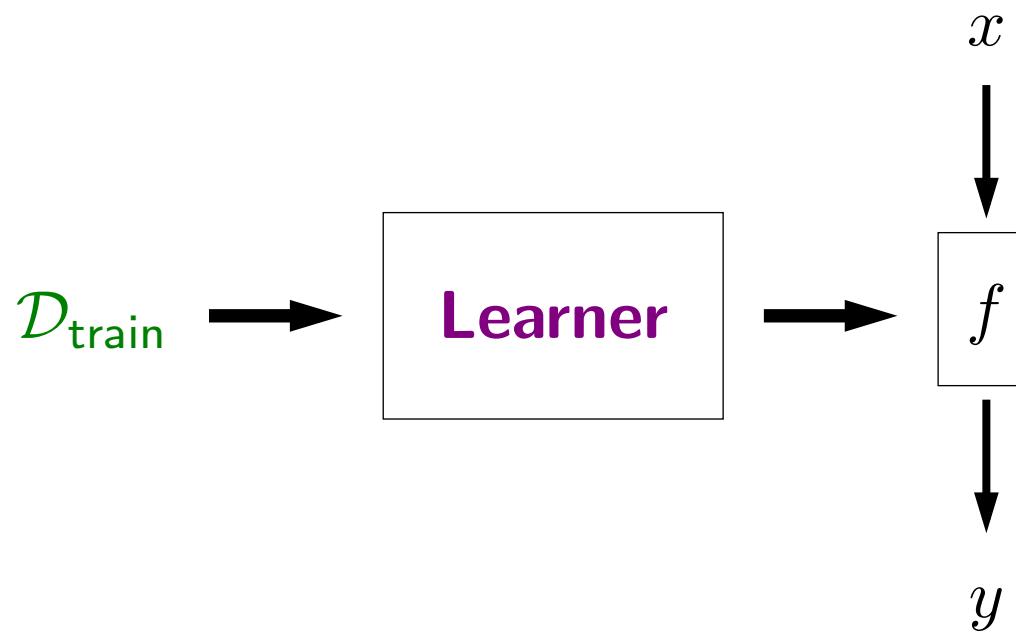
$$(x, y)$$

Training data: list of examples

$$\begin{aligned}\mathcal{D}_{\text{train}} = [ & \\ & ("...10m dollars...", +1), \\ & ("...CS221...", -1), \\ & ]\end{aligned}$$

- The starting point of machine learning is the data.
- For now, we will focus on **supervised learning**, in which our data provides both inputs and outputs, in contrast to unsupervised learning, which only provides inputs.
- A (supervised) **example** (also called a data point or instance) is simply an input-output pair  $(x, y)$ , which specifies that  $y$  is the ground-truth output for  $x$ .
- The **training data**  $\mathcal{D}_{\text{train}}$  is a multiset of examples (repeats are allowed, but this is not important), which forms a partial specification of the desired behavior of a predictor.

# Framework



- **Learning** is about taking the training data  $\mathcal{D}_{\text{train}}$  and producing a predictor  $f$ , which is a function that takes inputs  $x$  and tries to map them to outputs  $y = f(x)$ . One thing to keep in mind is that we want the predictor to approximately work even for examples that we have not seen in  $\mathcal{D}_{\text{train}}$ . This problem of generalization, which we will discuss two lectures from now, forces us to design  $f$  in a principled, mathematical way.
- We will first focus on examining what  $f$  is, independent of how the learning works. Then we will come back to learning  $f$  based on data.

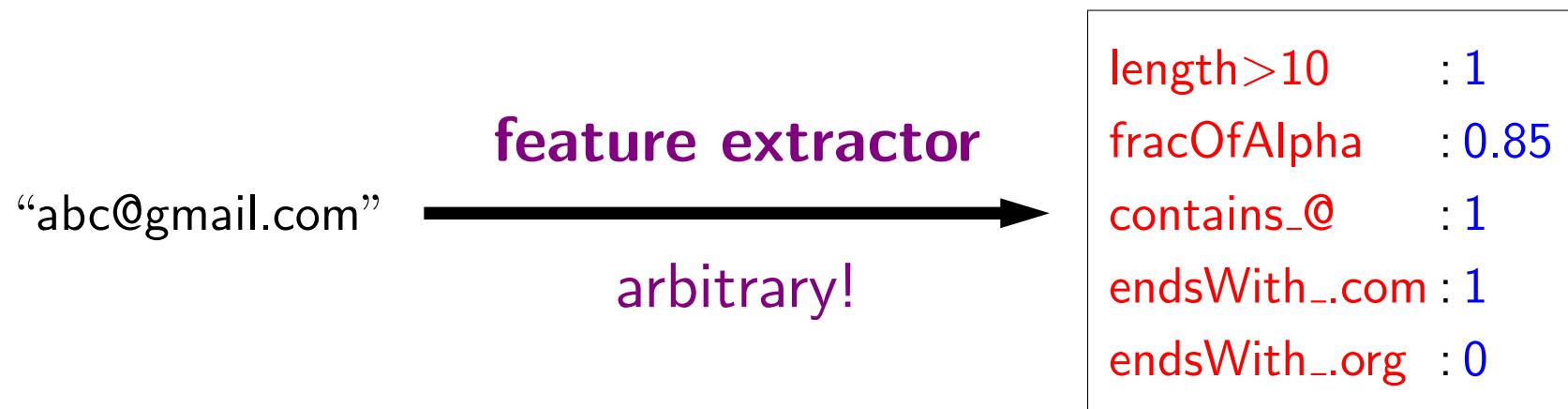


# Feature extraction

Example task: predict  $y$ , whether a string  $x$  is an email address

Question: what properties of  $x$  **might be** relevant for predicting  $y$ ?

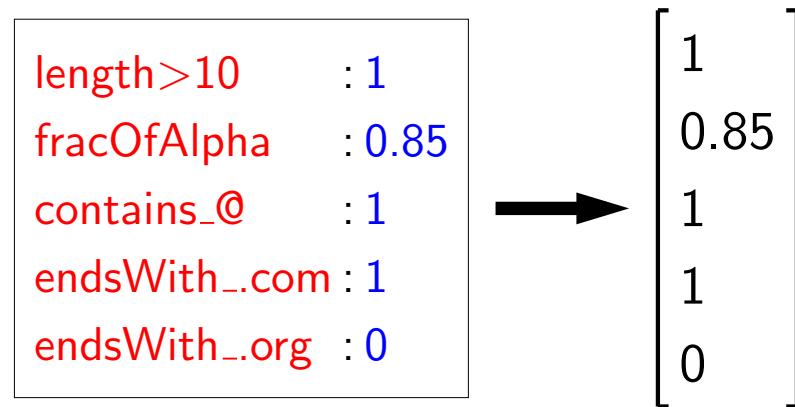
Feature extractor: Given input  $x$ , output a set of (**feature name**, **feature value**) pairs.



- We will consider predictors  $f$  based on **feature extractors**. Feature extraction is a bit of an art that requires intuition about both the task and also what machine learning algorithms are capable of.
- The general principle is that features should represent properties of  $x$  which **might be** relevant for predicting  $y$ . It is okay to add features which turn out to be irrelevant, since the learning algorithm can sort it out (though it might require more data to do so).

# Feature vector notation

Mathematically, feature vector doesn't need feature names:



## Definition: feature vector

For an input  $x$ , its feature vector is:

$$\phi(x) = [\phi_1(x), \dots, \phi_d(x)].$$

Think of  $\phi(x) \in \mathbb{R}^d$  as a point in a high-dimensional space.

- Each input  $x$  represented by a **feature vector**  $\phi(x)$ , which is computed by the feature extractor  $\phi$ . When designing features, it is useful to think of the feature vector as being a map from strings (feature names) to doubles (feature values). But formally, the feature vector  $\phi(x) \in \mathbb{R}^d$  is a real vector  $\phi(x) = [\phi_1(x), \dots, \phi_d(x)]$ , where each component  $\phi_j(x)$ , for  $j = 1, \dots, d$ , represents a feature.
- This vector-based representation allows us to think about feature vectors as a point in a (high-dimensional) vector space, which will later be useful for getting some geometric intuition.

# Weight vector

Weight vector: for each feature  $j$ , have real number  $w_j$  representing contribution of feature to prediction

```
length>10      :-1.2
fracOfAlpha    :0.6
contains_@      :3
endsWith_.com:2.2
endsWith_.org  :1.4
...
...
```

- So far, we have defined a feature extractor  $\phi$  that maps each input  $x$  to the feature vector  $\phi(x)$ . A **weight vector**  $w = [w_1, \dots, w_d]$  (also called a parameter vector or weights) specifies the contributions of each feature vector to the prediction.
- In the context of binary classification with binary features ( $\phi_j(x) \in \{0, 1\}$ ), the weights  $w_j \in \mathbb{R}$  have an intuitive interpretation. If  $w_j$  is positive, then the presence of feature  $j$  ( $\phi_j(x) = 1$ ) favors a positive classification. Conversely, if  $w_j$  is negative, then the presence of feature  $j$  favors a negative classification.
- Note that while the feature vector depends on the input  $x$ , the weight vector does not. This is because we want a single predictor (specified by the weight vector) that works on any input.

# Linear predictors

Weight vector  $\mathbf{w} \in \mathbb{R}^d$

length>10	:-1.2
fracOfAlpha	:0.6
contains_@	:3
endsWith_.com	:2.2
endsWith_.org	:1.4

Feature vector  $\phi(x) \in \mathbb{R}^d$

length>10	:1
fracOfAlpha	:0.85
contains_@	:1
endsWith_.com	:1
endsWith_.org	:0

**Score:** weighted combination of features

$$\mathbf{w} \cdot \phi(x) = \sum_{j=1}^d w_j \phi(x)_j$$

Example:  $-1.2(1) + 0.6(0.85) + 3(1) + 2.2(1) + 1.4(0) = 4.51$

- Given a feature vector  $\phi(x)$  and a weight vector  $w$ , we define the prediction **score** to be their inner product. The score intuitively represents the degree to which the classification is positive or negative.
- The predictor is linear because the score is a linear function of  $w$  (more on linearity in the next lecture).
- Again, in the context of binary classification with binary features, the score aggregates the contribution of each feature, weighted appropriately. We can think of each feature present as voting on the classification.

# Linear predictors

Weight vector  $\mathbf{w} \in \mathbb{R}^d$

Feature vector  $\phi(x) \in \mathbb{R}^d$

For binary classification:



**Definition: (binary) linear classifier**

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x)) = \begin{cases} +1 & \text{if } \mathbf{w} \cdot \phi(x) > 0 \\ -1 & \text{if } \mathbf{w} \cdot \phi(x) < 0 \\ ? & \text{if } \mathbf{w} \cdot \phi(x) = 0 \end{cases}$$

- We now have gathered enough intuition that we can formally define the predictor  $f$ . For each weight vector  $\mathbf{w}$ , we write  $f_{\mathbf{w}}$  to denote the predictor that depends on  $\mathbf{w}$  and takes the sign of the score.
- For the next few slides, we will focus on the case of binary classification. Recall that in this setting, we call the predictor a (binary) classifier.
- The case of  $f_{\mathbf{w}}(x) = ?$  is a boundary case that isn't so important. We can just predict +1 arbitrarily as a matter of convention.

# Geometric intuition

A binary classifier  $f_{\mathbf{w}}$  defines a hyperplane with normal vector  $\mathbf{w}$ .

( $\mathbb{R}^2 \implies$  hyperplane is a line;  $\mathbb{R}^3 \implies$  hyperplane is a plane)

Example:

$$\mathbf{w} = [2, -1]$$

$$\phi(x) \in \{[2, 0], [0, 2], [2, 4]\}$$

[whiteboard]

- So far, we have talked about linear predictors as weighted combinations of features. We can get a bit more insight by studying the **geometry** of the problem.
- Let's visualize the predictor  $f_w$  by looking at which points it classifies positive. Specifically, we can draw a ray from the origin to  $w$  (in two dimensions).
- Points which form an acute angle with  $w$  are classified as positive (dot product is positive), and points that form an obtuse angle with  $w$  are classified as negative. Points which are orthogonal  $\{z \in \mathbb{R}^d : w \cdot z = 0\}$  constitute the **decision boundary**.
- By changing  $w$ , we change the predictor  $f_w$  and thus the decision boundary as well.



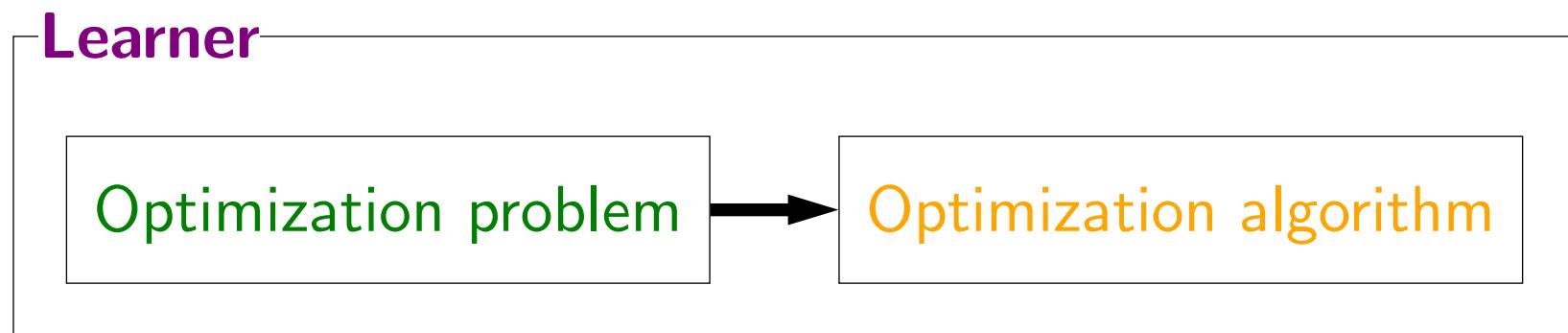
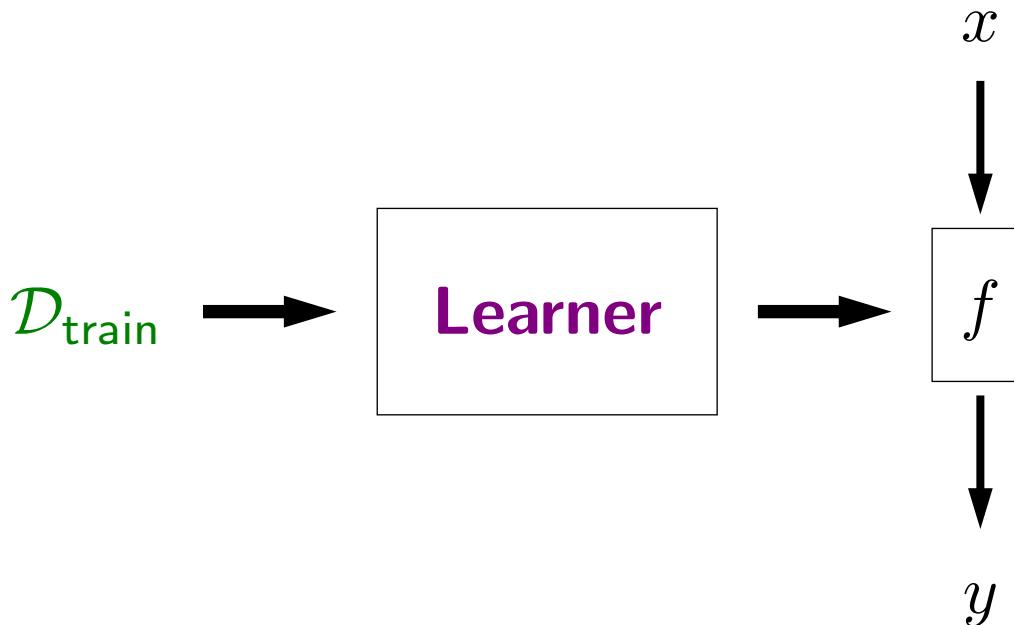
# Roadmap

Linear predictors

**Loss minimization**

Stochastic gradient descent

# Framework



- So far we have talked about linear predictors  $f_w$  which are based on a feature extractor  $\phi$  and a weight vector  $w$ . Now we turn to the problem of estimating (also known as fitting or learning)  $w$  from training data.
- The **loss minimization** framework is to cast learning as an optimization problem. Note the theme of separating your problem into a model (optimization problem) and an algorithm (optimization algorithm).

# Loss functions



## Definition: loss function

A loss function  $\text{Loss}(x, y, \mathbf{w})$  quantifies how unhappy you would be if you used  $\mathbf{w}$  to make a prediction on  $x$  when the correct output is  $y$ . It is the object we want to minimize.

# Score and margin

Correct label:  $y$

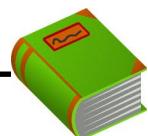
Predicted label:  $y' = f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$

Example:  $\mathbf{w} = [2, -1]$ ,  $\phi(x) = [2, 0]$ ,  $y = -1$



## Definition: score

The score on an example  $(x, y)$  is  $\mathbf{w} \cdot \phi(x)$ , how **confident** we are in predicting +1.



## Definition: margin

The margin on an example  $(x, y)$  is  $(\mathbf{w} \cdot \phi(x))y$ , how **correct** we are.

- Before we talk about what loss functions look like and how to learn  $\mathbf{w}$ , we introduce another important concept, the notion of a **margin**. Suppose the correct label is  $y \in \{-1, +1\}$ . The margin of an input  $x$  is  $\mathbf{w} \cdot \phi(x)y$ , which measures how correct the prediction that  $\mathbf{w}$  makes is. The larger the margin the better, and non-positive margins correspond to classification errors.
- Note that if we look at the actual prediction  $f_{\mathbf{w}}(x)$ , we can only ascertain whether the prediction was right or not. By looking at the score and the margin, we can get a more nuanced view into the behavior of the classifier.
- Geometrically, if  $\|\mathbf{w}\| = 1$ , then the margin of an input  $x$  is exactly the distance from its feature vector  $\phi(x)$  to the **decision boundary**.



# Question

When does a binary classifier err on an example?

margin less than 0

margin greater than 0

score less than 0

score greater than 0

# Binary classification

Example:  $\mathbf{w} = [2, -1]$ ,  $\phi(x) = [2, 0]$ ,  $y = -1$

Recall the binary classifier:

$$f_{\mathbf{w}}(x) = \text{sign}(\mathbf{w} \cdot \phi(x))$$

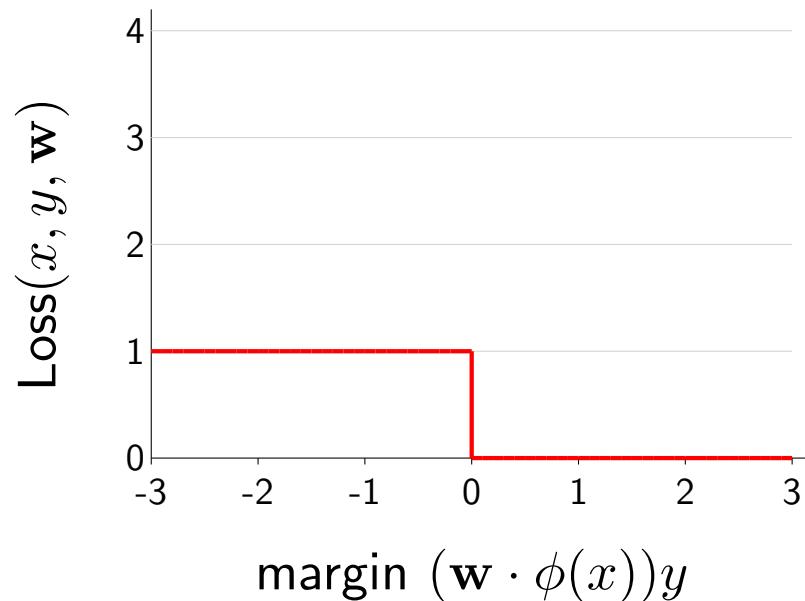


## Definition: zero-one loss

$$\begin{aligned}\text{Loss}_{0-1}(x, y, \mathbf{w}) &= \mathbf{1}[f_{\mathbf{w}}(x) \neq y] \\ &= \mathbf{1}[\underbrace{(\mathbf{w} \cdot \phi(x))y}_{\text{margin}} \leq 0]\end{aligned}$$

- Now let us define our first loss function, the **zero-one loss**. This corresponds exactly to our familiar notion of whether our predictor made a mistake or not. We can also write the loss in terms of the margin.

# Binary classification

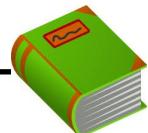
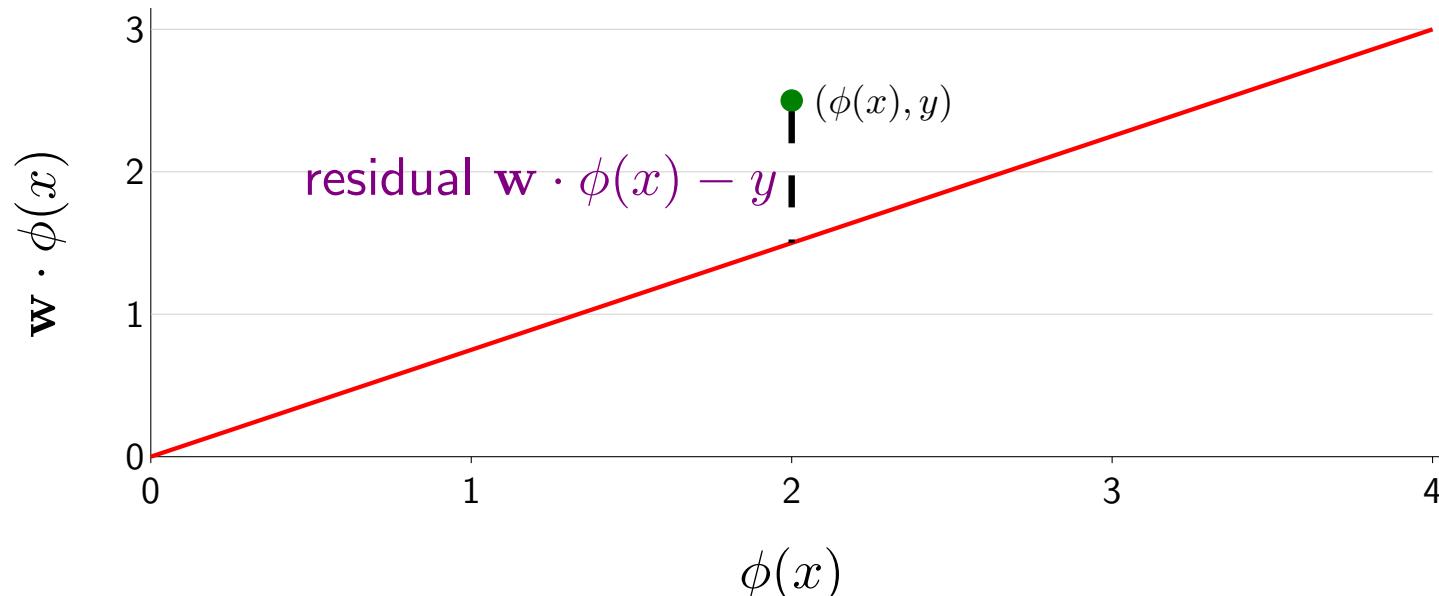


$$\text{Loss}_{0-1}(x, y, \mathbf{w}) = \mathbf{1}[(\mathbf{w} \cdot \phi(x))y \leq 0]$$

- We can plot the loss as a function of the margin. From the graph, it is clear that the loss is 1 when the margin is negative and 0 when it is positive.

# Linear regression

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$



## Definition: residual

The **residual** is  $(\mathbf{w} \cdot \phi(x)) - y$ , the amount by which prediction  $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$  overshoots the target  $y$ .

- Now let's turn for a moment to regression, where the output  $y$  is a real number rather than  $\{-1, +1\}$ . Here, the **zero-one loss** doesn't make sense, because it's unlikely that we're going to predict  $y$  exactly.
- Let's instead define the **residual** to measure how close the prediction  $f_w(x)$  is to the correct  $y$ . The residual will play the analogous role of the margin for classification and will let us craft an appropriate loss function.

# Linear regression

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$$



## Definition: squared loss

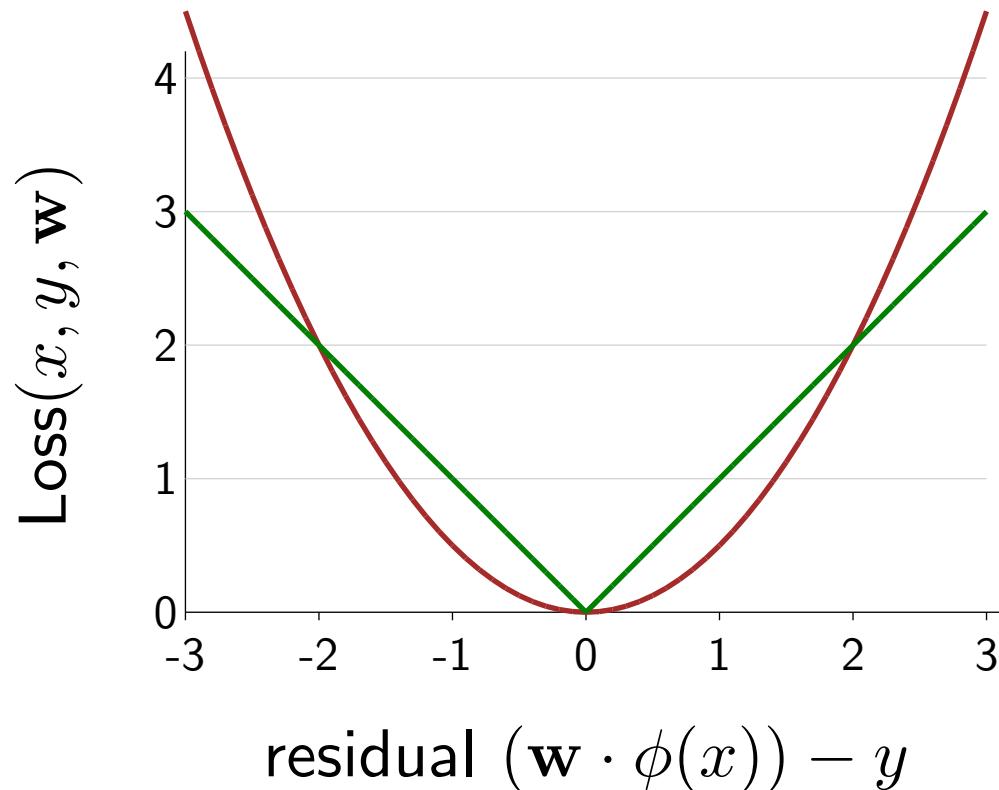
$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\underbrace{f_{\mathbf{w}}(x) - y}_{\text{residual}})^2$$

Example:

$$\mathbf{w} = [2, -1], \phi(x) = [2, 0], y = -1$$

$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = 25$$

# Regression loss functions



$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

$$\text{Loss}_{\text{absdev}}(x, y, \mathbf{w}) = |\mathbf{w} \cdot \phi(x) - y|$$

- A popular and convenient loss function to use in linear regression is the **squared loss**, which penalizes the residual of the prediction quadratically. If the predictor is off by a residual of 10, then the loss will be 100.
- An alternative to the squared loss is the **absolute deviation loss**, which simply takes the absolute value of the residual.

# Loss minimization framework

So far: one example,  $\text{Loss}(x, y, \mathbf{w})$  is easy to minimize.



**Key idea: minimize training loss**

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

$$\min_{\mathbf{w} \in \mathbb{R}^d} \text{TrainLoss}(\mathbf{w})$$

**Key:** need to set  $\mathbf{w}$  to make global tradeoffs — not every example can be happy.

- Note that on one example, both the squared and absolute deviation loss functions have the same minimum, so we cannot really appreciate the differences here. However, we are learning  $w$  based on a whole training set  $\mathcal{D}_{\text{train}}$ , not just one example. We typically minimize the **training loss** (also known as the training error or empirical risk), which is the average loss over all the training examples.
- Importantly, such an optimization problem requires making tradeoffs across all the examples (in general, we won't be able to set  $w$  to a single value that makes every example have low loss).

# Which regression loss to use? (skip)

Example:  $\mathcal{D}_{\text{train}} = \{(1, 0), (1, 2), (1, 1000)\}$ ,  $\phi(x) = x$

For least squares ( $L_2$ ) regression:

$$\text{Loss}_{\text{squared}}(x, y, \mathbf{w}) = (\mathbf{w} \cdot \phi(x) - y)^2$$

- $\mathbf{w}$  that minimizes training loss is **mean**  $y$
- **Mean**: tries to accommodate every example, popular

For least absolute deviation ( $L_1$ ) regression:

$$\text{Loss}_{\text{absdev}}(x, y, \mathbf{w}) = |\mathbf{w} \cdot \phi(x) - y|$$

- $\mathbf{w}$  that minimizes training loss is **median**  $y$
- **Median**: more robust to outliers

- Now the question of which loss we should use becomes more interesting.
- For example, consider the case where all the inputs are  $\phi(x) = 1$ . Essentially the problem becomes one of predicting a single value  $y^*$  which is the least offensive towards all the examples.
- If our loss function is the squared loss, then the optimal value is the mean  $y^* = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} y$ . If our loss function is the absolute deviation loss, then the optimal value is the median.
- The median is more robust to outliers: you can move the furthest point arbitrarily farther out without affecting the median. This makes sense given that the squared loss penalizes large residuals a lot more.
- In summary, this is an example of where the choice of the loss function has a qualitative impact on the weights learned, and we can study these differences in terms of the objective function without thinking about optimization algorithms.



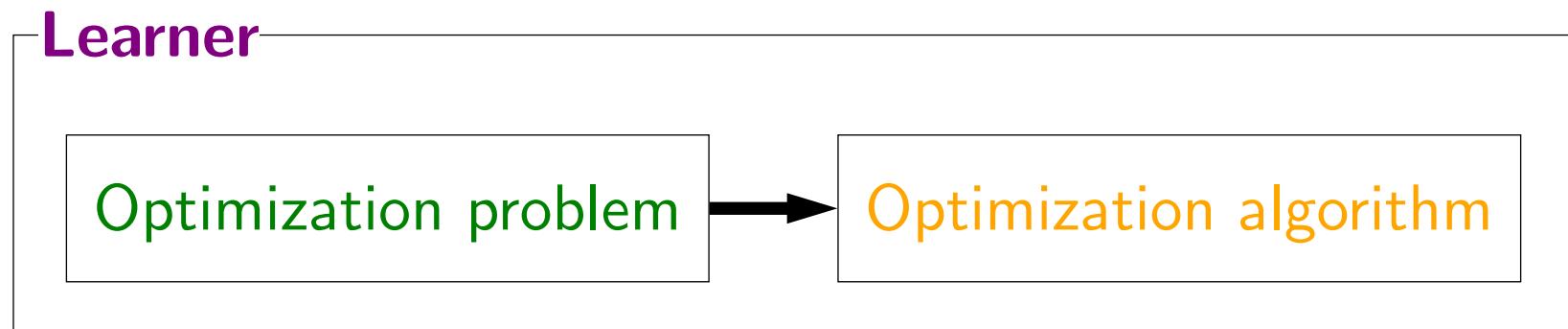
# Roadmap

Linear predictors

Loss minimization

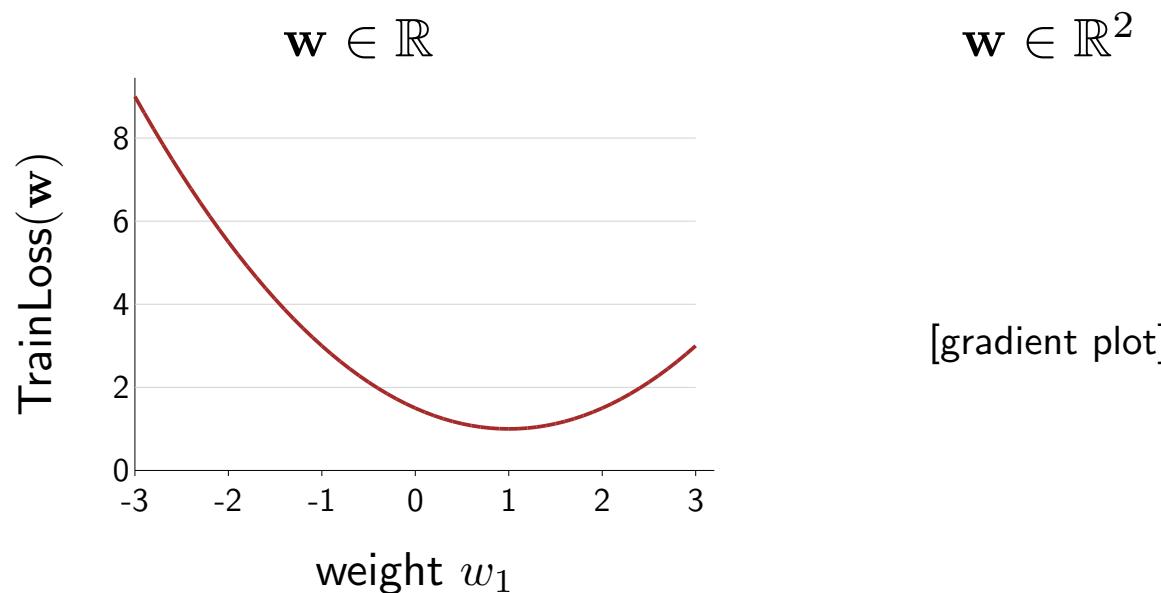
**Stochastic gradient descent**

# Learning as optimization



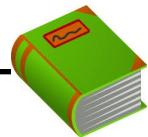
# Optimization problem

Objective:  $\min_{\mathbf{w} \in \mathbb{R}^d} \text{TrainLoss}(\mathbf{w})$



- Having defined a bunch of different objective functions that correspond to training loss, we would now like to optimize them — that is, obtain an algorithm that outputs the  $w$  where the objective function achieves the minimum value.

# How to optimize?



## Definition: gradient

The gradient  $\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$  is the direction that increases the loss the most.



## Algorithm: gradient descent

Initialize  $\mathbf{w} = [0, \dots, 0]$

For  $t = 1, \dots, T$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \underbrace{\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})}_{\text{gradient}}$$

- A general approach is to use **iterative optimization**, which essentially starts at some starting point  $w$  (say, all zeros), and tries to tweak  $w$  so that the objective function value decreases.
- To do this, we will rely on the gradient of the function, which tells us which direction to move in to decrease the objective the most. The gradient is a valuable piece of information, especially since we will often be optimizing in high dimensions ( $d$  on the order of thousands).
- This iterative optimization procedure is called **gradient descent**. Gradient descent has two **hyperparameters**, the **step size**  $\eta$  (which specifies how aggressively we want to pursue a direction) and the number of iterations  $T$ . Let's not worry about how to set them, but you can think of  $T = 100$  and  $\eta = 0.1$  for now.

# Least squares regression

Objective function:

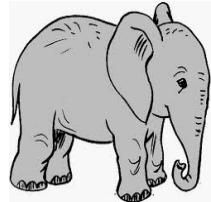
$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} (\mathbf{w} \cdot \phi(x) - y)^2$$

Gradient (use chain rule):

$$\nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} 2 \underbrace{(\mathbf{w} \cdot \phi(x) - y)}_{\text{prediction} - \text{target}} \phi(x)$$

[live solution]

- All that's left to do before we can use gradient descent is to compute the gradient of our objective function TrainLoss. The calculus can usually be done by hand; combinations of the product and chain rule suffice in most cases for the functions we care about.
- Note that the gradient often has a nice interpretation. For squared loss, it is the residual (prediction - target) times the feature vector  $\phi(x)$ .
- Note that for linear predictors, the gradient is always something times  $\phi(x)$  because  $w$  only affects the loss through  $w \cdot \phi(x)$ .



# Gradient descent is slow

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

Gradient descent:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

Problem: each iteration requires going over all training examples — expensive when have lots of data!

- We can now apply gradient descent on any of our objective functions that we defined before and have a working algorithm. But it is not necessarily the best algorithm.
- One problem (but not the only problem) with gradient descent is that it is slow. Those of you familiar with optimization will recognize that methods like Newton's method can give faster convergence, but that's not the type of slowness I'm talking about here.
- Rather, it is the slowness that arises in large-scale machine learning applications. Recall that the training loss is a sum over the training data. If we have one million training examples (which is, by today's standards, only a modest number), then each gradient computation requires going through those one million examples, and this must happen before we can make any progress. Can we make progress before seeing all the data?



# Stochastic gradient descent

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

Gradient descent (GD):

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

Stochastic gradient descent (SGD):

For each  $(x, y) \in \mathcal{D}_{\text{train}}$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$



**Key idea: stochastic updates**

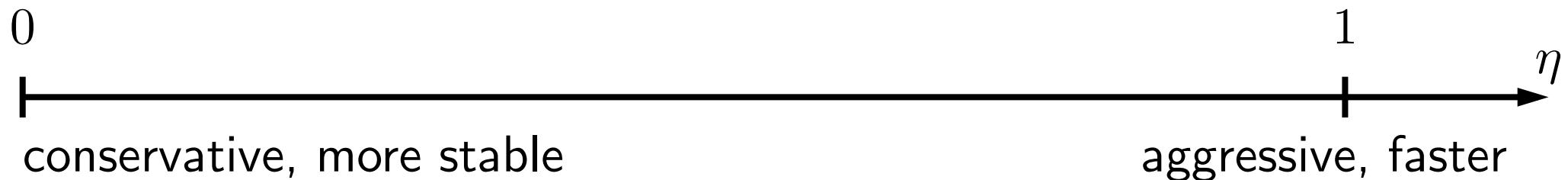
It's not about **quality**, it's about **quantity**.

- The answer is **stochastic gradient descent** (SGD). Rather than looping through all the training examples to compute a single gradient and making one step, SGD loops through the examples  $(x, y)$  and updates the weights  $w$  based on **each** example. Each update is not as good because we're only looking at one example rather than all the examples, but we can make many more updates this way.
- In practice, we often find that just performing one pass over the training examples with SGD, touching each example once, often performs comparably to taking ten passes over the data with GD.
- There are other variants of SGD. You can randomize the order in which you loop over the training data in each iteration, which is useful. Think about what would happen if you had all the positive examples first and the negative examples after that.

# Step size

$$\mathbf{w} \leftarrow \mathbf{w} - \underbrace{\eta}_{\text{step size}} \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

Question: what should  $\eta$  be?



Strategies:

- Constant:  $\eta = 0.1$
- Decreasing:  $\eta = 1/\sqrt{\# \text{ updates made so far}}$

- One remaining issue is choosing the step size, which in practice (and as we have seen) is actually quite important. Generally, larger step sizes are like driving fast. You can get faster convergence, but you might also get very unstable results and crash and burn. On the other hand, with smaller step sizes you get more stability, but you might get to your destination more slowly.
- A suggested form for the step size is to set the initial step size to 1 and let the step size decrease as the inverse of the square root of the number of updates we've taken so far. There are some nice theoretical results showing that SGD is guaranteed to converge in this case (provided all your gradients have bounded length).



# Summary so far

Linear predictors:

$$f_{\mathbf{w}}(x) \text{ based on score } \mathbf{w} \cdot \phi(x)$$

Loss minimization: learning as optimization

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

Stochastic gradient descent: optimization algorithm

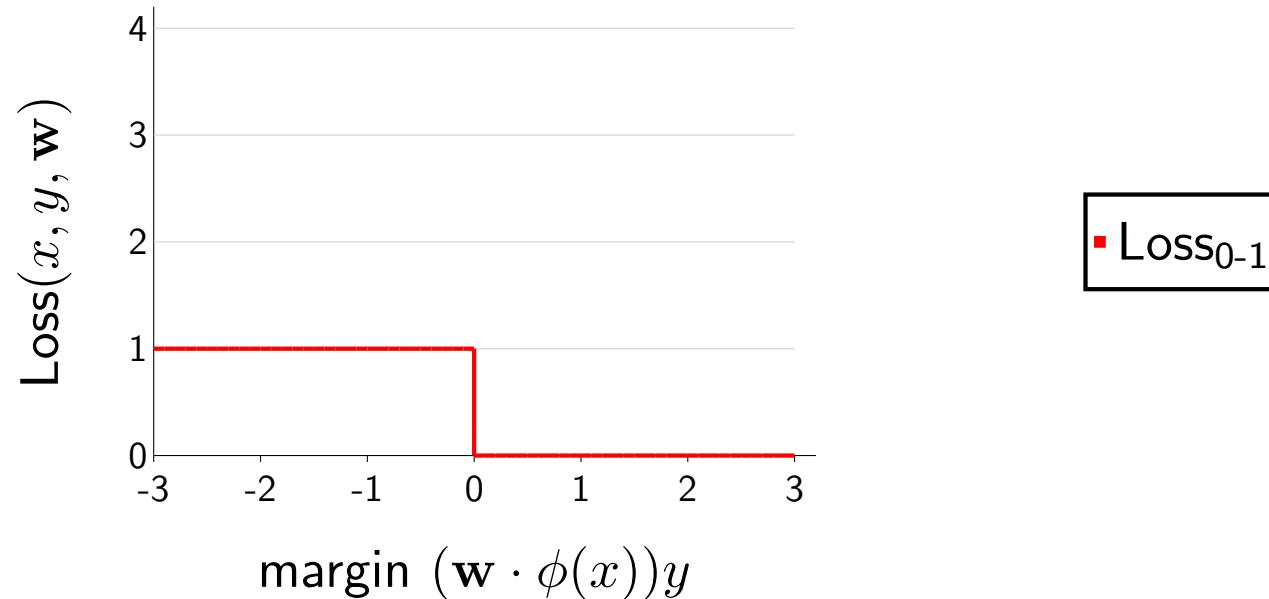
$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

Done for linear regression; what about classification?

- In summary we have seen (i) the functions we're considering (linear predictors), (ii) the criterion for choosing one (loss minimization), and (iii) an algorithm that goes after that criterion (SGD).
- We already worked out a linear regression example. What are good loss functions for binary classification?

# Zero-one loss

$$\text{Loss}_{0-1}(x, y, \mathbf{w}) = \mathbf{1}[(\mathbf{w} \cdot \phi(x))y \leq 0]$$



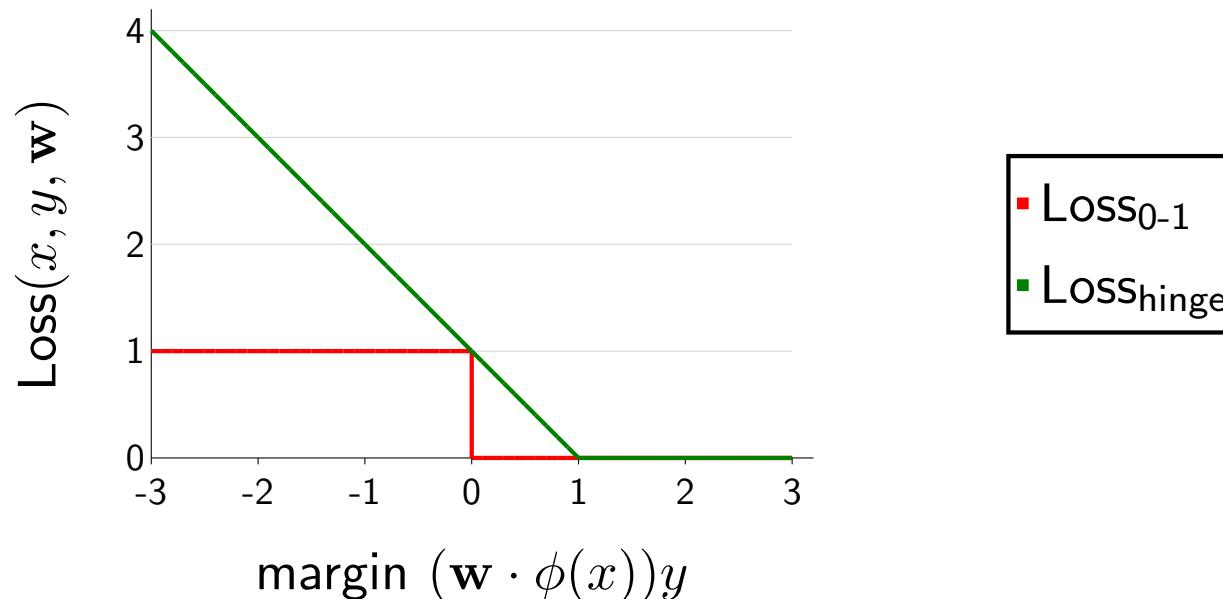
## Problems:

- Gradient of  $\text{Loss}_{0-1}$  is 0 everywhere, SGD not applicable
- $\text{Loss}_{0-1}$  is insensitive to how badly the model messed up

- Recall that we have the zero-one loss for classification. But the main problem with zero-one loss is that it's hard to optimize (in fact, it's provably NP hard in the worst case). And in particular, we cannot apply gradient-based optimization to it, because the gradient is zero (almost) everywhere.

# Hinge loss (SVMs)

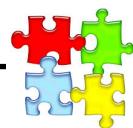
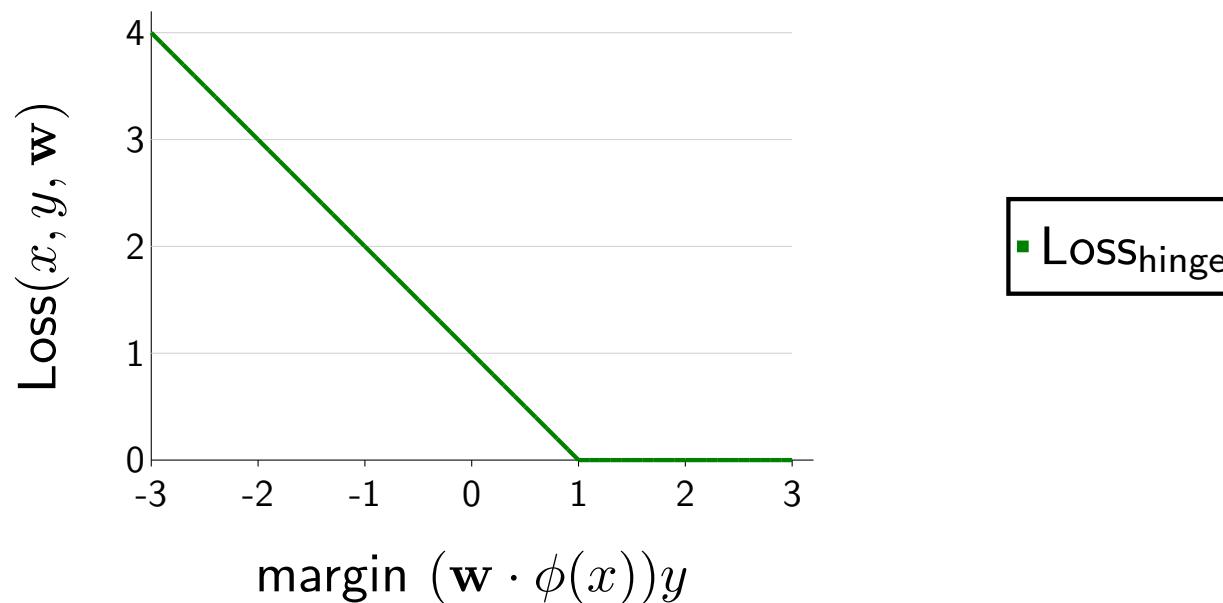
$$\text{Loss}_{\text{Hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$



- **Intuition:** hinge loss upper bounds 0-1 loss, has non-trivial gradient
- Try to increase margin if it is less than 1

- To fix this problem, we can use the **hinge loss**, which is an upper bound on the zero-one loss. Minimizing upper bounds are a general idea; the hope is that pushing down the upper bound leads to pushing down the actual function.
- Advanced: The hinge loss corresponds to the **Support Vector Machine** (SVM) objective function with one important difference. The SVM objective function also includes a **regularization penalty**  $\|\mathbf{w}\|^2$ , which prevents the weights from getting too large. We will get to regularization later in the course, so you needn't worry about this for now. But if you're curious, read on.
- Why should we penalize  $\|\mathbf{w}\|^2$ ? One answer is Occam's razor, which says to find the simplest hypothesis that explains the data. Here, simplicity is measured in the length of  $\mathbf{w}$ . This can be made formal using statistical learning theory (take CS229T if you want to learn more).
- Perhaps a less abstract and more geometric reason is the following. Recall that we defined the (algebraic) margin to be  $\mathbf{w} \cdot \phi(x)y$ . The actual (signed) distance from a point to the decision boundary is actually  $\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \phi(x)y$  — this is called the geometric margin. So the loss being zero (that is,  $\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = 0$ ) is equivalent to the algebraic margin being at least 1 (that is,  $\mathbf{w} \cdot \phi(x)y \geq 1$ ), which is equivalent to the geometric margin being larger than  $\frac{1}{\|\mathbf{w}\|}$  (that is,  $\frac{\mathbf{w}}{\|\mathbf{w}\|} \cdot \phi(x)y \geq \frac{1}{\|\mathbf{w}\|}$ ). Therefore, reducing  $\|\mathbf{w}\|$  increases the geometric margin. For this reason, SVMs are also referred to as max-margin classifiers.

# A gradient exercise



**Problem: Gradient of hinge loss**

Compute the gradient of

$$\text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \max\{1 - (\mathbf{w} \cdot \phi(x))y, 0\}$$

[whiteboard]

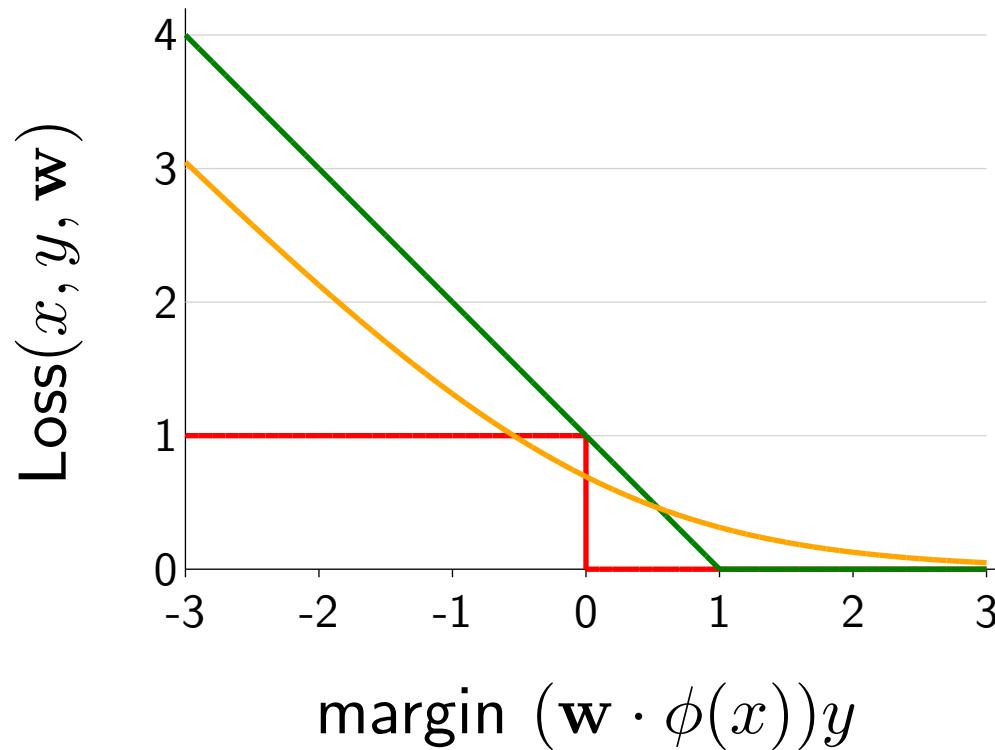
- You should try to "see" the solution before you write things down formally. Pictorially, it should be evident: when the margin is less than 1, then the gradient is the gradient of  $1 - (\mathbf{w} \cdot \phi(x))y$ , which is equal to  $-\phi(x)y$ . If the margin is larger than 1, then the gradient is the gradient of 0, which is 0. Combining the

two cases:  $\nabla_{\mathbf{w}} \text{Loss}_{\text{hinge}}(x, y, \mathbf{w}) = \begin{cases} -\phi(x)y & \text{if } \mathbf{w} \cdot \phi(x)y < 1 \\ 0 & \text{if } \mathbf{w} \cdot \phi(x)y > 1. \end{cases}$

- What about when the margin is exactly 1? Technically, the gradient doesn't exist because the hinge loss is not differentiable there. Fear not! Practically speaking, at the end of the day, we can take either  $-\phi(x)y$  or 0 (or anything in between).
- Technical note (can be skipped): given  $f(\mathbf{w})$ , the gradient  $\nabla f(\mathbf{w})$  is only defined at points  $\mathbf{w}$  where  $f$  is differentiable. However, subdifferentials  $\partial f(\mathbf{w})$  are defined at every point (for convex functions). The subdifferential is a set of vectors called subgradients  $z \in \partial f(\mathbf{w})$  which define linear underapproximations to  $f$ , namely  $f(\mathbf{w}) + z \cdot (\mathbf{w}' - \mathbf{w}) \leq f(\mathbf{w}')$  for all  $\mathbf{w}'$ .

# Logistic regression

$$\text{Loss}_{\text{logistic}}(x, y, \mathbf{w}) = \log(1 + e^{-(\mathbf{w} \cdot \phi(x))y})$$



- **Intuition:** Try to increase margin even when it already exceeds 1

- Another popular loss function used in machine learning is the **logistic loss**. The main property of the logistic loss is no matter how correct your prediction is, you will have non-zero loss, and so there is still an incentive (although a diminishing one) to push the margin even larger. This means that you'll update on every single example.
- There are some connections between logistic regression and probabilistic models, which we will get to later.



# Summary so far

$$\underbrace{\mathbf{w} \cdot \phi(x)}_{\text{score}}$$

	Classification	Linear regression
Predictor $f_{\mathbf{w}}$	$\text{sign(score)}$	score
Relate to correct $y$	margin (score $y$ )	residual (score – $y$ )
Loss functions	zero-one hinge logistic	squared absolute deviation
Algorithm	SGD	SGD

# Next lecture

Linear predictors:

$$f_{\mathbf{w}}(x) \text{ based on score } \mathbf{w} \cdot \phi(x)$$

Which feature vector  $\phi(x)$  to use?

Loss minimization:

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

How do we **generalize** beyond the training set?

CS221 Section 1

# Foundations

# Roadmap

Matrix Calculus

Python

Complexity

Recurrence Relations

Probability Theory

# Notation and Basic Properties

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \quad A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

$$\|\mathbf{v}\|_2^2 = \mathbf{v} \cdot \mathbf{v} = \mathbf{v}^T \mathbf{v}$$

$$(\mathbf{A} + \mathbf{B})^T = \mathbf{A}^T + \mathbf{B}^T$$

$$(\mathbf{AB})^T = \mathbf{B}^T \mathbf{A}^T$$

# Matrix Calculus

Compute  $\nabla_{\mathbf{w}} f(\mathbf{w})$

$$f(\mathbf{w}) = \mathbf{a} \cdot \mathbf{w} + b \|\mathbf{w}\|_2^2 + \mathbf{w}^\top C \mathbf{w}$$

# Matrix Calculus

Compute  $\nabla_{\mathbf{w}} f(\mathbf{w})$

$$f(\mathbf{w}) = \mathbf{a} \cdot \mathbf{w} + b \|\mathbf{w}\|_2^2 + \mathbf{w}^\top C \mathbf{w}$$

Observe that:

$$\nabla_{\mathbf{w}} \mathbf{a} \cdot \mathbf{w} = \mathbf{a}$$

$$\nabla_{\mathbf{w}} \|\mathbf{w}\|_2^2 = \nabla_{\mathbf{w}} \mathbf{w} \cdot \mathbf{w} = 2\mathbf{w}$$

$$\nabla_{\mathbf{w}} \mathbf{w}^\top C \mathbf{w} = (C + C^\top) \mathbf{w}$$

# Matrix Calculus

**example:** Find the gradient of  $g : \mathbf{R}^m \rightarrow \mathbf{R}$ ,

$$g(y) = \log \sum_{i=1}^m \exp(y_i).$$

# Matrix Calculus

**example:** Find the gradient of  $g : \mathbf{R}^m \rightarrow \mathbf{R}$ ,

$$g(y) = \log \sum_{i=1}^m \exp(y_i).$$

**solution.**

$$\nabla g(y) = \frac{1}{\sum_{i=1}^m \exp y_i} \begin{bmatrix} \exp y_1 \\ \vdots \\ \exp y_m \end{bmatrix}$$

# Roadmap

Matrix Calculus

Python

Complexity

Recurrence Relations

Probability Theory

# Syntactic Sugar

```
a = "Can I skip a CS221 homework? No dice!!"
```

The **split** command creates a list from a string with blank spaces as delimiters by default. You can also specify a different delimiter.

```
b = a.split()  
b
```

```
['Can', 'I', 'skip', 'a', 'CS221', 'homework?', 'No', 'dice!!']
```

```
c = [len(_) for _ in b]  
c
```

```
[3, 1, 4, 1, 5, 9, 2, 6]
```

Note that `_` is a valid variable name (usually for one-time use). An equivalent for-loop would be:

```
c = []  
for _ in b:  
    c.append(len(_))  
c
```

```
[3, 1, 4, 1, 5, 9, 2, 6]
```

# Syntactic Sugar

```
# Task: Find sum of X values less than 5

points = [(1, 2), (2, 6), (3, 3), (8, 9)]

x_total = 0
for point in points:
    if point[0] < 5:
        x_total = x_total + point[0]

x_total
```

6

```
sum([x for x, _ in points if x < 5])
```

6

**enumerate** is a useful built-in:

```
a = enumerate(['a', 'b', 'c'])

list(enumerate(['a', 'b', 'c']))

[(0, 'a'), (1, 'b'), (2, 'c')]
```

# Syntactic Sugar

```
c = [len(_) for _ in b]
```

```
c
```

```
[3, 1, 4, 1, 5, 9, 2, 6]
```

```
c[2] # The index starts from 0
```

```
4
```

```
c[-1] # You can count backward from the right: c[-1] is the last element
```

```
6
```

```
c[1:4] # Indexes are inclusive:exclusive
```

```
[1, 4, 1]
```

```
c[:4] # = c[0:4]
```

```
[3, 1, 4, 1]
```

```
c[4:] # = c[4:len(c)]
```

```
[5, 9, 2, 6]
```

```
c[:-1] # = c[0:len(c)-1], cutting out the last element in the list
```

```
[3, 1, 4, 1, 5, 9, 2]
```

# References

- Official Documentation (has a tutorial):

<https://docs.python.org/>

- Learn X in Y minutes:

<http://learnxinyminutes.com/docs/python/>

- You don't need to know numpy. But if you want to:

<http://nbviewer.ipython.org/gist/rpmuller/5920182>

# Roadmap

Matrix Calculus

Python

Complexity

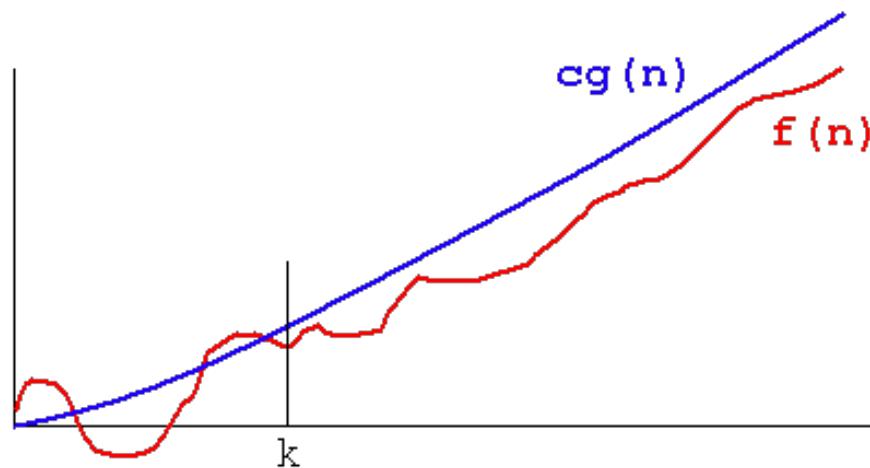
Recurrence Relations

Probability Theory

# Time Complexity

Big O Notation:

We say  $f(n)$  is  $O(g(n))$  if there exist positive constants  $C$  and  $K$  such that  $0 \leq f(n) \leq Cg(n)$  for all  $n \geq k$  (where  $C, K$  are constants)



# Time Complexity

Examples:

```
max_val = -infinite
for i = 1 to n:
    max_val = max(arr[i], max_val)
```

```
for i = 1 to n - 1:
    for j = 1 to n - i:
        if arr[j] > arr[j+1]:
            swap(arr[j], arr[j+1])
```

Note that:  $1 + 2 + \dots + k = k(k + 1)/2$

# Roadmap

Matrix Calculus

Python

Complexity

Recurrence Relations

Probability Theory

# Coin Payment

## Problem



Suppose you have an unlimited supply of coins with values 2, 3, and 5 cents

How many ways can you pay for an item costing 12 cents?  
*How about  $n$  cents?*

# Coin Payment

**Recurrence Relation:** Break down into smaller problems

**Memoization:** Remember what you already calculated

# Roadmap

Matrix Calculus

Python

Complexity

Recurrence Relations

Probability Theory

# Random Variables

Discrete:

$$\mathbb{P}(A = a) \quad \text{or} \quad p_A(a)$$

Continuous:

$$\mathbb{P}(A \leq a)$$

$$f_A(a)$$

$$\mathbb{P}(A \leq c) = \int_{--}^c f_A(a) da$$

# Random Variables

$A = 0 \quad A = 1 \quad A = 2 \quad A = 3$

$B = 0 \quad 0.1 \quad 0.25 \quad 0.1 \quad 0.05$

$B = 1 \quad 0.15 \quad 0 \quad 0.15 \quad 0.2$

- What is  $\mathbb{P}(A = 2)$
- What is  $\mathbb{P}(A = 2 | B = 1)$

# Random Variables

Independence:

$$\forall a, b, \quad \mathbb{P}(A = a, B = b) = \mathbb{P}(A = a)\mathbb{P}(B = b)$$

$$\forall a, b, \quad f_{A,B}(a, b) = f_A(a)f_B(b)$$

Expectation:

$$\mathbb{E}[A] = \sum_a a \mathbb{P}[A = a]$$

$$\mathbb{E}[A] = \int a f_A(a) da$$

# Random Variables

$A = 0 \quad A = 1 \quad A = 2 \quad A = 3$

$B = 0 \quad 0.1 \quad 0.25 \quad 0.1 \quad 0.05$

$B = 1 \quad 0.15 \quad 0 \quad 0.15 \quad 0.2$

- Are  $A$  and  $B$  independent?
- What are  $\mathbb{E}[A]$ ,  $\mathbb{E}[B]$ ,  $\mathbb{E}[A + B]$

Linearity of Expectation:  $\mathbb{E}[A + B] = \mathbb{E}[A] + \mathbb{E}[B]$

True even when  $A$  and  $B$  are dependent!

# Hat Toss

## Problem

Suppose  $n$  hatted people toss their hats into the air and pick up one hat at random

In expectation, how many people get their own hats back?

Hint: linearity of expectation

# Coin Tossing

## Problem

You are given a fair coin with sides heads and tails.

What is the expected number of flips until you get 3 heads in a row?

How about  $n$  heads in a row?

Questions?



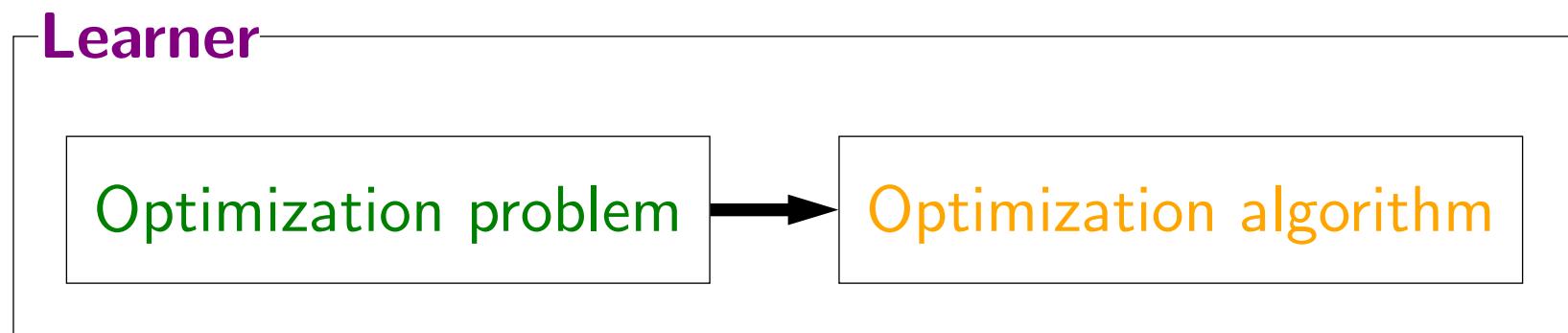
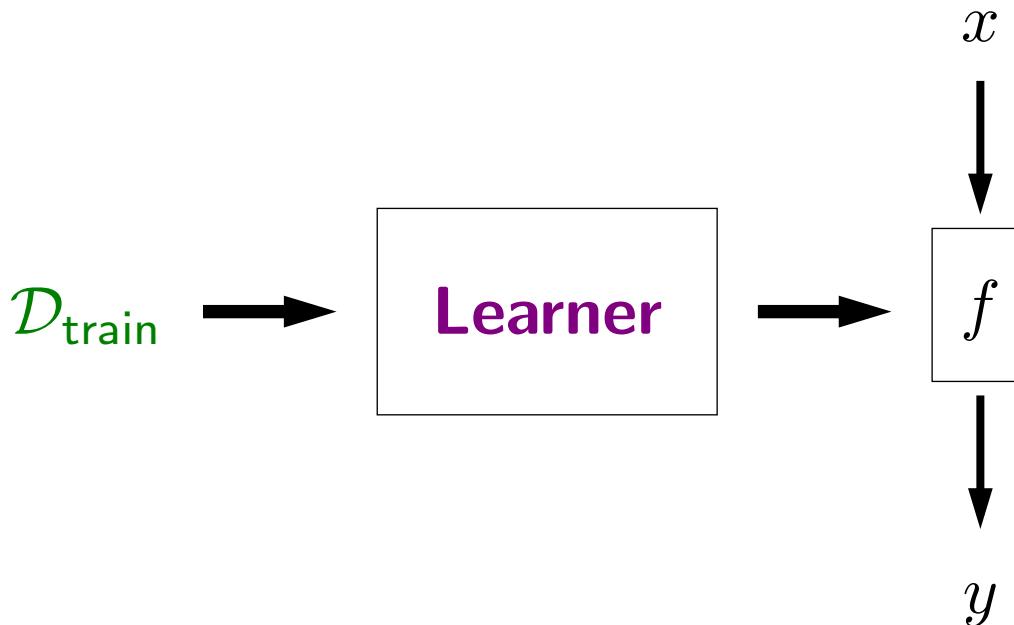
# Lecture 3: Machine learning II



# Announcements

- Homework 1 (foundations) due tomorrow at **11pm**; please test submit early, since you are responsible for any technical issues you encounter; don't email it to us
- Homework 2 (sentiment) is out
- Section this Thursday at 3:30pm

# Framework



- Recall from last time that learning is the process of taking training data and turning it into a model (predictor).
- Last time, we started by studying the predictor  $f$ , concerning ourselves with linear predictors based on the score  $\mathbf{w} \cdot \phi(x)$ , where  $\mathbf{w}$  is the weight vector we wish to learn and  $\phi$  is the feature extractor that maps an input  $x$  to some feature vector  $\phi(x) \in \mathbb{R}^d$ , turning something that is domain-specific (images, text) into a mathematical object.
- Then we looked at how to learn such a predictor by formulating an optimization problem and developing an algorithm to solve that problem.

# Review: optimization problem



**Key idea: minimize training loss**

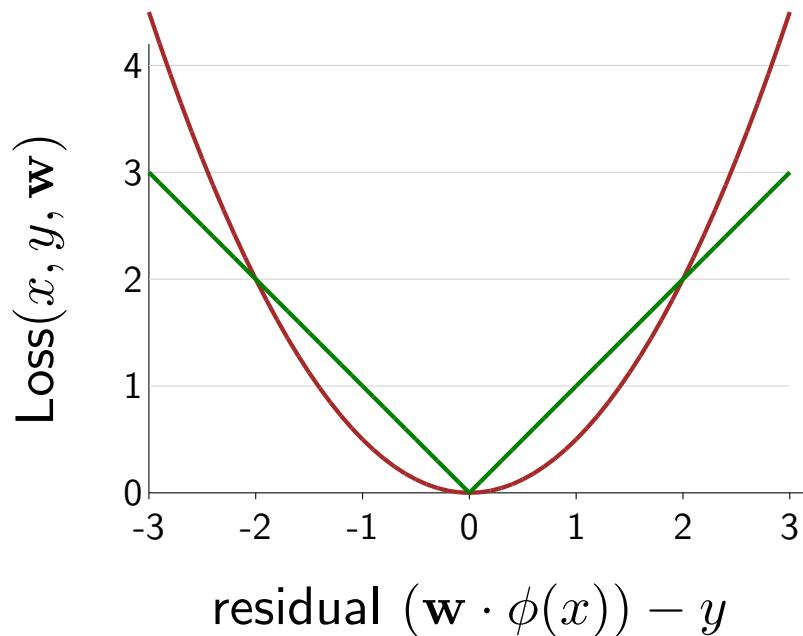
$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

$$\min_{\mathbf{w} \in \mathbb{R}^d} \text{TrainLoss}(\mathbf{w})$$

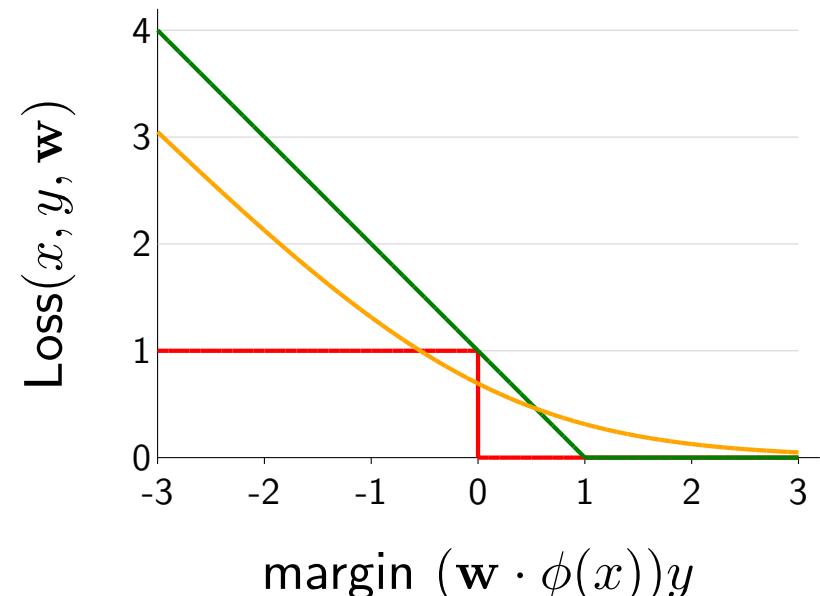
- Recall that the optimization problem was to minimize the training loss, which is the average loss over all the training examples.

# Review: loss functions

## Regression



## Binary classification



**Loss captures properties of the desired predictor**

- The actual loss function depends on what we're trying to accomplish. Generally, the loss function takes the score  $\mathbf{w} \cdot \phi(x)$ , compares it with the correct output  $y$  to form either the residual (for regression) or the margin (for classification).
- Regression losses are smallest when the residual is close to zero. Classification losses are smallest when the margin is large. Which loss function we choose depends on the desired properties. For example, the absolute deviation loss for regression is robust against outliers. The logistic loss for classification never relents in encouraging large margin.

# A regression example

Training data:

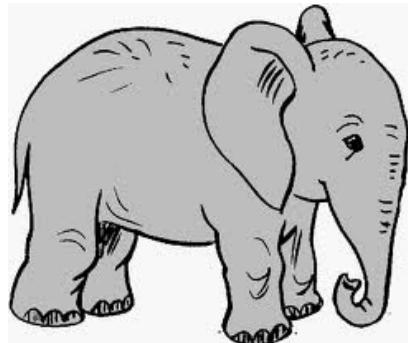
$x$	$y$	$\text{Loss}(x, y, w) = (\mathbf{w} \cdot \phi(x) - y)^2$
[1, 0]	2	$(w_1 - 2)^2$
[1, 0]	4	$(w_1 - 4)^2$
[0, 1]	-1	$(w_2 - (-1))^2$

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{3}((w_1 - 2)^2 + (w_1 - 4)^2 + (w_2 - (-1))^2)$$

[whiteboard]

- Note that we've been talking about the loss on a single example, and plotting it in 1D against the residual or the margin. Recall that what we're actually optimizing is the training loss, which sums over all data points. To help visualize the connection between a single loss plot and the more general picture, consider the simple example of linear regression on three data points:  $([1, 0], 2)$ ,  $([1, 0], 4)$ , and  $([0, 1], -1)$ , where  $\phi(x) = x$ .
- Let's try to draw the training loss, which is a function of  $\mathbf{w} = [w_1, w_2]$ . Specifically, the training loss is  $\frac{1}{3}((w_1 - 2)^2 + (w_1 - 4)^2 + (w_2 - (-1))^2)$ . The first two points contribute a quadratic term sensitive to  $w_1$ , and the third point contributes a quadratic term sensitive to  $w_2$ . When you combine them, you get a quadratic centered at  $[3, -1]$ .

# Review: optimization algorithms

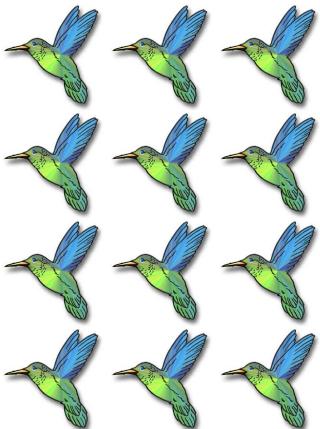


## Algorithm: gradient descent

Initialize  $\mathbf{w} = [0, \dots, 0]$

For  $t = 1, \dots, T$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$



## Algorithm: stochastic gradient descent

Initialize  $\mathbf{w} = [0, \dots, 0]$

For  $t = 1, \dots, T$ :

For  $(x, y) \in \mathcal{D}_{\text{train}}$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta_t \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

- Finally, we introduced two very simple algorithms to minimize the training loss, both based on iteratively computing the gradient of the objective with respect to the parameters  $w$  and stepping in the opposite direction of the gradient. Think about a ball at the current weight vector and rolling it down on the surface of the training loss objective.
- Gradient descent (GD) computes the gradient of the full training loss, which can be slow for large datasets.
- Stochastic gradient descent (SGD), which approximates the gradient of the training loss with the loss at a single example, generally takes less time.
- In both cases, one must be careful to set the step size  $\eta$  properly (not too big, not too small).



# Question

Can we obtain decision boundaries which are circles by using linear classifiers?

Yes

No

- The answer is yes.
- This might seem paradoxical since we are only working with linear classifiers. But as we will see later, **linear** refers to the relationship between the weight vector  $w$  and the prediction score (not the input  $x$ , which might not even be a real vector), whereas the decision boundary refers to how the prediction varies as a function of  $x$ .
- Advanced: Sometimes people might think that linear classifiers are not expressive, and that you need neural networks to get expressive and non-linear classifiers. This is false. You can build arbitrarily expressive models with the machinery of linear classifiers (see kernel methods). The advantages of neural networks are the computational benefits and the inductive bias that comes from the particular neural network architecture.



# Roadmap

Features

Neural networks

Gradients without tears

Nearest neighbors

- The first half of this lecture is about thinking about the feature extractor  $\phi$ . Features are a critical part of machine learning which often do not get as much attention as they deserve. Ideally, they would be given to us by a domain expert, and all we (as machine learning people) have to do is to stick them into our learning algorithm. While one can get considerable mileage out of doing this, the interface between general-purpose machine learning and domain knowledge is often nuanced, so to be successful, it pays to understand this interface.
- In the second half of this lecture, we return to learning, rip out the linear predictors that we had from before, and show how we can build more powerful neural network classifiers given the features that we extracted.

# Two components

Score (drives prediction):

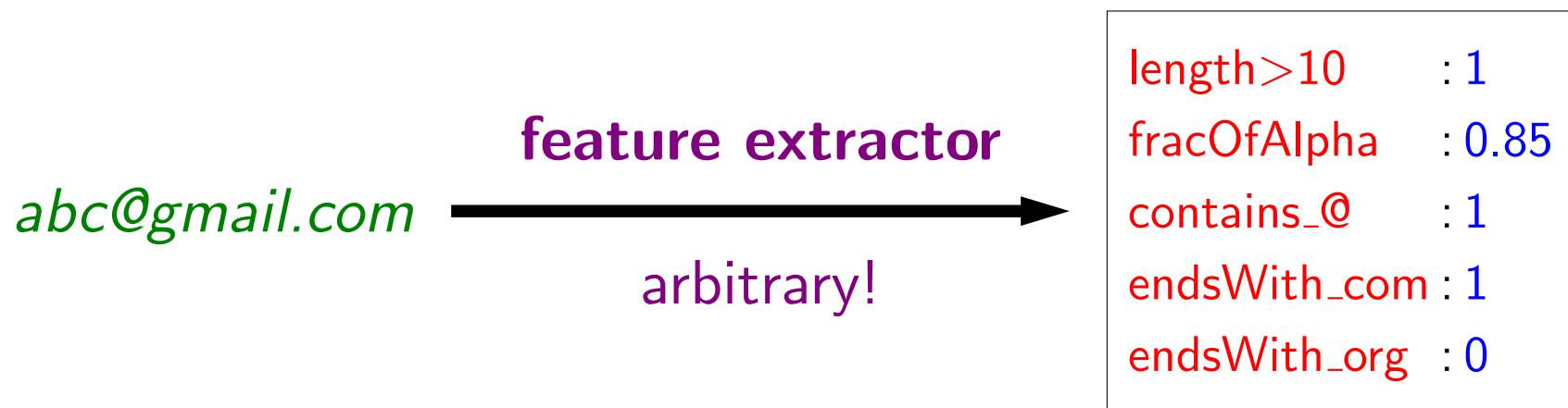
$$\mathbf{w} \cdot \phi(x)$$

- Previous: **learning** chooses  $\mathbf{w}$  via optimization
- Next: **feature extraction** specifies  $\phi(x)$  based on domain knowledge

- As a reminder, the prediction is driven by the score  $\mathbf{w} \cdot \phi(x)$ . In regression, we predict the score directly, and in binary classification, we predict the sign of the score.
- Both  $\mathbf{w}$  and  $\phi(x)$  play an important role in prediction. So far, we have fixed  $\phi(x)$  and used learning to set  $\mathbf{w}$ . Now, we will explore how  $\phi(x)$  affects the prediction.

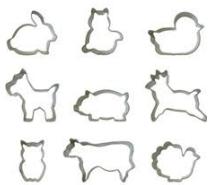
# Organization of features

Task: predict whether a string is an email address



Which features to include? Need an organizational principle...

- How would we go about creating good features?
- Here, we used our prior knowledge to define certain features (`contains_@`) which we believe are helpful for detecting email addresses.
- But this is ad-hoc: which strings should we include? We need a more systematic way to go about this.



# Feature templates



## Definition: feature template (informal)

A **feature template** is a group of features all computed in a similar way.

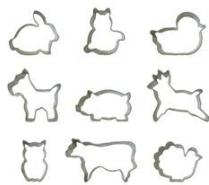
Input:

*abc@gmail.com*

Some feature templates:

- Length greater than \_\_\_
- Last three characters equals \_\_\_
- Contains character \_\_\_
- Pixel intensity of position \_\_\_, \_\_\_

- A useful organization principle is a **feature template**, which groups all the features which are computed in a similar way. (People often use the word "feature" when they really mean "feature template".)
- A feature template also allows us to define a set of related features (contains\_@, contains\_a, contains\_b). This reduces the amount of burden on the feature engineer since we don't need to know which particular characters are useful, but only that existence of certain single characters is a useful cue to look at.
- We can write each feature template as a English description with a blank (\_\_\_\_), which is to be filled in with an arbitrary string. Also note that feature templates are most natural for defining binary features, ones which take on value 1 (true) or 0 (false).
- Note that an isolated feature (fraction of alphanumeric characters) can be treated as a trivial feature template with no blanks to be filled.
- As another example, if  $x$  is a  $k \times k$  image, then  $\{\text{pixelIntensity}_{ij} : 1 \leq i, j \leq k\}$  is a feature template consisting of  $k^2$  features, whose values are the pixel intensities at various positions of  $x$ .



# Feature templates

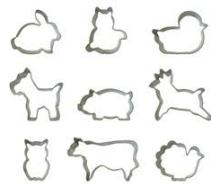
Feature template: last three characters equals \_\_\_

*abc@gmail.com*



```
endsWith_aaa : 0
endsWith_aab : 0
endsWith_aac : 0
...
endsWith_com : 1
...
endsWith_zzz : 0
```

- This is an example of one feature template mapping onto a group of  $m^3$  features, where  $m$  (26 in this example) is the number of possible characters.



# Sparsity in feature vectors

Feature template: last character equals \_\_\_

*abc@gmail.com*



endsWith_a	: 0
endsWith_b	: 0
endsWith_c	: 0
endsWith_d	: 0
endsWith_e	: 0
endsWith_f	: 0
endsWith_g	: 0
endsWith_h	: 0
endsWith_i	: 0
endsWith_j	: 0
endsWith_k	: 0
endsWith_l	: 0
endsWith_m	: 1
endsWith_n	: 0
endsWith_o	: 0
endsWith_p	: 0
endsWith_q	: 0
endsWith_r	: 0
endsWith_s	: 0
endsWith_t	: 0
endsWith_u	: 0
endsWith_v	: 0
endsWith_w	: 0
endsWith_x	: 0
endsWith_y	: 0
endsWith_z	: 0

Inefficient to represent all the zeros...

- In general, a feature template corresponds to many features. It would be inefficient to represent all the features explicitly. Fortunately, the feature vectors are often **sparse**, meaning that most of the feature values are 0. It is common for all but one of the features to be 0. This is known as a **one-hot representation** of a discrete value such as a character.

# Feature vector representations

```
fracOfAlpha : 0.85  
contains_a   : 0  
...  
contains_@   : 1  
...
```

Array representation (good for dense features):

```
[0.85, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

Map representation (good for sparse features):

```
{"fracOfAlpha": 0.85, "contains_@": 1}
```

- Let's now talk a bit more about implementation. There are two common ways to define features: using arrays or using maps.
- **Arrays** assume a fixed ordering of the features and represent the feature values as an array. This representation is appropriate when the number of nonzeros is significant (the features are dense). Arrays are especially efficient in terms of space and speed (and you can take advantage of GPUs). In computer vision applications, features (e.g., the pixel intensity features) are generally dense, so array representation is more common.
- However, when we have sparsity (few nonzeros), it is typically more efficient to represent the feature vector as a **map** from strings to doubles rather than a fixed-size array of doubles. The features not in the map implicitly have a default value of zero. This sparse representation is very useful in natural language processing, and is what allows us to work effectively over trillions of features. In Python, one would define a feature vector  $\phi(x)$  as the dictionary `{"endsWith_+x[-3:]: 1}`. Maps do incur extra overhead compared to arrays, and therefore maps are much slower when the features are not sparse.
- Finally, it is important to be clear when describing features. Saying "length" might mean that there is one feature whose value is the length of  $x$  or that there could be a feature template "length is equal to  $\_\_\_$ ". These two encodings of the same information can have a drastic impact on prediction accuracy when using a linear predictor, as we'll see later.

# Hypothesis class

Predictor:

$$f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x) \quad \text{or} \quad \text{sign}(\mathbf{w} \cdot \phi(x))$$



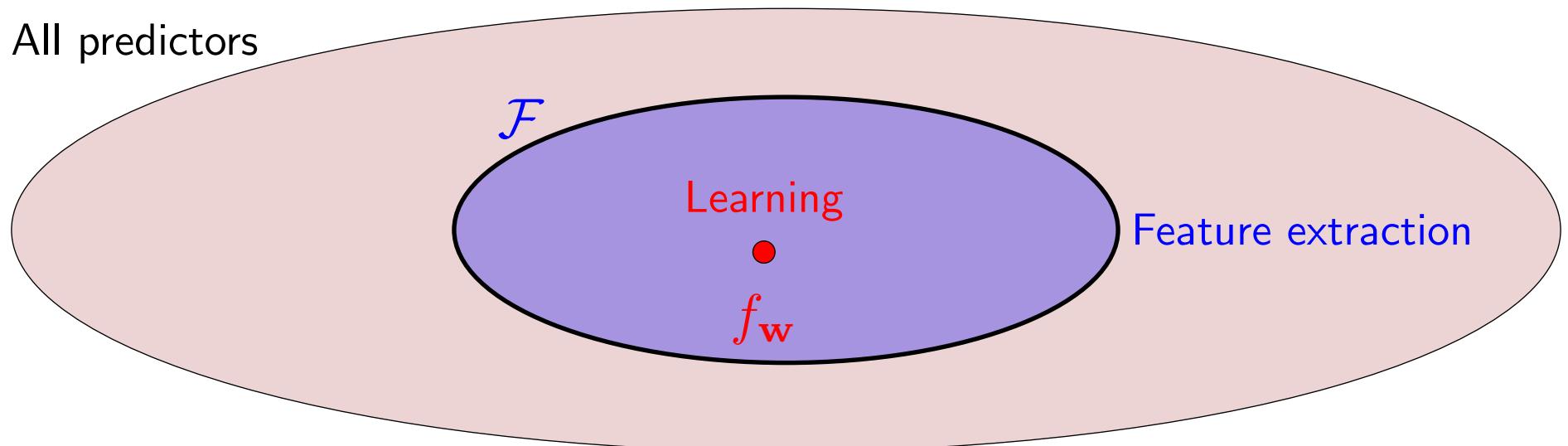
## Definition: hypothesis class

A **hypothesis class** is the set of possible predictors with a fixed  $\phi(x)$  and varying  $\mathbf{w}$ :

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathbb{R}^d\}$$

- Having discussed how feature templates can be used to organize groups of features and allow us to leverage sparsity, let us further study how features impact prediction.
- The key notion is that of a **hypothesis class**, which is the set of all possible predictors that you can get by varying the weight vector  $w$ . Thus, the feature extractor  $\phi$  specifies a hypothesis class  $\mathcal{F}$ . This allows us to take data and learning out of the picture.

# Feature extraction + learning



- Feature extraction: set  $\mathcal{F}$  based on domain knowledge
- Learning: set  $f_w \in \mathcal{F}$  based on data

- Stepping back, we can see the two stages more clearly. First, we perform feature extraction (given domain knowledge) to specify a hypothesis class  $\mathcal{F}$ . Second, we perform learning (given training data) to obtain a particular predictor  $f_w \in \mathcal{F}$ .
- Note that if the hypothesis class doesn't contain any good predictors, then no amount of learning can help. So the question when extracting features is really whether they are powerful enough to **express** predictors which are good. It's okay and expected that  $\mathcal{F}$  will contain a bunch of bad ones as well.
- Later, we'll see reasons for keeping the hypothesis class small (both for computational and statistical reasons), because we can't get the optimal  $w$  for any feature extractor  $\phi$  we choose.

# Example: beyond linear functions

Regression:  $x \in \mathbb{R}, y \in \mathbb{R}$

Linear functions:

$$\phi(x) = x$$

$$\mathcal{F}_1 = \{x \mapsto w_1 x + w_2 x^2 : w_1 \in \mathbb{R}, w_2 = 0\}$$

Quadratic functions:

$$\phi(x) = [x, x^2]$$

$$\mathcal{F}_2 = \{x \mapsto w_1 x + w_2 x^2 : w_1 \in \mathbb{R}, w_2 \in \mathbb{R}\}$$

[whiteboard]

- Given a fixed feature extractor  $\phi$ , let us consider the space of all predictors  $f_w$  obtained by sweeping  $w$  over all possible values.
- If we use  $\phi(x) = x$ , then we get linear functions that go through the origin.
- However, we want to have functions that "bend" (or are not monotonic). For example, if we want to predict someone's health given his or her body temperature, there is a sweet spot temperature (37 C) where the health is optimal; both higher and lower values should cause the health to decline.
- If we use  $\phi(x) = [x, x^2]$ , then we get quadratic functions that go through the origin, which are a strict superset of the linear functions, and therefore are strictly more expressive.

# Example: even more flexible functions

Regression:  $x \in \mathbb{R}, y \in \mathbb{R}$

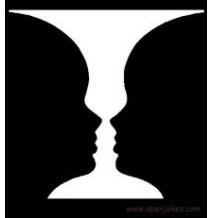
Piecewise constant functions:

$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \dots]$$

$$\mathcal{F}_3 = \{x \mapsto \sum_{j=1}^{10} w_j \mathbf{1}[j-1 < x \leq j] : \mathbf{w} \in \mathbb{R}^{10}\}$$

[whiteboard]

- However, even quadratic functions can be limiting because they have to rise and fall in a certain (parabolic) way. What if we wanted a more flexible, freestyle approach?
- We can create piecewise constant functions by defining features that "fire" (are 1) on particular regions of the input (e.g.,  $1 < x \leq 2$ ). Each feature gets associated with its own weight, which in this case corresponds to the desired function value in that region.
- Thus by varying the weight vector, we get piecewise constant functions with a particular discretization level. We can increase or decrease the discretization level as we need.
- Advanced: what happens if  $x$  were not a scalar, but a  $d$ -dimensional vector? We could perform discretization in  $\mathbb{R}^d$ , but the number of features grows exponentially in  $d$ , which leads to a phenomenon called the curse of dimensionality.



# Linear in what?

Prediction driven by score:

$$\mathbf{w} \cdot \phi(x)$$

Linear in  $\mathbf{w}$ ? Yes

Linear in  $\phi(x)$ ? Yes

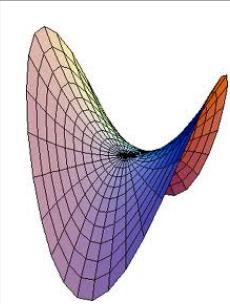
Linear in  $x$ ? No! ( $x$  not necessarily even a vector)



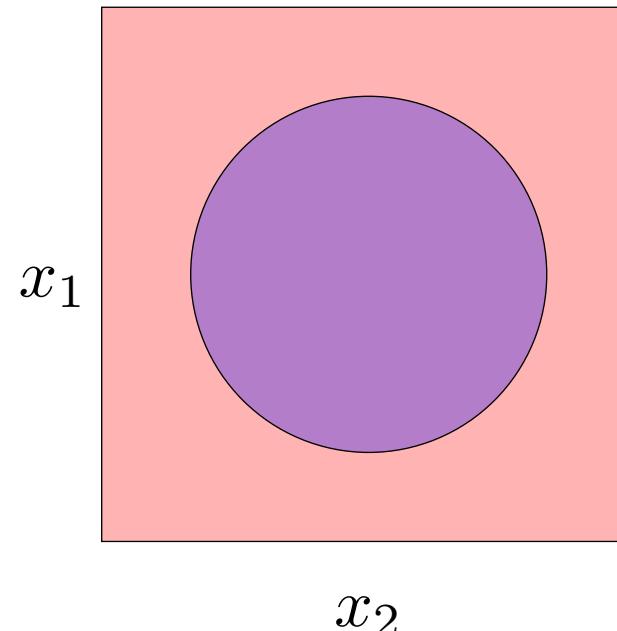
## Key idea: non-linearity

- Predictors  $f_{\mathbf{w}}(x)$  can be expressive **non-linear** functions and decision boundaries of  $x$ .
- Score  $\mathbf{w} \cdot \phi(x)$  is **linear** function of  $\mathbf{w}$ , which permits efficient learning.

- Wait a minute...how were we able to get non-linear predictions using linear predictors?
- It is important to remember that for linear predictors, it is the score  $\mathbf{w} \cdot \phi(x)$  that is linear in  $\mathbf{w}$  and  $\phi(x)$  (read it off directly from the formula). In particular, the score is not linear in  $x$  (it sometimes doesn't even make sense because  $x$  need not be a vector at all — it could be a string or a PDF file. Also, neither the predictor  $f_{\mathbf{w}}$  (unless we're doing linear regression) nor the loss function  $\text{TrainLoss}(\mathbf{w})$  are linear in anything.
- The significance is as follows: From the feature extraction viewpoint, we can define arbitrary features that yield very **non-linear** functions in  $x$ . From the learning viewpoint (only looking at  $\phi(x)$ , not  $x$ ), **linearity** plays an important role in being able to optimize the weights efficiently (as it leads to convex optimization problems).



# Geometric viewpoint



$$\phi(x) = [1, x_1, x_2, x_1^2 + x_2^2]$$

How to relate **non-linear** decision boundary in  $x$  space with **linear** decision boundary in  $\phi(x)$  space?

[demo]

- Let's try to understand the relationship between the non-linearity in  $x$  and linearity in  $\phi(x)$ . We consider binary classification where our input is  $x = [x_1, x_2] \in \mathbb{R}^2$  a point on the plane. With the quadratic features  $\phi(x)$ , we can carve out the decision boundary corresponding to an ellipse (think about the formula for an ellipse and break it down into monomials).
- We can now look at the feature vectors  $\phi(x)$ , which include an extra dimension. In this 3D space, a linear predictor (defined by the hyperplane) actually corresponds to the non-linear predictor in the original 2D space.

# An example task



## Example: detecting responses

Input  $x$ :

two consecutive messages in a chat

Output  $y \in \{+1, -1\}$ :

whether the second message is a response to the first

Recall: feature extractor  $\phi$  should pick out properties of  $x$  that might be useful for prediction of  $y$

- Let's apply what you've learned about feature extraction to a concrete problem. The motivation here is that messaging platforms often just show a single stream of messages, when there is generally a grouping of messages into coherent conversations. How can we build a classifier that can group messages automatically? We can formulate this as a binary classification problem where we look at two messages and determine whether these two are part of the same conversation or not.



# Question

What feature templates would you use for predicting whether the second message is a response to the first?

time elapsed

time elapsed is between \_\_\_ and \_\_\_

first message contains \_\_\_

second message contains \_\_\_

two messages both contain \_\_\_

two messages have \_\_\_ common words



# Summary so far

- Feature templates: organize related (sparse) features
- Hypothesis class: defined by features (what is possible)
- Linear classifiers: can produce non-linear decision boundaries



# Roadmap

Features

**Neural networks**

Gradients without tears

Nearest neighbors

- What we've shown so far is that by being mildly clever with choosing the feature extractor  $\phi$ , we can actually get quite a bit of mileage out of our so-called linear predictors.
- However, sometimes we don't know what features are good to use, either because the prediction task is non-intuitive or we don't have time to figure out which features are suitable. Sometimes, we think we might know what features are good, but then it turns out that they aren't (this happens a lot!).
- In the spirit of machine learning, we'd like to automate things as much as possible. In this context, it means creating algorithms that can take whatever crude features we have and turn them into refined predictions, thereby shifting the burden off feature extraction and moving it to learning.
- Neural networks have been around for many decades, but they fell out of favor because they were difficult to train. In the last decade, there has been a huge resurgence of interest in neural networks since they perform so well and training seems to not be such an issue when you have tons of data and compute.
- In a sense, neural networks allow one to automatically learn the features of a linear classifier which are geared towards the desired task, rather than specifying them all by hand.

# Motivation



## Example: predicting car collision

**Input:** position of two oncoming cars  $x = [x_1, x_2]$

**Output:** whether safe ( $y = +1$ ) or collide ( $y = -1$ )

True function: safe if cars sufficiently far

$$y = \text{sign}(|x_1 - x_2| - 1)$$

Examples:

$x$	$y$
[1, 3]	+1
[3, 1]	+1
[1, 0.5]	-1

- As a motivating example, consider the problem of predicting whether two cars at positions  $x_1$  and  $x_2$  are going to collide. Suppose the true output is 1 (safe) whenever the cars are separated by a distance of at least 1. Clearly, this the decision is not linear.

# Decomposing the problem

Test if car 1 is far right of car 2:

$$h_1 = \mathbf{1}[x_1 - x_2 \geq 1]$$

Test if car 2 is far right of car 1:

$$h_2 = \mathbf{1}[x_2 - x_1 \geq 1]$$

Safe if at least one is true:

$$y = \text{sign}(h_1 + h_2)$$

$x$	$h_1$	$h_2$	$y$
[1, 3]	0	1	+1
[3, 1]	1	0	+1
[1, 0.5]	0	0	-1

- The intuition is to break up the problem into two subproblems, which test if car 1 (car 2) is to the far right.
- Given these two binary values  $h_1, h_2$ , we can declare safety if at least one of them is true.

# Learning strategy

Define:  $\phi(x) = [1, x_1, x_2]$

Intermediate hidden subproblems:

$$h_1 = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0] \quad \mathbf{v}_1 = [-1, +1, -1]$$

$$h_2 = \mathbf{1}[\mathbf{v}_2 \cdot \phi(x) \geq 0] \quad \mathbf{v}_2 = [-1, -1, +1]$$

Final prediction:

$$f_{\mathbf{V}, \mathbf{w}}(x) = \text{sign}(\mathbf{w}_1 h_1 + \mathbf{w}_2 h_2) \quad \mathbf{w} = [1, 1]$$



**Key idea: joint learning**

Goal: learn both hidden subproblems  $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2)$  and combination weights  $\mathbf{w} = [w_1, w_2]$

- Having written  $y$  in a specific way, let us try to generalize to a family of predictors (this seems to be a recurring theme).
- We can define  $\mathbf{v}_1 = [-1, 1, -1]$  and  $\mathbf{v}_2 = [-1, -1, 1]$  and  $w_1 = w_2 = 1$  to accomplish this.
- At a high-level, we have defined two intermediate subproblems, that of predicting  $h_1$  and  $h_2$ . These two values are hidden in the sense that they are not specified to be anything. They just need to be set in a way such that  $y$  is linearly predictable from them.

# Gradients

Problem: gradient of  $h_1$  with respect to  $\mathbf{v}_1$  is 0

$$h_1 = \mathbf{1}[\mathbf{v}_1 \cdot \phi(x) \geq 0]$$

[whiteboard]



## Definition: logistic function

The logistic function maps  $(-\infty, \infty)$  to  $[0, 1]$ :

$$\sigma(z) = (1 + e^{-z})^{-1}$$

Derivative:

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

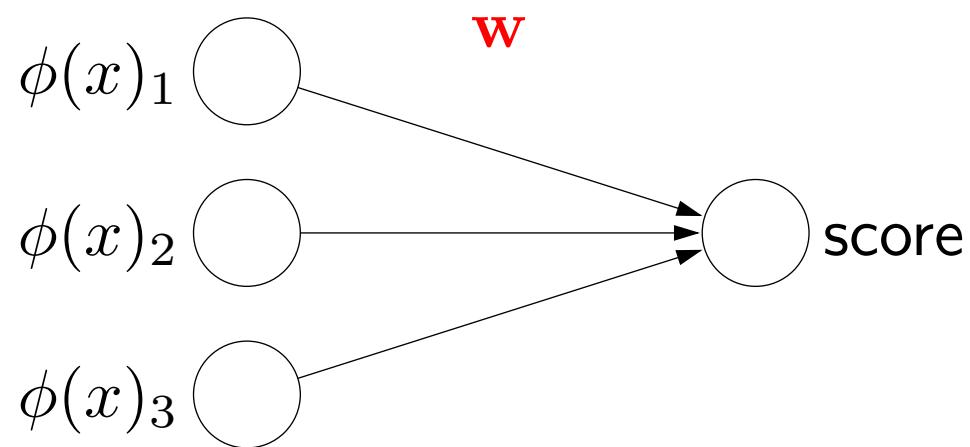
Solution:

$$h_1 = \sigma(\mathbf{v}_1 \cdot \phi(x))$$

- If we try to train the weights  $\mathbf{v}_1, \mathbf{v}_2, w_1, w_2$ , we will immediately notice a problem: the gradient of  $h_1$  with respect to  $\mathbf{v}_1$  is always zero because of the hard thresholding function.
- Therefore, we define a function **logistic function**  $\sigma(z)$ , which looks roughly like the step function  $\mathbf{1}[z \geq 0]$ , but has non-zero gradients everywhere.
- One thing to bear in mind is that even though the gradients are non-zero, they can be quite small when  $|z|$  is large. This is what makes optimizing neural networks hard.

# Linear functions

Linear functions:



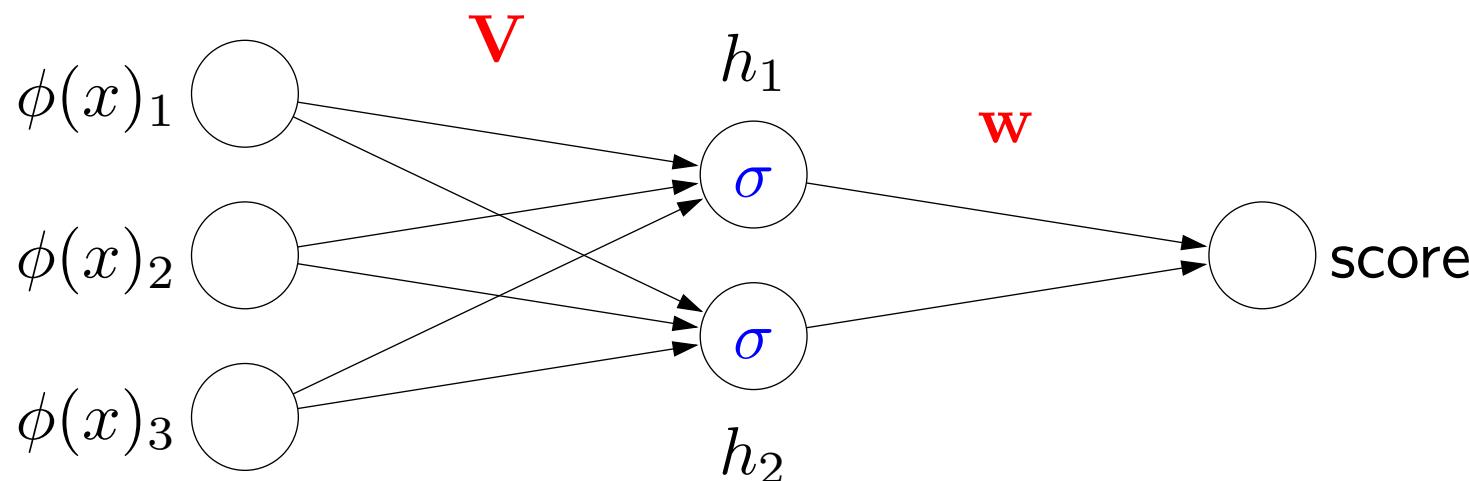
Output:

$$\text{score} = \mathbf{w} \cdot \phi(\mathbf{x})$$

- Let's try to visualize the functions.
- Recall that a linear function takes the input  $\phi(x) \in \mathbb{R}^d$  and directly take the dot product with the weight vector  $w$  to form the score, the basis for prediction in both binary classification and regression.

# Neural networks

Neural network (one hidden layer):



Intermediate hidden units:

$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \quad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot \mathbf{h}$$

- A (one-layer) neural network first maps an input  $\phi(x) \in \mathbb{R}^d$  onto a hidden **intermediate representation**  $\mathbf{h} \in \mathbb{R}^k$ , which in turn is mapped to the score via a linear function.
- Specifically, let  $k$  be the number of hidden units. For each hidden unit  $j = 1, \dots, k$ , we have a weight vector  $\mathbf{v}_j \in \mathbb{R}^d$ , which is used to determine the value of the hidden node  $h_j \in \mathbb{R}$  (also called the **activation**) according to  $h_j = \sigma(\mathbf{v}_j \cdot \phi(x))$ , where  $\sigma$  is the activation function. The activation function can be a number of different things, but its main property is that it is a non-linear function.
- Let  $\mathbf{h} = [h_1, \dots, h_k]$  be the vector of activations. This activation vector is now combined with another weight vector  $\mathbf{w} \in \mathbb{R}^k$  to produce the final score.
- The logistic function is an instance of an **activation function**, and is the classic one that was used in the past. These days, most people use a **rectifier** function, commonly known as a rectified linear unit (ReLU), which is defined as  $\text{ReLU}(z) = \max(z, 0)$ . The ReLU has two advantages: (i) its gradient doesn't vanish as  $z$  grows, which makes it empirically easier to train; and (ii) it only involves a max operation, which is computationally easier to compute than the exponential function.

# Neural networks

Interpretation: intermediate hidden units as learned features of a linear predictor



## Key idea: feature learning

Before: apply linear predictor on manually specify features

$$\phi(x)$$

Now: apply linear predictor on automatically learned features

$$h(x) = [h_1(x), \dots, h_k(x)]$$

- The noteworthy aspect here is that the activation vector  $\mathbf{h}$  behaves a lot like our feature vector  $\phi(x)$  that we were using for linear prediction. The difference is that mapping from input  $\phi(x)$  to  $\mathbf{h}$  is learned automatically, not manually constructed (as was the case before). Therefore, a neural network can be viewed as learning the features of a linear classifier. Of course, the type of features that can be learned must be of the form  $x \mapsto \sigma(\mathbf{v}_j \cdot \phi(x))$ . Even for deep neural networks, no matter how deep the neural network is, the top layer is always a linear function, and the layers below can be interpreted as defining a (possibly very complex) feature map.
- Whether this is a suitable form depends on the nature of the application. Empirically, though, neural networks have been quite successful, since learning the features from the data with the explicit objective of minimizing the loss can yield better features than ones which are manually crafted. Since 2010, there have been some advances in getting neural networks to work, and they have become the state-of-the-art in many tasks. For example, all the major companies (Google, Microsoft, IBM) all recently switched over to using neural networks for speech recognition. In computer vision, (convolutional) neural networks are completely dominant in object recognition.



# Roadmap

Features

Neural networks

**Gradients without tears**

Nearest neighbors

# Motivation: loss minimization

Optimization problem:

$$\min_{\mathbf{V}, \mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

$$\text{TrainLoss}(\mathbf{V}, \mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x, y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{V}, \mathbf{w})$$

$$\text{Loss}(x, y, \mathbf{V}, \mathbf{w}) = (y - f_{\mathbf{V}, \mathbf{w}}(x))^2$$

$$f_{\mathbf{V}, \mathbf{w}}(x) = \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x))$$

Goal: compute gradient

$$\nabla_{\mathbf{V}, \mathbf{w}} \text{TrainLoss}(\mathbf{V}, \mathbf{w})$$

- The main thing left to do for neural networks is to be able to train them. Conceptually, this should be straightforward: just take the gradient and run SGD.
- While this is true, computing the gradient, even though it is not hard, can be quite tedious to do by hand.

# Approach

Mathematically: just grind through the chain rule

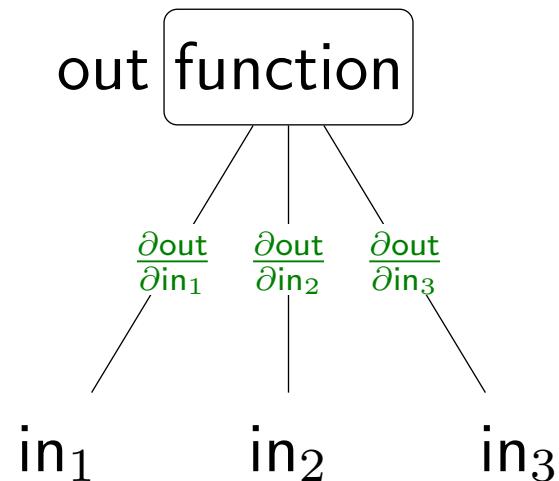
Next: visualize the computation using a **computation graph**

Advantages:

- Avoid long equations
- Reveal structure of computations (modularity, efficiency, dependencies) — TensorFlow/PyTorch are built on this

- We will illustrate a graphical way of organizing the computation of gradients, which is built out of a few components.
- This graphical approach will show the structure of the function and will not only make gradients easy to compute, but also shed more light onto the predictor and loss function.
- In fact, these days if you use a package such as TensorFlow or PyTorch, you can write down the expressions symbolically and the gradient is computed for you. This is done essentially using the computational procedure that we will see.

# Functions as boxes



Partial derivatives (gradients): how much does the output change if an input changes?

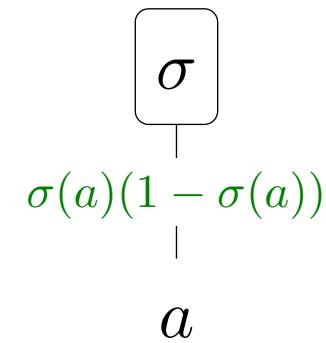
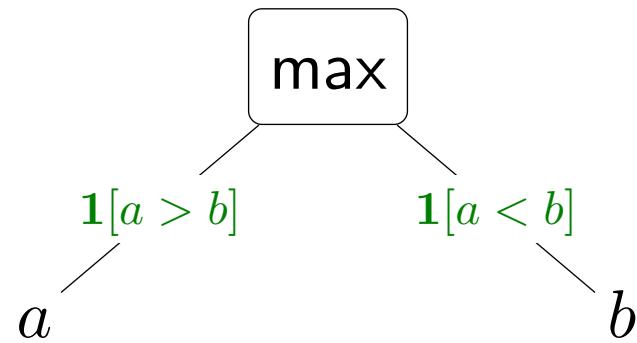
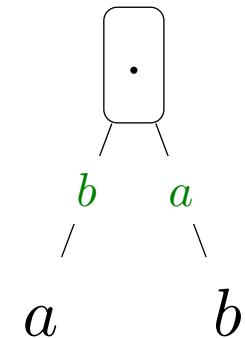
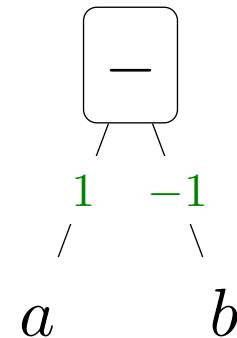
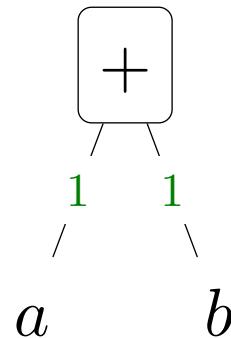
Example:

$$2\text{in}_1 + (\text{in}_2 + \epsilon)\text{in}_3 = \text{out} + \text{in}_3\epsilon$$

- The first conceptual step is to think of functions as boxes that take a set of inputs and produces an output. Then the partial derivatives (gradients if the input is vector-valued) are just a measure of sensitivity: if we perturb  $\text{in}_1$  by a small amount  $\epsilon$ , how much does the output  $\text{out}$  change? The answer is  $\frac{\partial \text{out}}{\partial \text{in}_1} \cdot \epsilon$ . For convenience, we write the partial derivative on the edge connecting the input to the output.



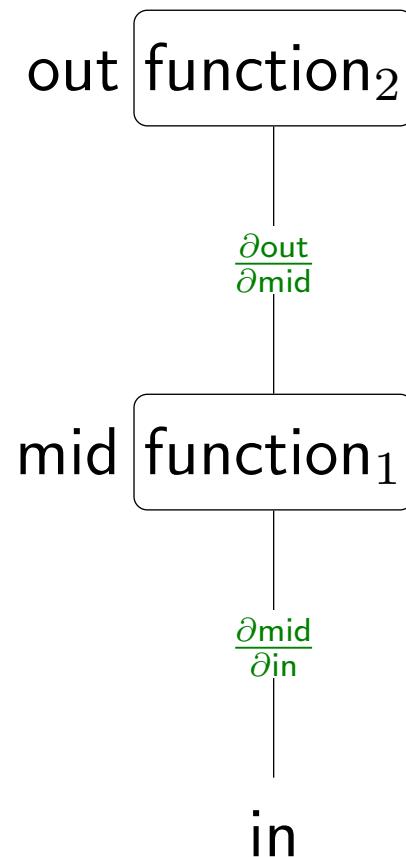
# Basic building blocks



- Here are 5 examples of simple functions and their partial derivatives. These should be familiar from basic calculus. All we've done is present them in a visually more intuitive way.
- But it turns out that these simple functions are all we need to build up many of the more complex and potentially scarier looking functions that we'll encounter in machine learning.



# Composing functions



Chain rule:

$$\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$$

- The second conceptual point is to think about **composing**. Graphically, this is very natural: the output of one function  $f$  simply gets fed as the input into another function  $g$ .
- Now how does  $in$  affect  $out$  (what is the partial derivative)? The key idea is that the partial derivative **decomposes** into a product of the two partial derivatives on the two edges. You should recognize this is no more than the chain rule in graphical form.
- More generally, the partial derivative of  $y$  with respect to  $x$  is simply the product of all the green expressions on the edges of the path connecting  $x$  and  $y$ . This visual intuition will help us better understand more complex functions, which we will turn to next.

# Binary classification with hinge loss

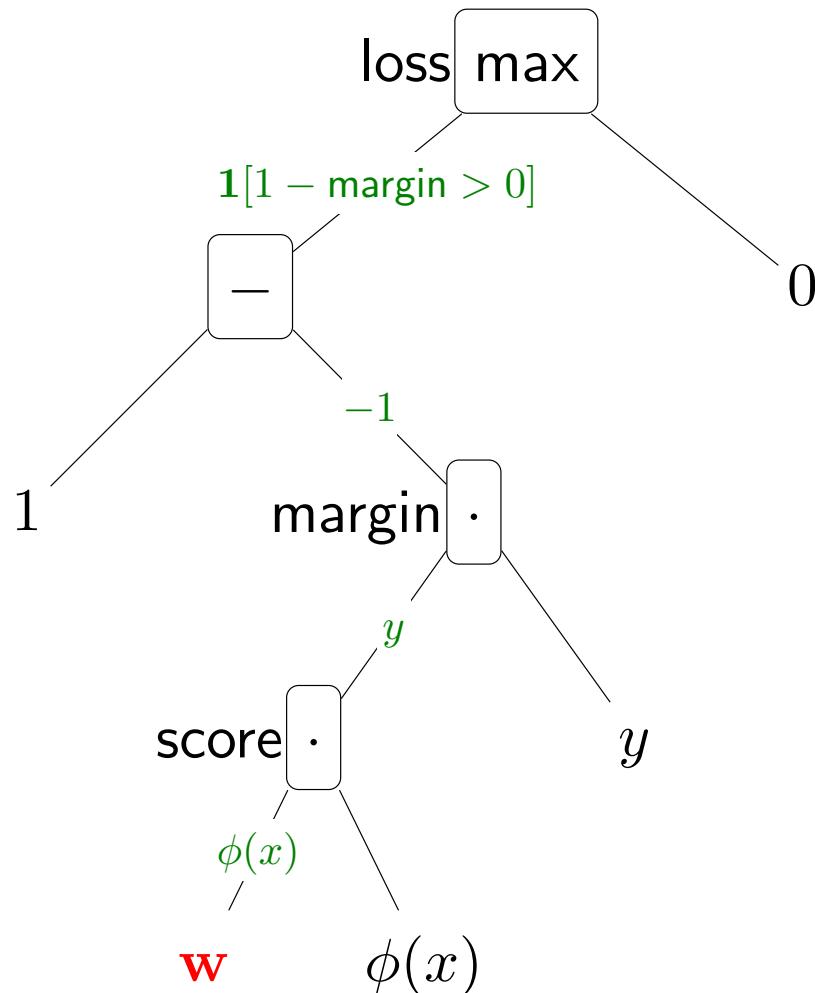
Hinge loss:

$$\text{Loss}(x, y, \mathbf{w}) = \max\{1 - \mathbf{w} \cdot \phi(x)y, 0\}$$

Compute:

$$\frac{\partial \text{Loss}(x, y, \mathbf{w})}{\partial \mathbf{w}}$$

# Binary classification with hinge loss



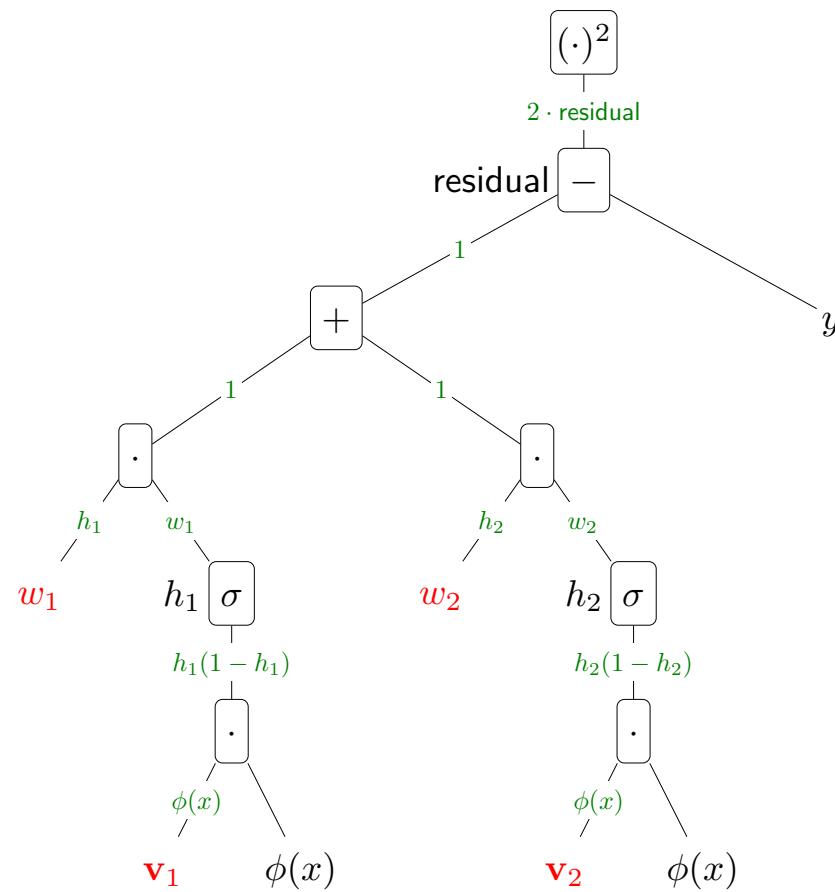
Gradient: multiply the edges

$$-1[\text{margin} < 1]\phi(x)y$$

- Let us start with a simple example: the hinge loss for binary classification.
- In red, we have highlighted the weights  $w$  with respect to which we want to take the derivative. The central question is how small perturbations in  $w$  affect a change in the output (loss). Intermediate nodes have been labeled with interpretable names (score, margin).
- The actual gradient is the product of the edge-wise gradients from  $w$  to the loss output.

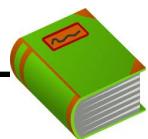
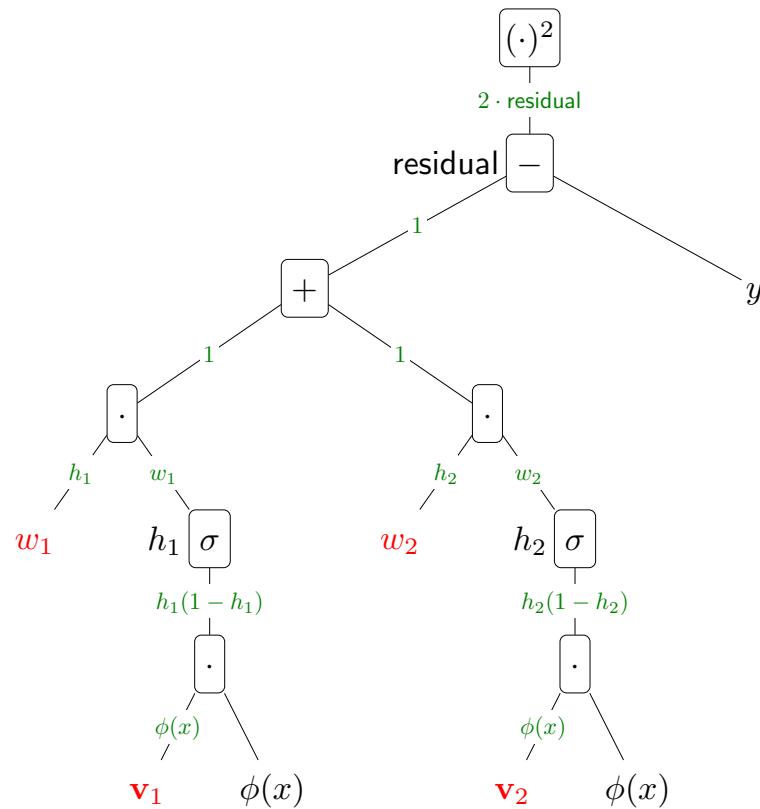
# Neural network

$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$



- Now, we can apply the same strategy to neural networks. Here we are using the squared loss for concreteness, but one can also use the logistic or hinge losses.
- Note that there is some really nice modularity here: you can pick any predictor (linear or neural network) to drive the score, and the score can be fed into any loss function (squared, hinge, etc.).

# Backpropagation



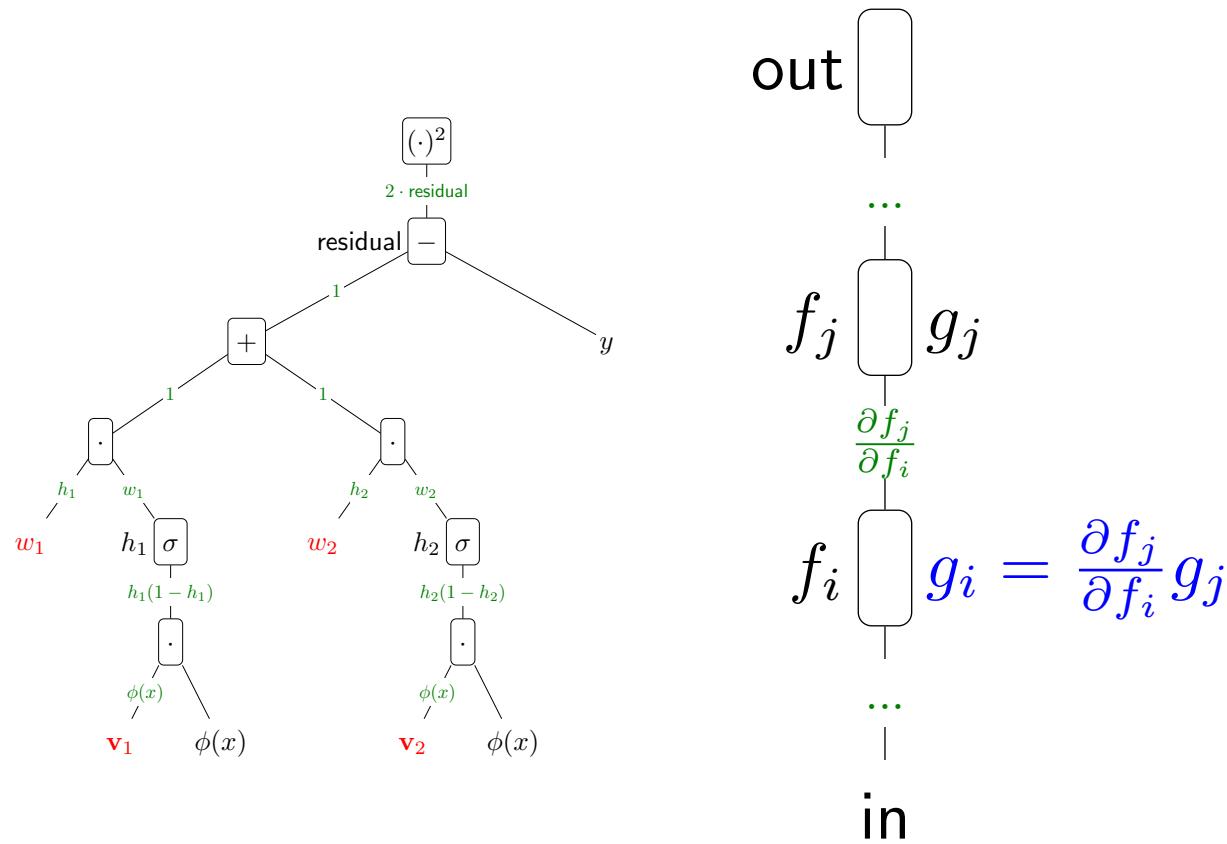
## Definition: Forward/backward values

Forward:  $f_i$  is value for subexpression rooted at  $i$

Backward:  $g_i = \frac{\partial \text{out}}{\partial f_i}$  is how  $f_i$  influences output

- So far, we have mainly used the graphical representation to visualize the computation of function values and gradients for our conceptual understanding. But it turns out that the graph has algorithmic implications too.
- Recall that to train any sort of model using (stochastic) gradient descent, we need to compute the gradient of the loss (top output node) with respect to the weights (leaf nodes highlighted in red).
- We also saw that these gradients (partial derivatives) are just the product of the local derivatives (green stuff) along the path from a leaf to a root. So we can just go ahead and compute these gradients: for each red node, multiply the quantities on the edges. However, notice that many of the paths share subpaths in common, so sometimes there's an opportunity to save computation (think dynamic programming).
- To make this sharing more explicit, for each node  $i$  in the tree, define the forward value  $f_i$  to be the value of the subexpression rooted at that tree, which depends on the inputs underneath that subtree. For example, the parent node of  $w_1$  corresponds to the expression  $w_1\sigma(\mathbf{v}_1 \cdot \phi(x))$ . The  $f_i$ 's are the intermediate computations required to even evaluate the function at the root.
- Next, for each node  $i$  in the tree, define the backward value  $g_i$  to be the gradient of the output with respect to  $f_i$ , the forward value of node  $i$ . This measures the change that would happen in the output (root node) induced by changes to  $f_i$ .
- Note that both  $f_i$  and  $g_i$  can either be scalars, vectors, or matrices, but have the same dimensionality.

# Backpropagation



## Algorithm: backpropagation

Forward pass: compute each  $f_i$  (from leaves to root)

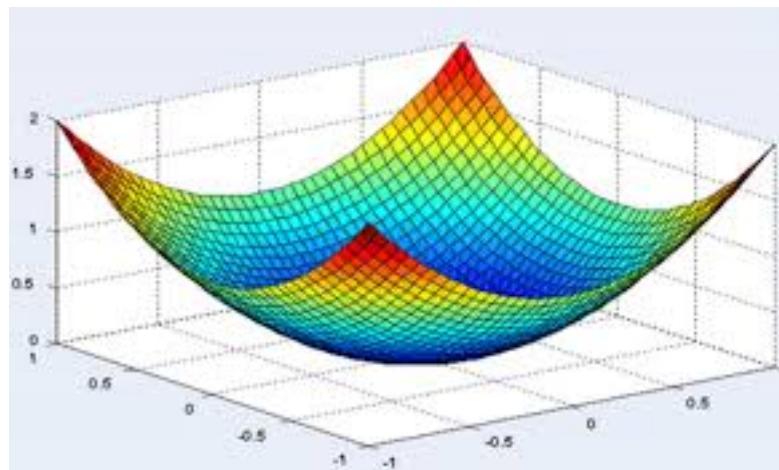
Backward pass: compute each  $g_i$  (from root to leaves)

- We now define the backpropagation algorithm on arbitrary computation graphs.
- First, in the forward pass, we go through all the nodes in the computation graph from leaves to the root, and compute  $f_i$ , the value of each node  $i$ , recursively given the node values of the children of  $i$ . These values will be used in the backward pass.
- Next, in the backward pass, we go through all the nodes from the root to the leaves and compute  $g_i$  recursively from  $f_i$  and  $g_j$ , the backward value for the parent of  $i$  using the key recurrence  $g_i = \frac{\partial f_j}{\partial f_i} g_j$  (just the chain rule).
- In this example, the backward pass gives us the gradient of the output node (the gradient of the loss) with respect to the weights (the red nodes).

# Note on optimization

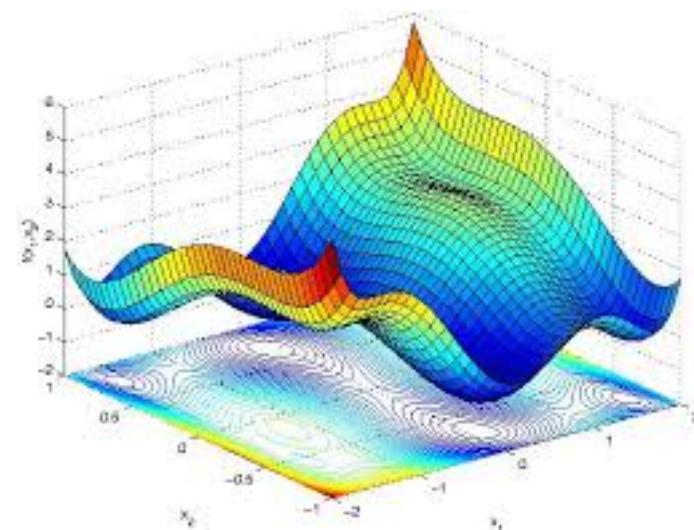
TrainLoss( $\mathbf{w}$ )

Linear functions



(convex loss)

Neural networks



(non-convex)

Optimization of neural networks is generally **hard**

- While we can go through the motions of running the backpropagation algorithm to compute gradients, what is the result of running SGD?
- For linear predictors (using the squared loss or hinge loss),  $\text{TrainLoss}(\mathbf{w})$  is a convex function, which means that SGD (with an appropriately set step size) is theoretically guaranteed to converge to the global optimum.
- However, for neural networks,  $\text{TrainLoss}(\mathbf{w})$  is typically non-convex which means that there are multiple local optima, and SGD is not guaranteed to converge to the global optimum. There are many settings that SGD fails both theoretically and empirically, but in practice, SGD on neural networks can work with proper attention to tuning hyperparameters. The gap between theory and practice is not well understood and an active area of research.



# Roadmap

Features

Neural networks

Gradients without tears

**Nearest neighbors**

- Linear predictors were governed by a simple dot product  $\mathbf{w} \cdot \phi(x)$ . Neural networks chained together these simple primitives to yield something more complex. Now, we will consider **nearest neighbors**, which yields complexity by another mechanism: computing similarities between examples.

# Nearest neighbors



## Algorithm: nearest neighbors

Training: just store  $\mathcal{D}_{\text{train}}$

Predictor  $f(x')$ :

- Find  $(x, y) \in \mathcal{D}_{\text{train}}$  where  $\|\phi(x) - \phi(x')\|$  is smallest
- Return  $y$



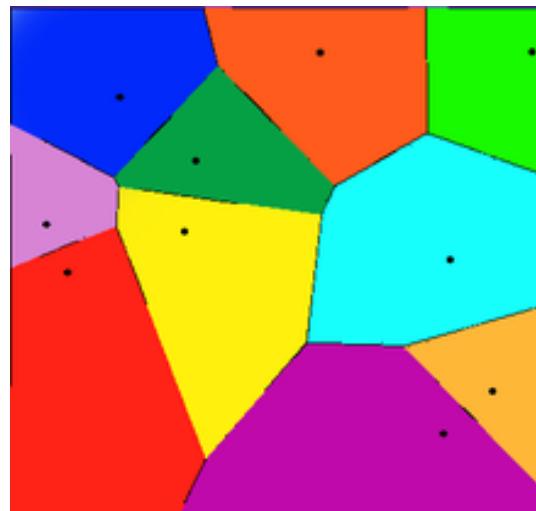
## Key idea: similarity

Similar examples tend to have similar outputs.

- **Nearest neighbors** is perhaps conceptually one of the simplest learning algorithms. In a way, there is no learning. At training time, we just store the entire training examples. At prediction time, we get an input  $x'$  and we just find the input in our training set that is **most similar**, and return its output.
- In a practical implementation, finding the closest input is non-trivial. Popular choices are using k-d trees or locality-sensitive hashing. We will not worry about this issue.
- The intuition being expressed here is that similar (nearby) points tend to have similar outputs. This is a reasonable assumption in most cases; all else equal, having a body temperature of 37 and 37.1 is probably not going to affect the health prediction by much.

# Expressivity of nearest neighbors

Decision boundary: based on Voronoi diagram



- Much more expressive than quadratic features
- **Non-parametric:** the hypothesis class adapts to number of examples
- Simple and powerful, but kind of brute force

- Let's look at the decision boundary of nearest neighbors. The input space is partitioned into regions, such that each region has the same closest point (this is a Voronoi diagram), and each region could get a different output.
- Notice that this decision boundary is much more expressive than what you could get with quadratic features. In particular, one interesting property is that the complexity of the decision boundary adapts to the number of training examples. As we increase the number of training examples, the number of regions will also increase. Such methods are called **non-parametric**.



# Summary of learners

- Linear predictors: combine raw features

prediction is **fast**, **easy** to learn, **weak** use of features

- Neural networks: combine learned features

prediction is **fast**, **hard** to learn, **powerful** use of features

- Nearest neighbors: predict according to similar examples

prediction is **slow**, **easy** to learn, **powerful** use of features

- Let us conclude now. First, we discussed some general principles for designing good features for linear predictors. Just with the machinery of linear prediction, we were able to obtain rich predictors.
- Second, we focused on expanding the expressivity of our predictors fixing a particular feature extractor  $\phi$ .
- We covered three algorithms: **linear predictors** combine the features linearly (which is rather weak), but is easy and fast.
- **Neural networks** effectively learn non-linear features, which are then used in a linear way. This is what gives them their power and prediction speed, but they are harder to learn (due to the non-convexity of the objective function).
- **Nearest neighbors** is based on computing similarities with training examples. They are powerful and easy to learn, but are slow to use for prediction because they involve enumerating (or looking up points in) the training data.



# Lecture 4: Machine learning III



# Announcements

- Homework 1 (foundations): Thursday 11pm is 2 late day **hard deadline**
- Section Thursday 3:30pm: backpropagation example, nearest neighbors, scikit-learn

# Review: feature extractor



- Last lecture, we spoke at length about the importance of features, how to organize them using feature templates, and how we can get interesting non-linearities by choosing the feature extractor  $\phi$  judiciously. This is you using all your domain knowledge about the problem.

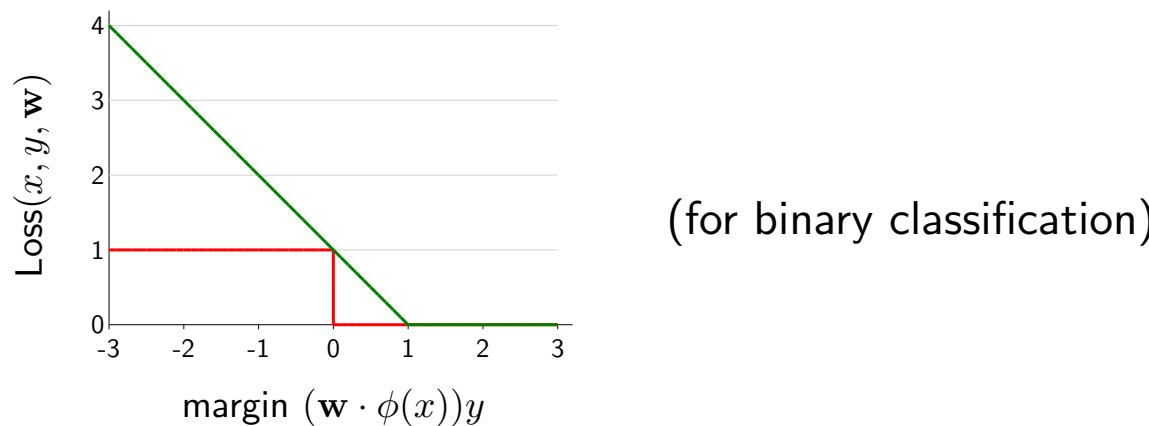
# Review: prediction score

- Linear predictor:  $\text{score} = \mathbf{w} \cdot \phi(x)$
- Neural network:  $\text{score} = \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x))$

- Given the feature extractor  $\phi$ , we can use that to define a prediction score, either using a linear predictor or a neural network. If you use neural networks, you typically have to work less hard at designing features, but you end up with a harder learning problem. There is a human-machine tradeoff here.

# Review: loss function

$\text{Loss}(x, y, \mathbf{w})$ :



$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

Stochastic gradient descent:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{Loss}(x, y, \mathbf{w})$$

- The prediction score is the basis of many types of predictions, including regression and binary classification. The loss function connects the prediction score with the correct output  $y$ , and measures how unhappy we are with a particular weight vector  $w$ .
- This leads to an optimization problem, that of finding the  $w$  that yields the lowest training loss. We saw that a simple algorithm, stochastic gradient descent, works quite well.



# Question

What's the true objective of machine learning?

minimize error on the training set

minimize training error with regularization

minimize error on the test set

minimize error on unseen future examples

learn about machines

- We have written the average training loss as the objective function, but it turns out that that's not really the true goal. That's only what we tell our optimization friends so that there's something concrete and actionable. The true goal is to minimize error on unseen future examples; in other words, we need to **generalize**. As we'll see, this is perhaps the most important aspect of machine learning and statistics — albeit a more elusive one.



# Roadmap

**Generalization**

Unsupervised learning

Summary

# Training error

$$\text{TrainLoss}(\mathbf{w}) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

Is this a good objective?

- Now let's be a little more critical about what we've set out to optimize. So far, we've declared that we want to minimize the training loss.



# A strawman algorithm



## Algorithm: rote learning

Training: just store  $\mathcal{D}_{\text{train}}$ .

Predictor  $f(x)$ :

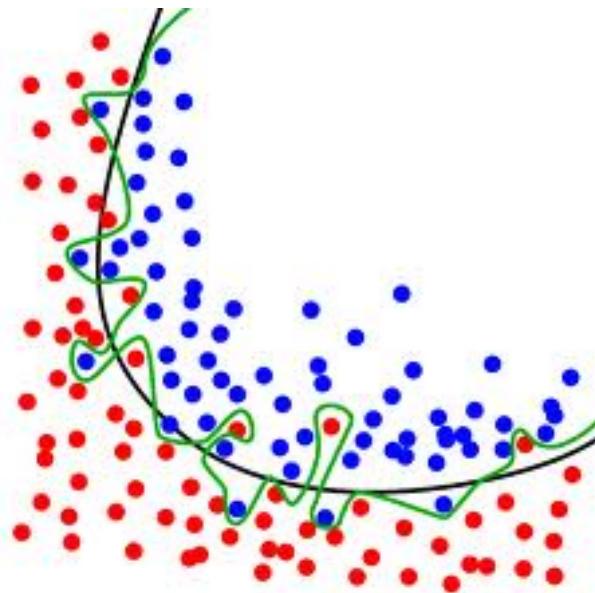
If  $(x, y) \in \mathcal{D}_{\text{train}}$ : return  $y$ .

Else: **segfault**.

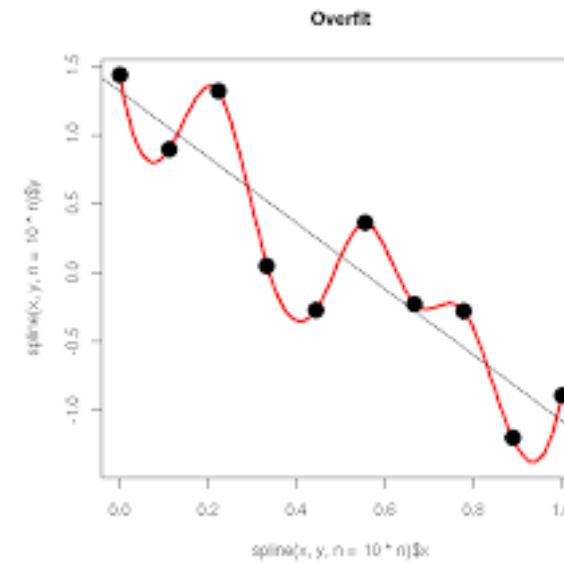
Minimizes the objective perfectly (zero), but clearly bad...

- Clearly, machine learning can't be about just minimizing the training loss. The rote learning algorithm does a perfect job of that, and yet is clearly a bad idea. It **overfits** to the training data and doesn't **generalize** to unseen examples.

# Overfitting pictures



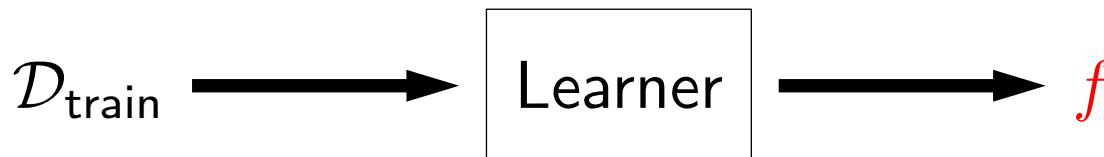
Classification



Regression

- Here are two pictures that illustrate what can go wrong if you only try to minimize the training loss for binary classification and regression.
- On the left, we see that the green decision boundary gets zero training loss by separating all the blue points from the red ones. However, the smoother and simpler black curve is intuitively more likely to be the better classifier.
- On the right, we see that the predictor that goes through all the points will get zero training loss, but intuitively, the black line is perhaps a better option.
- In both cases, what is happening is that by over-optimizing on the training set, we risk fitting **noise** in the data.

# Evaluation



How good is the predictor  $f$ ?



**Key idea: the real learning objective**

Our goal is to minimize **error on unseen future examples**.

Don't have unseen examples; next best thing:



**Definition: test set**

**Test set**  $\mathcal{D}_{\text{test}}$  contains examples not used for training.

- So what is the true objective then? Taking a step back, what we're doing is building a system which happens to use machine learning, and then we're going to deploy it. What we really care about is how accurate that system is on those **unseen future** inputs.
- Of course, we can't access unseen future examples, so the next best thing is to create a **test set**. As much as possible, we should treat the test set as a pristine thing that's unseen and from the future. We definitely should not tune our predictor based on the test error, because we wouldn't be able to do that on future examples.
- Of course at some point we have to run our algorithm on the test set, but just be aware that each time this is done, the test set becomes less good of an indicator of how well you're actually doing.

# Generalization

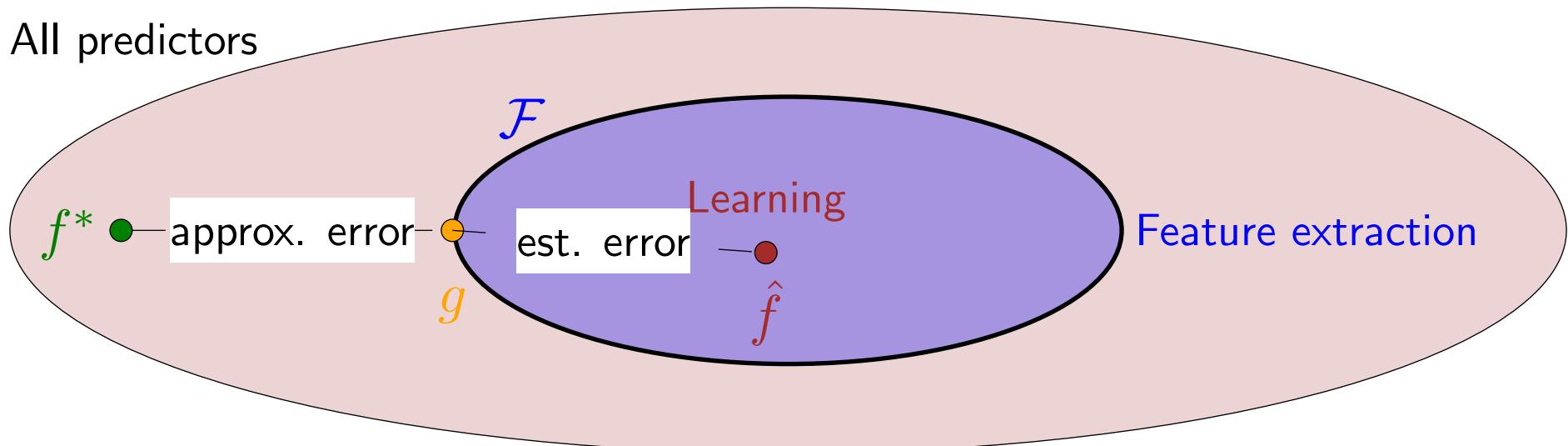
When will a learning algorithm **generalize** well?



- So far, we have an intuitive feel for what overfitting is. How do we make this precise? In particular, when does a learning algorithm generalize from the training set to the test set?

# Approximation and estimation error

All predictors

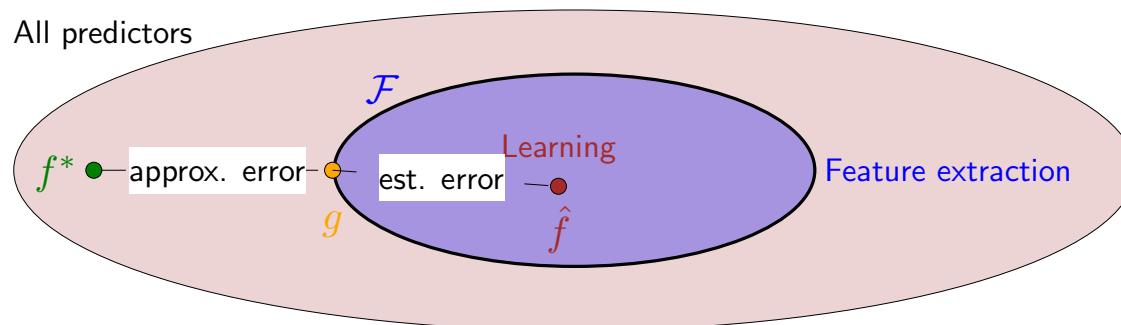


- Approximation error: how good is the hypothesis class?
- Estimation error: how good is the learned predictor **relative to** the potential of the hypothesis class?

$$\underbrace{\text{Err}(\hat{f}) - \text{Err}(g)}_{\text{estimation}} + \underbrace{\text{Err}(g) - \text{Err}(f^*)}_{\text{approximation}}$$

- Here's a cartoon that can help you understand the balance between fitting and generalization. Out there somewhere, there is a magical predictor  $f^*$  that classifies everything perfectly. This predictor is unattainable; all we can hope to do is to use a combination of our domain knowledge and data to approximate that. The question is: how far are we away from  $f^*$ ?
- Recall that our learning framework consists of (i) choosing a hypothesis class  $\mathcal{F}$  (by defining the feature extractor) and then (ii) choosing a particular predictor  $\hat{f}$  from  $\mathcal{F}$ .
- **Approximation error** is how far the entire hypothesis class is from the target predictor  $f^*$ . Larger hypothesis classes have lower approximation error. Let  $g \in \mathcal{F}$  be the best predictor in the hypothesis class in the sense of minimizing test error  $g = \arg \min_{f \in \mathcal{F}} \text{Err}(f)$ . Here, distance is just the differences in test error:  $\text{Err}(g) - \text{Err}(f^*)$ .
- **Estimation error** is how good the predictor  $\hat{f}$  returned by the learning algorithm is with respect to the best in the hypothesis class:  $\text{Err}(\hat{f}) - \text{Err}(g)$ . Larger hypothesis classes have higher estimation error because it's harder to find a good predictor based on limited data.
- We'd like both approximation and estimation errors to be small, but there's a tradeoff here.

# Effect of hypothesis class size



As the hypothesis class size increases...

Approximation error decreases because:

taking min over larger set

Estimation error increases because:

harder to estimate something more complex

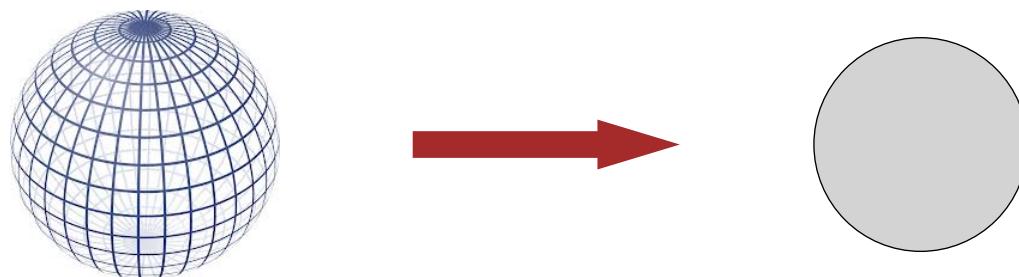
**How do we control the hypothesis class size?**

- The approximation error decreases monotonically as the hypothesis class size increases for a simple reason: you're taking a minimum over a larger set.
- The estimation error increases monotonically as the hypothesis class size increases for a deeper reason involving statistical learning theory (explained in CS229T).
- For each weight vector  $\mathbf{w}$ , we have a predictor  $f_{\mathbf{w}}$  (for classification,  $f_{\mathbf{w}}(x) = \mathbf{w} \cdot \phi(x)$ ). So the hypothesis class  $\mathcal{F} = \{f_{\mathbf{w}}\}$  is all the predictors as  $\mathbf{w}$  ranges. By controlling the number of possible values of  $\mathbf{w}$  that the learning algorithm is allowed to choose from, we control the size of the hypothesis class and thus guard against overfitting.

# Strategy 1: dimensionality

$$\mathbf{w} \in \mathbb{R}^d$$

Reduce the dimensionality  $d$ :



- One straightforward strategy is to change the dimensionality, which is the number of features. For example, linear functions are lower-dimensional than quadratic functions.

# Controlling the dimensionality

Manual feature (template) selection:

- Add features if they help
- Remove features if they don't help

Automatic feature selection (beyond the scope of this class):

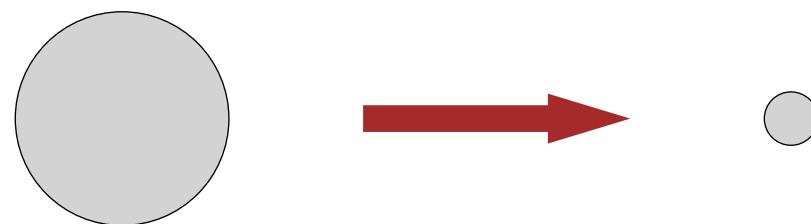
- Forward selection
- Boosting
- $L_1$  regularization

- Mathematically, you can think about removing a feature  $\phi(x)_{37}$  as simply only allowing its corresponding weight to be zero ( $w_{37} = 0$ ).
- Operationally, if you have a few feature templates, then it's probably easier to just manually include or exclude them — this will give you more intuition.
- If you have a lot of individual features, you can apply more automatic methods for selecting features, but these are beyond the scope of this class.

# Strategy 2: norm

$$\mathbf{w} \in \mathbb{R}^d$$

Reduce the norm (length)  $\|\mathbf{w}\|$ :



[whiteboard:  $x \mapsto w_1 x$ ]

# Controlling the norm

Regularized objective:

$$\min_{\mathbf{w}} \text{TrainLoss}(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|^2$$



## Algorithm: gradient descent

Initialize  $\mathbf{w} = [0, \dots, 0]$

For  $t = 1, \dots, T$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta (\nabla_{\mathbf{w}} [\text{TrainLoss}(\mathbf{w})] + \lambda \mathbf{w})$$

Same as gradient descent, except shrink the weights towards zero by  $\lambda$ .

- A related way to keep the weights small is called **regularization**, which involves adding an additional term to the objective function which penalizes the norm (length) of  $w$ . This is probably the most common way to control the norm.
- This form of regularization is also known as  $L_2$  regularization, or weight decay in deep learning literature.
- We can use gradient descent on this regularized objective, and this simply leads to an algorithm which subtracts a scaled down version of  $w$  in each iteration. This has the effect of keeping  $w$  closer to the origin than it otherwise would be.
- Note: Support Vector Machines are exactly hinge loss + regularization.

# Controlling the norm: early stopping



## Algorithm: gradient descent

Initialize  $\mathbf{w} = [0, \dots, 0]$

For  $t = 1, \dots, T$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} \text{TrainLoss}(\mathbf{w})$$

Idea: simply make  $T$  smaller

Intuition: if have fewer updates, then  $\|\mathbf{w}\|$  can't get too big.

Lesson: try to minimize the training error, but don't try too hard.

- A really cheap way to keep the weights small is to do **early stopping**. As we run more iterations of gradient descent, the objective function improves. If we cared about the objective function, this would always be a good thing. However, our true objective is not the training loss.
- Each time we update the weights,  $w$  has the potential of getting larger, so by running gradient descent a fewer number of iterations, we are implicitly ensuring that  $w$  stays small.
- Though early stopping seems hacky, there is actually some theory behind it. And one paradoxical note is that we can sometimes get better solutions by performing less computation.

# Summary so far



**Key idea: keep it simple**

Try to minimize training error, but keep the hypothesis class small.



- We've seen several ways to control the size of the hypothesis class (and thus reducing variance) based on either reducing the dimensionality or reducing the norm.
- It is important to note that what matters is the **size** of the hypothesis class, not how "complex" the predictors in the hypothesis class look. To put it another way, using complex features backed by 1000 lines of code doesn't hurt you if there are only 5 of them.
- Now the question is: how do we actually decide how big to make the hypothesis class, and in what ways (which features)?

# Hyperparameters



## Definition: hyperparameters

Properties of the learning algorithm (features, regularization parameter  $\lambda$ , number of iterations  $T$ , step size  $\eta$ , etc.).

How do we choose hyperparameters?

Choose hyperparameters to minimize  $\mathcal{D}_{\text{train}}$  error? **No** - solution would be to include all features, set  $\lambda = 0$ ,  $T \rightarrow \infty$ .

Choose hyperparameters to minimize  $\mathcal{D}_{\text{test}}$  error? **No** - choosing based on  $\mathcal{D}_{\text{test}}$  makes it an unreliable estimate of error!

# Validation

Problem: can't use test set!

Solution: randomly take out 10-50% of training data and use it instead of the test set to estimate test error.



## Definition: validation set

A **validation set** is taken out of the training data which acts as a surrogate for the **test set**.

- However, if we make the hypothesis class too small, then the approximation error gets too big. In practice, how do we decide the appropriate size? Generally, our learning algorithm has multiple **hyperparameters** to set. These hyperparameters cannot be set by the learning algorithm on the training data because we would just choose a degenerate solution and overfit. On the other hand, we can't use the test set either because then we would spoil the test set.
- The solution is to invent something that looks like a test set. There's no other data lying around, so we'll have to steal it from the training set. The resulting set is called the **validation set**.
- With this validation set, now we can simply try out a bunch of different hyperparameters and choose the setting that yields the lowest error on the validation set. Which hyperparameter values should we try? Generally, you should start by getting the right order of magnitude (e.g.,  $\lambda = 0.0001, 0.001, 0.01, 0.1, 1, 10$ ) and then refining if necessary.
- In  $K$ -fold **cross-validation**, you divide the training set into  $K$  parts. Repeat  $K$  times: train on  $K - 1$  of the parts and use the other part as a validation set. You then get  $K$  validation errors, from which you can report both the mean and the variance, which gives you more reliable information.

# Development cycle



## Problem: simplified named-entity recognition

Input: a string  $x$  (e.g., *Governor [Gavin Newsom] in*)

Output:  $y$ , whether  $x$  contains a person or not (e.g., +1)



## Algorithm: recipe for success

- Split data into train, val, test
- Look at data to get intuition
- Repeat:
  - Implement feature / tune hyperparameters
  - Run learning algorithm
  - Sanity check train and val error rates, weights
  - Look at errors to brainstorm improvements
- Run on test set to get final error rates

[live solution]

- This slide represents the most important yet most overlooked part of machine learning: how to actually apply it in practice.
- We have so far talked about the mathematical foundation of machine learning (loss functions and optimization), and discussed some of the conceptual issues surrounding overfitting, generalization, and the size of hypothesis classes. But what actually takes most of your time is not writing new algorithms, but going through a **development cycle**, where you iteratively improve your system.
- Suppose you're given a binary classification task (backed by a dataset). What is the process by which you get to a working system? There are many ways to do this; here is one that I've found to be effective.
- The key is to stay connected with the data and the model, and have intuition about what's going on. Make sure to empirically examine the data before proceeding to the actual machine learning. It is imperative to understand the nature of your data in order to understand the nature of your problem. (You might even find that your problem admits a simple, clean solution sans machine learning.) Understanding trained models can be hard sometimes, as machine learning algorithms (even linear classifiers) are often not the easiest things to understand when you have thousands of parameters.

- First, maintain data hygiene. Hold out a test set from your data that you don't look at until you're done. Start by looking at the data to get intuition. You can start to brainstorm what features / predictors you will need. You can compute some basic statistics.
- Then you enter a loop: implement a new feature. There are three things to look at: error rates, weights, and predictions. First, sanity check the error rates and weights to make sure you don't have an obvious bug. Then do an **error analysis** to see which examples your predictor is actually getting wrong. The art of practical machine learning is turning these observations into new features.
- Finally, run your system once on the test set and report the number you get. If your test error is much higher than your validation error, then you probably did too much tweaking and were **overfitting** (at a meta-level) the validation set.



# Roadmap

Generalization

**Unsupervised learning**

Summary

# Supervision?

## Supervised learning:

- Prediction:  $\mathcal{D}_{\text{train}}$  contains input-output pairs  $(x, y)$
- Fully-labeled data is very **expensive** to obtain (we can maybe get thousands of labeled examples)

## Unsupervised learning:

- Clustering:  $\mathcal{D}_{\text{train}}$  only contains inputs  $x$
- Unlabeled data is much **cheaper** to obtain (we can maybe get billions of unlabeled examples)

- We have so far covered the basics of **supervised learning**. If you get a labeled training set of  $(x, y)$  pairs, then you can train a predictor. However, where do these examples  $(x, y)$  come from? If you're doing image classification, someone has to sit down and label each image, and generally this tends to be expensive enough that we can't get that many examples.
- On the other hand, there are tons of **unlabeled examples** sitting around (e.g., Flickr for photos, Wikipedia, news articles for text documents). The main question is whether we can harness all that unlabeled data to help us make better predictions? This is the goal of **unsupervised learning**.

# Word clustering

**Input:** raw text (100 million words of news articles)...

**Output:**

Cluster 1: Friday Monday Thursday Wednesday Tuesday Saturday Sunday weekends Sundays Saturdays

Cluster 2: June March July April January December October November September August

Cluster 3: water gas coal liquid acid sand carbon steam shale iron

Cluster 4: great big vast sudden mere sheer gigantic lifelong scant colossal

Cluster 5: man woman boy girl lawyer doctor guy farmer teacher citizen

Cluster 6: American Indian European Japanese German African Catholic Israeli Italian Arab

Cluster 7: pressure temperature permeability density porosity stress velocity viscosity gravity tension

Cluster 8: mother wife father son husband brother daughter sister boss uncle

Cluster 9: machine device controller processor CPU printer spindle subsystem compiler plotter

Cluster 10: John George James Bob Robert Paul William Jim David Mike

Cluster 11: anyone someone anybody somebody

Cluster 12: feet miles pounds degrees inches barrels tons acres meters bytes

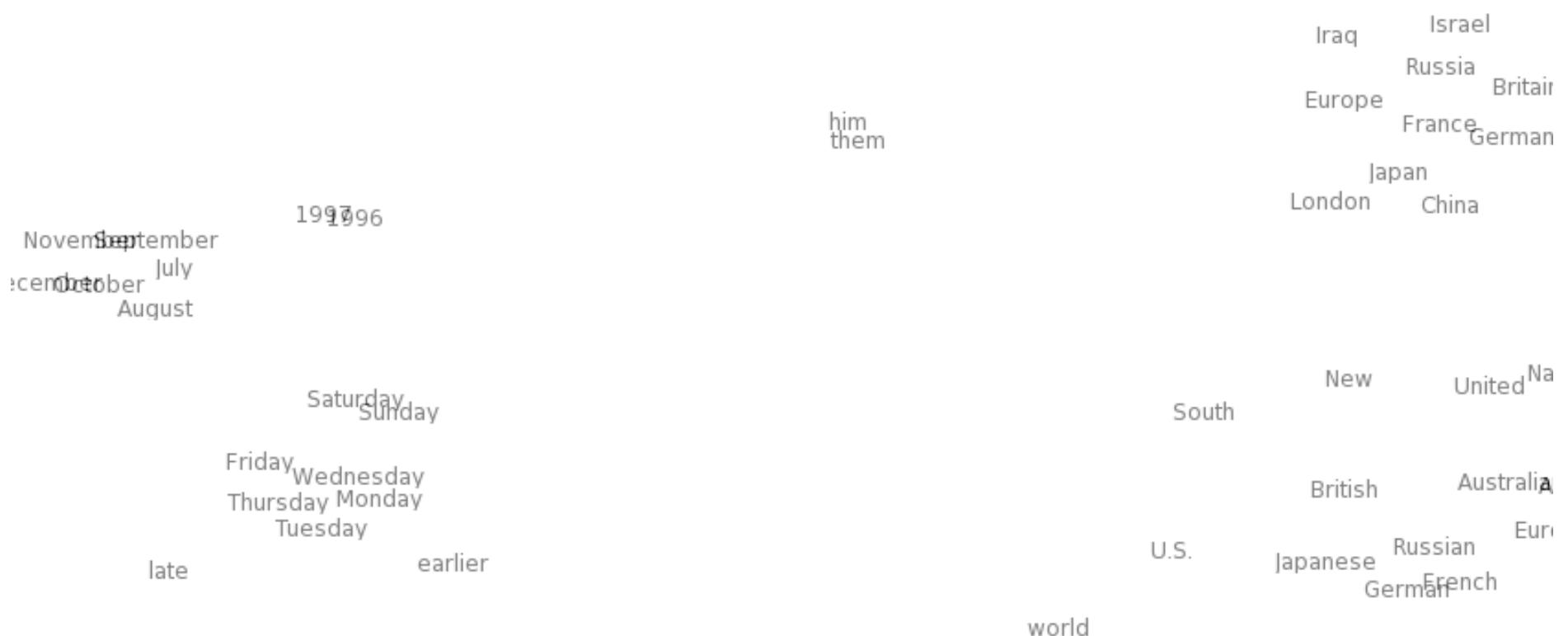
Cluster 13: director chief professor commissioner commander treasurer founder superintendent dean custodian

Cluster 14: had hadn't hath would've could've should've must've might've

Cluster 15: head body hands eyes voice arm seat eye hair mouth

- Empirically, unsupervised learning has produced some pretty impressive results. HMMs (more specifically, Brown clustering) can be used to take a ton of raw text and cluster related words together.
- It is important to note that no one told the algorithm what days of the week were or months or family relations. The clustering algorithm discovered this structure automatically by simply examining the statistics of raw text.

# Word vectors

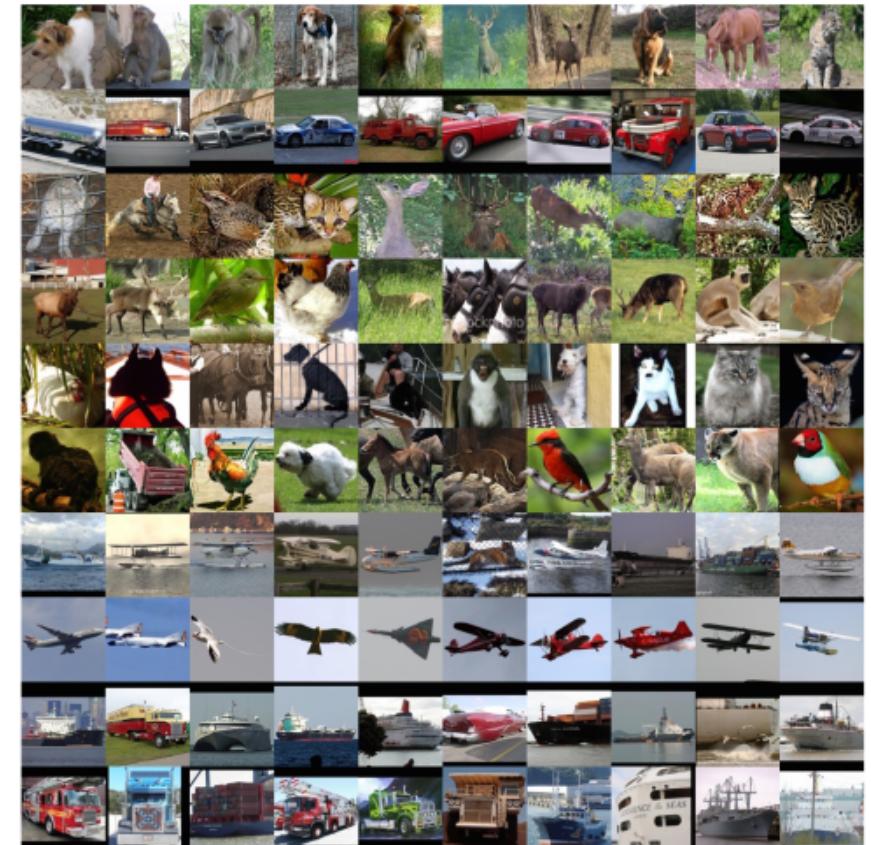


- A related idea are word vectors, which became popular after Tomas Mikolov created word2vec in 2013 (though the idea of vector space representations had been around for a while).
- Instead of representing a word by discrete clusters, a word is represented by a vector, which gives us a notion of similarity between words.
- More recently, **contextualized word representations** such as ELMo, BERT, XLNet, ALBERT, etc. have been very impactful. These methods also are unsupervised in that they only require raw text as input, but they produce representations of words in context. These representations essentially serve as good features for any NLP task, and empirically these methods have resulted in significant gains.

# Clustering with deep embeddings



(a) MNIST



(b) STL-10

- In an example from vision, one can learn a feature representation (embedding) for images along with a clustering of them.



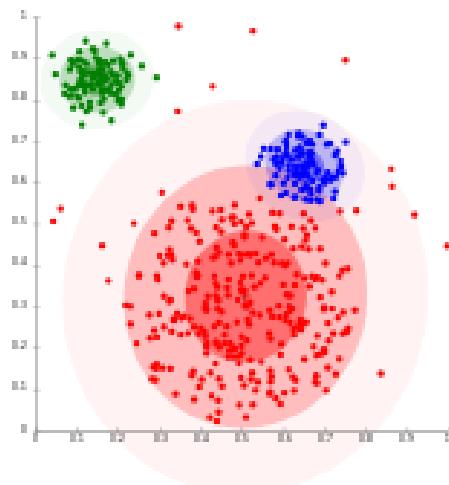
## Key idea: unsupervised learning

Data has lots of rich **latent** structures; want methods to discover this **structure** automatically.

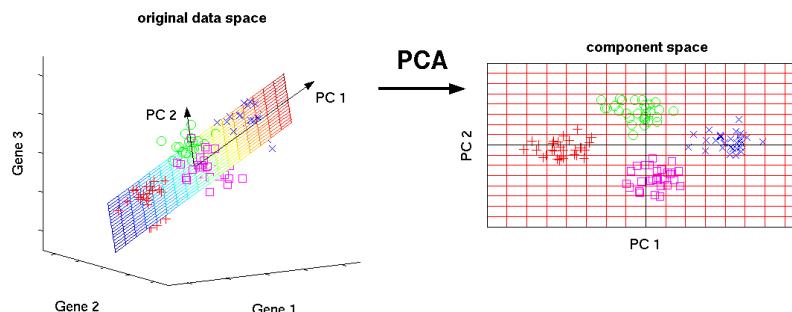
- Unsupervised learning in some sense is the holy grail: you don't have to tell the machine anything — it just "figures it out." However, one must not be overly optimistic here: there is no free lunch. You ultimately still have to tell the algorithm something, at least in the way you define the features or set up the optimization problem.

# Types of unsupervised learning

Clustering (e.g., K-means):



Dimensionality reduction (e.g., PCA):



- There are many forms of unsupervised learning, corresponding to different types of latent structures you want to pull out of your data. In this class, we will focus on one of them: clustering.

# Clustering



## Definition: clustering

Input: training set of input points

$$\mathcal{D}_{\text{train}} = \{x_1, \dots, x_n\}$$

Output: assignment of each point to a cluster

$$[z_1, \dots, z_n] \text{ where } z_i \in \{1, \dots, K\}$$

Intuition: Want similar points to be in same cluster, dissimilar points to be in different clusters

[whiteboard]

- The task of clustering is to take a set of points as input and return a partitioning of the points into  $K$  clusters. We will represent the partitioning using an **assignment vector**  $z = [z_1, \dots, z_n]$ . For each  $i$ ,  $z_i \in \{1, \dots, K\}$  specifies which of the  $K$  clusters point  $i$  is assigned to.

# K-means objective

Setup:

- Each cluster  $k = 1, \dots, K$  is represented by a **centroid**  $\mu_k \in \mathbb{R}^d$
- **Intuition:** want each point  $\phi(x_i)$  close to its assigned centroid  $\mu_{z_i}$

Objective function:

$$\text{Loss}_{\text{kmeans}}(z, \mu) = \sum_{i=1}^n \|\phi(x_i) - \mu_{z_i}\|^2$$

Need to choose centroids  $\mu$  and assignments  $z$  **jointly**

- **K-means** is a particular method for performing clustering which is based on associating each cluster with a **centroid**  $\mu_k$  for  $k = 1, \dots, K$ . The intuition is to assign the points to clusters **and** place the centroid for each cluster so that each point  $\phi(x_i)$  is close to its assigned centroid  $\mu_{z_i}$ .

# K-means: simple example



## Example: one-dimensional

Input:  $\mathcal{D}_{\text{train}} = \{0, 2, 10, 12\}$

Output:  $K = 2$  centroids  $\mu_1, \mu_2 \in \mathbb{R}$

If know centroids  $\mu_1 = 1, \mu_2 = 11$ :

$$z_1 = \arg \min \{(0 - 1)^2, (0 - 11)^2\} = 1$$

$$z_2 = \arg \min \{(2 - 1)^2, (2 - 11)^2\} = 1$$

$$z_3 = \arg \min \{(10 - 1)^2, (10 - 11)^2\} = 2$$

$$z_4 = \arg \min \{(12 - 1)^2, (12 - 11)^2\} = 2$$

If know assignments  $z_1 = z_2 = 1, z_3 = z_4 = 2$ :

$$\mu_1 = \arg \min_{\mu} (0 - \mu)^2 + (2 - \mu)^2 = 1$$

$$\mu_2 = \arg \min_{\mu} (10 - \mu)^2 + (12 - \mu)^2 = 11$$

- How do we solve this optimization problem? We can't just use gradient descent because there are discrete variables (assignment variables  $z_i$ ). We can't really use dynamic programming because there are continuous variables (the centroids  $\mu_k$ ).
- To motivate the solution, consider a simple example with four points. As always, let's try to break up the problem into subproblems.
- What if we knew the optimal centroids? Then computing the assignment vectors is trivial (for each point, choose the closest center).
- What if we knew the optimal assignments? Then computing the centroids is also trivial (one can check that this is just averaging the points assigned to that center).
- The only problem is that we don't know the optimal centroids or assignments, and unlike in dynamic programming, the two depend on one another cyclically.

# K-means algorithm

$$\min_z \min_{\mu} \text{Loss}_{\text{kmeans}}(z, \mu)$$



**Key idea: alternating minimization**

Tackle **hard** problem by solving two easy problems.

- And now the leap of faith is this: start with an arbitrary setting of the centroids (not optimal). Then alternate between choosing the best assignments given the centroids, and choosing the best centroids given the assignments. This is the K-means algorithm.

# K-means algorithm (Step 1)

Goal: given centroids  $\mu_1, \dots, \mu_K$ , assign each point to the best centroid.



## Algorithm: Step 1 of K-means

For each point  $i = 1, \dots, n$ :

Assign  $i$  to cluster with closest centroid:

$$z_i \leftarrow \arg \min_{k=1, \dots, K} \|\phi(x_i) - \mu_k\|^2.$$

- **Step 1** of K-means fixes the centroids. Then we can optimize the K-means objective with respect to  $z$  alone quite easily. It is easy to show that the best label for  $z_i$  is the cluster  $k$  that minimizes the distance to the centroid  $\mu_k$  (which is fixed).

# K-means algorithm (Step 2)

Goal: given cluster assignments  $z_1, \dots, z_n$ , find the best centroids  $\mu_1, \dots, \mu_K$ .



## Algorithm: Step 2 of K-means

For each cluster  $k = 1, \dots, K$ :

Set  $\mu_k$  to average of points assigned to cluster  $k$ :

$$\mu_k \leftarrow \frac{1}{|\{i : z_i = k\}|} \sum_{i:z_i=k} \phi(x_i)$$

- Now, turning things around, let's suppose we knew what the assignments  $z$  were. We can again look at the K-means objective function and try to optimize it with respect to the centroids  $\mu$ . The best  $\mu_k$  is to place the centroid at the average of all the points assigned to cluster  $k$ ; this is **step two**.

# K-means algorithm

Objective:

$$\min_z \min_{\mu} \text{Loss}_{\text{kmeans}}(z, \mu)$$



## Algorithm: K-means

Initialize  $\mu_1, \dots, \mu_K$  randomly.

For  $t = 1, \dots, T$ :

    Step 1: set assignments  $z$  given  $\mu$

    Step 2: set centroids  $\mu$  given  $z$

[demo]

- Now we have the two ingredients to state the full K-means algorithm. We start by initializing all the centroids randomly. Then, we iteratively alternate back and forth between steps 1 and 2, optimizing  $z$  given  $\mu$  and vice-versa.

# K-means: simple example



Example: one-dimensional

Input:  $\mathcal{D}_{\text{train}} = \{0, 2, 10, 12\}$

Output:  $K = 2$  centroids  $\mu_1, \mu_2 \in \mathbb{R}$

Initialization (random):  $\mu_1 = 0, \mu_2 = 2$

Iteration 1:

- Step 1:  $z_1 = 1, z_2 = 2, z_3 = 2, z_4 = 2$
- Step 2:  $\mu_1 = 0, \mu_2 = 8$

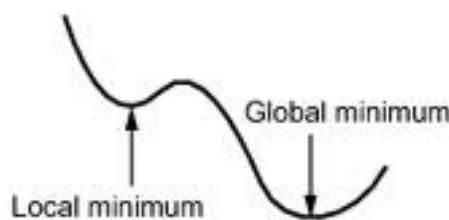
Iteration 2:

- Step 1:  $z_1 = 1, z_2 = 1, z_3 = 2, z_4 = 2$
- Step 2:  $\mu_1 = 1, \mu_2 = 11$

- Here is an example of an execution of K-means where we converged to the correct answer.

# Local minima

K-means is guaranteed to converge to a local minimum, but is not guaranteed to find the global minimum.



[demo: getting stuck in local optima, seed = 100]

## Solutions:

- Run multiple times from different random initializations
- Initialize with a heuristic (K-means++)

- K-means is guaranteed to decrease the loss function each iteration and will converge to a local minimum, but it is not guaranteed to find the global minimum, so one must exercise caution when applying K-means.
- One solution is to simply run K-means several times from multiple random initializations and then choose the solution that has the lowest loss.
- Or we could try to be smarter in how we initialize K-means. K-means++ is an initialization scheme which places centroids on training points so that these centroids tend to be distant from one another.



# Unsupervised learning summary

- Leverage tons of unlabeled data
- Difficult optimization:

latent variables  $z$



parameters  $\mu$



# Roadmap

Generalization

Unsupervised learning

**Summary**



# Summary

- Feature extraction (think hypothesis classes) [modeling]
- Prediction (linear, neural network, k-means) [modeling]
- Loss functions (compute gradients) [modeling]
- Optimization (stochastic gradient, alternating minimization) [learning]
- Generalization (think development cycle) [modeling]

- This concludes our tour of the foundations of machine learning, although machine learning will come up again later in the course. You should have gotten more than just a few isolated equations and algorithms. It is really important to think about the overarching principles in a modular way.
- First, feature extraction is where you put your domain knowledge into. In designing features, it's useful to think in terms of the induced **hypothesis classes** — what kind of functions can your learning algorithm potentially learn?
- These features then drive **prediction**: either linearly or through a neural network. We can even think of k-means as trying to predict the data points using the centroids.
- **Loss functions** connect predictions with the actual training examples.
- Note that all of the design decisions up to this point are about modeling. Algorithms are very important, but only come in once we have the right **optimization problem** to solve.
- Finally, machine learning requires a leap of faith. How does optimizing anything at training time help you **generalize** to new unseen examples at test time? Learning can only work when there's a common core that cuts past all the idiosyncrasies of the examples. This is exactly what features are meant to capture.

# A brief history

1795: Gauss proposed least squares (astronomy)

1940s: logistic regression (statistics)

1952: Arthur Samuel built program that learned to play checkers (AI)

1957: Rosenblatt invented Perceptron algorithm (like SGD)

1969: Minsky and Papert "killed" machine learning

1980s: neural networks (backpropagation, from 1960s)

1990: interface with optimization/statistics, SVMs

2000s-: structured prediction, revival of neural networks, etc.

- Many of the ideas surrounding fitting functions was known in other fields long before computers, let alone AI.
- When computers arrived on the scene, learning was definitely on people's radar, although this was detached from the theoretical, statistical and optimization foundations.
- In 1969, Minsky and Papert wrote a famous book *Perceptrons*, which showed the limitations of linear classifiers with the famous XOR example (similar to our car collision example), which killed off this type of research. AI largely turned to symbolic methods.
- Since the 1980s, machine learning has increased its role in AI, been placed on a more solid mathematical foundation with its connection with optimization and statistics.
- While there is a lot of optimism today about the potential of machine learning, there are still a lot of unsolved problems.

# Challenges

## Capabilities:

- More complex prediction problems (translation, generation)
- Unsupervised learning: automatically discover structure

## Responsibilities:

- Feedback loops: predictions affect user behavior, which generates data
- Fairness: build classifiers that don't discriminate?
- Privacy: can we pool data together
- Interpretability: can we understand what algorithms are doing?

- Going ahead, one major thrust is to improve the capabilities of machine learning. Broadly construed, machine learning is about learning predictors from some input to some output. The simplest case is when the output is just a label, but increasingly, researchers have been using the same machine learning tools for doing translation (output is a sentence), speech synthesis (output is a waveform), and image generation (output is an image).
- Another important direction is being able to leverage the large amounts of unlabeled data to learn good representations. Can we automatically discover the underlying structure (e.g., a 3D model of the world from videos)? Can we learn a causal model of the world? How can we make sure that the representations we are learning are useful for some other task?
- A second major thrust has to do with the context in which machine learning is now routinely being applied, for example in high-stakes scenarios such as self-driving cars. But machine learning does not exist in a vacuum. When machine learning systems are deployed to real users, it changes user behavior, and since the same systems are being trained on this user-generated data, this results in feedback loops.
- We also want to build ML systems which are fair. The real world is not fair; thus the data generated from it will reflect these discriminatory biases. Can we overcome these biases?
- The strength of machine learning lies in being able to aggregate information across many individuals. However, this appears to require a central organization that collects all this data, which seems like poor practice from the point of view of protecting privacy. Can we perform machine learning while protecting individual privacy? For example, local differential privacy mechanisms inject noise into an individual's measurement before sending it to the central server.
- Finally, there is the issue of trust of machine learning systems in high-stakes situations. As these systems become more complex, it becomes harder for humans to "understand" how and why a system is making a particular decision.

# Machine learning



**Key idea: learning**

Programs should improve with experience.

So far: reflex-based models

Next time: state-based models

- If we generalize for a moment, machine learning is really about programs that can improve with experience.
- So far, we have only focused on reflex-based models where the program only outputs a yes/no or a number, and the experience is examples of input-output pairs.
- Next time, we will start looking at models which can perform higher-level reasoning, but machine learning will remain our companion for the remainder of the class.

# Section 2: Learning

Chuma Kabaghe, Chuanbo Pan

- Beyond Linear Functions
- Backpropagation
- Nearest Neighbors
- Scikit-learn Tutorial

# Beyond Linear Functions

# Example: beyond linear functions

Regression:  $x \in \mathbb{R}, y \in \mathbb{R}$

Linear functions:

$$\phi(x) = x$$

$$\mathcal{F}_1 = \{x \mapsto w_1 x + w_2 x^2 : w_1 \in \mathbb{R}, w_2 = 0\}$$

Quadratic functions:

$$\phi(x) = [x, x^2]$$

$$\mathcal{F}_2 = \{x \mapsto w_1 x + w_2 x^2 : w_1 \in \mathbb{R}, w_2 \in \mathbb{R}\}$$

[whiteboard]

# Example: even more flexible functions

Regression:  $x \in \mathbb{R}, y \in \mathbb{R}$

Piecewise constant functions:

$$\phi(x) = [\mathbf{1}[0 < x \leq 1], \mathbf{1}[1 < x \leq 2], \dots]$$

$$\mathcal{F}_3 = \{x \mapsto \sum_{j=1}^{10} w_j \mathbf{1}[j-1 < x \leq j] : \mathbf{w} \in \mathbb{R}^{10}\}$$

[whiteboard]

# Backpropagation

# Backpropagation

$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$

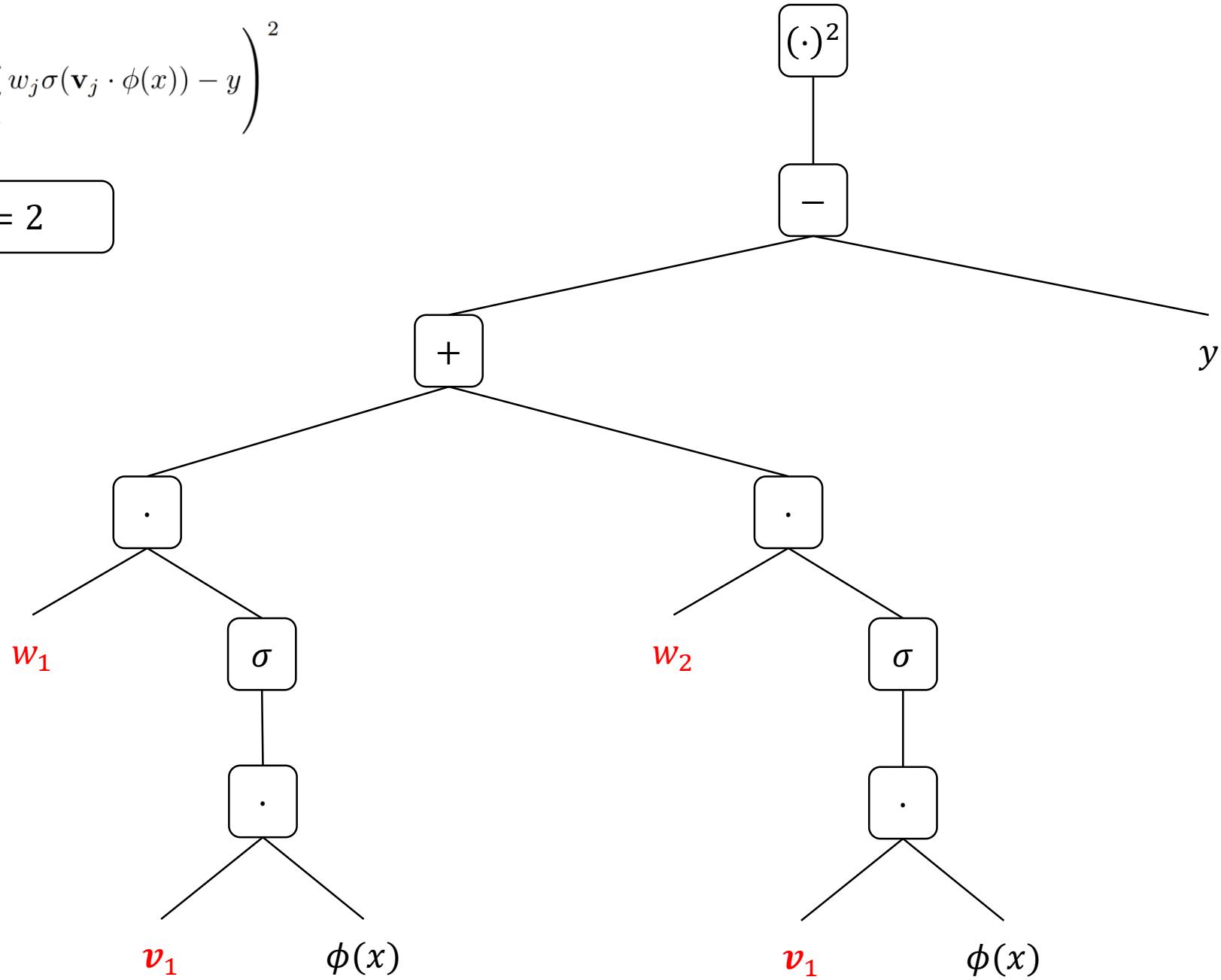
# Backpropagation

$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$

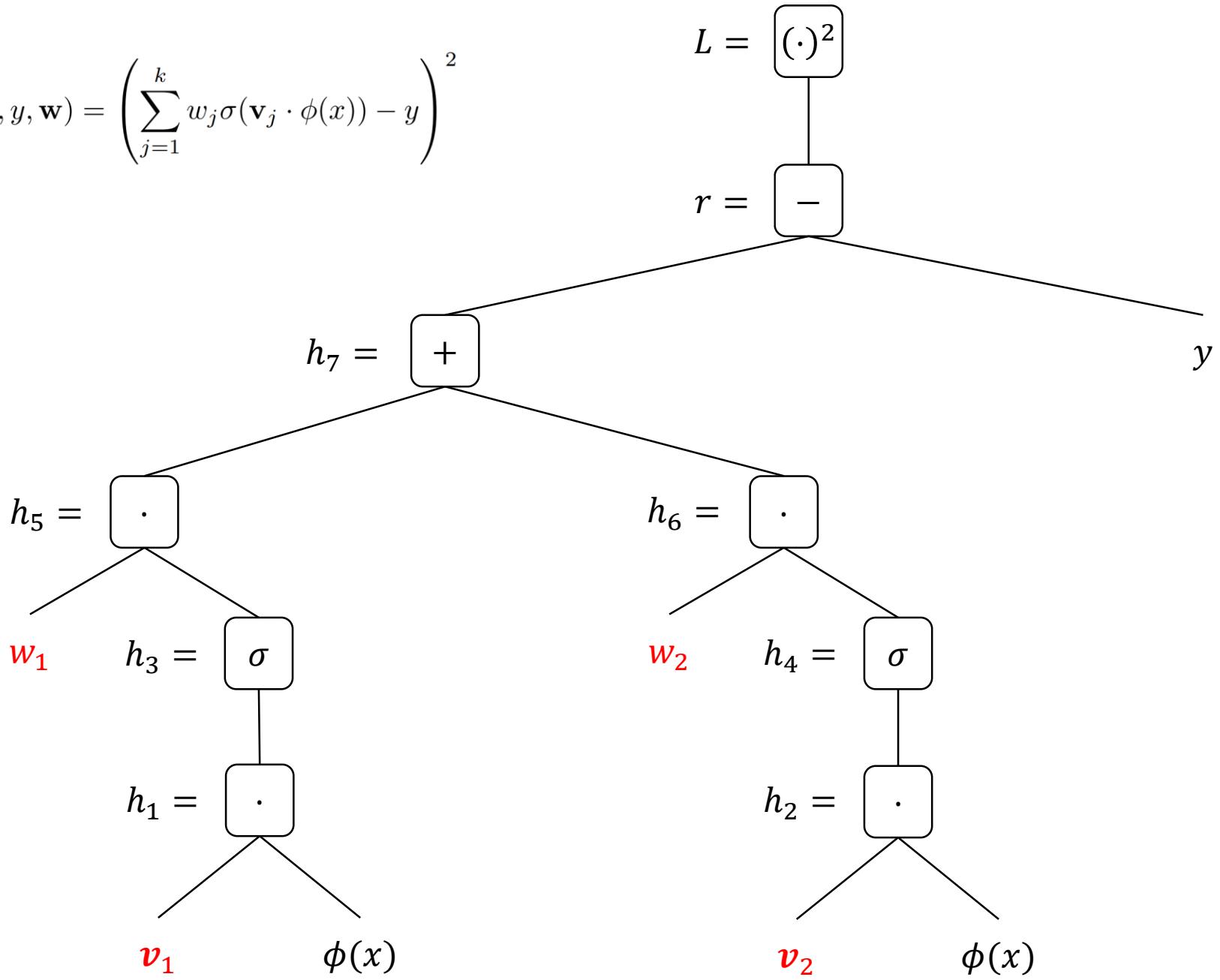
The diagram illustrates the backpropagation formula. A green rectangular box encloses the term  $\sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x))$ . Two blue arrows point from the text "Weights" above to the  $w_j$  coefficients in the summation. Below the green box, the word "Predicted" is centered. To the right of the green box, another green rectangular box encloses the variable  $y$ . A black curved arrow points from the text "Actual" below to the  $y$  inside this box.

$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$

*Consider  $k = 2$*



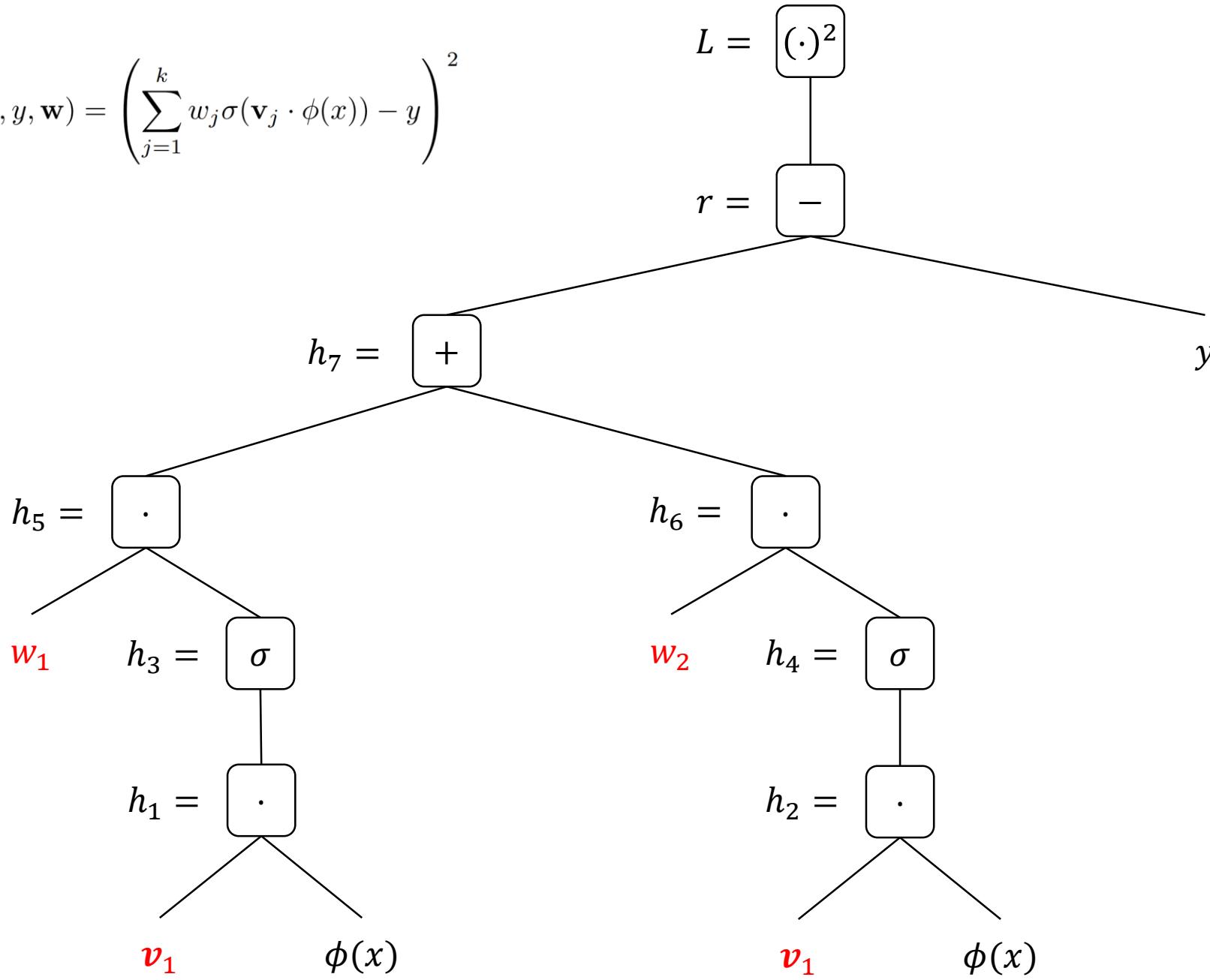
$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$



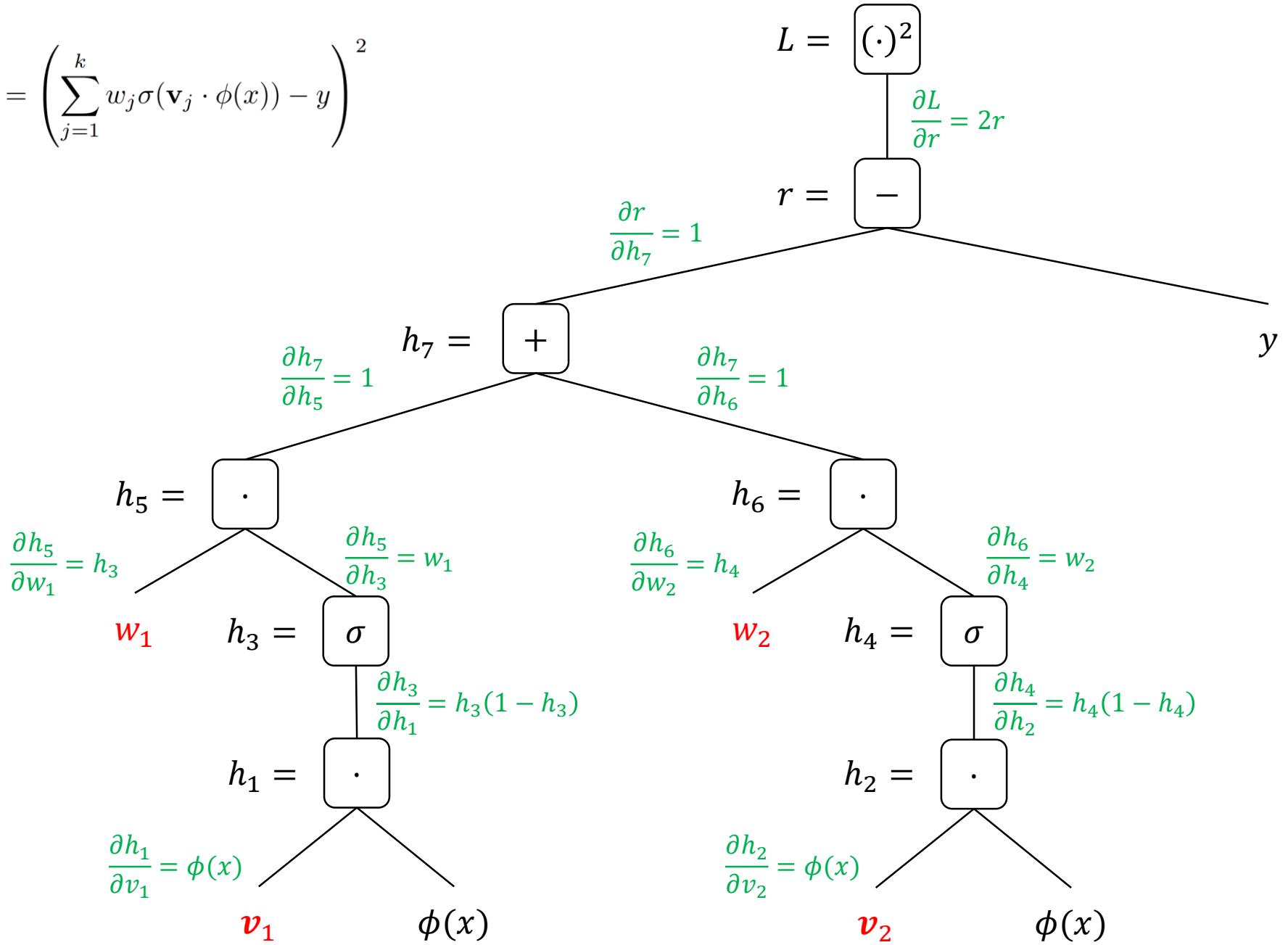
### Forward Pass

Compute each node and store intermediate values

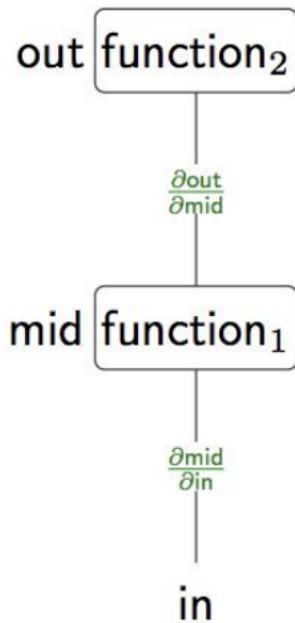
$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$



$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$

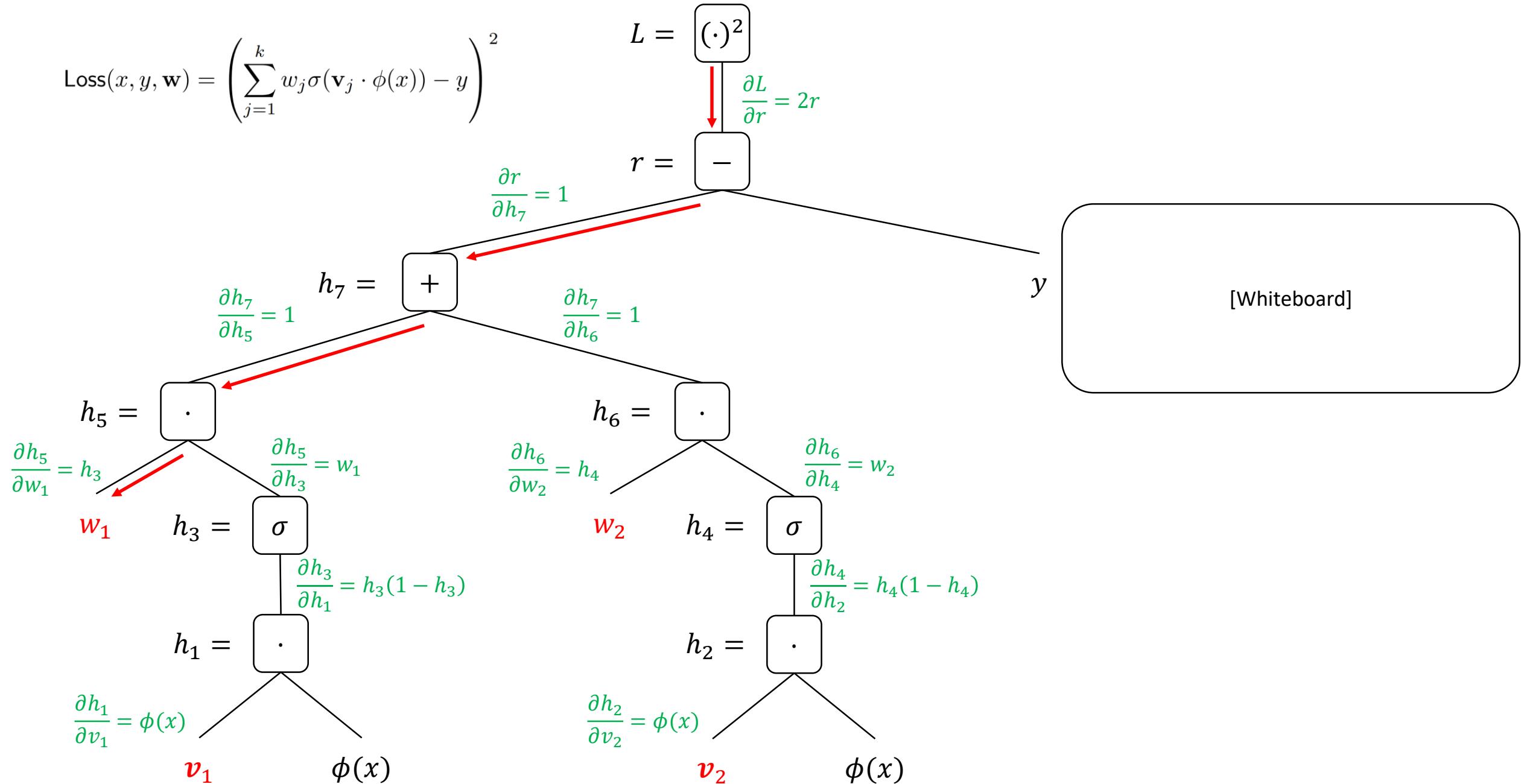


# Backpropagation

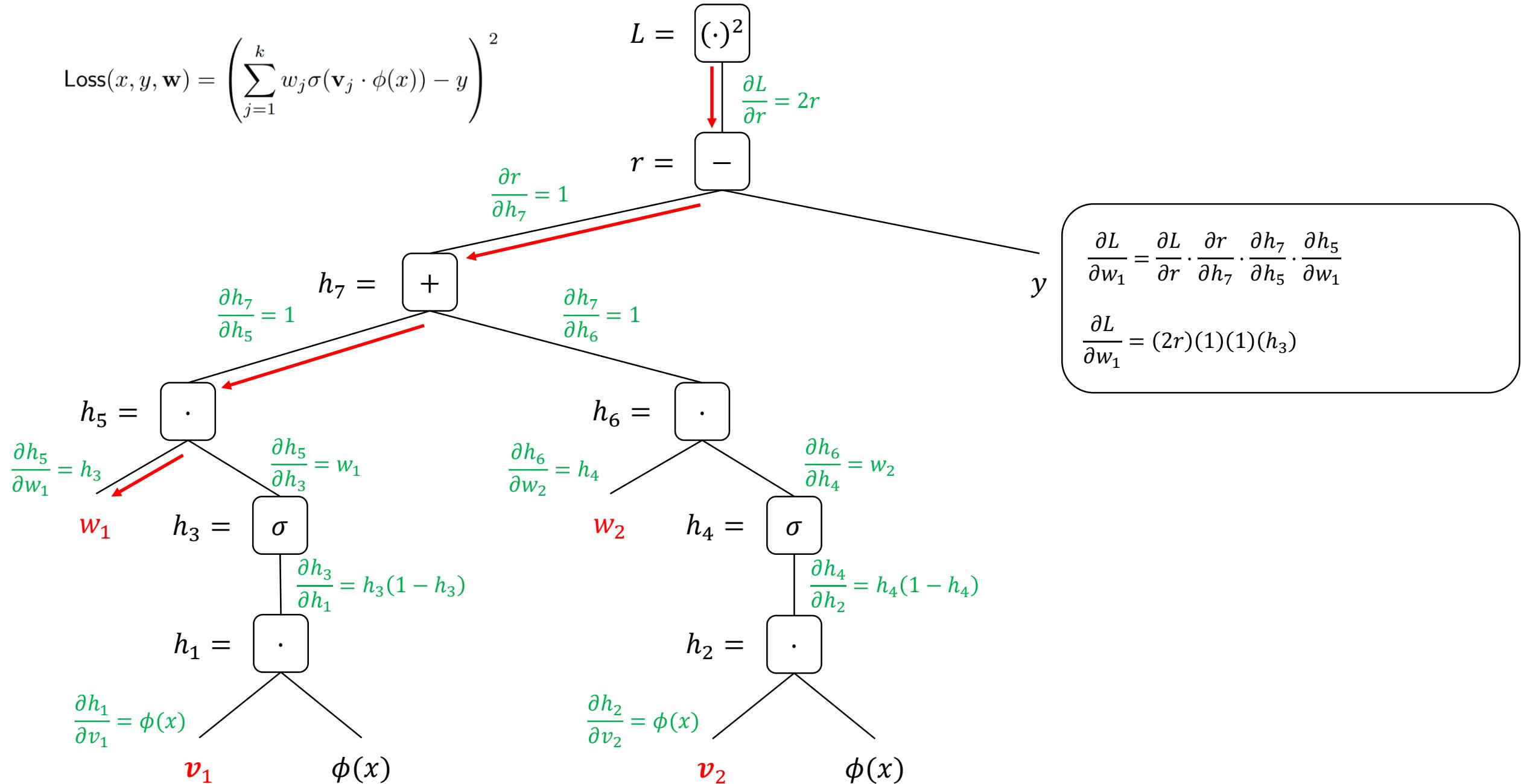


Chain rule:  $\frac{\partial \text{out}}{\partial \text{in}} = \frac{\partial \text{out}}{\partial \text{mid}} \frac{\partial \text{mid}}{\partial \text{in}}$

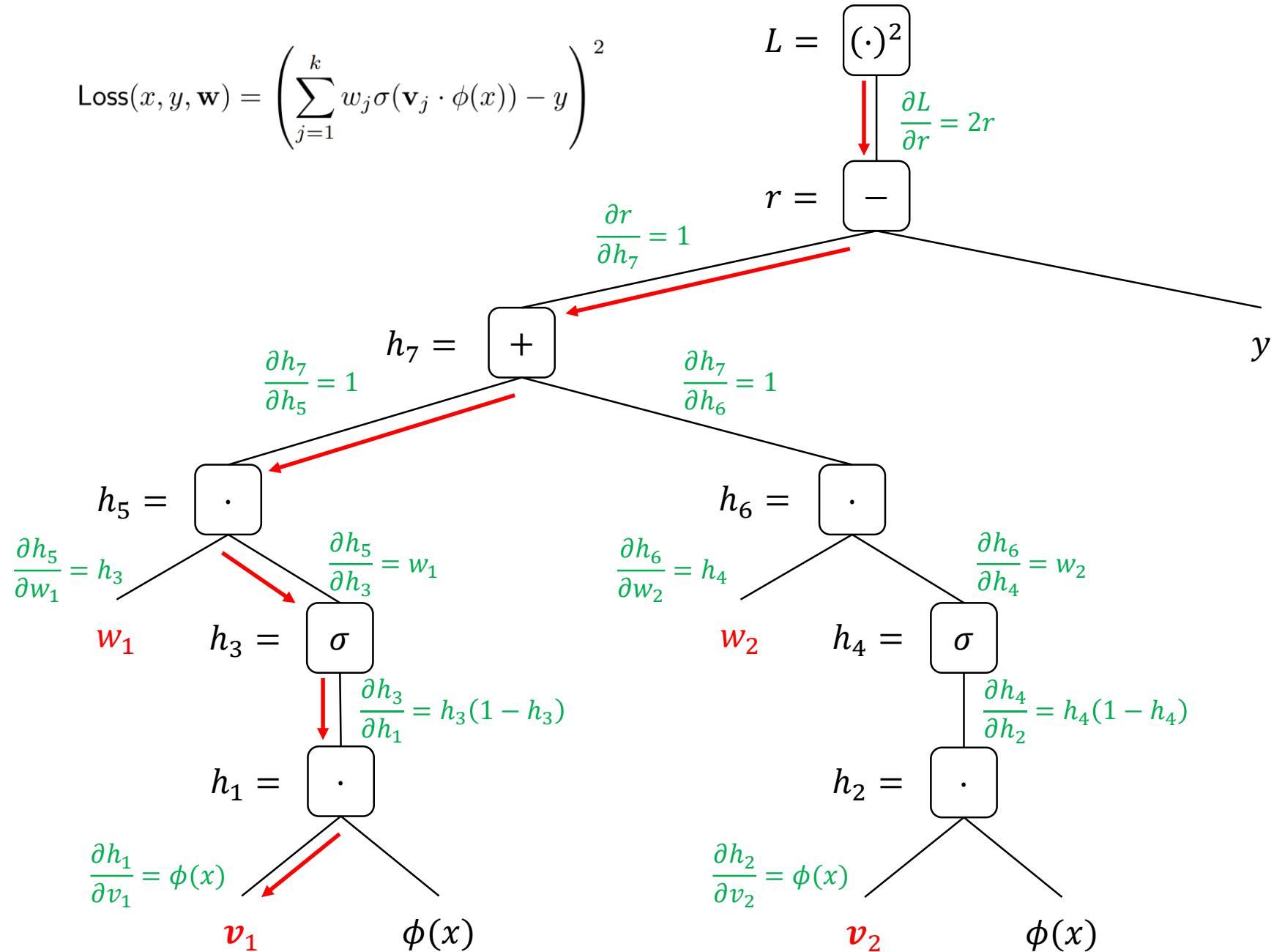
$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$



$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$



$$\text{Loss}(x, y, \mathbf{w}) = \left( \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(x)) - y \right)^2$$



# Backpropagation

Backprop: <http://cs231n.github.io/optimization-2/>

Vector, Matrix, and Tensor Derivatives: <http://cs231n.Stanford.edu/vecDerivs.pdf>



# Nearest Neighbors

# Nearest neighbors



## Algorithm: nearest neighbors

Training: just store  $\mathcal{D}_{\text{train}}$

Predictor  $f(x')$ :

- Find  $(x, y) \in \mathcal{D}_{\text{train}}$  where  $\|\phi(x) - \phi(x')\|$  is smallest
- Return  $y$

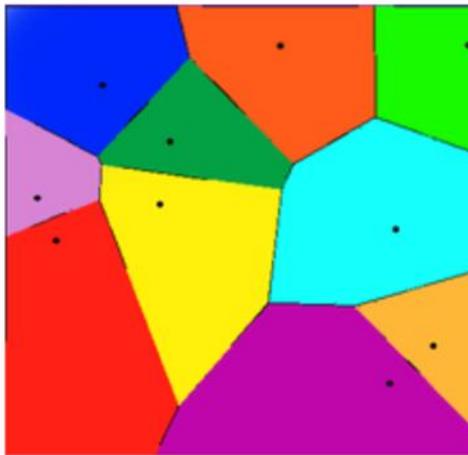


## Key idea: similarity

Similar examples tend to have similar outputs.

# Expressivity of nearest neighbors

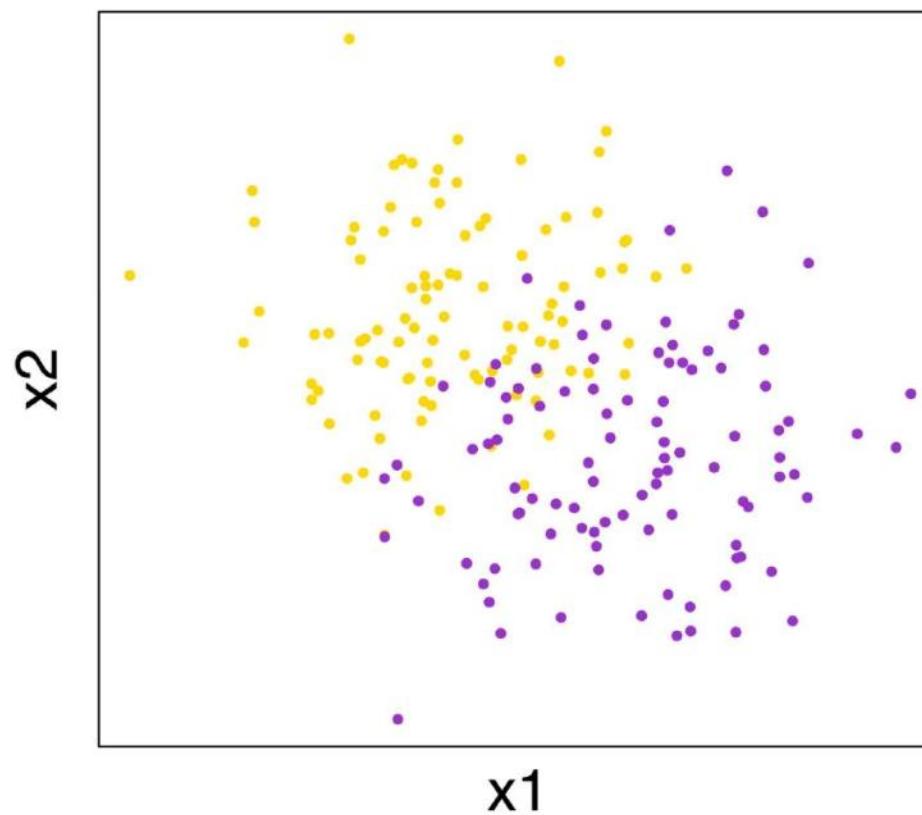
Decision boundary: based on Voronoi diagram



- Much more expressive than quadratic features
- **Non-parametric:** the hypothesis class adapts to number of examples
- Simple and powerful, but kind of brute force

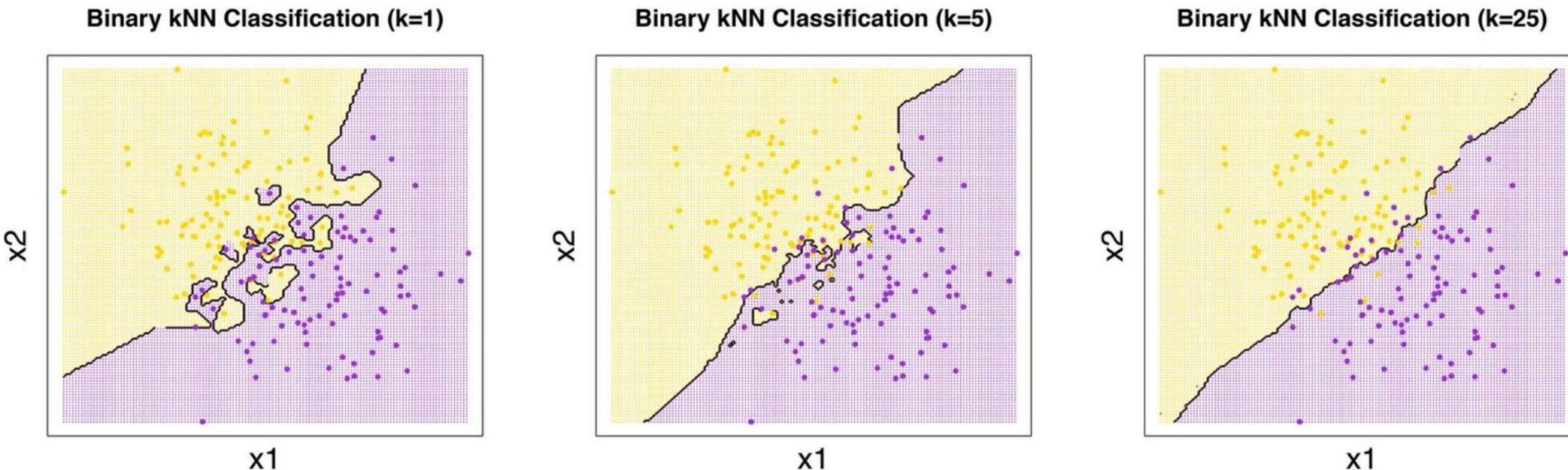
# $k$ -nearest neighbors

**Binary kNN Classification Training Set**

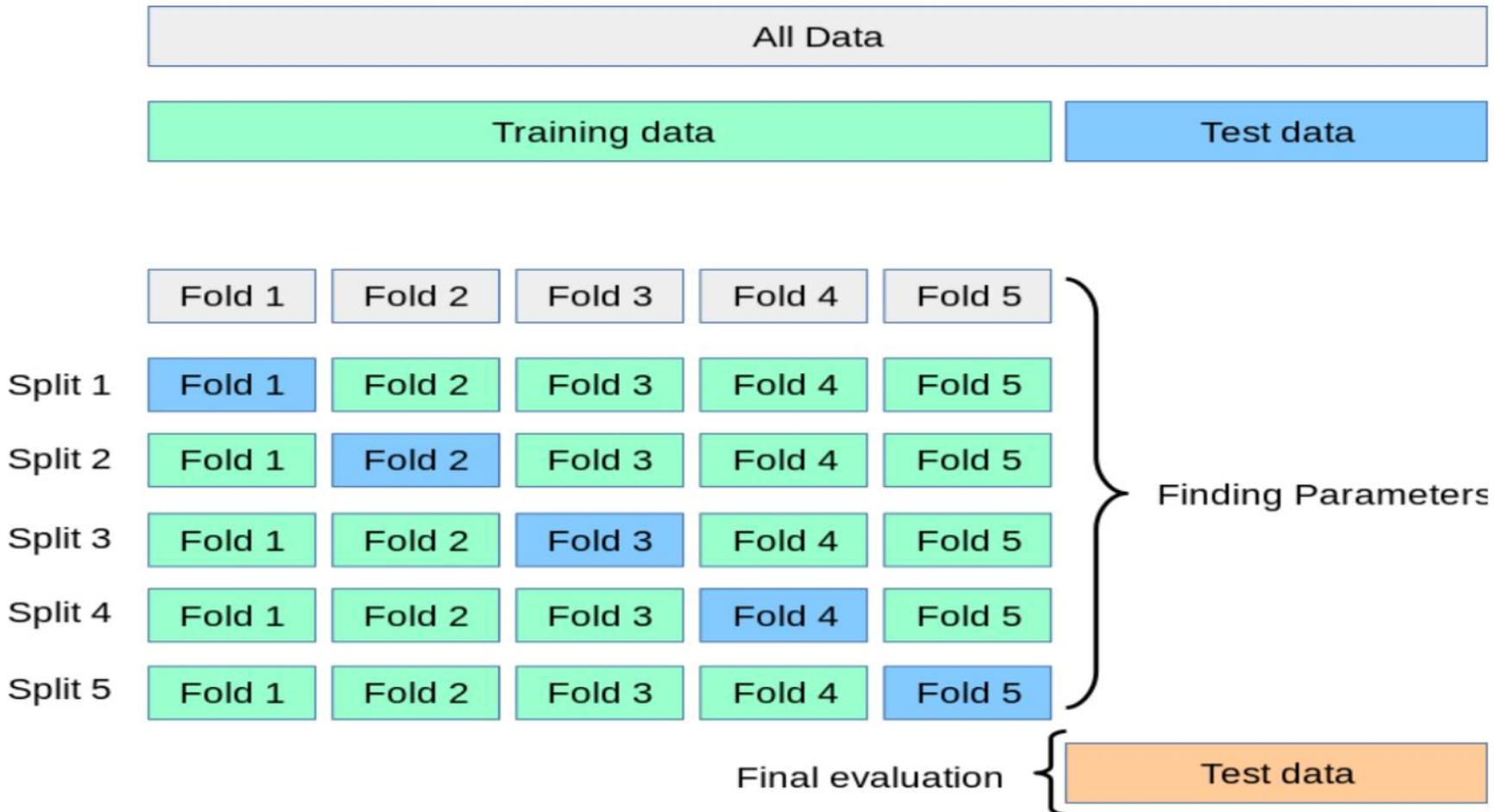


# $k$ -nearest neighbors

Effect of  $k$ :



# $k$ -fold cross-validation



[https://scikit-learn.org/stable/modules/cross\\_validation.html](https://scikit-learn.org/stable/modules/cross_validation.html)

# Scikit-learn Tutorial

[Switch to ipython notebook]



# Lecture 5: Search I





# Question

A **farmer** wants to get his **cabbage**, **goat**, and **wolf** across a river. He has a boat that only holds two. He cannot leave the cabbage and goat alone or the goat and wolf alone. How many river crossings does he need?

4

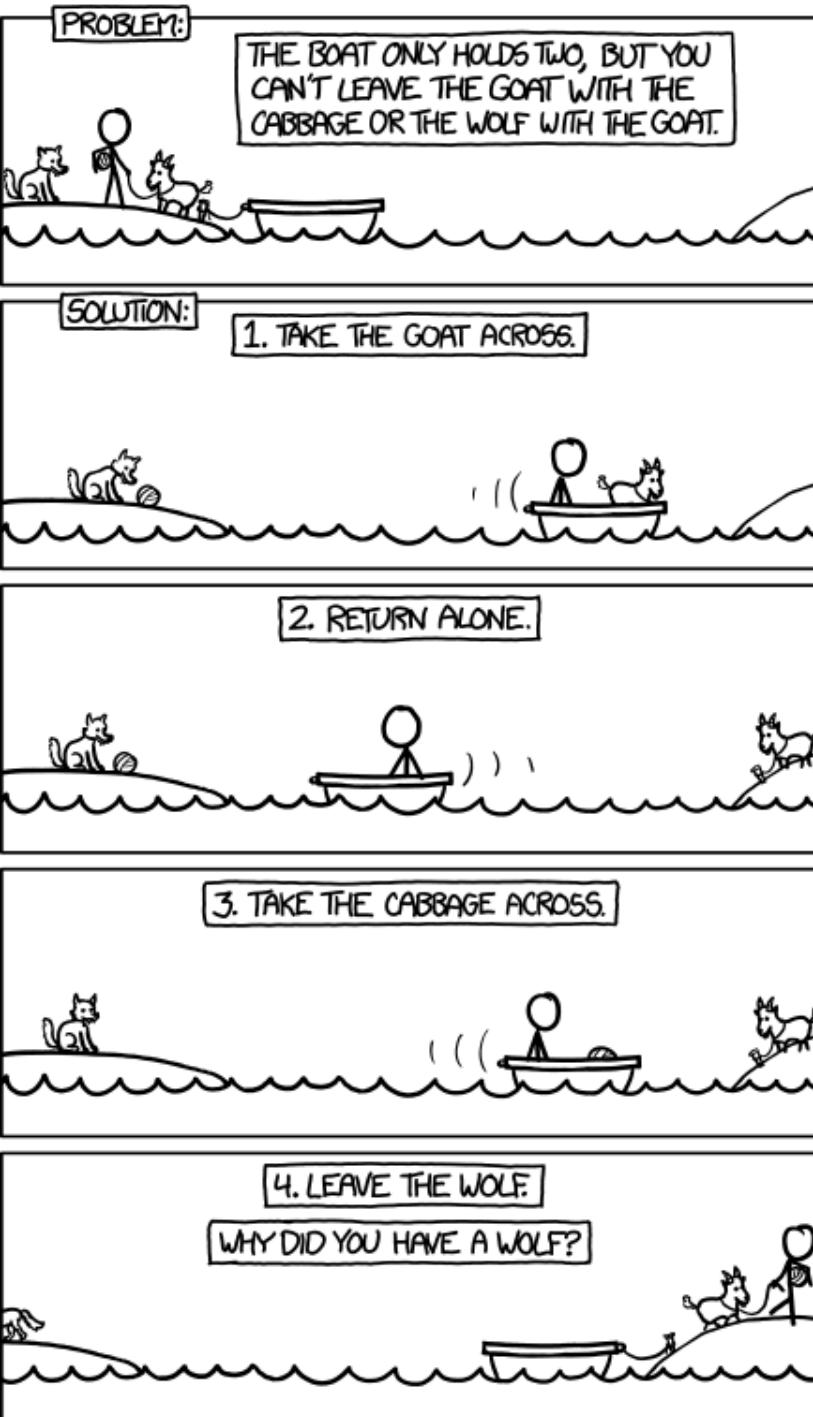
5

6

7

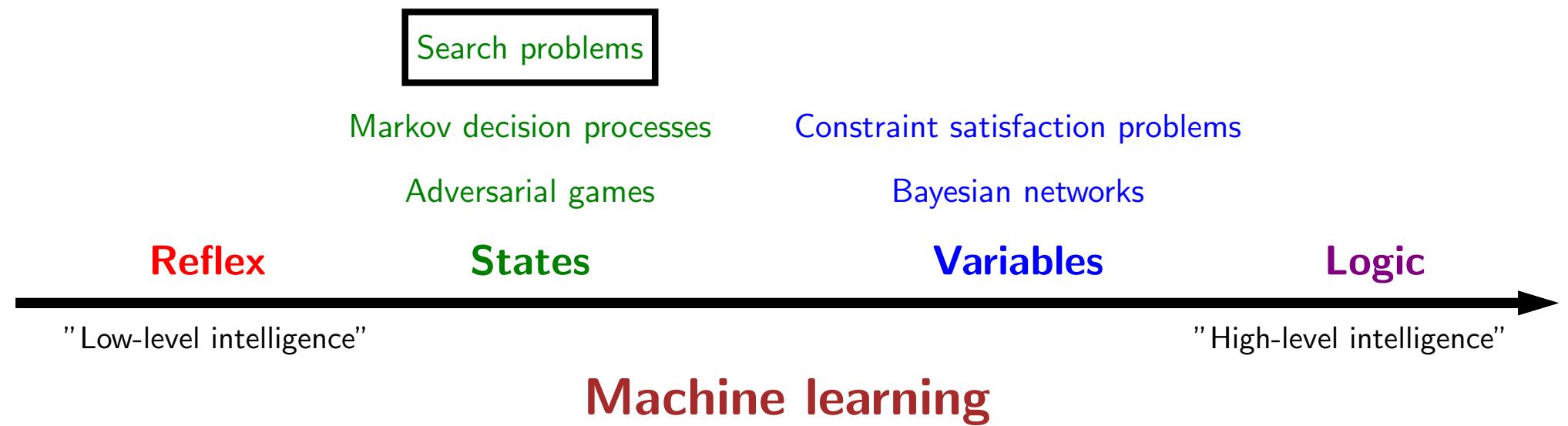
no solution

- When you solve this problem, try to think about how you did it. You probably simulated the scenario in your head, trying to send the farmer over with the goat and observing the consequences. If nothing got eaten, you might continue with the next action. Otherwise, you undo that move and try something else.
- But the point is not for you to be able to solve this one problem manually. The real question is: How can we get a machine to do solve all problems like this automatically? One of the things we need is a systematic approach that considers all the possibilities. We will see that **search problems** define the possibilities, and **search algorithms** explore these possibilities.



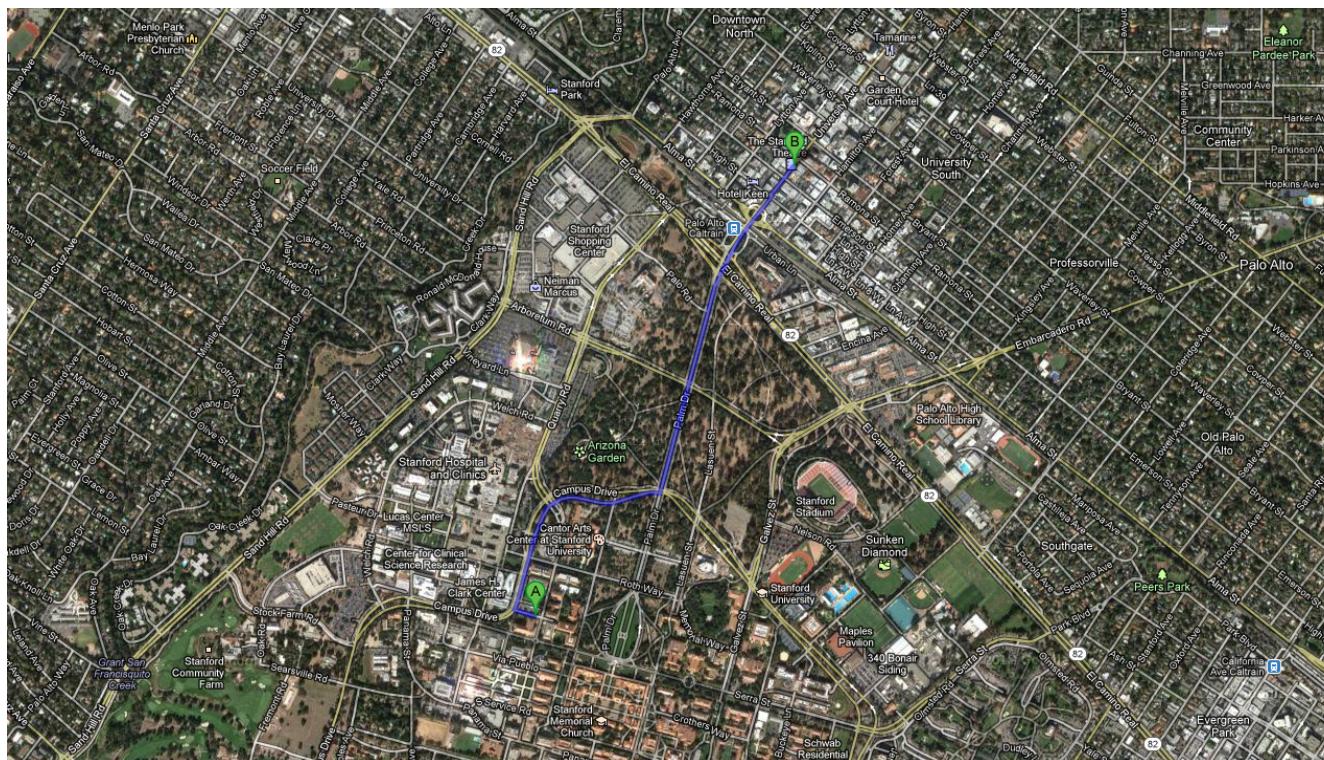
- This example, taken from *xkcd*, points out the cautionary tale that sometimes you can do better if you change the model (perhaps the value of having a wolf is zero) instead of focusing on the algorithm.

# Course plan



- So far, we have worked with only the simplest types of models — reflex models. We used these as a starting point to explore machine learning. Now we will proceed to the first type of state-based models, search problems.

# Application: route finding

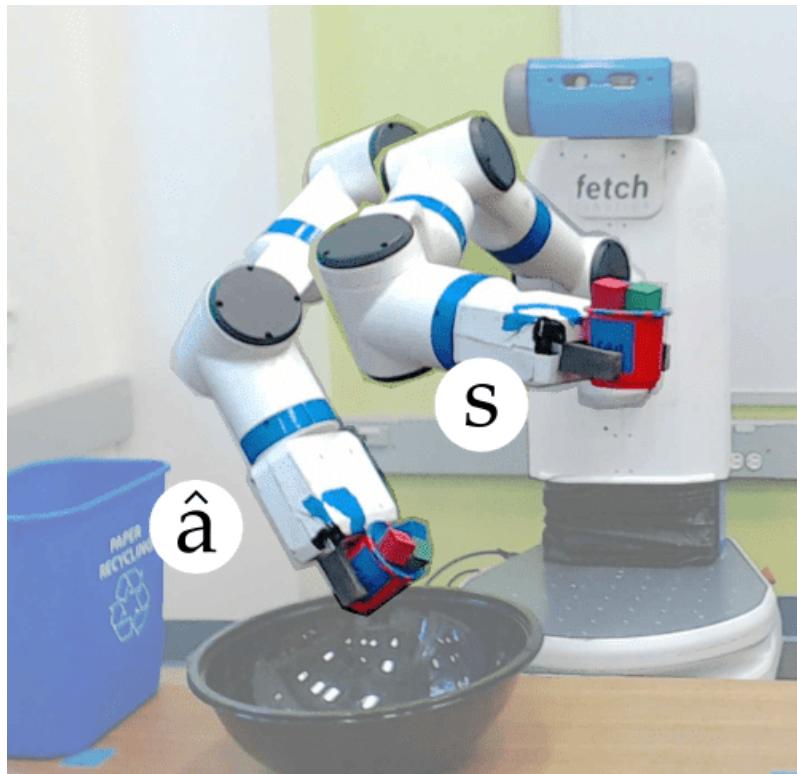


**Objective:** shortest? fastest? most scenic?

**Actions:** go straight, turn left, turn right

- Route finding is perhaps the most canonical example of a search problem. We are given as the input a map, a source point and a destination point. The goal is to output a sequence of actions (e.g., go straight, turn left, or turn right) that will take us from the source to the destination.
- We might evaluate action sequences based on an objective (distance, time, or pleasantness).

# Application: robot motion planning

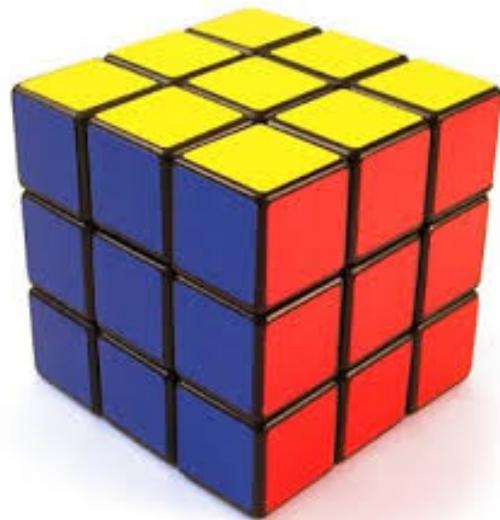


**Objective:** fastest? most energy efficient? safest? most expressive?

**Actions:** translate and rotate joints

- In robot motion planning, the goal is get a robot to move from one position/pose to another. The desired output trajectory consists of individual actions, each action corresponding to moving or rotating the joints by a small amount.
- Again, we might evaluate action sequences based on various resources like time or energy.

# Application: solving puzzles



**Objective:** reach a certain configuration

**Actions:** move pieces (e.g., Move12Down)

- In solving various puzzles, the output solution can be represented by a sequence of individual actions. In the Rubik's cube, an action is rotating one slice of the cube. In the 15-puzzle, an action is moving one square to an adjacent free square.
- In puzzles, even finding one solution might be an accomplishment. The more ambitious might want to find the best solution (say, minimize the number of moves).

# Application: machine translation

*la maison bleue*



*the blue house*

**Objective:** fluent English and preserves meaning

**Actions:** append single words (e.g., the)

- In machine translation, the goal is to output a sentence that's the translation of the given input sentence. The output sentence can be built out of actions, each action appending a word or a phrase to the current output.

# Beyond reflex

Classifier (reflex-based models):



Search problem (state-based models):



**Key: need to consider future consequences of an action!**

- Last week, we finished our tour of machine learning of **reflex-based models** (e.g., linear predictors and neural networks) that output either a  $+1$  or  $-1$  (for binary classification) or a real number (for regression).
- While reflex-based models were appropriate for some applications such as sentiment classification or spam filtering, the applications we will look at today, such as solving puzzles, demand more.
- To tackle these new problems, we will introduce **search problems**, our first instance of a **state-based model**.
- In a search problem, in a sense, we are still building a predictor  $f$  which takes an input  $x$ , but  $f$  will now return an entire **action sequence**, not just a single action. Of course you should object: can't I just apply a reflex model iteratively to generate a sequence? While that is true, the search problems that we're trying to solve importantly require reasoning about the consequences of the entire action sequence, and cannot be tackled by myopically predicting one action at a time.
- Tangent: Of course, saying "cannot" is a bit strong, since sometimes a search problem can be solved by a reflex-based model. You could have a massive lookup table that told you what the best action was for any given situation. It is interesting to think of this as a time/memory tradeoff where reflex-based models are performing an implicit kind of caching. Going on a further tangent, one can even imagine **compiling** a state-based model into a reflex-based model; if you're walking around Stanford for the first time, you might have to really plan things out, but eventually it kind of becomes reflex.
- We have looked at many real-world examples of this paradigm. For each example, the key is to decompose the output solution into a sequence of primitive actions. In addition, we need to think about how to evaluate different possible outputs.

# Paradigm

Modeling

Inference

Learning

- Recall the modeling-inference-learning paradigm. For reflex-based classifiers, modeling consisted of choosing the features and the neural network architecture; inference was trivial forward computation of the output given the input; and learning involved using stochastic gradient descent on the gradient of the loss function, which might involve backpropagation.
- Today, we will focus on the modeling and inference part of search problems. The next lecture will cover learning.

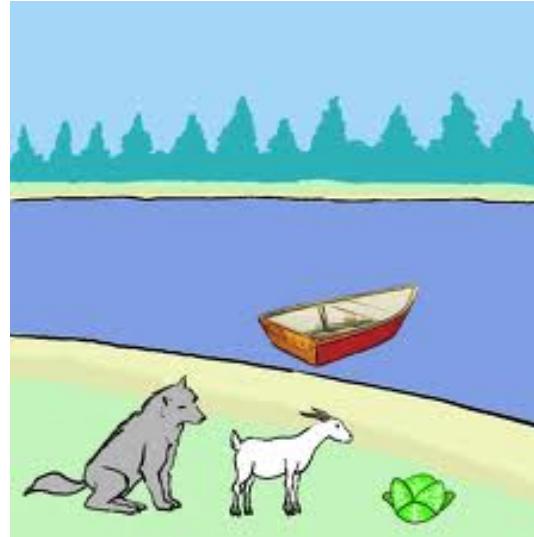


# Roadmap

Tree search

Dynamic programming

Uniform cost search



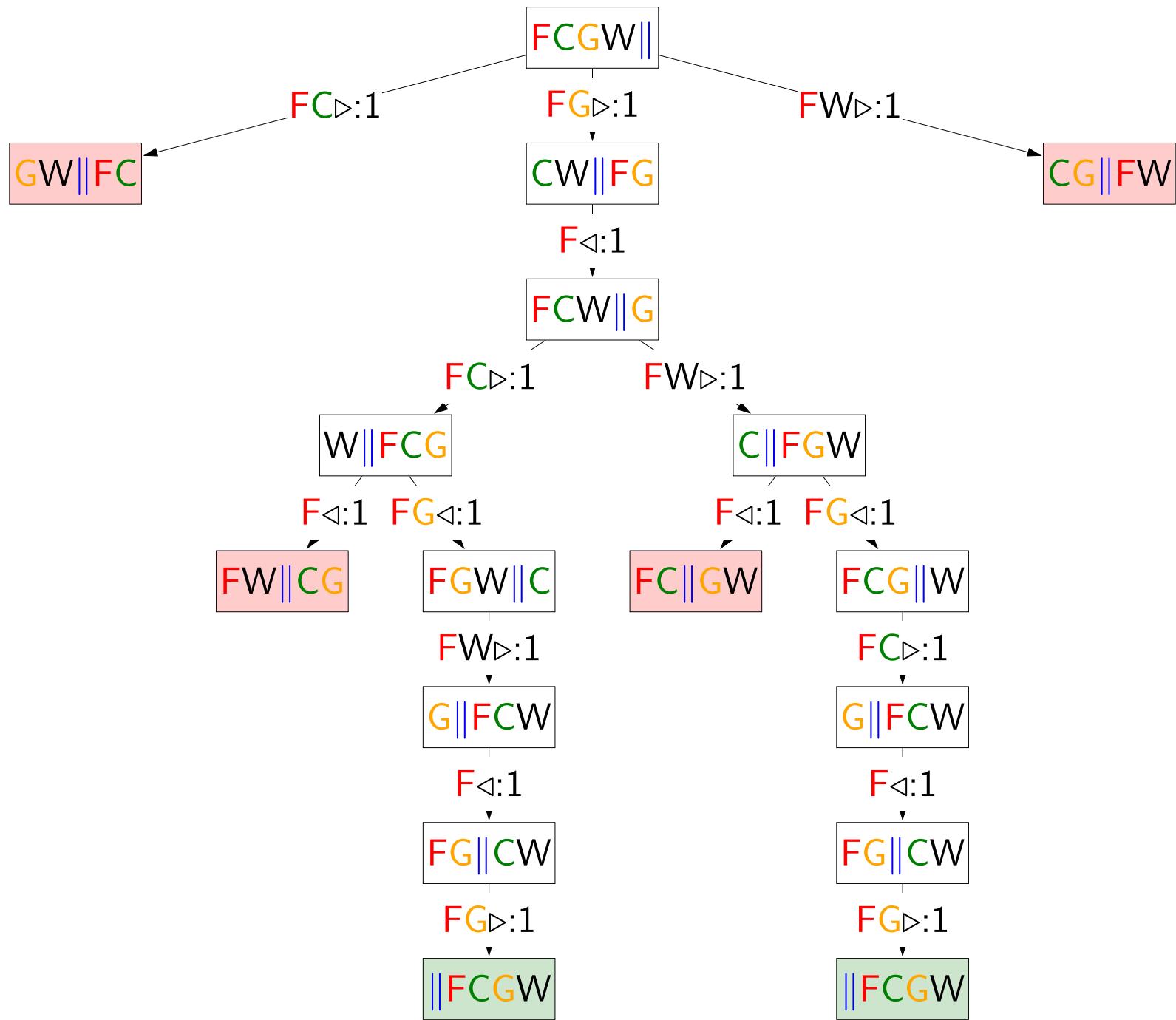
Farmer    Cabbage    Goat    Wolf

Actions:

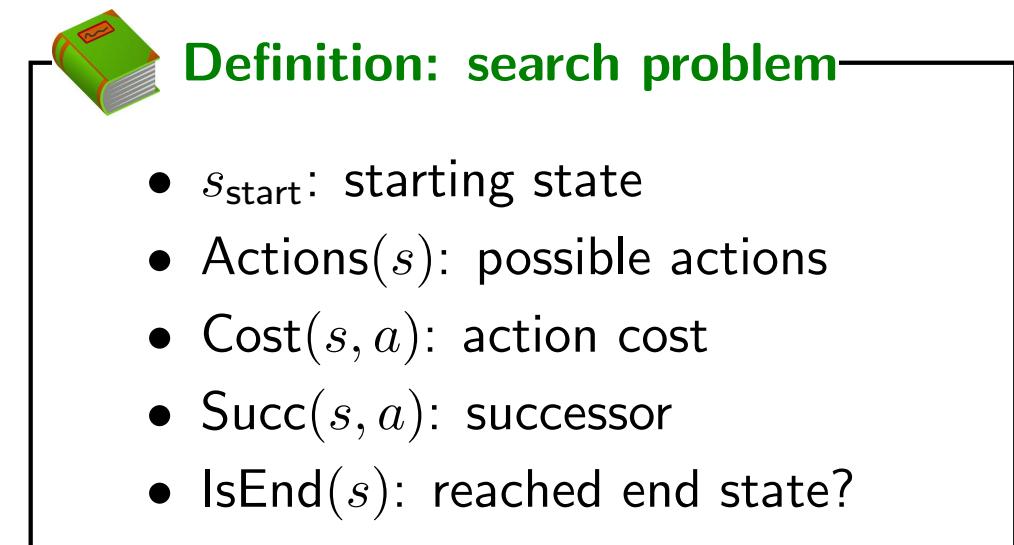
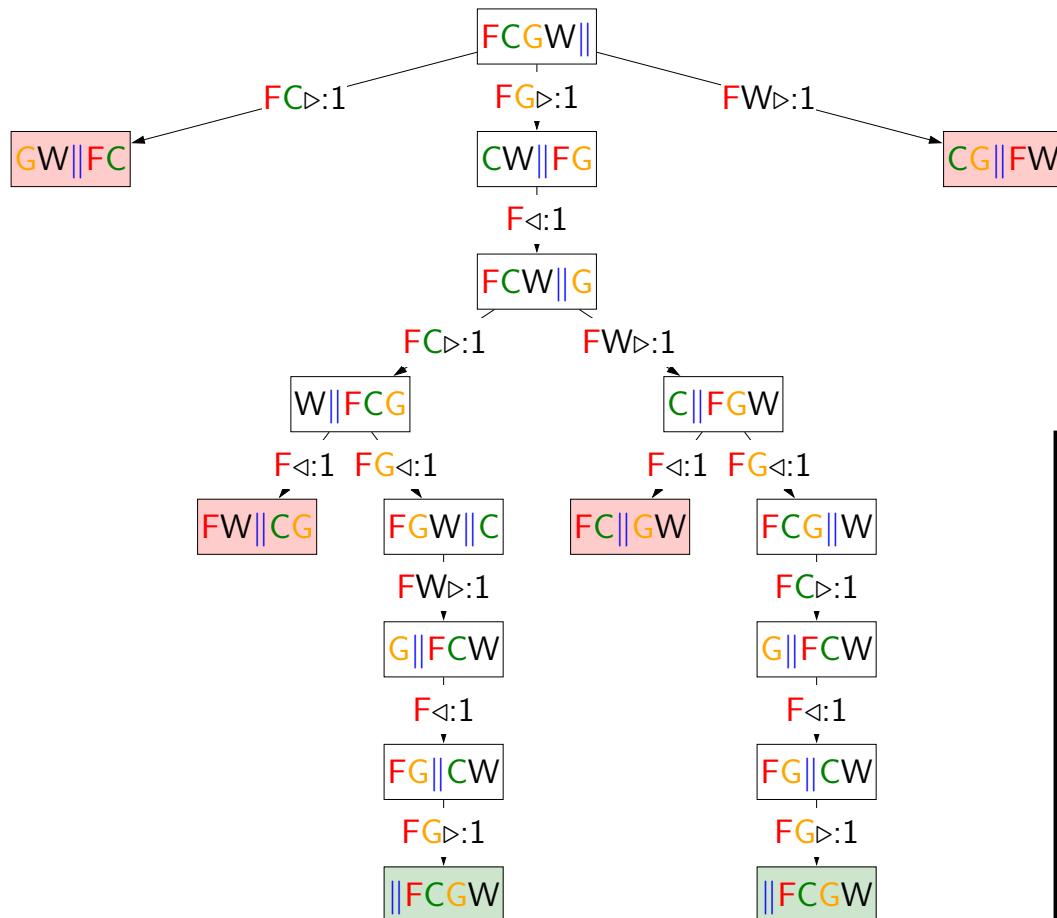
$F \triangleright$	$F \triangleleft$
$FC \triangleright$	$FC \triangleleft$
$FG \triangleright$	$FG \triangleleft$
$FW \triangleright$	$FW \triangleleft$

Approach: build a **search tree** ("what if?")

- We first start with our boat crossing puzzle. While you can possibly solve it in more clever ways, let us approach it in a very brain-dead, simple way, which allows us to introduce the notation for search problems.
- For this problem, we have eight possible actions, which will be denoted by a concise set of symbols. For example, the action  $\text{FG}\triangleright$  means that the farmer will take the goat across to the right bank;  $\text{F}\triangleleft$  means that the farmer is coming back to the left bank alone.



# Search problem



- We will build what we will call a **search tree**. The root of the tree is the start state  $s_{\text{start}}$ , and the leaves are the end states ( $\text{IsEnd}(s)$  is true). Each edge leaving a node  $s$  corresponds to a possible action  $a \in \text{Actions}(s)$  that could be performed in state  $s$ . The edge is labeled with the action and its cost, written  $a : \text{Cost}(s, a)$ . The action leads deterministically to the successor state  $\text{Succ}(s, a)$ , represented by the child node.
- In summary, each root-to-leaf path represents a possible action sequence, and the sum of the costs of the edges is the cost of that path. The goal is to find the root-to-leaf path that ends in a valid end state with minimum cost.
- Note that in code, we usually do not build the search tree as a concrete data structure. The search tree is used merely to visualize the computation of the search algorithms and study the structure of the search problem.
- For the boat crossing example, we have assumed each action (a safe river crossing) costs 1 unit of time. We disallow actions that return us to an earlier configuration. The green nodes are the end states. The red nodes are not end states but have no successors (they result in the demise of some animal or vegetable). From this search tree, we see that there are exactly two solutions, each of which has a total cost of 7 steps.



# Transportation example



## Example: transportation

Street with blocks numbered 1 to  $n$ .

Walking from  $s$  to  $s + 1$  takes 1 minute.

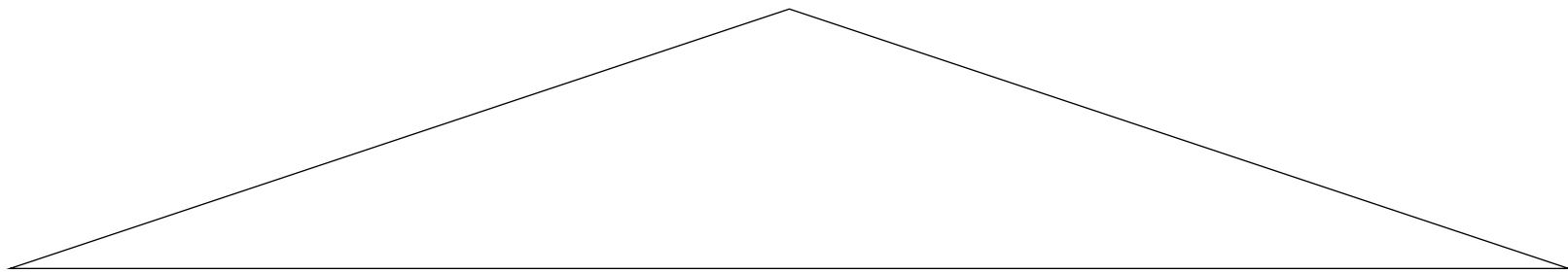
Taking a magic tram from  $s$  to  $2s$  takes 2 minutes.

How to travel from 1 to  $n$  in the least time?

[semi-live solution: `TransportationProblem`]

- Let's consider another problem and practice modeling it as a search problem. Recall that this means specifying precisely what the states, actions, goals, costs, and successors are.
- To avoid the ambiguity of natural language, we will do this directly in code, where we define a `SearchProblem` class and implement the methods: `startState`, `isEnd` and `succAndCost`.

# Backtracking search



[whiteboard: search tree]

If  $b$  actions per state, maximum depth is  $D$  actions:

- Memory:  $O(D)$  (**small**)
- Time:  $O(b^D)$  (**huge**) [ $2^{50} = 1125899906842624$ ]

- Now let's put modeling aside and suppose we are handed a search problem. How do we construct an algorithm for finding a **minimum cost path** (not necessarily unique)?
- We will start with **backtracking search**, the simplest algorithm which just tries all paths. The algorithm is called recursively on the current state  $s$  and the path leading up to that state. If we have reached a goal, then we can update the minimum cost path with the current path. Otherwise, we consider all possible actions  $a$  from state  $s$ , and recursively search each of the possibilities.
- Graphically, backtracking search performs a depth-first traversal of the search tree. What is the time and memory complexity of this algorithm?
- To get a simple characterization, assume that the search tree has maximum depth  $D$  (each path consists of  $D$  actions/edges) and that there are  $b$  available actions per state (the **branching factor** is  $b$ ).
- It is easy to see that backtracking search only requires  $O(D)$  memory (to maintain the stack for the recurrence), which is as good as it gets.
- However, the running time is proportional to the number of nodes in the tree, since the algorithm needs to check each of them. The number of nodes is  $1 + b + b^2 + \dots + b^D = \frac{b^{D+1} - 1}{b - 1} = O(b^D)$ . Note that the total number of nodes in the search tree is on the same order as the number of leaves, so the cost is always dominated by the last level.
- In general, there might not be a finite upper bound on the depth of a search tree. In this case, there are two options: (i) we can simply cap the maximum depth and give up after a certain point or (ii) we can disallow visits to the same state.
- It is worth mentioning that the greedy algorithm that repeatedly chooses the lowest action myopically won't work. Can you come up with an example?

# Backtracking search

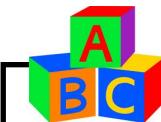


## Algorithm: backtracking search

```
def backtrackingSearch( $s$ , path):
    If IsEnd( $s$ ): update minimum cost path
    For each action  $a \in \text{Actions}(s)$ :
        Extend path with  $\text{Succ}(s, a)$  and  $\text{Cost}(s, a)$ 
        Call backtrackingSearch( $\text{Succ}(s, a)$ , path)
    Return minimum cost path
```

[semi-live solution: `backtrackingSearch`]

# Depth-first search



**Assumption: zero action costs**

Assume action costs  $\text{Cost}(s, a) = 0$ .

Idea: Backtracking search + stop when find the first end state.

If  $b$  actions per state, maximum depth is  $D$  actions:

- Space: still  $O(D)$
- Time: still  $O(b^D)$  worst case, but could be much better if solutions are easy to find

- Backtracking search will always work (i.e., find a minimum cost path), but there are cases where we can do it faster. But in order to do that, we need some additional assumptions — there is no free lunch.
- Suppose we make the assumption that all the action costs are zero. In other words, all we care about is finding a valid action sequence that reaches the goal. Any such sequence will have the minimum cost: zero.
- In this case, we can just modify backtracking search to not keep track of costs and then stop searching as soon as we reach a goal. The resulting algorithm is **depth-first search** (DFS), which should be familiar to you. The worst time and space complexity are of the same order as backtracking search. In particular, if there is no path to an end state, then we have to search the entire tree.
- However, if there are many ways to reach the end state, then we can stop much earlier without exhausting the search tree. So DFS is great when there are an abundance of solutions.

# Breadth-first search



**Assumption: constant action costs**

Assume action costs  $\text{Cost}(s, a) = c$  for some  $c \geq 0$ .

Idea: explore all nodes in order of increasing depth.

Legend:  $b$  actions per state, solution has  $d$  actions

- Space: now  $O(b^d)$  (a lot worse!)
- Time:  $O(b^d)$  (better, depends on  $d$ , not  $D$ )

- **Breadth-first search** (BFS), which should also be familiar, makes a less stringent assumption, that all the action costs are the same non-negative number. This effectively means that all the paths of a given length have the same cost.
- BFS maintains a queue of states to be explored. It pops a state off the queue, then pushes its successors back on the queue.
- BFS will search all the paths consisting of one edge, two edges, three edges, etc., until it finds a path that reaches a end state. So if the solution has  $d$  actions, then we only need to explore  $O(b^d)$  nodes, thus taking that much time.
- However, a potential show-stopper is that BFS also requires  $O(b^d)$  space since the queue must contain all the nodes of a given level of the search tree. Can we do better?

# DFS with iterative deepening



**Assumption: constant action costs**

Assume action costs  $\text{Cost}(s, a) = c$  for some  $c \geq 0$ .

Idea:

- Modify DFS to stop at a maximum depth.
- Call DFS for maximum depths 1, 2, ... .

DFS on  $d$  asks: is there a solution with  $d$  actions?

Legend:  $b$  actions per state, solution size  $d$

- Space:  $O(d)$  (saved!)
- Time:  $O(b^d)$  (same as BFS)

- Yes, we can do better with a trick called **iterative deepening**. The idea is to modify DFS to make it stop after reaching a certain depth. Therefore, we can invoke this modified DFS to find whether a valid path exists with at most  $d$  edges, which as discussed earlier takes  $O(d)$  space and  $O(b^d)$  time.
- Now the trick is simply to invoke this modified DFS with cutoff depths of  $1, 2, 3, \dots$  until we find a solution or give up. This algorithm is called DFS with iterative deepening (DFS-ID). In this manner, we are guaranteed optimality when all action costs are equal (like BFS), but we enjoy the parsimonious space requirements of DFS.
- One might worry that we are doing a lot of work, searching some nodes many times. However, keep in mind that both the number of leaves and the number of nodes in a search tree is  $O(b^d)$  so asymptotically DFS with iterative deepening is the same time complexity as BFS.



# Tree search algorithms

Legend:  $b$  actions/state, solution depth  $d$ , maximum depth  $D$

Algorithm	Action costs	Space	Time
Backtracking	any	$O(D)$	$O(b^D)$
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant $\geq 0$	$O(b^d)$	$O(b^d)$
DFS-ID	constant $\geq 0$	$O(d)$	$O(b^d)$

- Always exponential time
- Avoid exponential space with DFS-ID

- Here is a summary of all the tree search algorithms, the assumptions on the action costs, and the space and time complexities.
- The take-away is that we can't avoid the exponential time complexity, but we can certainly have linear space complexity. Space is in some sense the more critical dimension in search problems. Memory cannot magically grow, whereas time "grows" just by running an algorithm for a longer period of time, or even by parallelizing it across multiple machines (e.g., where each processor gets its own subtree to search).



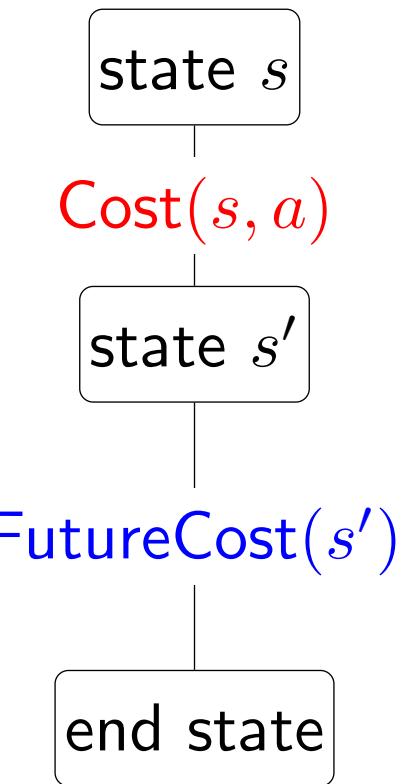
# Roadmap

Tree search

**Dynamic programming**

Uniform cost search

# Dynamic programming



Minimum cost path from state  $s$  to a end state:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

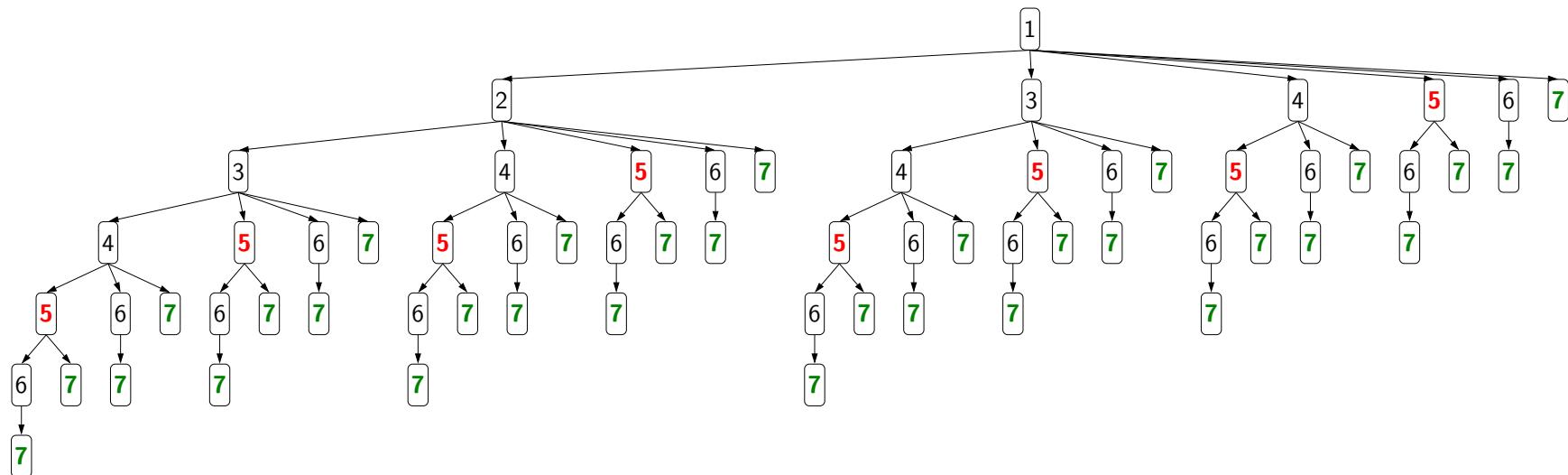
- Now let's see if we can avoid the exponential running time of tree search. Our first algorithm will be dynamic programming. We have already seen dynamic programming in specific contexts. Now we will use the search problem abstraction to define a single dynamic program for all search problems.
- First, let us try to think about the minimum cost path in the search tree recursively. Define  $\text{FutureCost}(s)$  as the cost of the minimum cost path from  $s$  to some end state. The minimum cost path starting with a state  $s$  to an end state must take a first action  $a$ , which results in another state  $s'$ , from which we better take a minimum cost path to the end state.
- Written in symbols, we have a nice recurrence. Throughout this course, we will see many recurrences of this form. The basic form is a base case (when  $s$  is an end state) and an inductive case, which consists of taking the minimum over all possible actions  $a$  from  $s$ , taking an initial step resulting in an **immediate** action cost  $\text{Cost}(s, a)$  and a **future** cost.

# Motivating task



## Example: route finding

Find the minimum cost path from city 1 to city  $n$ , only moving forward. It costs  $c_{ij}$  to go from  $i$  to  $j$ .

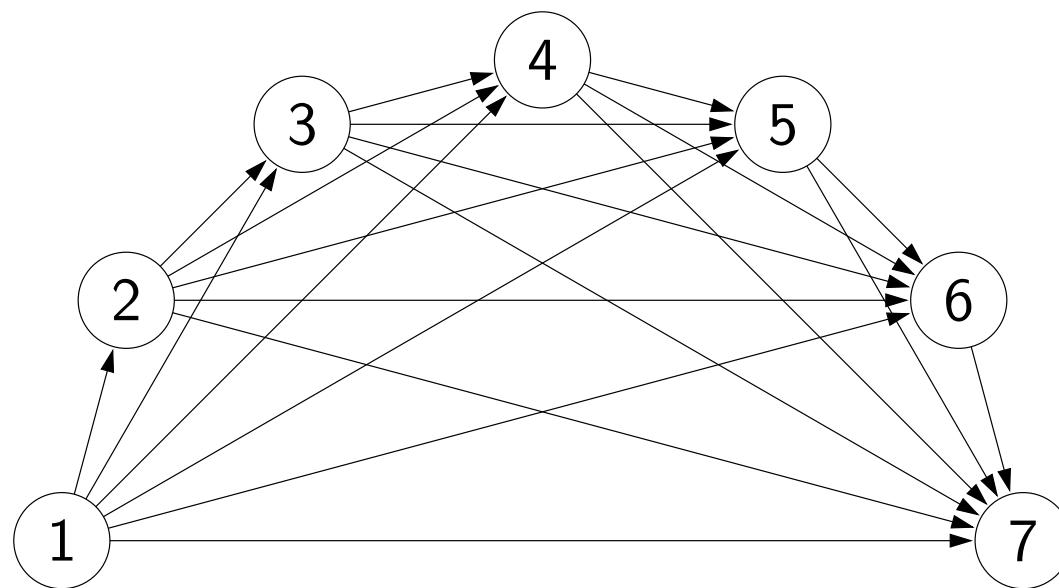


**Observation:** future costs only depend on current city

- Now let us see if we can avoid the exponential time. If we consider the simple route finding problem of traveling from city 1 to city  $n$ , the search tree grows exponentially with  $n$ .
- However, upon closer inspection, we note that this search tree has a lot of repeated structures. Moreover (and this is important), the future costs (the minimum cost of reaching a end state) of a state only depends on the current city! So therefore, all the subtrees rooted at city 5, for example, have the same minimum cost!
- If we can just do that computation once, then we will have saved big time. This is the central idea of **dynamic programming**.
- We've already reviewed dynamic programming in the first lecture. The purpose here is to construct one generic dynamic programming solution that will work on any search problem. Again, this highlights the useful division between modeling (defining the search problem) and algorithms (performing the actual search).

# Dynamic programming

**State:** ~~past sequence of actions~~ current city



**Exponential saving in time and space!**

- Let us collapse all the nodes that have the same city into one. We no longer have a tree, but a directed acyclic graph with only  $n$  nodes rather than exponential in  $n$  nodes.
- Note that dynamic programming is only useful if we can define a search problem where the number of states is small enough to fit in memory.

# Dynamic programming



## Algorithm: dynamic programming

```
def DynamicProgramming( $s$ ):  
    If already computed for  $s$ , return cached answer.  
    If IsEnd( $s$ ): return solution  
    For each action  $a \in \text{Actions}(s)$ : ...
```

[semi-live solution: Dynamic Programming]



## Assumption: acyclicity

The state graph defined by  $\text{Actions}(s)$  and  $\text{Succ}(s, a)$  is acyclic.

- The dynamic programming algorithm is exactly backtracking search with one twist. At the beginning of the function, we check to see if we've already computed the future cost for  $s$ . If we have, then we simply return it (which takes constant time if we use a hash map). Otherwise, we compute it and save it in the cache so we don't have to recompute it again. In this way, for every state, we are only computing its value once.
- For this particular example, the running time is  $O(n^2)$ , the number of edges.
- One important point is that the graph must be acyclic for dynamic programming to work. If there are cycles, the computation of a future cost for  $s$  might depend on  $s'$  which might depend on  $s$ . We will infinite loop in this case. To deal with cycles, we need uniform cost search, which we will describe later.

# Dynamic programming



## Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

past actions (all cities)      1 3 4 6

state (current city)            1 3 4 6

- So far, we have only considered the example where the cost only depends on the current city. But let's try to capture exactly what's going on more generally.
- This is perhaps the most important idea of this lecture: **state**. A state is a summary of all the past actions sufficient to choose future actions optimally.
- What state is really about is forgetting the past. We can't forget everything because the action costs in the future might depend on what we did on the past. The more we forget, the fewer states we have, and the more efficient our algorithm. So the name of the game is to find the minimal set of states that suffice. It's a fun game.

# Handling additional constraints

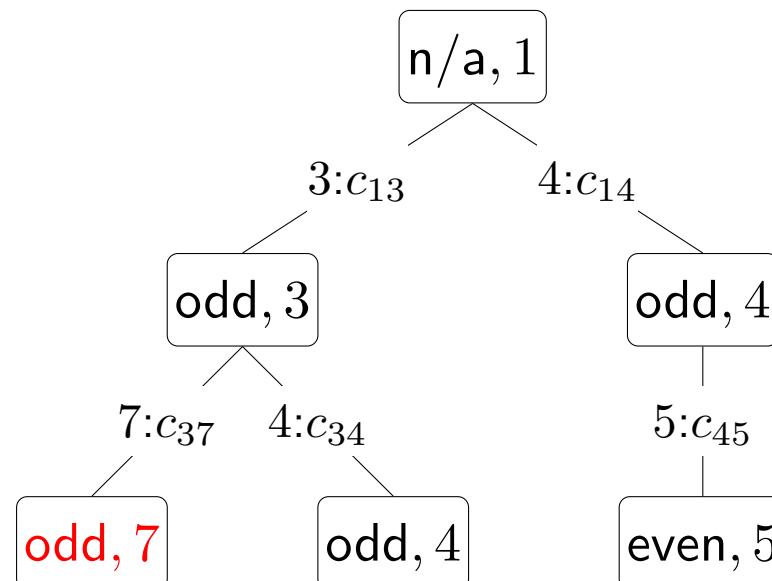


## Example: route finding

Find the minimum cost path from city 1 to city  $n$ , only moving forward. It costs  $c_{ij}$  to go from  $i$  to  $j$ .

**Constraint: Can't visit three odd cities in a row.**

**State:** (whether previous city was odd, current city)



- Let's add a constraint that says we can't visit three odd cities in a row. If we only keep track of the current city, and we try to move to a next city, we cannot enforce this constraint because we don't know what the previous city was. So let's add the previous city into the state.
- This will work, but we can actually make the state smaller. We only need to keep track of whether the previous city was an odd numbered city to enforce this constraint.
- Note that in doing so, we have  $2n$  states rather than  $n^2$  states, which is a substantial savings. So the lesson is to pay attention to what information you actually need in the state.

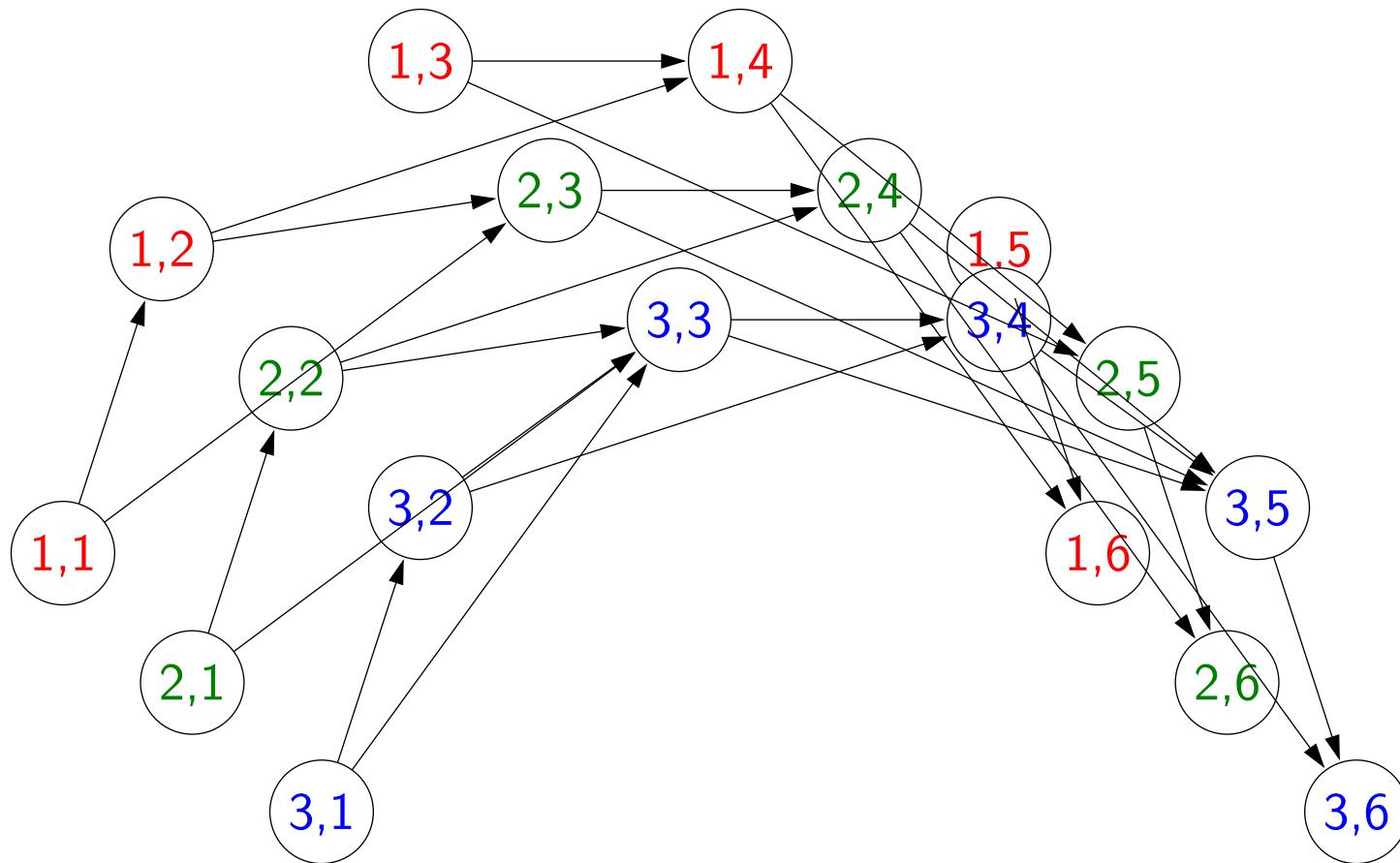


# Question

Objective: travel from city 1 to city  $n$ , visiting at least 3 odd cities.  
What is the minimal state?

# State graph

State:  $(\min(\text{number of odd cities visited}, 3), \text{current city})$



- Our first thought might be to remember how many odd cities we have visited so far (and the current city).
- But if we're more clever, we can notice that once the number of odd cities is 3, we don't need to keep track of whether that number goes up to 4 or 5, etc. So the state we actually need to keep is  $(\min(\text{number of odd cities visited}, 3), \text{current city})$ . Thus, our state space is  $O(n)$  rather than  $O(n^2)$ .
- We can visualize what augmenting the state does to the state graph. Effectively, we are copying each node 4 times, and the edges are redirected to move between these copies.
- Note that some states such as  $(2, 1)$  aren't reachable (if you're in city 1, it's impossible to have visited 2 odd cities already); the algorithm will not touch those states and that's perfectly okay.



# Question

Objective: travel from city 1 to city  $n$ , visiting more odd than even cities.  
What is the minimal state?

- An initial guess might be to keep track of the number of even cities and the number of odd cities visited.
- But we can do better. We have to just keep track of the number of odd cities minus the number of even cities and the current city. We can write this more formally as  $(n_1 - n_2, \text{current city})$ , where  $n_1$  is the number of odd cities visited so far and  $n_2$  is the number of even cities visited so far.



# Summary

- **State:** summary of past actions sufficient to choose future actions optimally
- **Dynamic programming:** backtracking search with **memoization**
  - potentially exponential savings

Dynamic programming only works for acyclic graphs...what if there are cycles?



# Roadmap

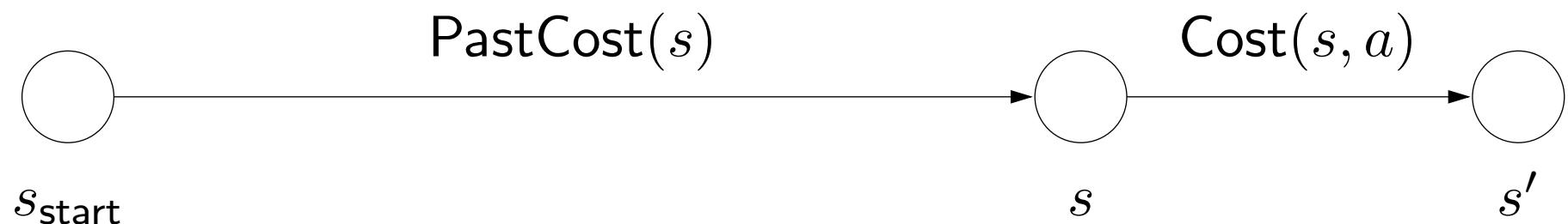
Tree search

Dynamic programming

**Uniform cost search**

# Ordering the states

Observation: prefixes of optimal path are optimal



Key: if graph is acyclic, dynamic programming makes sure we compute  $\text{PastCost}(s)$  before  $\text{PastCost}(s')$

If graph is cyclic, then we need another mechanism to order states...

- Recall that we used dynamic programming to compute the future cost of each state  $s$ , the cost of the minimum cost path from  $s$  to a end state.
- We can analogously define  $\text{PastCost}(s)$ , the cost of the minimum cost path from the start state to  $s$ . If instead of having access to the successors via  $\text{Succ}(s, a)$ , we had access to predecessors (think of reversing the edges in the state graph), then we could define a dynamic program to compute all the  $\text{PastCost}(s)$ .
- Dynamic programming relies on the absence of cycles, so that there is always a clear order in which to compute all the past costs. If the past costs of all the predecessors of a state  $s$  are computed, then we could compute the past cost of  $s$  by taking the minimum.
- Note that  $\text{PastCost}(s)$  will always be computed before  $\text{PastCost}(s')$  if there is an edge from  $s$  to  $s'$ . In essence, the past costs will be computed according to a topological ordering of the nodes.
- However, when there are cycles, no topological ordering exists, so we need another way to order the states.

# Uniform cost search (UCS)



**Key idea: state ordering**

UCS enumerates states in order of increasing past cost.



**Assumption: non-negativity**

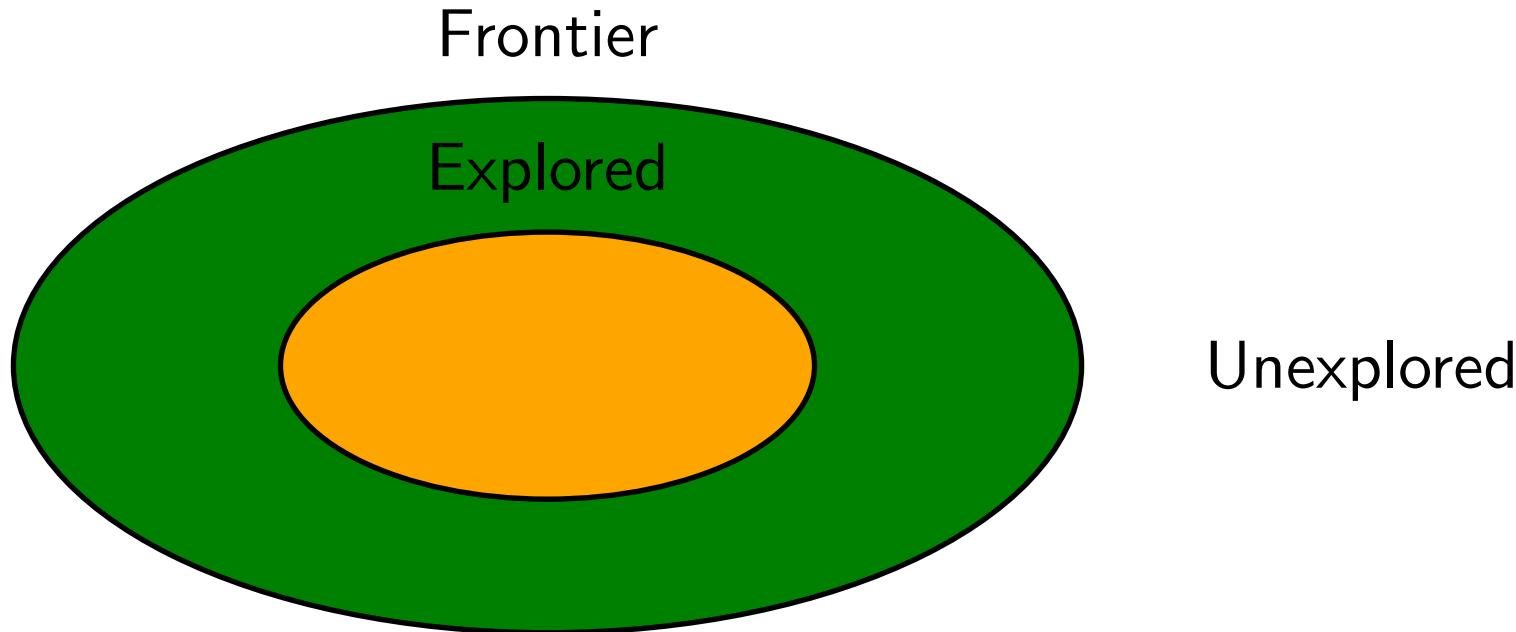
All action costs are non-negative:  $\text{Cost}(s, a) \geq 0$ .

UCS in action:



- The key idea that uniform cost search (UCS) uses is to compute the past costs in order of increasing past cost. To make this efficient, we need to make an important assumption that all action costs are non-negative.
- This assumption is reasonable in many cases, but doesn't allow us to handle cases where actions have payoff. To handle negative costs (positive payoffs), we need the Bellman-Ford algorithm. When we talk about value iteration for MDPs, we will see a form of this algorithm.
- Note: those of you who have studied algorithms should immediately recognize UCS as Dijkstra's algorithm. Logically, the two are indeed equivalent. There is an important implementation difference: UCS takes as input a **search problem**, which implicitly defines a large and even infinite graph, whereas Dijkstra's algorithm (in the typical exposition) takes as input a fully concrete graph. The implicitness is important in practice because we might be working with an enormous graph (a detailed map of world) but only need to find the path between two close by points (Stanford to Palo Alto).
- Another difference is that Dijkstra's algorithm is usually thought of as finding the shortest path from the start state to every other node, whereas UCS is explicitly about finding the shortest path to an end state. This difference is sharpened when we look at the A\* algorithm next time, where knowing that we're trying to get to the goal can yield a much faster algorithm. The name uniform cost search refers to the fact that we are exploring states of the same past cost uniformly (the video makes this visually clear); in contrast, A\* will explore states which are biased towards the end state.

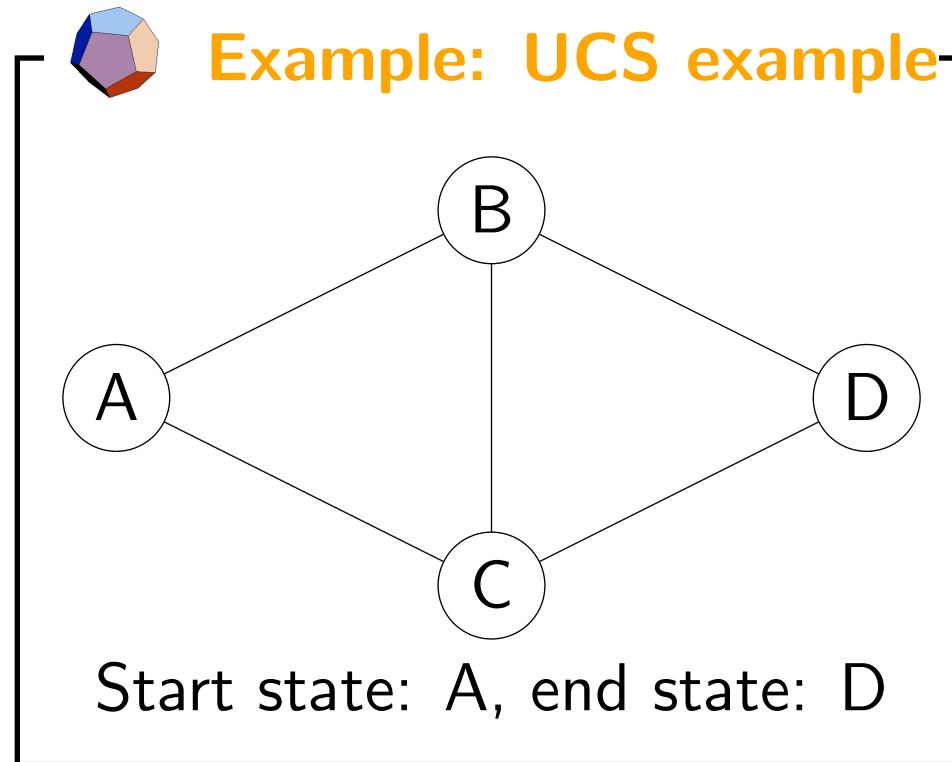
# High-level strategy



- **Explored:** states we've found the optimal path to
- **Frontier:** states we've seen, still figuring out how to get there cheaply
- **Unexplored:** states we haven't seen

- The general strategy of UCS is to maintain three sets of nodes: explored, frontier, and unexplored. Throughout the course of the algorithm, we will move states from unexplored to frontier, and from frontier to explored.
- The key invariant is that we have computed the minimum cost paths to all the nodes in the explored set. So when the end state moves into the explored set, then we are done.

# Uniform cost search example



[whiteboard]

Minimum cost path:

$A \rightarrow B \rightarrow C \rightarrow D$  with cost 3

- Before we present the full algorithm, let's walk through a concrete example.
- Initially, we put A on the frontier. We then take A off the frontier and mark it as explored. We add B and C to the frontier with past costs 1 and 100, respectively.
- Next, we remove from the frontier the state with the minimum past cost (priority), which is B. We mark B as explored and consider successors A, C, D. We ignore A since it's already explored. The past cost of C gets updated from 100 to 2. We add D to the frontier with initial past cost 101.
- Next, we remove C from the frontier; its successors are A, B, D. A and B are already explored, so we only update D's past cost from 101 to 3.
- Finally, we pop D off the frontier, find that it's a end state, and terminate the search.

# Uniform cost search (UCS)



## Algorithm: uniform cost search [Dijkstra, 1956]

Add  $s_{\text{start}}$  to **frontier** (priority queue)

Repeat until frontier is empty:

    Remove  $s$  with smallest priority  $p$  from frontier

    If  $\text{IsEnd}(s)$ : return solution

    Add  $s$  to **explored**

    For each action  $a \in \text{Actions}(s)$ :

        Get successor  $s' \leftarrow \text{Succ}(s, a)$

        If  $s'$  already in explored: continue

        Update **frontier** with  $s'$  and priority  $p + \text{Cost}(s, a)$

[semi-live solution: Uniform Cost Search]

- Implementation note: we use `util.PriorityQueue` which supports `removeMin` and `update`. Note that `frontier.update(state, pastCost)` returns whether `pastCost` improves the existing estimate of the past cost of state.

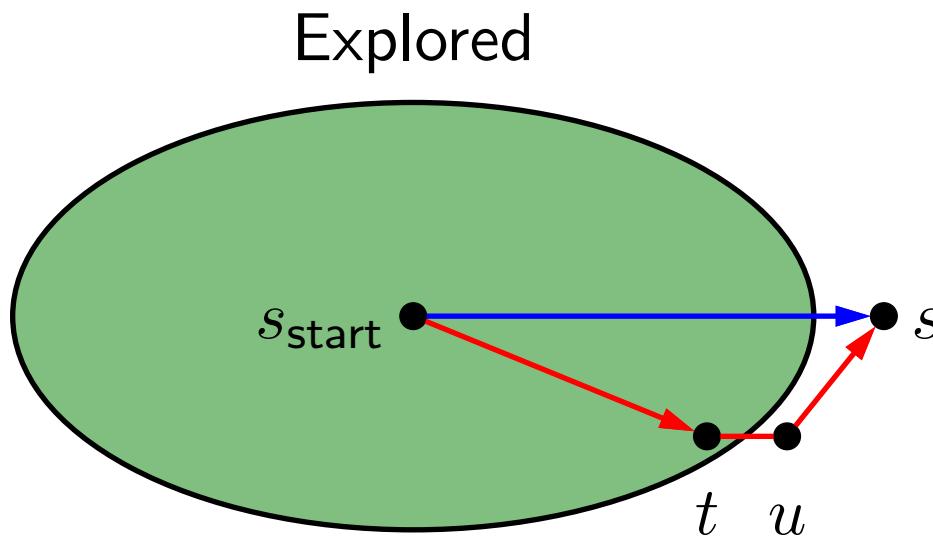
# Analysis of uniform cost search



## Theorem: correctness

When a state  $s$  is popped from the frontier and moved to explored, its priority is  $\text{PastCost}(s)$ , the minimum cost to  $s$ .

Proof:



- Let  $p_s$  be the priority of  $s$  when  $s$  is popped off the frontier. Since all costs are non-negative,  $p_s$  increases over the course of the algorithm.
- Suppose we pop  $s$  off the frontier. Let the blue path denote the path with cost  $p_s$ .
- Consider any alternative red path from the start state to  $s$ . The red path must leave the explored region at some point; let  $t$  and  $u = \text{Succ}(t, a)$  be the first pair of states straddling the boundary. We want to show that the red path cannot be cheaper than the blue path via a string of inequalities.
- First, by definition of  $\text{PastCost}(t)$  and non-negativity of edge costs, the cost of the red path is at least the cost of the part leading to  $u$ , which is  $\text{PastCost}(t) + \text{Cost}(t, a) = p_t + \text{Cost}(t, a)$ , where the last equality is by the inductive hypothesis.
- Second, we have  $p_t + \text{Cost}(t, a) \geq p_u$  since we updated the frontier based on  $(t, a)$ .
- Third, we have that  $p_u \geq p_s$  because  $s$  was the minimum cost state on the frontier.
- Note that  $p_s$  is the cost of the blue path.

# DP versus UCS

$N$  total states,  $n$  of which are closer than end state

<b>Algorithm</b>	<b>Cycles?</b>	<b>Action costs</b>	<b>Time/space</b>
DP	no	any	$O(N)$
UCS	yes	$\geq 0$	$O(n \log n)$

**Note:** UCS potentially explores fewer states, but requires more overhead to maintain the priority queue

**Note:** assume number of actions per state is constant (independent of  $n$  and  $N$ )

- DP and UCS have complementary strengths and weaknesses; neither dominates the other.
- DP can handle negative action costs, but is restricted to acyclic graphs. It also explores all  $N$  reachable states from  $s_{\text{start}}$ , which is inefficient. This is unavoidable due to negative action costs.
- UCS can handle cyclic graphs, but is restricted to non-negative action costs. An advantage is that it only needs to explore  $n$  states, where  $n$  is the number of states which are cheaper to get to than any end state. However, there is an overhead with maintaining the priority queue.
- One might find it unsatisfying that UCS can only deal with non-negative action costs. Can we just add a large positive constant to each action cost to make them all non-negative? It turns out this doesn't work because it penalizes longer paths more than shorter paths, so we would end up solving a different problem.



# Summary

- Tree search: memory efficient, suitable for huge state spaces but exponential worst-case running time
- State: summary of past actions sufficient to choose future actions optimally
- Graph search: dynamic programming and uniform cost search construct optimal paths (exponential savings!)
- Next time: learning action costs, searching faster with A\*

- We started out with the idea of a search problem, an abstraction that provides a clean interface between modeling and algorithms.
- Tree search algorithms are the simplest: just try exploring all possible states and actions. With backtracking search and DFS with iterative deepening, we can scale up to huge state spaces since the memory usage only depends on the number of actions in the solution path. Of course, these algorithms necessarily take exponential time in the worst case.
- To do better, we need to think more about bookkeeping. The most important concept from this lecture is the idea of a **state**, which contains all the information about the past to act optimally in the future. We saw several examples of traveling between cities under various constraints, where coming up with the proper minimal state required a deep understanding of search.
- With an appropriately defined state, we can apply either dynamic programming or UCS, which have complementary strengths. The former handles negative action costs and the latter handles cycles. Both require space proportional to the number of states, so we need to make sure that we did a good job with the modeling of the state.



# Lecture 6: Search II





# Question

Suppose we want to travel from city 1 to city  $n$  (going only forward) and back to city 1 (only going backward). It costs  $c_{ij} \geq 0$  to go from  $i$  to  $j$ . Which of the following algorithms can be used to find the minimum cost path (select all that apply)?

depth-first search

breadth-first search

dynamic programming

uniform cost search

- Let's first start by figuring out what the search problem actually is. Any action sequence needs to satisfy the constraint that we move forward to  $n$  and then move backwards to 1. So we need to keep track of the current city  $i$  as well as the direction in the state.
- We can write down the details, but all that matters for this question is that the graph is acyclic (note that the graph implied by  $c_{ij}$  over cities is not acyclic, but keeping track of directionality makes it acyclic). Also, all edge costs are non-negative.
- Now, let's think about which algorithms will work. Recall the various assumptions of the algorithms. DFS won't work because it assumes all edge costs are zero. BFS also won't work because it assumes all edge costs are the same. Dynamic programming will work because the graph is acyclic. Uniform cost search will also work because all the edge costs are non-negative.



## Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

past actions (all cities)      1 3 4 6 5 3

state (current city)            1 3 ← 6 5 3

# Review



## Definition: search problem

- $s_{\text{start}}$ : starting state
- $\text{Actions}(s)$ : possible actions
- $\text{Cost}(s, a)$ : action cost
- $\text{Succ}(s, a)$ : successor
- $\text{IsEnd}(s)$ : reached end state?

**Objective:** find the minimum cost path from  $s_{\text{start}}$  to an  $s$  satisfying  $\text{IsEnd}(s)$ .

# Paradigm

Modeling

Inference

Learning



# Roadmap

**Learning costs**

A\* search

Relaxation



# Search

Transportation example

Start state: 1

Walk action: from  $s$  to  $s + 1$  (cost: 1)

Tram action: from  $s$  to  $2s$  (cost: 2)

End state:  $n$



search algorithm

walk walk tram tram tram walk tram tram

(minimum cost path)

- Recall the magic tram example from the last lecture. Given a search problem (specification of the start state, end test, actions, successors, and costs), we can use a search algorithm (DP or UCS) to yield a solution, which is a sequence of actions of minimum cost reaching an end state from the start state.



# Learning

— Transportation example —

Start state: 1

Walk action: from  $s$  to  $s + 1$  (cost: ?)

Tram action: from  $s$  to  $2s$  (cost: ?)

End state:  $n$

walk walk tram tram tram walk tram tram



learning algorithm

walk cost: 1, tram cost: 2

- Now suppose we don't know what the costs are, but we observe someone getting from 1 to  $n$  via some sequence of walking and tram-taking. Can we figure out what the costs are? This is the goal of learning.

# Learning as an inverse problem

Forward problem (search):

$$\text{Cost}(s, a) \longrightarrow (a_1, \dots, a_k)$$

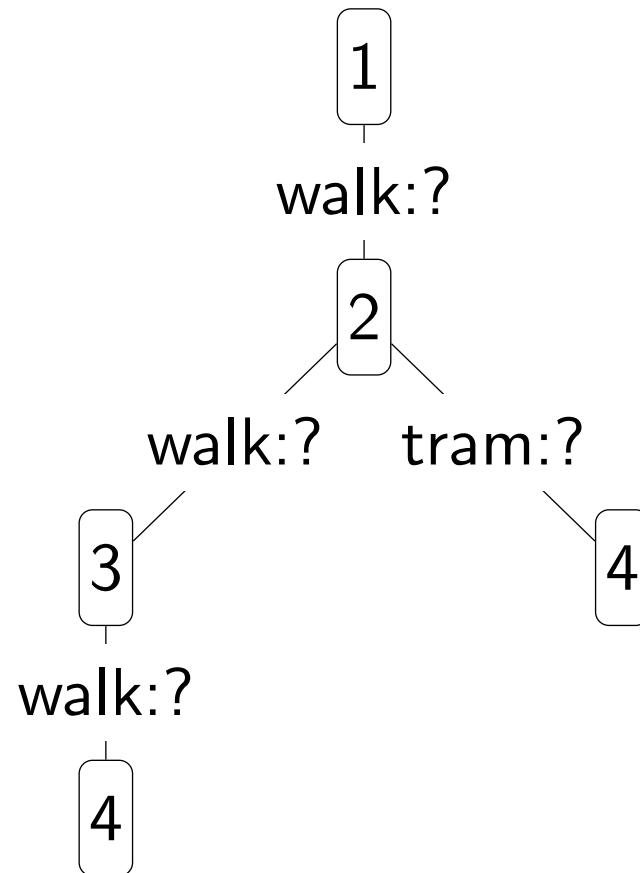
Inverse problem (learning):

$$(a_1, \dots, a_k) \longrightarrow \text{Cost}(s, a)$$

- More generally, so far we have thought about search as a "forward" problem: given costs, finding the optimal sequence of actions.
- Learning concerns the "inverse" problem: given the desired sequence of actions, reverse engineer the costs.

# Prediction (inference) problem

Input  $x$ : search problem without costs



Output  $y$ : solution path

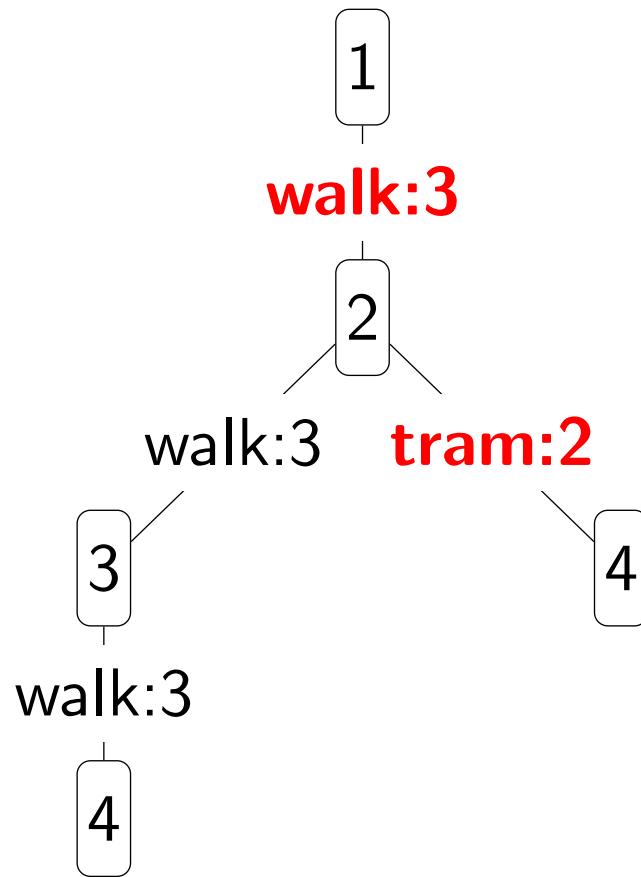
walk walk walk

- Let's cast the problem as predicting an output  $y$  given an input  $x$ . Here, the input  $x$  is the search problem (visualized as a search tree) without the costs provided. The output  $y$  is the desired solution path. The question is what the costs should be set to so that  $y$  is actually the minimum cost path of the resulting search problem.

# Tweaking costs

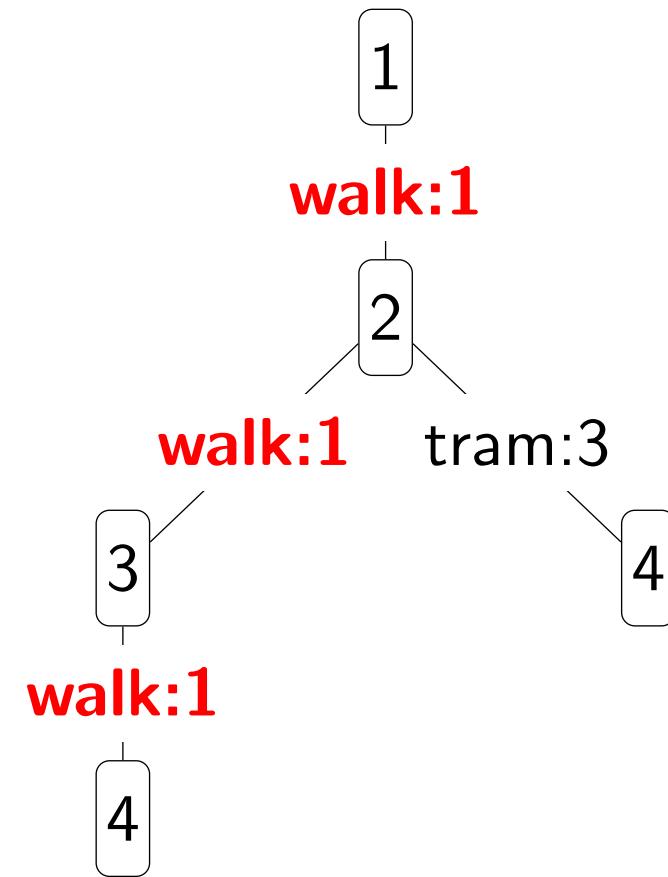
Costs: {walk:3, tram:2}

Minimum cost path:



Costs: {walk:1, tram:3}

Minimum cost path:



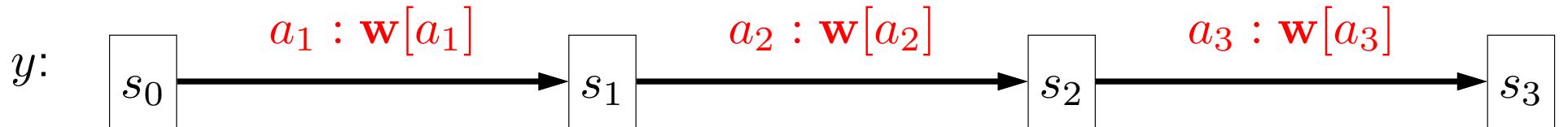
- Suppose the walk cost is 3 and the tram cost is 2. Then, we would obviously predict the [walk, tram] path, which has lower cost.
- But this is not our desired output, because we actually saw the person walk all the way from 1 to 4. How can we update the action costs so that the minimum cost path is walking?
- Intuitively, we want the tram cost to be more and the walk cost to be less. Specifically, let's increase the cost of every action on the predicted path and decrease the cost of every action on the true path. Now, the predicted path coincides with the true observed path. Is this a good strategy in general?

# Modeling costs (simplified)

Assume costs depend only on the action:

$$\text{Cost}(s, a) = \mathbf{w}[a]$$

Candidate output path:



Path cost:

$$\text{Cost}(y) = \mathbf{w}[a_1] + \mathbf{w}[a_2] + \mathbf{w}[a_3]$$

- For each action  $a$ , we define a weight  $w[a]$  representing the cost of action  $a$ . Without loss of generality, let us assume that the cost of the action does not depend on the state  $s$ .
- Then the cost of a path  $y$  is simply the sum of the weights of the actions on the path. Every path has some cost, and recall that the search algorithm will return the minimum cost path.

# Learning algorithm



## Algorithm: Structured Perceptron (simplified)

- For each action:  $\mathbf{w}[a] \leftarrow 0$
- For each iteration  $t = 1, \dots, T$ :
  - For each training example  $(x, y) \in \mathcal{D}_{\text{train}}$ :
    - Compute the minimum cost path  $y'$  given  $\mathbf{w}$
    - For each action  $a \in y$ :  $\mathbf{w}[a] \leftarrow \mathbf{w}[a] - 1$
    - For each action  $a \in y'$ :  $\mathbf{w}[a] \leftarrow \mathbf{w}[a] + 1$
- Try to decrease cost of true  $y$  (from training data)
- Try to increase cost of predicted  $y'$  (from search)

[semi-live solution]

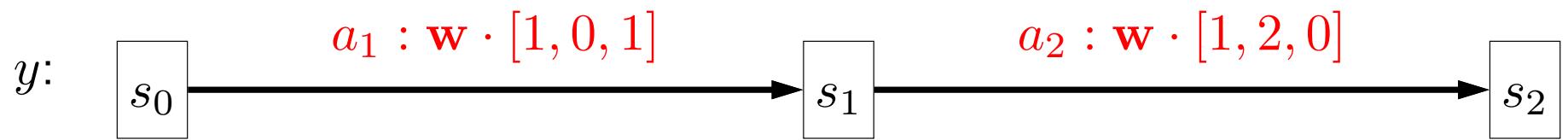
- We are now in position to state the (simplified version of) **structured Perceptron** algorithm.
- Advanced: the Perceptron algorithm performs stochastic gradient descent (SGD) on a modified hinge loss with a constant step size of  $\eta = 1$ . The modified hinge loss is  $\text{Loss}(x, y, \mathbf{w}) = \max\{-(\mathbf{w} \cdot \phi(x))y, 0\}$ , where the margin of 1 has been replaced with a zero. The structured Perceptron is a generalization of the Perceptron algorithm, which is stochastic gradient descent on  $\text{Loss}(x, y, \mathbf{w}) = \max_{y'} \{\sum_{a \in y} \mathbf{w}[a] - \sum_{a \in y'} \mathbf{w}[a]\}$  (note the relationship to the multiclass hinge loss). Even if you don't really understand the loss function, you can still understand the algorithm, since it is very intuitive.
- We iterate over the training examples. Each  $(x, y)$  is a tuple where  $x$  is a search problem without costs and  $y$  is the true minimum-cost path. Given the current weights  $w$  (action costs), we run a search algorithm to find the minimum-cost path  $y'$  according to those weights. Then we update the weights to favor actions that appear in the correct output  $y$  (by reducing their costs) and disfavor actions that appear in the predicted output  $y'$  (by increasing their costs). Note that if we are not making a mistake (that is, if  $y = y'$ ), then there is no update.
- Collins (2002) proved (based on the proof of the original Perceptron algorithm) that if there exists a weight vector that will make zero mistakes on the training data, then the Perceptron algorithm will converge to one of those weight vectors in a finite number of iterations.

# Generalization to features (skip)

Costs are parametrized by feature vector:

$$\text{Cost}(s, a) = \mathbf{w} \cdot \phi(s, a)$$

Example:



$$\mathbf{w} = [3, -1, -1]$$

Path cost:

$$\text{Cost}(y) = 2 + 1 = 3$$

- So far, the cost of an action  $a$  is simply  $\mathbf{w}[a]$ . We can generalize this to allow the cost to be a general dot product  $\mathbf{w} \cdot \phi(s, a)$ , which (i) allows the features to depend on both the state and the action and (ii) allows multiple features per edge. For example, we can have different costs for walking and tram-taking depending on which part of the city we are in.
- We can equivalently write the cost of an entire output  $y$  as  $\mathbf{w} \cdot \phi(y)$ , where  $\phi(y) = \phi(s_0, a_1) + \phi(s_1, a_2)$  is the sum of the feature vectors over all actions.

# Learning algorithm (skip)



## Algorithm: Structured Perceptron [Collins, 2002]

- For each action:  $\mathbf{w} \leftarrow 0$
- For each iteration  $t = 1, \dots, T$ :
  - For each training example  $(x, y) \in \mathcal{D}_{\text{train}}$ :
    - Compute the minimum cost path  $y'$  given  $\mathbf{w}$
    - $\mathbf{w} \leftarrow \mathbf{w} - \phi(y) + \phi(y')$
  - Try to decrease cost of true  $y$  (from training data)
  - Try to increase cost of predicted  $y'$  (from search)

# Applications

- Part-of-speech tagging

*Fruit flies like a banana.*  Noun Noun Verb Det Noun

- Machine translation

*la maison bleue*  *the blue house*

- The structured Perceptron was first used for natural language processing tasks. Given it's simplicity, the Perceptron works reasonably well. With a few minor tweaks, you get state-of-the-art algorithms for structured prediction, which can be applied to many tasks such as machine translation, gene prediction, information extraction, etc.
- On a historical note, the structured Perceptron merges two relatively classic communities. The first is search algorithms (uniform cost search was developed by Dijkstra in 1956). The second is machine learning (Perceptron was developed by Rosenblatt in 1957). It was only over 40 years later that the two met.



# Roadmap

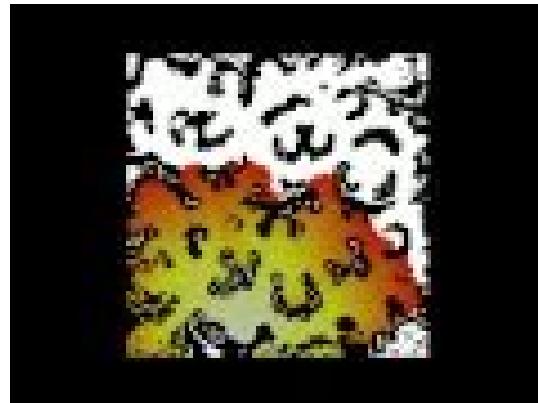
Learning costs

A\* search

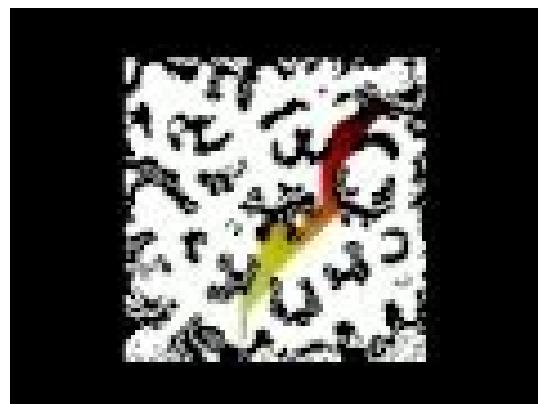
Relaxation

# A\* algorithm

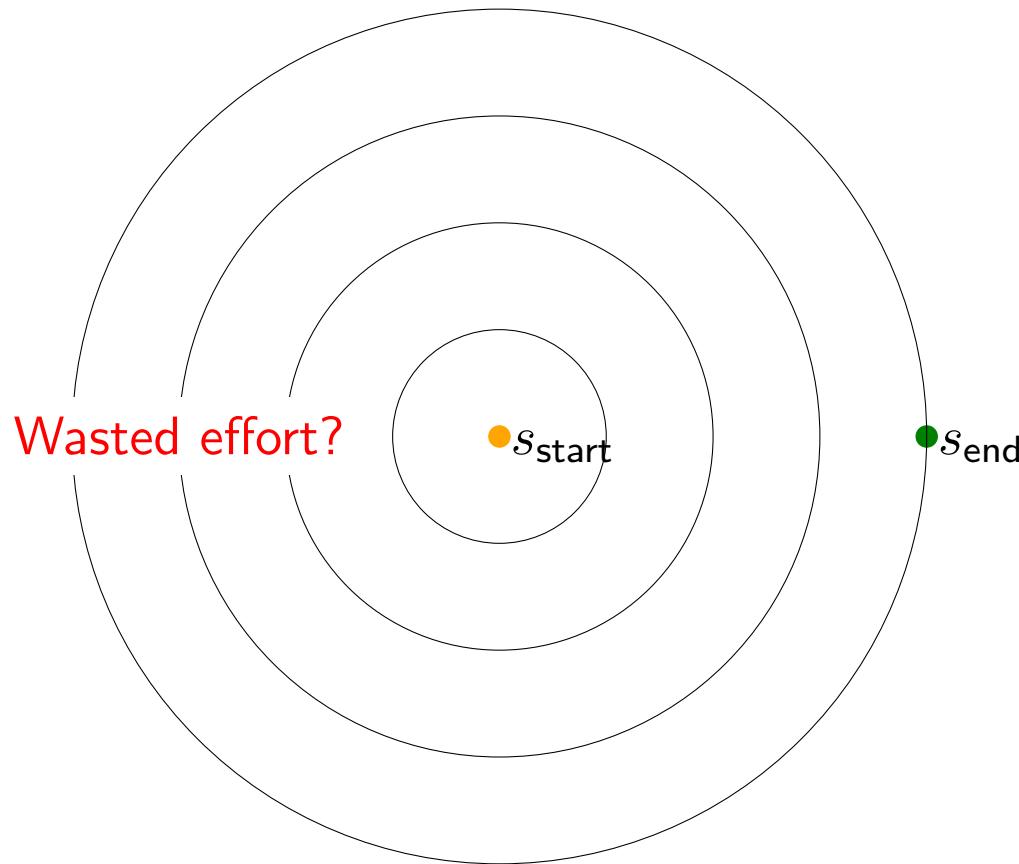
UCS in action:



A\* in action:



# Can uniform cost search be improved?



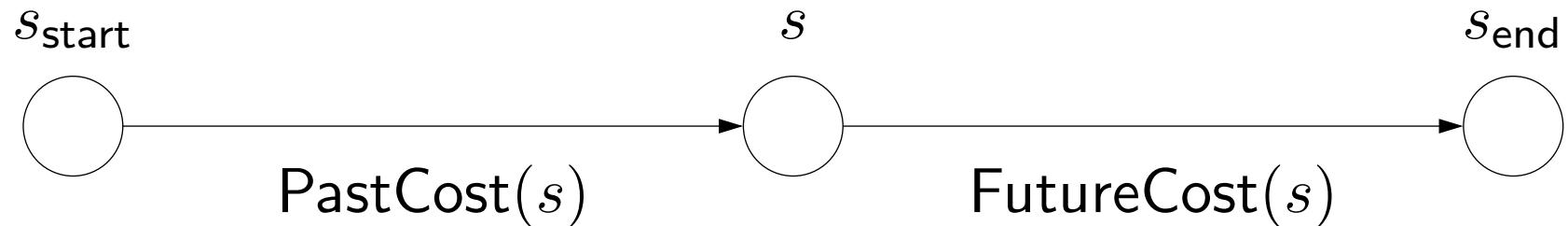
**Problem:** UCS orders states by cost from  $s_{\text{start}}$  to  $s$

**Goal:** take into account cost from  $s$  to  $s_{\text{end}}$

- Now our goal is to make UCS faster. If we look at the UCS algorithm, we see that it explores states based on how far they are away from the start state. As a result, it will explore many states which are close to the start state, but in the opposite direction of the end state.
- Intuitively, we'd like to bias UCS towards exploring states which are closer to the end state, and that's exactly what A\* does.

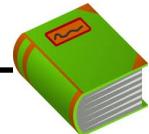
# Exploring states

UCS: explore states in order of  $\text{PastCost}(s)$



Ideal: explore in order of  $\text{PastCost}(s) + \text{FutureCost}(s)$

A\*: explore in order of  $\text{PastCost}(s) + h(s)$



## Definition: Heuristic function

A heuristic  $h(s)$  is any estimate of FutureCost( $s$ ).

- First, some terminology:  $\text{PastCost}(s)$  is the minimum cost from the start state to  $s$ , and  $\text{FutureCost}(s)$  is the minimum cost from  $s$  to an end state. Without loss of generality, we can just assume we have one end state. (If we have multiple ones, create a new official goal state which is the successor of all the original end states.)
- Recall that UCS explores states in order of  $\text{PastCost}(s)$ . It'd be nice if we could explore states in order of  $\text{PastCost}(s) + \text{FutureCost}(s)$ , which would definitely take the end state into account, but computing  $\text{FutureCost}(s)$  would be as expensive as solving the original problem.
- A\* relies on a **heuristic**  $h(s)$ , which is an estimate of  $\text{FutureCost}(s)$ . For A\* to work,  $h(s)$  must satisfy some conditions, but for now, just think of  $h(s)$  as an approximation. We will soon show that A\* will explore states in order of  $\text{PastCost}(s) + h(s)$ . This is nice, because now states which are estimated (by  $h(s)$ ) to be really far away from the end state will be explored later, even if their  $\text{PastCost}(s)$  is small.

# A\* search



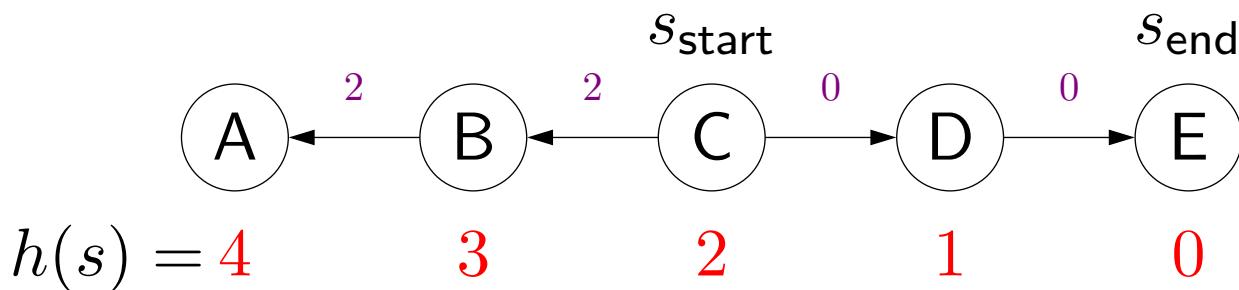
**Algorithm: A\* search [Hart/Nilsson/Raphael, 1968]**

Run uniform cost search with **modified edge costs**:

$$\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s)$$

**Intuition:** add a penalty for how much action  $a$  takes us away from the end state

**Example:**



$$\text{Cost}'(C, B) = \text{Cost}(C, B) + h(B) - h(C) = 1 + (3 - 2) = 2$$

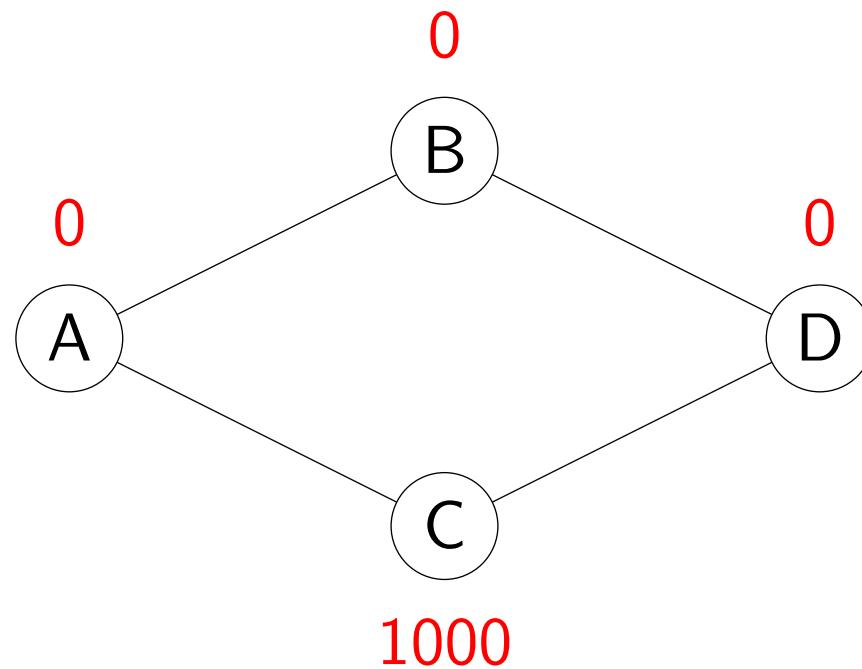
- Here is the full A\* algorithm: just run UCS with modified edge costs.
- You might feel tricked because we promised you a shiny new algorithm, but actually, you just got a refurbished version of UCS. (This is a slightly unorthodox presentation of A\*. The normal presentation is modifying UCS to prioritize by  $\text{PastCost}(s) + h(s)$  rather than  $\text{PastCost}(s)$ .) But I think the modified edge costs view shows a deeper connection to UCS, and we don't even have to modify the UCS code at all.
- How should we think of these modified edge costs? It's the same edge cost  $\text{Cost}(s, a)$  plus an additional term. This term is difference between the estimated future cost of the new state  $\text{Succ}(s, a)$  and that of the current state  $s$ . In other words, we're measuring how much farther from the end state does action  $a$  take us. If this difference is positive, then we're penalizing the action  $a$  more. If this difference is negative, then we're favoring this action  $a$ .
- Let's look at a small example. All edge costs are 1. Let's suppose we define  $h(s)$  to be the actual  $\text{FutureCost}(s)$ , the minimum cost to the end state. In general, this is not the case, but let's see what happens in the best case. The modified edge costs are 2 for actions moving away from the end state and 0 for actions moving towards the end state.
- In this case, UCS with original edge costs 1 will explore all the nodes. However, A\* (UCS with modified edge costs) will explore only the three nodes on the path to the end state.

# An example heuristic

Will any heuristic work?

No.

Counterexample:



Doesn't work because of **negative modified edge costs!**

- So far, we've just said that  $h(s)$  is just an approximation of  $\text{FutureCost}(s)$ . But can it be any approximation?
- The answer is no, as the counterexample clearly shows. The modified edge costs would be 1 (A to B), 1002 (A to C), 5 (B to D), and -999 (C to D). UCS would go to B first and then to D, finding a cost 6 path rather than the optimal cost 3 path through C.
- If our heuristic is lying to us (bad approximation of future costs), then running A\* (UCS on modified costs) could lead to a suboptimal solution. Note that the reason this heuristic doesn't work is the same reason UCS doesn't work when there are negative action costs.

# Consistent heuristics

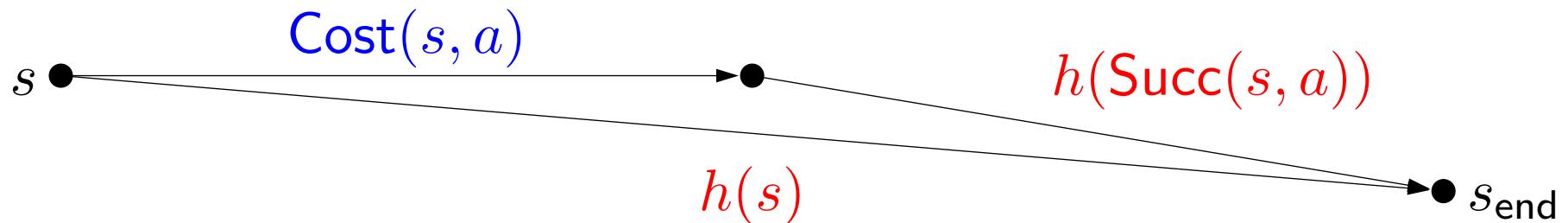


## Definition: consistency

A heuristic  $h$  is **consistent** if

- $\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s) \geq 0$
- $h(s_{\text{end}}) = 0$ .

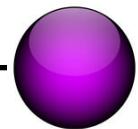
Condition 1: needed for UCS to work (triangle inequality).



Condition 2:  $\text{FutureCost}(s_{\text{end}}) = 0$  so match it.

- We need  $h(s)$  to be **consistent**, which means two things. First, the modified edge costs are non-negative (this is the main property). This is important for UCS to find the minimum cost path (remember that UCS only works when all the edge costs are non-negative).
- Second,  $h(s_{\text{end}}) = 0$ , which is just saying: be reasonable. The minimum cost from the end state to the end state is trivially 0, so just use 0.
- We will come back later to the issue of getting a hold of a consistent heuristic, but for now, let's assume we have one and see what we can do with it.

# Correctness of A\*



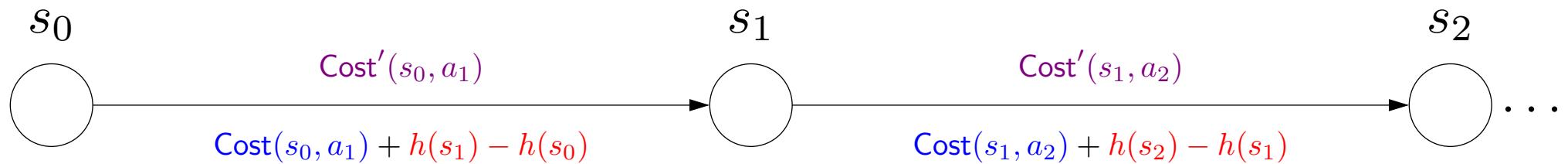
## Proposition: correctness

If  $h$  is consistent, A\* returns the minimum cost path.

- The main theoretical result for A\* is that if we use any consistent heuristic, then we will be guaranteed to find the minimum cost path.

# Proof of A\* correctness

- Consider any path  $[s_0, a_1, s_1, \dots, a_L, s_L]$ :



- Key identity:

$$\underbrace{\sum_{i=1}^L \text{Cost}'(s_{i-1}, a_i)}_{\text{modified path cost}} = \underbrace{\sum_{i=1}^L \text{Cost}(s_{i-1}, a_i)}_{\text{original path cost}} + \underbrace{h(s_L) - h(s_0)}_{\text{constant}}$$

- Therefore, A\* (finding the minimum cost path using modified costs) solves the original problem (even though edge costs are all different!)

- To show the correctness of A\*, let's take any path of length  $L$  from  $s_0 = s_{\text{start}}$  to  $s_L = s_{\text{end}}$ . Let us compute the modified path cost by just adding up the modified edge costs. Just to simplify notation, let  $c_i = \text{Cost}(s_{i-1}, a_i)$  and  $h_i = h(s_i)$ . The modified path cost is  $(c_1 + h_1 - h_0) + (c_2 + h_2 - h_1) + \cdots + (c_L + h_L - h_{L-1})$ . Notice that most of the  $h_i$ 's actually cancel out (this is known as **telescoping sums**).
- We end up with  $\sum_{i=1}^L c_i$ , which is the original path cost plus  $h_L - h_0$ . First, notice that  $h_L = 0$  because  $s_L$  is an end state and by the second condition of consistency,  $h(s_L) = 0$ . Second,  $h_0$  is just a constant (in that it doesn't depend on the path at all), since all paths must start with the start state.
- Therefore, the modified path cost is equal to the original path cost plus a constant. A\*, which is running UCS on the modified edge costs, is equivalent to running UCS on the original edge costs, which minimizes the original path cost.
- This is kind of remarkable: all the edge costs are modified in A\*, but yet the final path cost is the same (up to a constant)!

# Efficiency of A\*



## Theorem: efficiency of A\*

A\* explores all states  $s$  satisfying

$$\text{PastCost}(s) \leq \text{PastCost}(s_{\text{end}}) - h(s)$$

Interpretation: the larger  $h(s)$ , the better

Proof: A\* explores all  $s$  such that

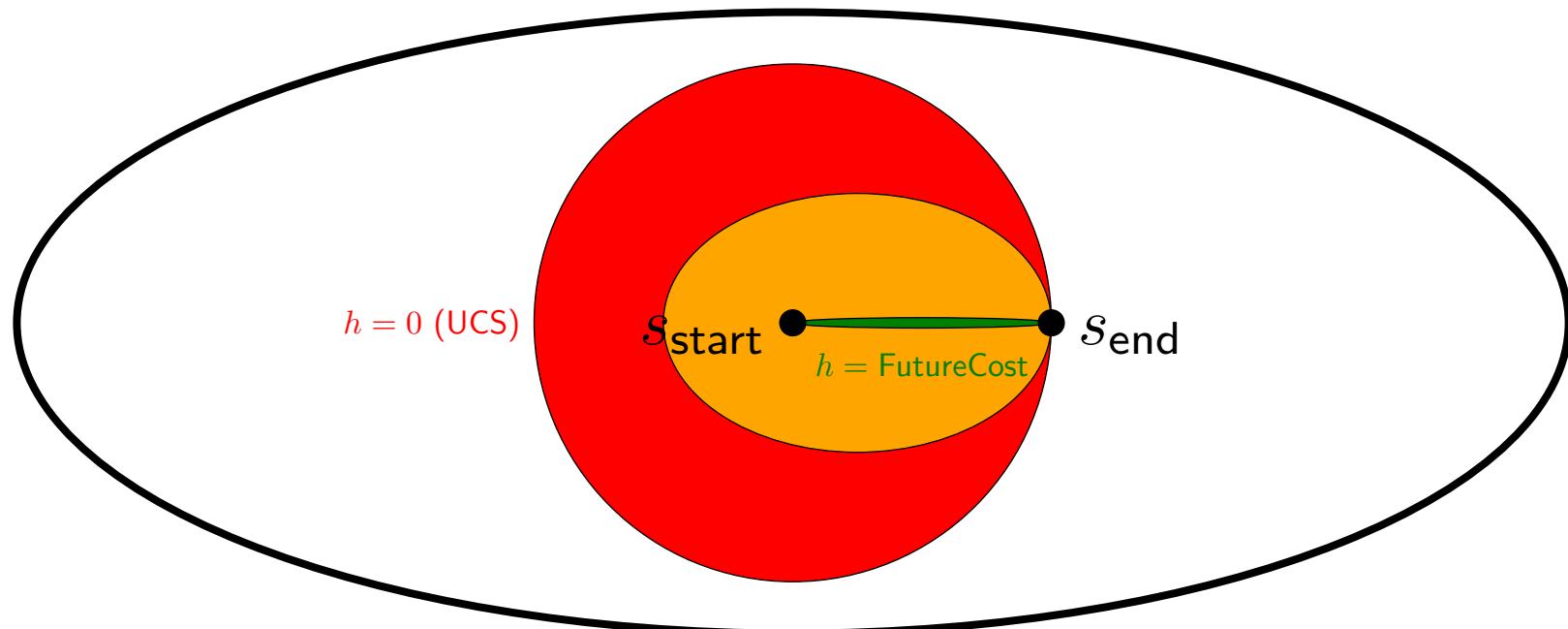
$$\text{PastCost}(s) + h(s)$$

$\leq$

$$\text{PastCost}(s_{\text{end}})$$

- We've proven that A\* is correct (finds the minimum cost path) for any consistent heuristic  $h$ . But for A\* to be interesting, we need to show that it's more efficient than UCS (on the original edge costs). We will measure speed in terms of the number of states which are explored prior to exploring an end state.
- Our second theorem is about the efficiency of A\*: recall that UCS explores states in order of past cost, so that it will explore every state whose past cost is less than the past cost of the end state.
- A\* explores all states for which  $\text{PastCost}'(s) = \text{PastCost}(s) + h(s) - h(s_{\text{start}})$  is less than  $\text{PastCost}'(s_{\text{end}}) = \text{PastCost}(s_{\text{end}}) + h(s_{\text{end}}) - h(s_{\text{start}})$ , or equivalently  $\text{PastCost}(s) + h(s) \leq \text{PastCost}(s_{\text{end}})$  since  $h(s_{\text{end}}) = 0$ .
- From here, it's clear that we want  $h(s)$  to be as large as possible so we can push as many states over the  $\text{PastCost}(s_{\text{end}})$  threshold, so that we don't have to explore them. Of course, we still need  $h$  to be consistent to maintain correctness.
- For example, suppose  $\text{PastCost}(s_1) = 1$  and  $h(s_1) = 1$  and  $\text{PastCost}(s_{\text{end}}) = 2$ . Then we would have to explore  $s_1$  ( $1 + 1 \leq 2$ ). But if we were able to come up with a better heuristic where  $h(s_1) = 2$ , then we wouldn't have to explore  $s_1$  ( $1 + 2 > 2$ ).

# Amount explored



- If  $h(s) = 0$ , then A\* is same as UCS.
- If  $h(s) = \text{FutureCost}(s)$ , then A\* only explores nodes on a minimum cost path.
- Usually  $h(s)$  is somewhere in between.

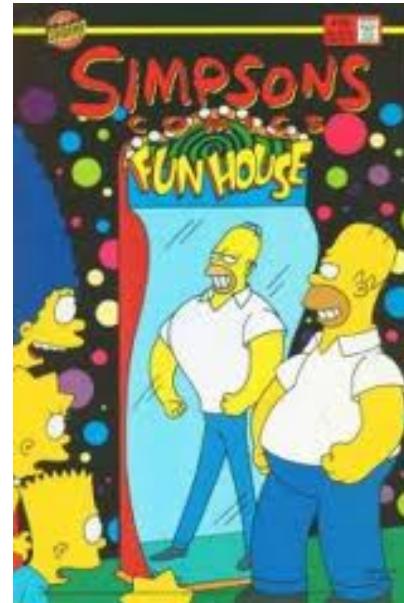
- In this diagram, each ellipse corresponds to the set of states which are explored by A\* with various heuristics. In general, any heuristic we come up with will be between the trivial heuristic  $h(s) = 0$  which corresponds to UCS and the oracle heuristic  $h(s) = \text{FutureCost}(s)$  which is unattainable.

# A\* search



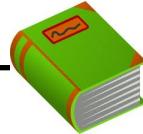
**Key idea: distortion**

A\* distorts edge costs to favor end states.



- What exactly is A\* doing to the edge costs? Intuitively, it's biasing us towards the end state.

# Admissibility



## Definition: admissibility

A heuristic  $h(s)$  is admissible if  
$$h(s) \leq \text{FutureCost}(s)$$

Intuition: admissible heuristics are optimistic



## Theorem: consistency implies admissibility

If a heuristic  $h(s)$  is **consistent**, then  $h(s)$  is **admissible**.

Proof: use induction on  $\text{FutureCost}(s)$

- So far, we've just assumed that  $\text{FutureCost}(s)$  is the best possible heuristic (ignoring for the moment that it's impractical to compute). Let's actually prove this now.
- To do this, we just have to show that any consistent heuristic  $h(s)$  satisfies  $h(s) \leq \text{FutureCost}(s)$  (since by the previous theorem, the larger the heuristic, the better). In fact, this property has a special name: we say that  $h(s)$  is **admissible**. In other words, an admissible heuristic  $h(s)$  **underestimates** the future cost: it is optimistic.
- The proof proceeds by induction on increasing  $\text{FutureCost}(s)$ . In the base case, we have  $0 = h(s_{\text{end}}) \leq \text{FutureCost}(s_{\text{end}}) = 0$  by the second condition of consistency.
- In the inductive case, let  $s$  be a state and let  $a$  be an optimal action leading to  $s' = \text{Succ}(s, a)$  that achieves the minimum cost path to the end state; in other words,  $\text{FutureCost}(s) = \text{Cost}(s, a) + \text{FutureCost}(s')$ . Since  $\text{Cost}(s, a) \geq 0$ , we have that  $\text{FutureCost}(s') \leq \text{FutureCost}(s)$ , so by the inductive hypothesis,  $h(s') \leq \text{FutureCost}(s')$ . To show the same holds for  $s$ , consider:  $h(s) \leq \text{Cost}(s, a) + h(s') \leq \text{Cost}(s, a) + \text{FutureCost}(s') = \text{FutureCost}(s)$ , where the first inequality follows by consistency of  $h(s)$ , the second inequality follows by the inductive hypothesis, and the third equality follows because  $a$  was chosen to be the optimal action. Therefore, we conclude that  $h(s) \leq \text{FutureCost}(s)$ .
- Aside: People often talk about admissible heuristics. Using A\* with an admissible heuristic is only guaranteed to find the minimum cost path for tree search algorithms, where we don't use an explored list. However, the UCS and A\* algorithms we are considering in this class are graph search algorithms, which require consistent heuristics, not just admissible heuristics, to find the minimum cost path. There are some admissible heuristics which are not consistent, but most natural ones are consistent.



# Roadmap

Learning costs

A\* search

**Relaxation**

How do we get good heuristics? Just relax...



# Relaxation

**Intuition:** ideally, use  $h(s) = \text{FutureCost}(s)$ , but that's as hard as solving the original problem.



## Key idea: relaxation

Constraints make life hard. Get rid of them.

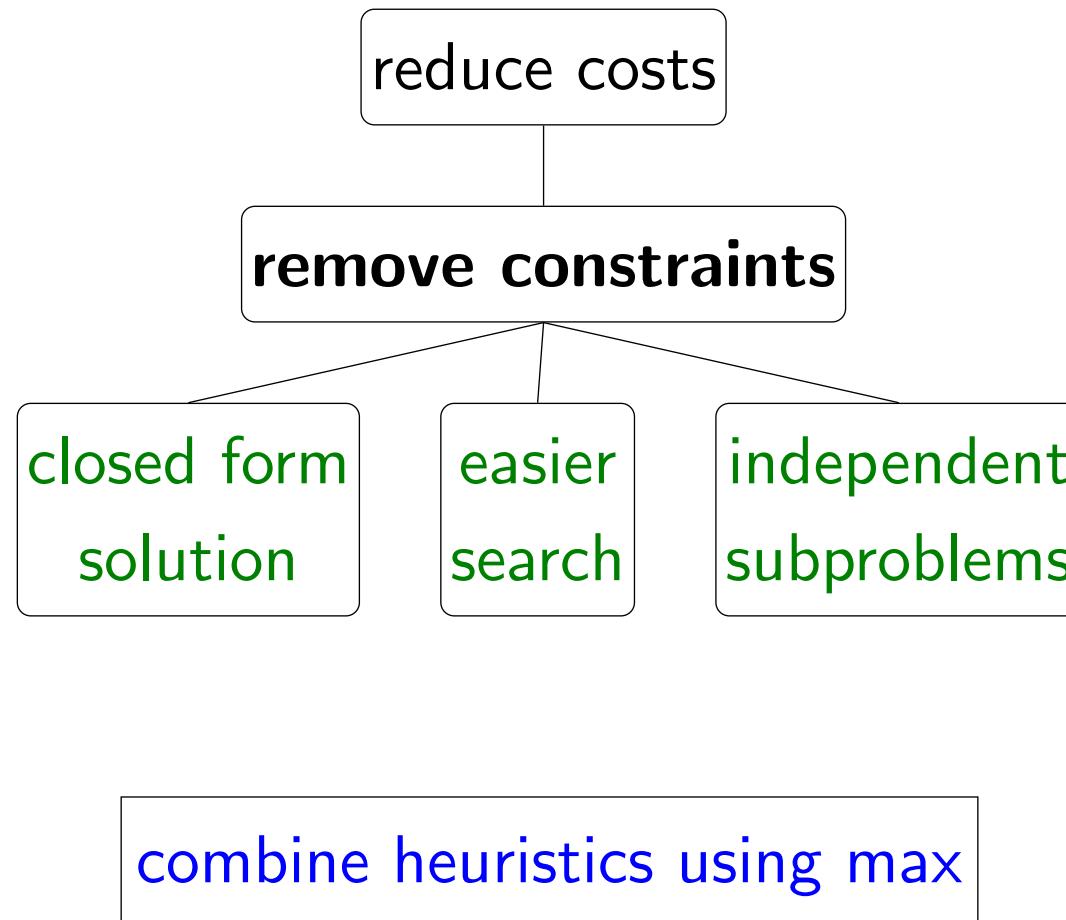
But this is just for the heuristic!



- So far, given a heuristic  $h(s)$ , we can run A\* using it and get a savings which depends on how large  $h(s)$  is. However, we've only seen two heuristics:  $h(s) = 0$  and  $h(s) = \text{FutureCost}(s)$ . The first does nothing (gives you back UCS), and the second is hard to compute.
- What we'd like to do is to come up with a general principle for coming up with heuristics. The idea is that of a **relaxation**: instead of computing  $\text{FutureCost}(s)$  on the original problem, let us compute  $\text{FutureCost}(s)$  on an easier problem, where the notion of easy will be made more formal shortly.
- Note that coming up with good heuristics is about **modeling**, not algorithms. We have to think carefully about our problem domain and see what kind of structure we can exploit in it.



# Relaxation overview

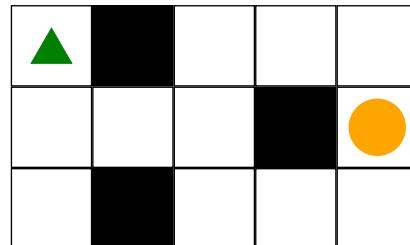


# Closed form solution

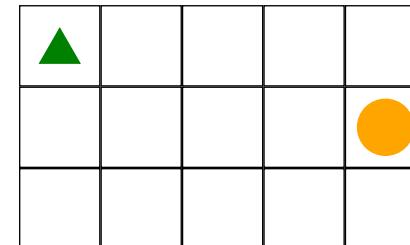


## Example: knock down walls

Goal: move from triangle to circle



Hard



Easy

Heuristic:

$$h(s) = \text{ManhattanDistance}(s, (2, 5))$$

$$\text{e.g., } h((1, 1)) = 5$$

- Here's a simple example. Suppose states are positions  $(r, c)$  on the grid. Possible actions are moving up, down, left, or right, provided they don't move you into a wall or off the grid; and all edge costs are 1. The start state is at the triangle at  $(1, 1)$ , and the end state is the circle at position  $(2, 5)$ .
- With an arbitrary configuration of walls, we can't compute  $\text{FutureCost}(s)$  except by doing search. However, if we just **relaxed** the original problem by removing the walls, then we can compute  $\text{FutureCost}(s)$  in **closed form**: it's just the Manhattan distance between  $s$  and  $s_{\text{end}}$ . Specifically,  $\text{ManhattanDistance}((r_1, c_1), (r_2, c_2)) = |r_1 - r_2| + |c_1 - c_2|$ .



# Easier search



## Example: original problem

Start state: 1

Walk action: from  $s$  to  $s + 1$  (cost: 1)

Tram action: from  $s$  to  $2s$  (cost: 2)

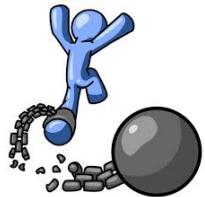
End state:  $n$

**Constraint: can't have more tram actions than walk actions.**

State: (location, **#walk - #tram**)

Number of states goes from  $O(n)$  to  $O(n^2)$ !

- Let's revisit our magic tram example. Suppose now that a decree comes from above that says you can't have take the tram more times than you walk. This makes our lives considerably worse, since if we wanted to respect this constraint, we have to keep track of additional information (augment the state).
- In particular, we need to keep track of the number of walk actions that we've taken so far minus the number of tram actions we've taken so far, and enforce that this number does not go negative. Now the number of states we have is much larger and thus, search becomes a lot slower.



# Easier search



## Example: relaxed problem

Start state: 1

Walk action: from  $s$  to  $s + 1$  (cost: 1)

Tram action: from  $s$  to  $2s$  (cost: 2)

End state:  $n$

~~Constraint: can't have more tram actions than walk actions.~~

**Original state:** (location, ~~#walk - #tram~~)

**Relaxed state:** location

- What if we just ignore that constraint and solve the original problem? That would be much easier/faster.  
But how do we construct a consistent heuristic from the solution from the relaxed problem?

# Easier search

- Compute relaxed  $\text{FutureCost}_{\text{rel}}(\text{location})$  for **each** location  $(1, \dots, n)$  using dynamic programming or UCS



## Example: reversed relaxed problem

Start state:  $n$

Walk action: from  $s$  to  $s - 1$  (cost: 1)

Tram action: from  $s$  to  $s/2$  (cost: 2)

End state: 1

Modify UCS to compute all past costs in reversed relaxed problem  
(equivalent to future costs in relaxed problem!)

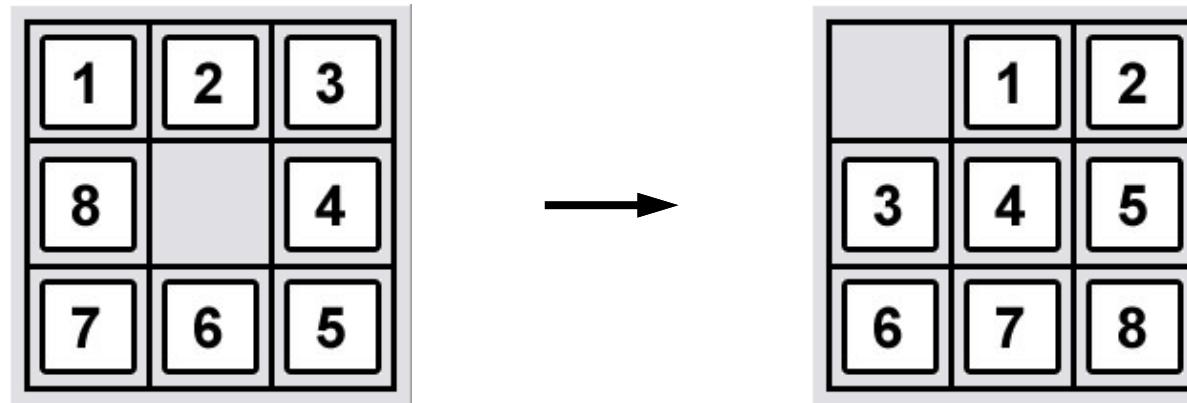
- Define heuristic for original problem:

$$h((\text{location}, \#\text{walk}-\#\text{tram})) = \text{FutureCost}_{\text{rel}}(\text{location})$$

- We want to now construct a heuristic  $h(s)$  based on the future costs under the relaxed problem.
- For this, we need the future costs for all the relaxed states. One straightforward way to do this is by using dynamic programming. However, if we have cycles, then we need to use uniform cost search.
- But recall that UCS only computes the past costs of all states up until the end. So we need to make two changes. First, we simply don't stop at the end, but keep on going until we've explored all the states. Second, we define a **reversed relaxed problem** (where all the edges are just reversed), and call UCS on that. UCS will return past costs in the reversed relaxed problem which correspond exactly to future costs in the relaxed problem.
- Finally, we need to construct the actual heuristic. We have to be a bit careful because the state spaces of the relaxed and original problems are different. For this, we set the heuristic  $h(s)$  to the future cost of the relaxed version of  $s$ .
- Note that the minimum cost returned by A\* (UCS on the modified problem) is the true minimum cost minus the value of the heuristic at the start state.

# Independent subproblems

[8 puzzle]



Original problem: tiles **cannot** overlap (constraint)

Relaxed problem: tiles **can** overlap (no constraint)

Relaxed solution: 8 indep. problems, each in closed form



**Key idea: independence**

Relax original problem into independent subproblems.

- So far, we've seen that some heuristics  $h(s)$  can be computed in closed form and others can be computed by doing a cheaper search. But there's another way to define heuristics  $h(s)$  which are efficient to compute.
- In the 8-puzzle, the goal is to slide the tiles around to produce the desired configuration, but with the constraint that no two tiles can occupy the same position. However, we can throw that constraint out the window to get a relaxed problem. Now, the new problem is really easy, because the tiles can now move **independently**. So we've taken one giant problem and turned it into 8 smaller problems. Each of the smaller problems can now be solved separately (in this case, in closed form, but in other cases, we can also just do search).
- It's worth remembering that all of these relaxed problems are simply used to get the value of the heuristic  $h(s)$  to guide the full search. The actual solutions to these relaxed problems are not used.

# General framework

## Removing constraints

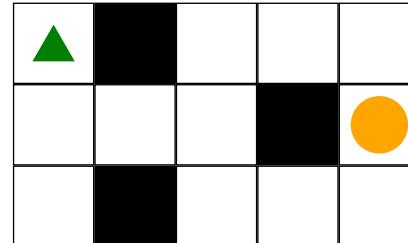
(knock down walls, walk/tram freely, overlap pieces)



## Reducing edge costs

(from  $\infty$  to some finite cost)

Example:



Original:  $\text{Cost}((1, 1), \text{East}) = \infty$

Relaxed:  $\text{Cost}_{\text{rel}}((1, 1), \text{East}) = 1$

- We have seen three instances where removing constraints yields simpler solutions, either via closed form, easier search, or independent subproblems. But we haven't formally proved that the heuristics you get are consistent!
- Now we will analyze all three cases in a unified framework. Removing constraints can be thought of as adding edges (you can go between pairs of states that you weren't able to before). Adding edges is equivalent to reducing the edge cost from infinity to something finite (the resulting edge cost).

# General framework



## Definition: relaxed search problem

A **relaxation**  $P_{\text{rel}}$  of a search problem  $P$  has costs that satisfy:

$$\text{Cost}_{\text{rel}}(s, a) \leq \text{Cost}(s, a).$$



## Definition: relaxed heuristic

Given a relaxed search problem  $P_{\text{rel}}$ , define the **relaxed heuristic**  $h(s) = \text{FutureCost}_{\text{rel}}(s)$ , the minimum cost from  $s$  to an end state using  $\text{Cost}_{\text{rel}}(s, a)$ .

- More formally, we define a relaxed search problem as one where the relaxed edge costs are no larger than the original edge costs.
- The relaxed heuristic is simply the future cost of the relaxed search problem, which by design should be efficiently computable.

## General framework



### Theorem: consistency of relaxed heuristics

Suppose  $h(s) = \text{FutureCost}_{\text{rel}}(s)$  for some relaxed problem  $P_{\text{rel}}$ .

Then  $h(s)$  is a consistent heuristic.

Proof:

$$h(s) \leq \text{Cost}_{\text{rel}}(s, a) + h(\text{Succ}(s, a)) \quad [\text{triangle inequality}]$$

$$\leq \text{Cost}(s, a) + h(\text{Succ}(s, a)) \quad [\text{relaxation}]$$

- We now need to check that our defined heuristic  $h(s)$  is actually consistent, so that using it actually will yield the minimum cost path of our original problem (not the relaxed problem, which is just a means towards an end).
- Checking consistency is actually quite easy. The first inequality follows because  $h(s) = \text{FutureCost}_{\text{rel}}(s)$ , and all future costs correspond to the minimum cost paths. So taking action  $a$  from state  $s$  better be no better than taking the best action from state  $s$  (this is all in the search problem  $P_{\text{rel}}$ ).
- The second inequality follows just by the definition of a relaxed search problem.
- The significance of this theorem is that we only need to think about coming up with relaxations rather than worrying directly about checking consistency.

# Tradeoff

## Efficiency:

$h(s) = \text{FutureCost}_{\text{rel}}(s)$  must be easy to compute

Closed form, easier search, independent subproblems

## Tightness:

heuristic  $h(s)$  should be close to  $\text{FutureCost}(s)$

Don't remove too many constraints

- How should one go about designing a heuristic?
- First, the heuristic should be easy to compute. As the main point of A\* is to make things more efficient, if the heuristic is as hard as to compute as the original search problem, we've gained nothing (an extreme case is no relaxation at all, in which case  $h(s) = \text{FutureCost}(s)$ ).
- Second, the heuristic should tell us some information about where the goal is. In the extreme case, we relax all the way and we have  $h(s) = 0$ , which corresponds to running UCS. (Perhaps it is reassuring that we never perform worse than UCS.)
- So the art of designing heuristics is to balance informativeness with computational efficiency.

# Max of two heuristics

How do we combine two heuristics?



## Proposition: max heuristic

Suppose  $h_1(s)$  and  $h_2(s)$  are consistent.

Then  $h(s) = \max\{h_1(s), h_2(s)\}$  is consistent.

Proof: exercise

- In many situations, you'll be able to come up with two heuristics which are both reasonable, but no one dominates the other. Which one should you choose? Fortunately, you don't have to choose because you can use both of them!
- The key point is the max of two consistent heuristics is consistent. Why max? Remember that we want heuristic values to be larger. And furthermore, we can prove that taking the max leads to a consistent heuristic.
- Stepping back a bit, there is a deeper takeaway with A\* and relaxation here. The idea is that when you are confronted with a difficult problem, it is wise to start by solving easier versions of the problem (being an optimist). The result of solving these easier problems can then be a useful guide in solving the original problem.
- For example, if the relaxed problem turns out to have no solution, then you don't even have to bother solving the original problem, because a solution can't possibly exist (you've been optimistic by using the relaxation).



# Summary

- Structured Perceptron (reverse engineering): learn cost functions (search + learning)
- A\*: add in heuristic estimate of future costs
- Relaxation (breaking the rules): framework for producing consistent heuristics
- Next time: when actions have unknown consequences...

# CS221 Section 3: Search

DP, UCS and A\*

# Contents

- 1. Uniform Cost Search**
2. Defining States
3. Dynamic Programming
4. A\* Search

# Uniform Cost Search

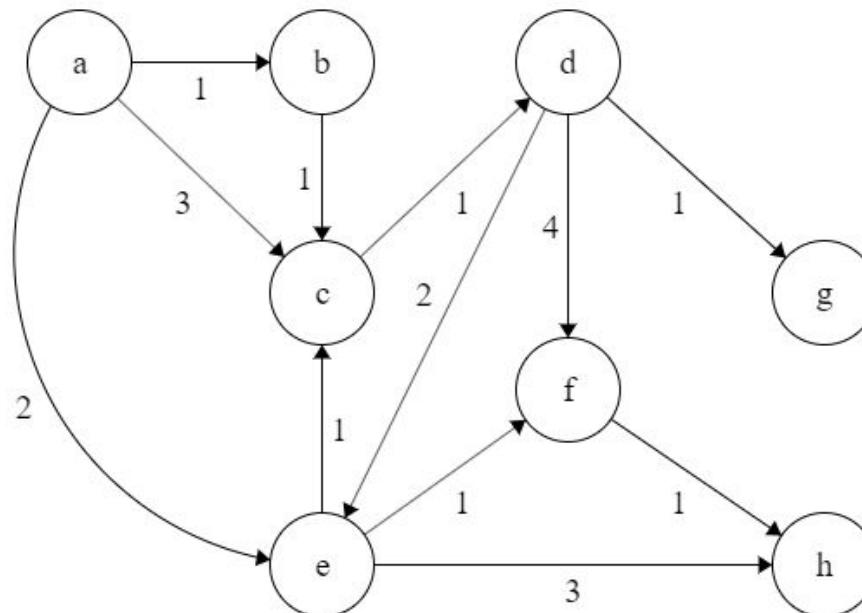
**Idea:** In UCS, we find the shortest cost to a node by using the fact we already know the shortest path to a set of nodes.

**Recall:** We have the following three sets

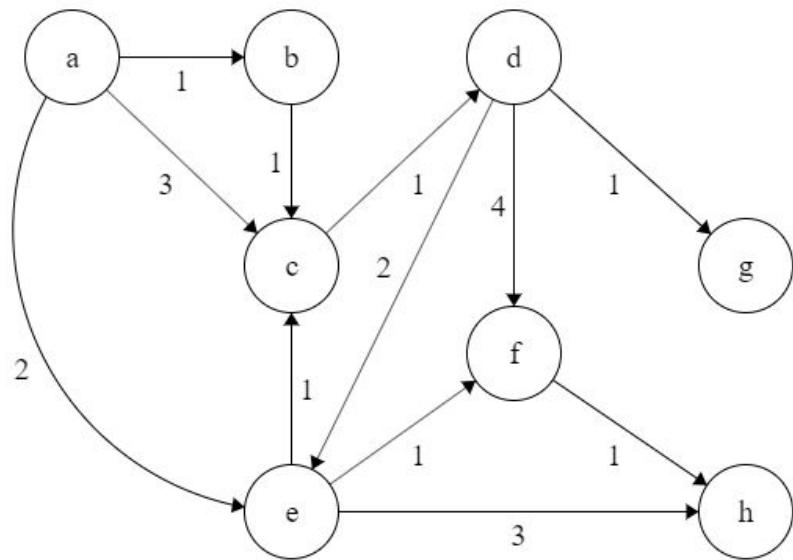
- Explored Set: contains nodes we know the path length to
- Frontier Set: contains nodes that are neighbors of those in the explored set, but we don't know their costs yet
- Unexplored Set: Nodes in the graph we haven't encountered

# Problem - UCS

In the following graph, find the costs to reach each node given that we start on node **a**.



# Problem - UCS



**Explored**

[a : 0]

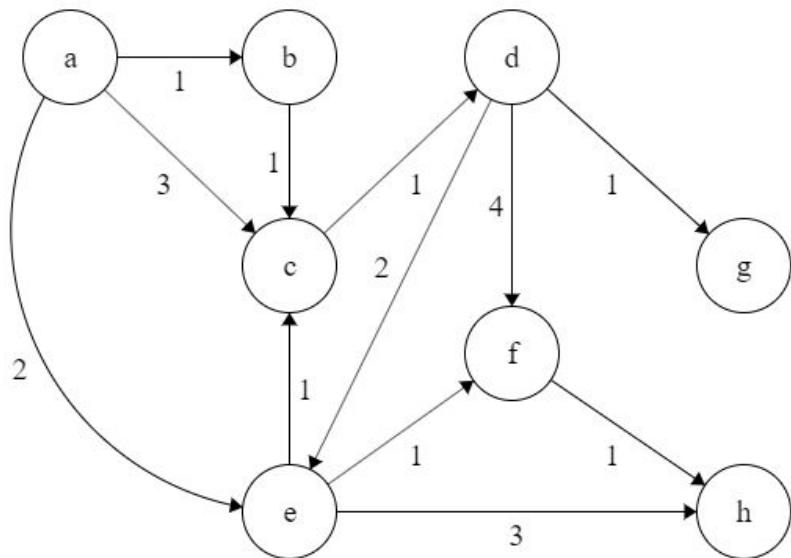
**Frontier**

[b : 0 + 1, e : 0 + 2, c : 0 + 3]

**Unexplored**

We start with node **a**. We add all neighbors of **a** to the frontier. Note: [a : 0] means it takes 0 cost to get to node a.

# Problem - UCS



**Explored**

[a : 0, b : 1]

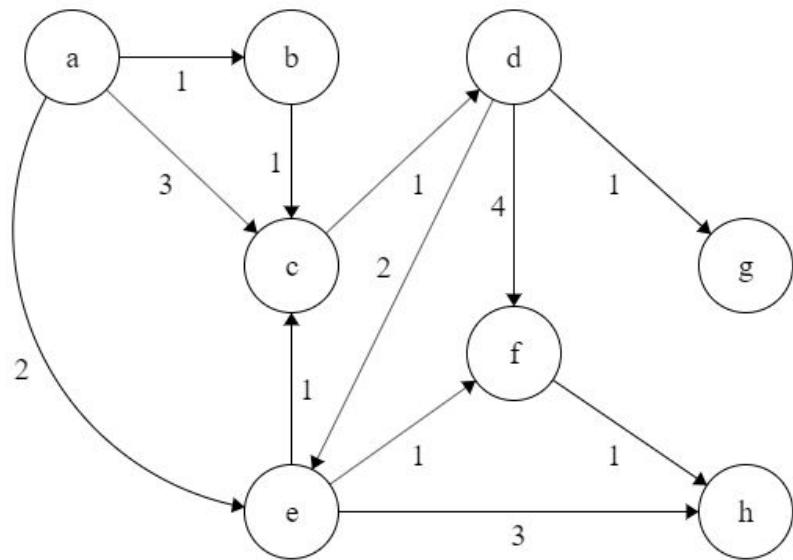
**Frontier**

[c : 1 + 1, e : 0 + 2]

**Unexplored**

In the frontier, **b** has the lowest cost. Thus, we can add it to the explored set. We add all neighbors of **b** to the frontier, updating costs to reach some nodes if necessary (we updated **c**).

# Problem - UCS



**Explored**

[a : 0, b : 1, c : 2]

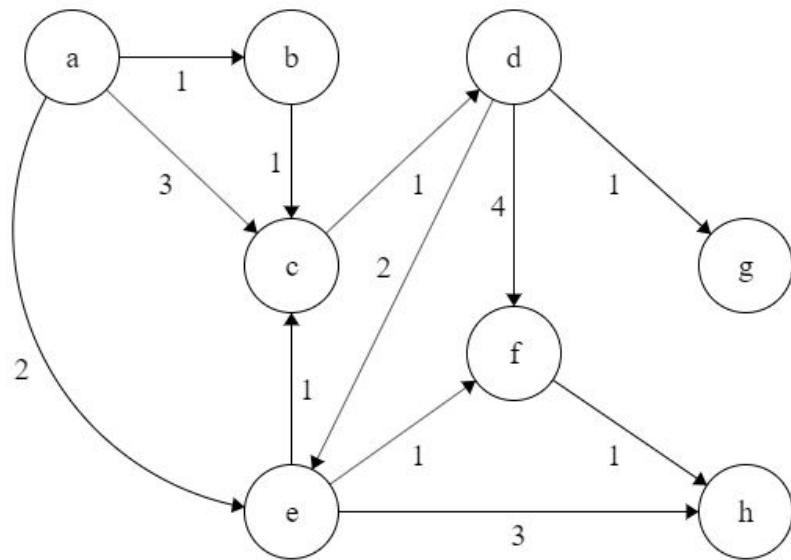
**Frontier**

[e : 0 + 2, d : 2 + 1]

**Unexplored**

In the frontier, **c** has the lowest cost (ties broken alphabetically here). Thus, we can add it to the explored set. We add all neighbors of **c** to the frontier, updating as necessary.

# Problem - UCS



**Explored**

[a : 0, b : 1, c : 2, e : 2]

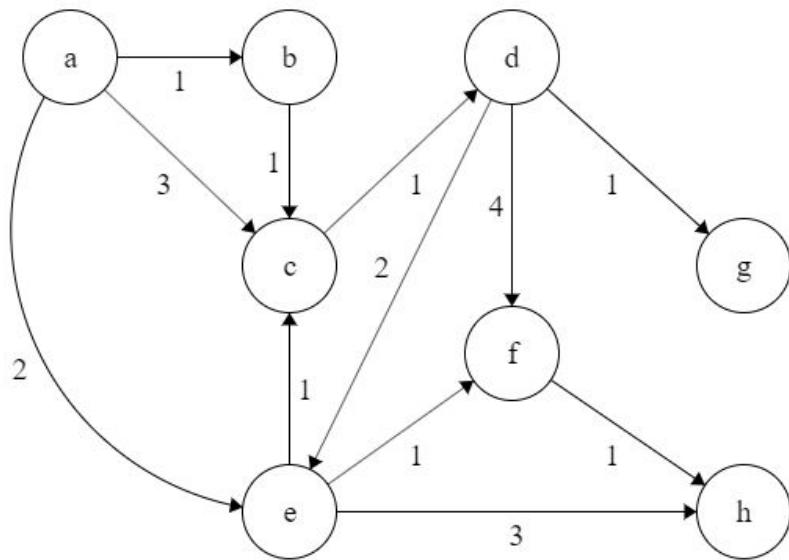
**Frontier**

[d : 2 + 1, f : 2 + 1, h : 2 + 3]

**Unexplored**

In the frontier, **e** has the lowest cost. Thus, we can add it to the explored set. We add all neighbors of **e** to the frontier, updating as necessary.

# Problem - UCS



**Explored**

[a : 0, b : 1, c : 2, e : 2, d : 3]

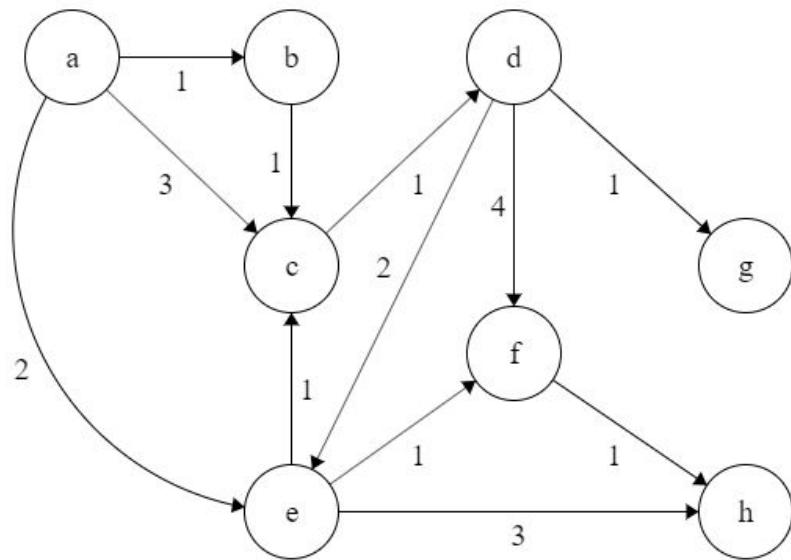
**Frontier**

[f : 2 + 1, g : 3 + 1, h : 2 + 3]

**Unexplored**

In the frontier, **d** has the lowest cost. Thus, we can add it to the explored set. We add all neighbors of **d** to the frontier, updating as necessary.

# Problem - UCS



## Explored

[a : 0, b : 1, c : 2, e : 2, d : 3, f : 3]

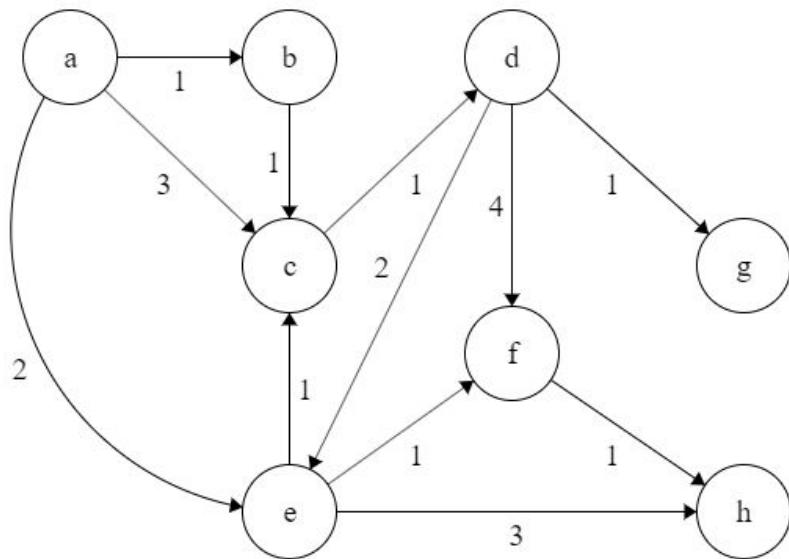
## Frontier

[g : 3 + 1, h : 3 + 1]

## Unexplored

In the frontier, **f** has the lowest cost. Thus, we can add it to the explored set. We add all neighbors of **f** to the frontier, updating as necessary.

# Problem - UCS



## Explored

[a : 0, b : 1, c : 2, e : 2, d : 3, f : 3, g : 4]

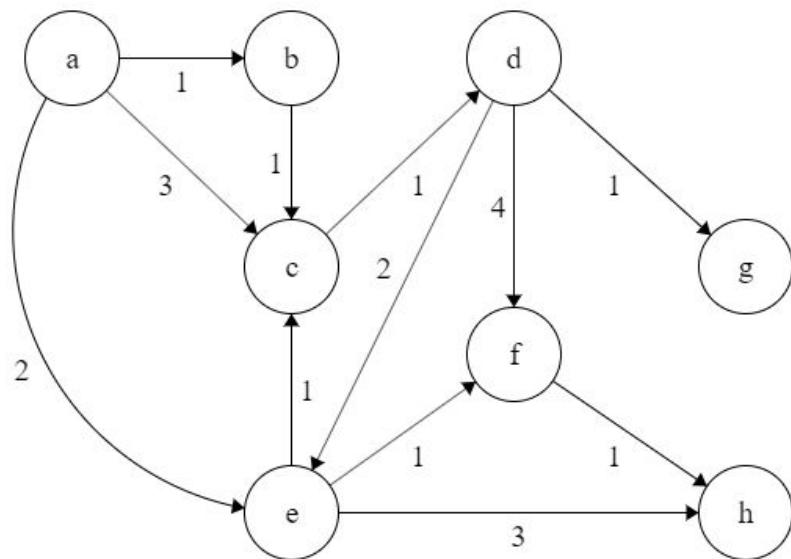
## Frontier

[h : 3 + 1]

## Unexplored

In the frontier, **g** has the lowest cost. Thus, we can add it to the explored set. We add all neighbors of **f** to the frontier, updating as necessary.

# Problem - UCS



**Explored**

[a : 0, b : 1, c : 2, e : 2, d : 3, f : 3, g : 4,  
h : 4]

**Frontier**

**Unexplored**

In the frontier, h has the lowest cost. Thus, we can add it to the explored set. There are no more nodes in the frontier, so we are done.

# Uniform Cost Search



## Algorithm: uniform cost search [Dijkstra, 1956]

Add  $s_{\text{start}}$  to **frontier** (priority queue)

Repeat until frontier is empty:

    Remove  $s$  with smallest priority  $p$  from frontier

    If **IsEnd**( $s$ ): return solution

    Add  $s$  to **explored**

    For each action  $a \in \text{Actions}(s)$ :

        Get successor  $s' \leftarrow \text{Succ}(s, a)$

        If  $s'$  already in explored: continue

        Update **frontier** with  $s'$  and priority  $p + \text{Cost}(s, a)$

# Contents

1. Uniform Cost Search
- 2. Defining States**
3. Dynamic Programming
4. A\* Search

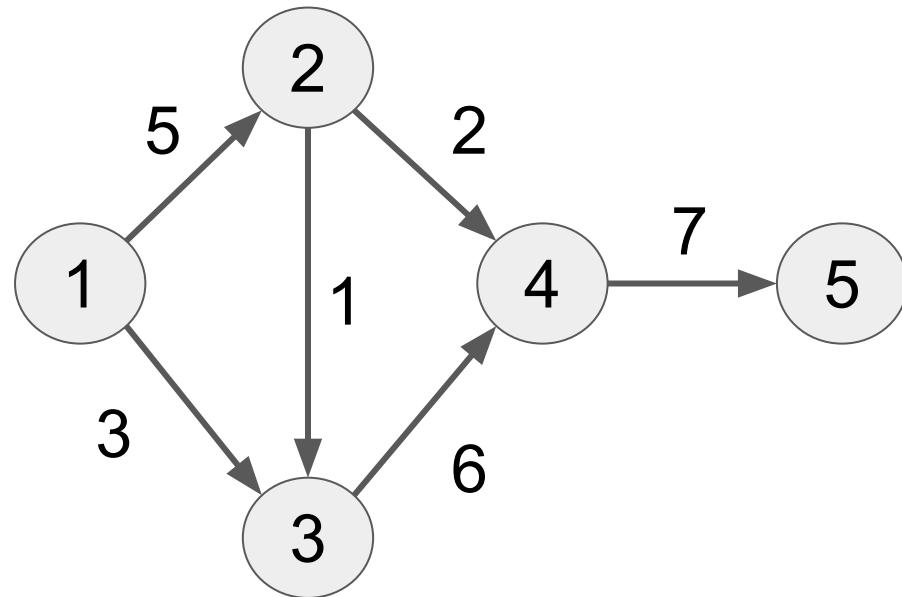
# Problem

There exists **N** cities, conveniently labelled from **1** to **N**.

There are roads connecting some pairs of cities. The road connecting city **i** and city **j** takes **c(i,j)** time to traverse. However, one can only travel from a city with smaller label to a city with larger label (i.e. each road is one-directional).

From city **1**, we want to travel to city **N**. What is the shortest time required to make this trip, given the additional constraint that we should visit more odd-labeled cities than even labeled cities?

# Example



Best path is [1, 3, 4, 5] with cost 16.

[1, 2, 4, 5] has cost 14 but visits equal number of odd and even cities.

# State Representation



**Key idea: state**

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

# State Representation

We need to know where we are currently at: **current\_city**

We need to know how many odd and even cities we have visited thus far: **#odd, #even**

State Representation: **(current\_city, #odd, #even)**

Total number of states:  **$O(N^3)$**

# Can We Do Better?

Check if all the information is really required

We store **#odd** and **#even** so that we can check whether  
**#odd - #even > 0** at  $(N, \#odd, \#even)$

Why not store **#odd - #even** directly instead?

**(current\_city, #odd - #even)** --  $O(N^2)$  states

# Solving the Problem

Since we are computing shortest path, which is some form of optimization, we consider DP and UCS.

Recall:

- DP can handle negative edges but works only on DAGs
- UCS works on general graphs, but cannot handle negative edges

Since we have a DAG and all edges are positive, both work! We already went through UCS, so we solve this with DP.

# Contents

1. Uniform Cost Search
2. Defining States
- 3. Dynamic Programming**
4. A\* Search

# Solving the Problem: Dynamic Programming

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

If  $s$  has no successors, we set it as undefined

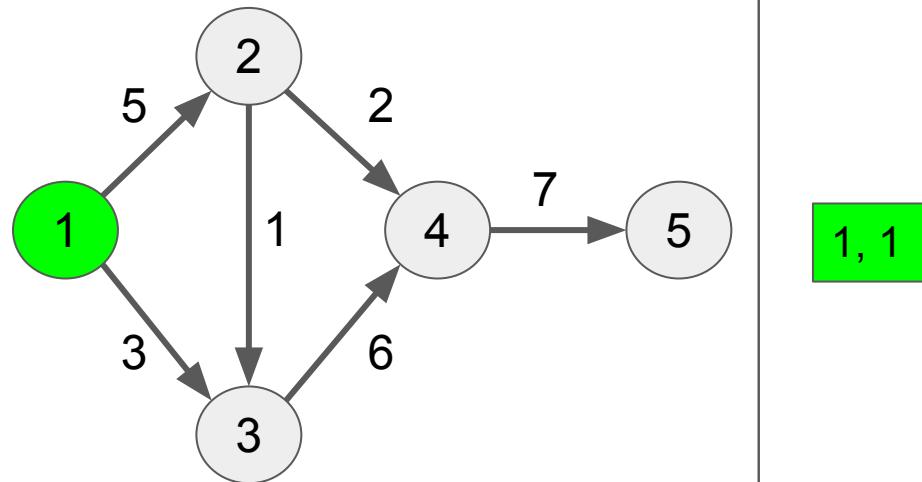
# Simulation of DP

Visiting

Successors

Completed

Regular Graph



State Graph

1, 1

Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

# Simulation of DP

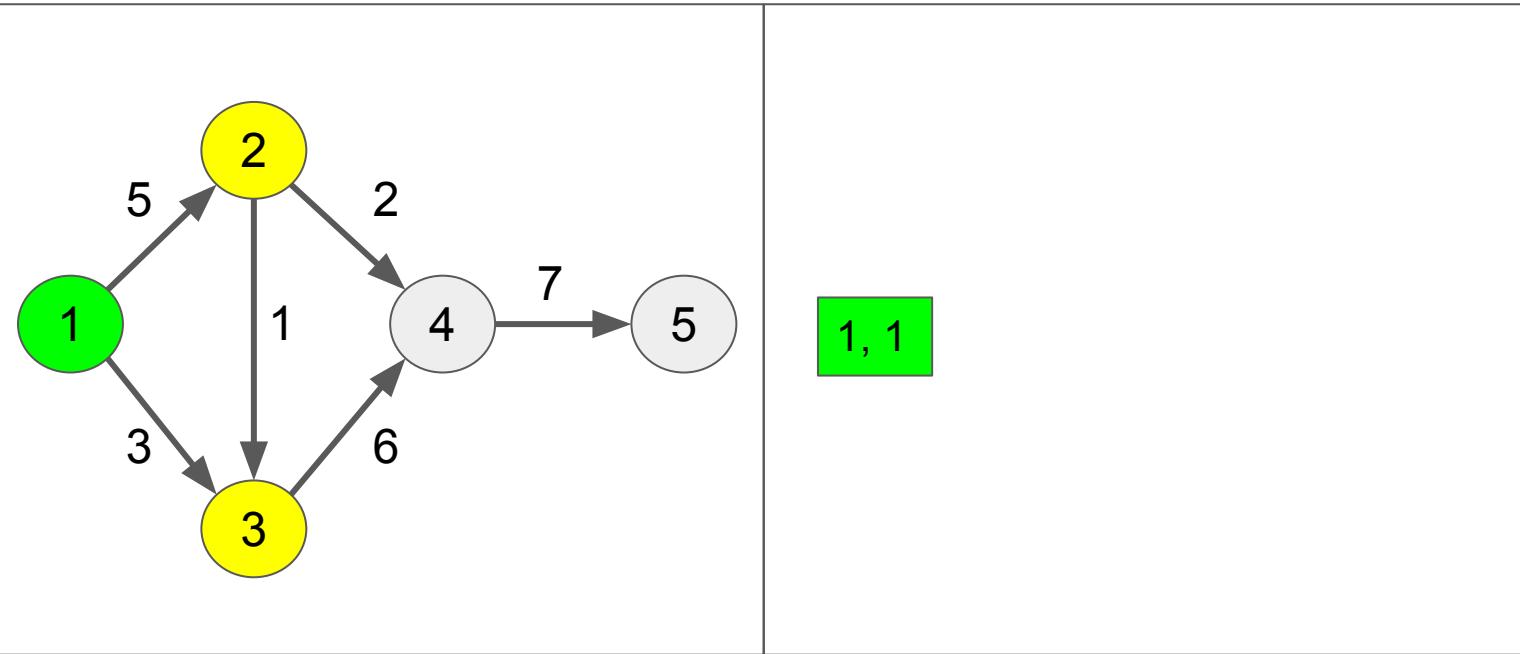
Visiting

Successors

Completed

Regular Graph

State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

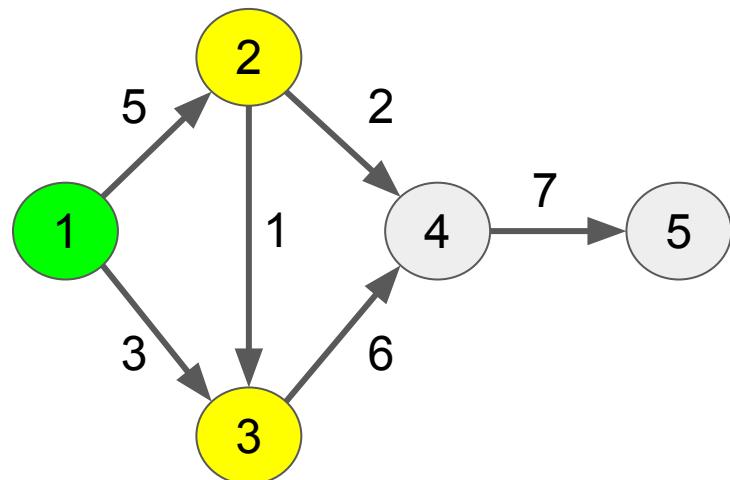
# Simulation of DP

Visiting

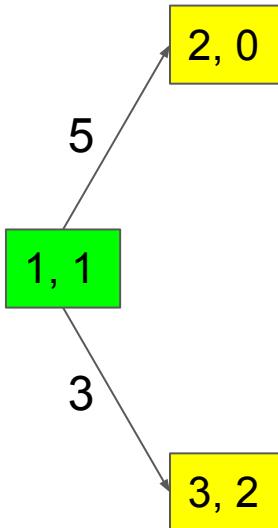
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

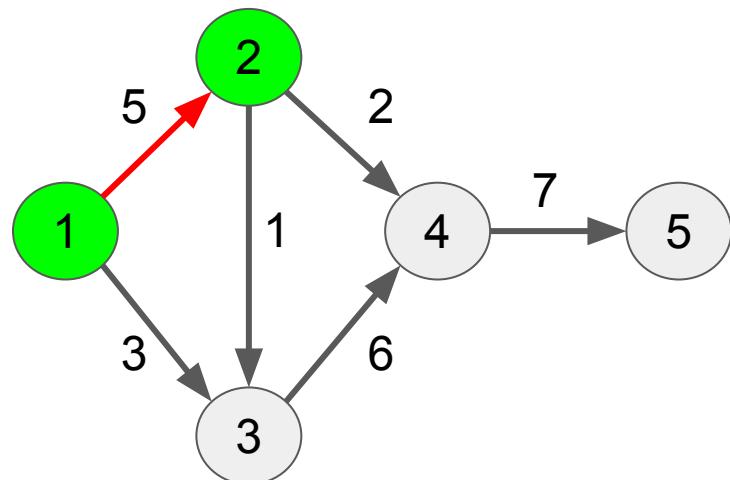
# Simulation of DP

Visiting

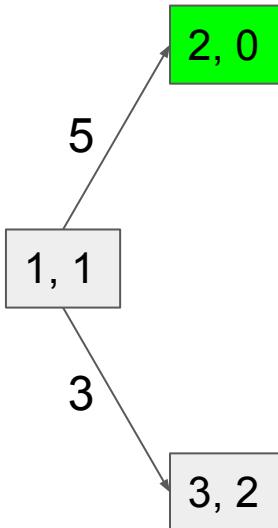
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

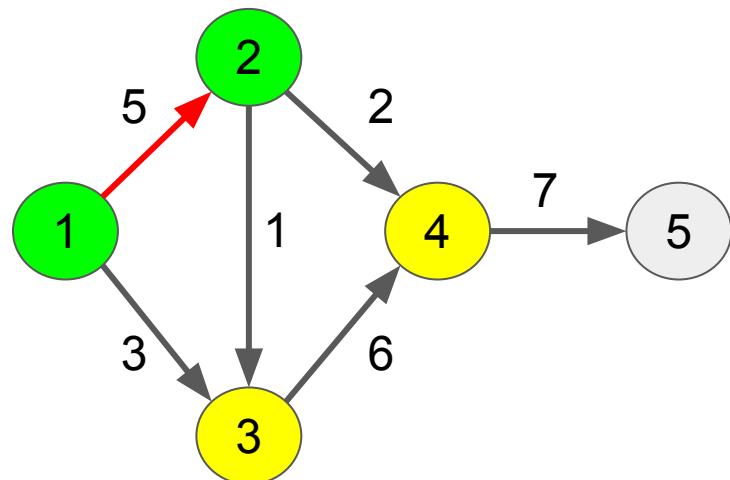
# Simulation of DP

Visiting

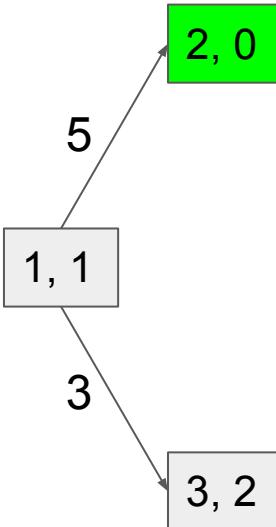
Successors

Completed

Regular Graph



State Graph



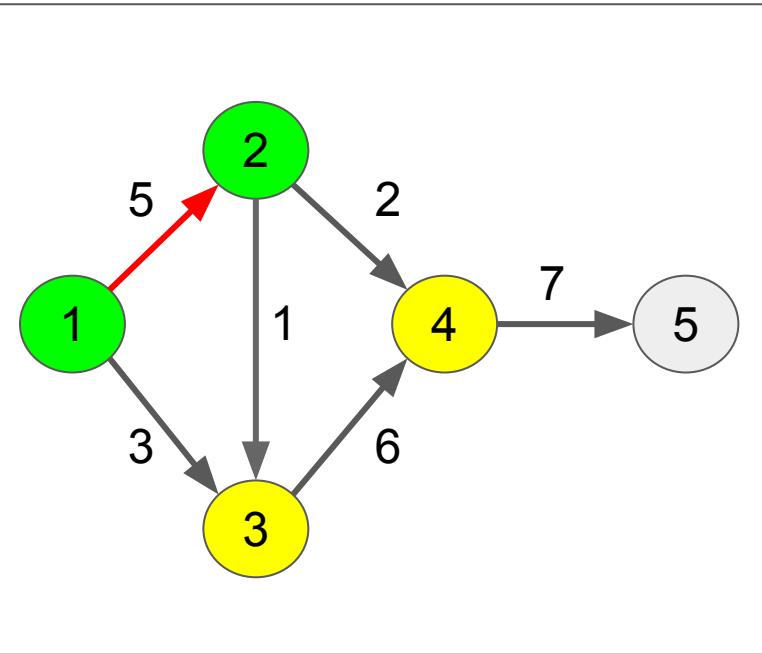
Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

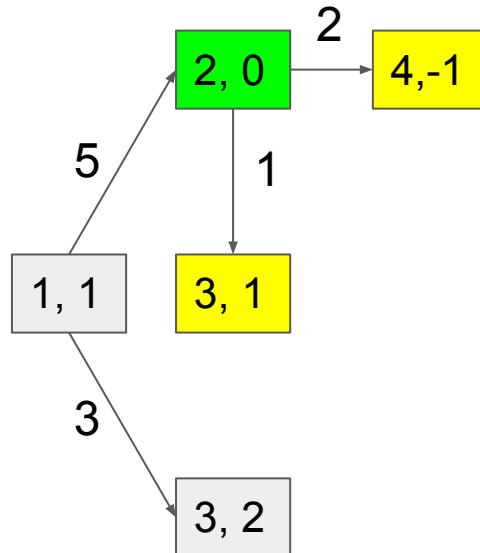
# Simulation of DP

Visiting	Successors	Completed
----------	------------	-----------

## Regular Graph



## State Graph



$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

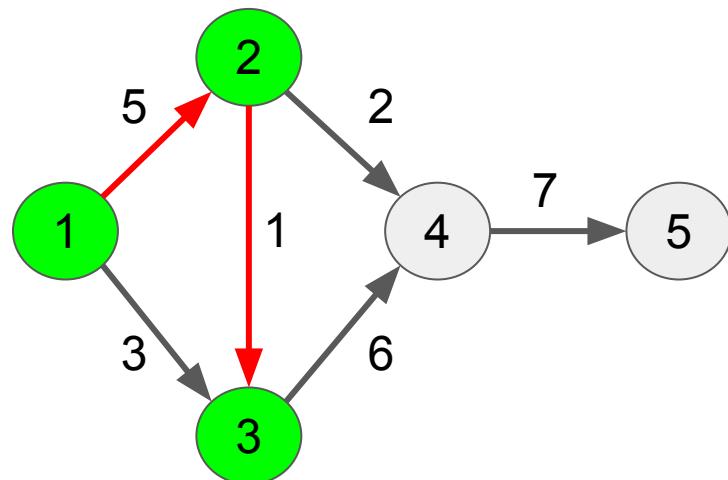
# Simulation of DP

Visiting

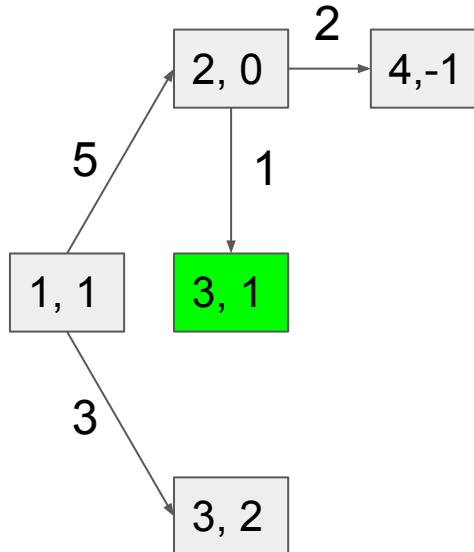
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

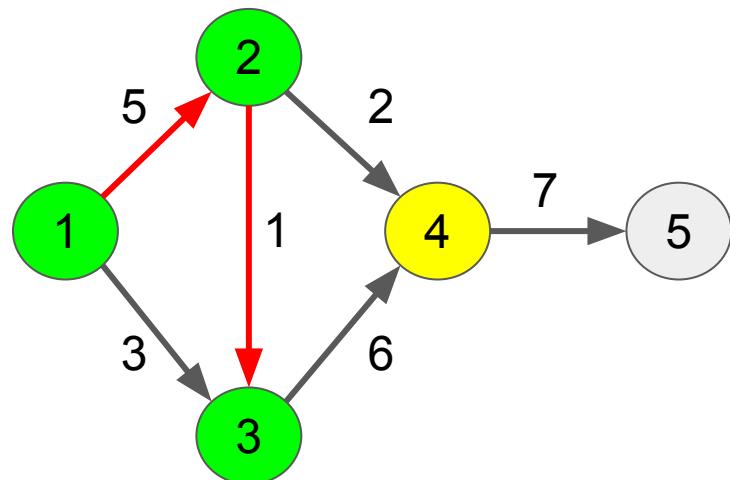
# Simulation of DP

Visiting

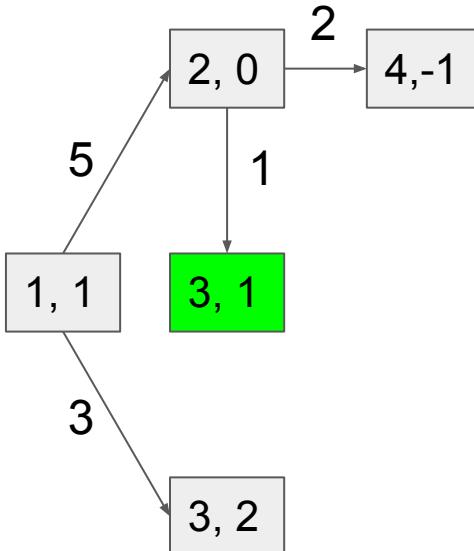
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

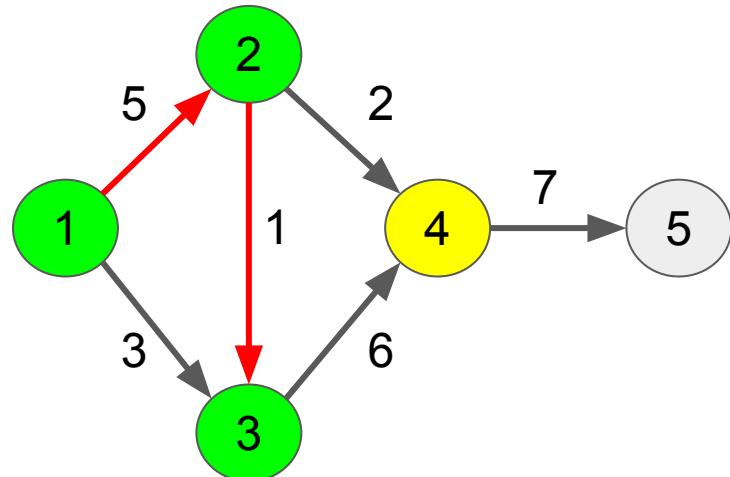
# Simulation of DP

Visiting

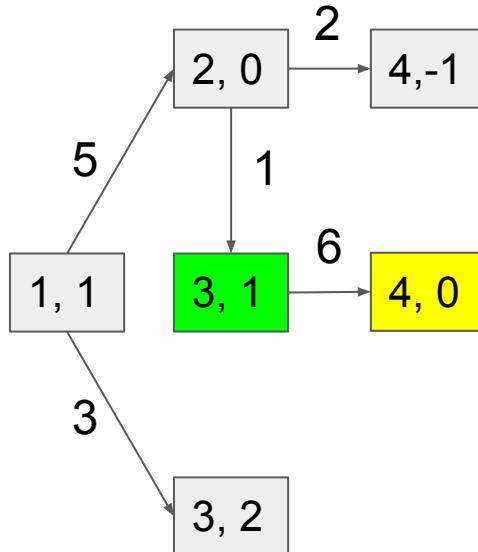
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

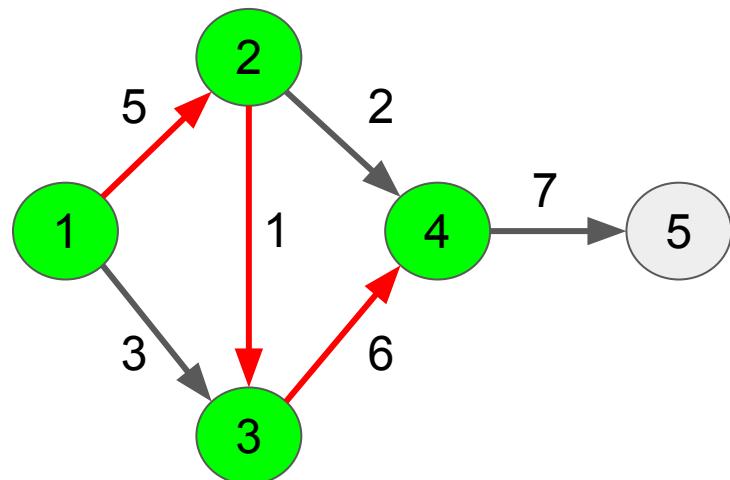
# Simulation of DP

Visiting

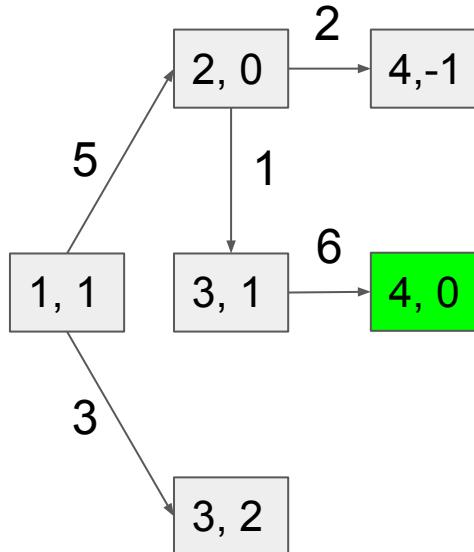
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

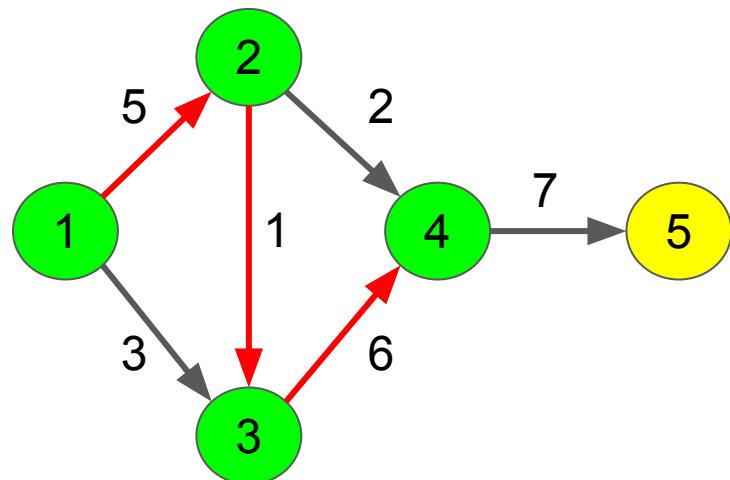
# Simulation of DP

Visiting

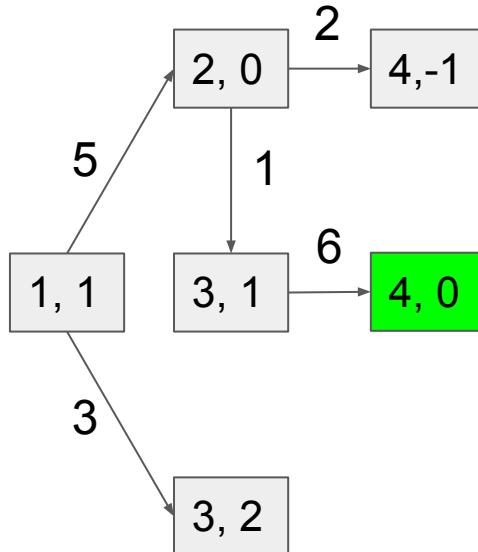
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

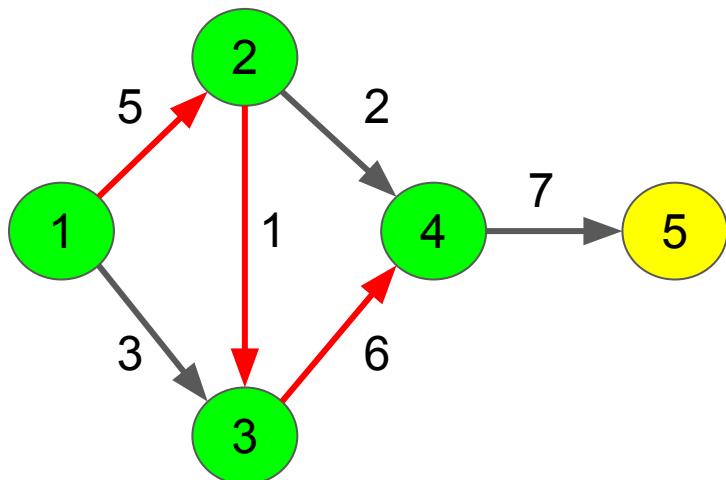
# Simulation of DP

Visiting

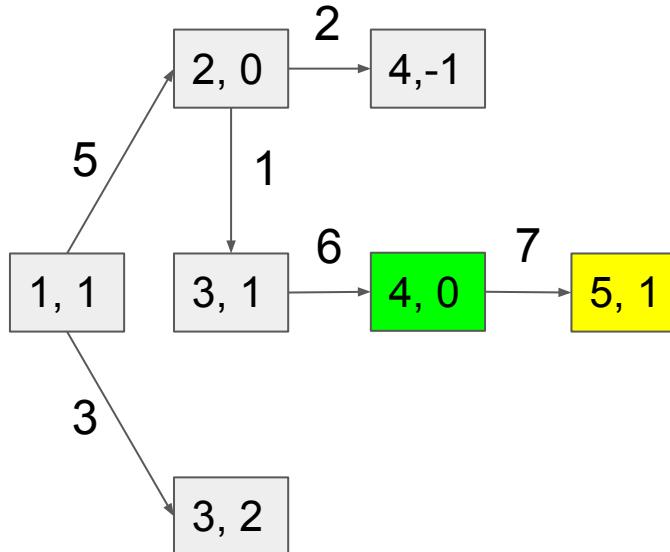
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

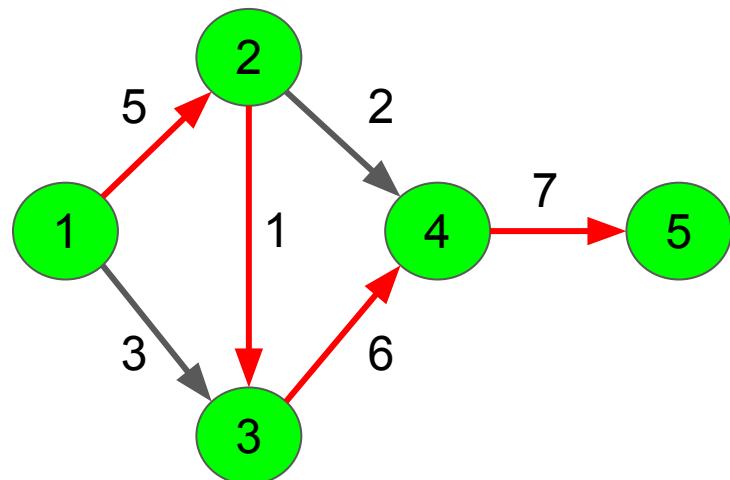
# Simulation of DP

Visiting

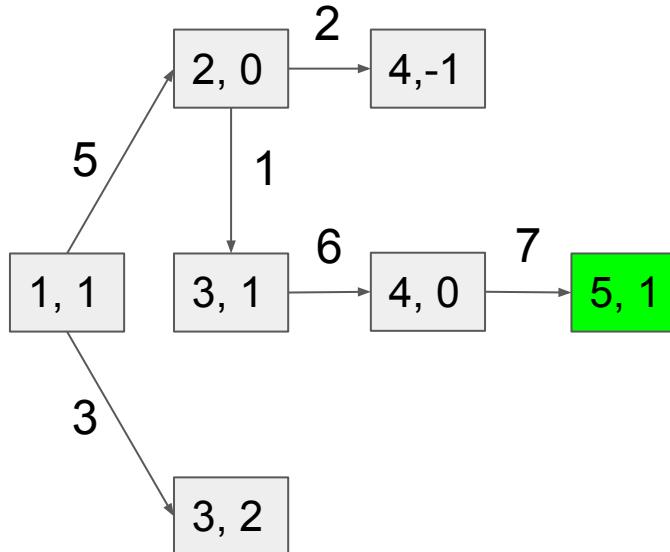
Successors

Completed

Regular Graph



State Graph



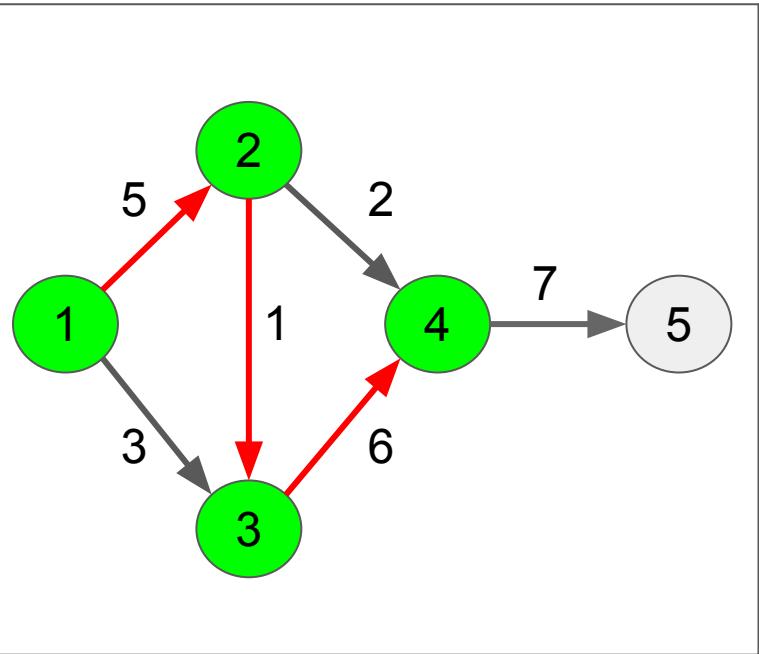
Cache	
Key	Value
(5, 1)	0

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

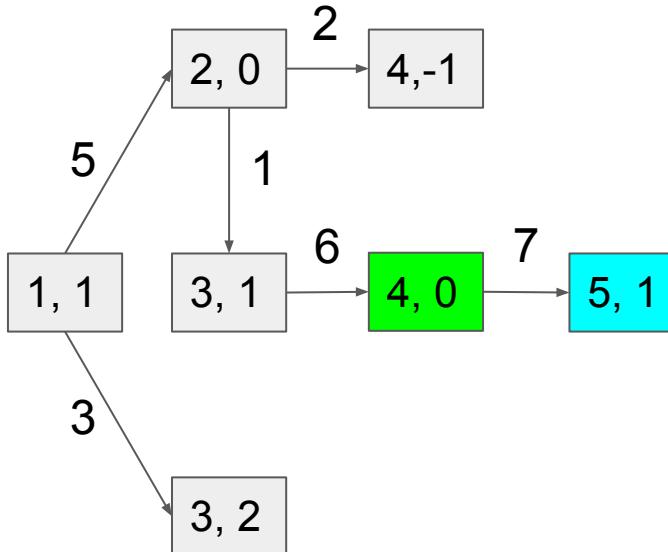
# Simulation of DP

Visiting	Successors	Completed
----------	------------	-----------

## Regular Graph



## State Graph



$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

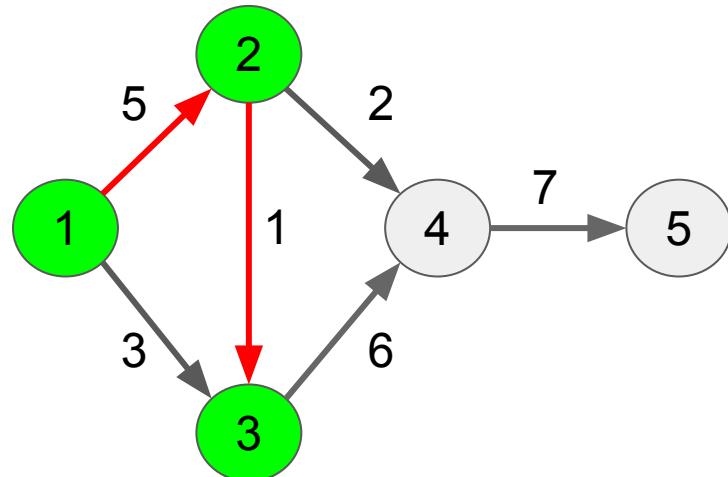
# Simulation of DP

Visiting

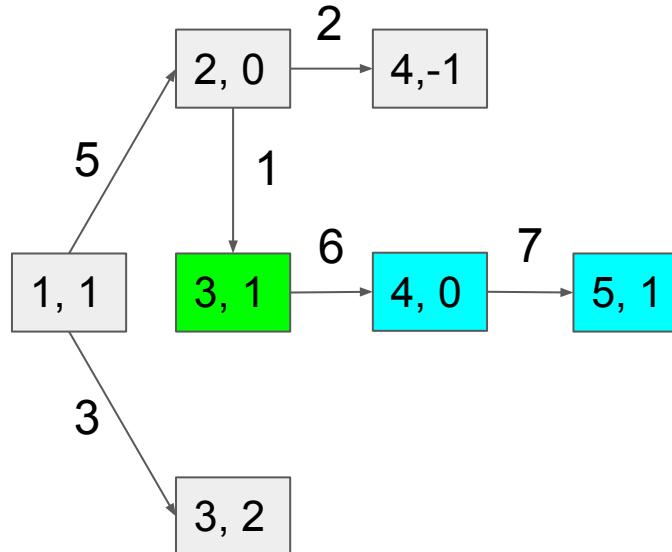
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
$(5, 1)$	0
$(4, 0)$	7
$(3, 1)$	$7 + 6 = 13$

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

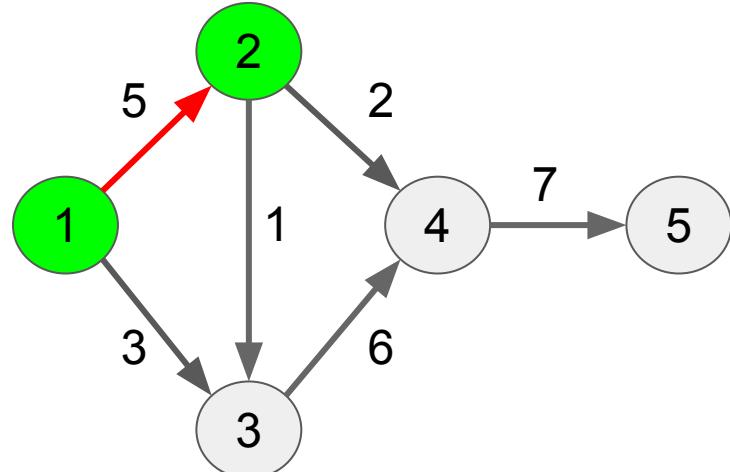
# Simulation of DP

Visiting

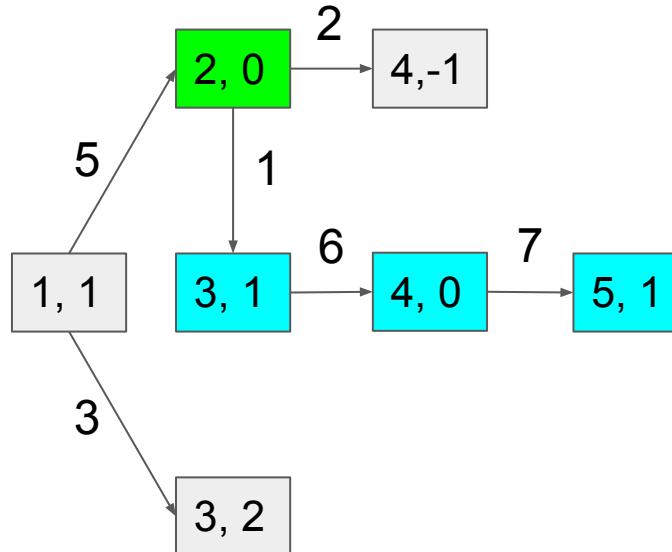
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

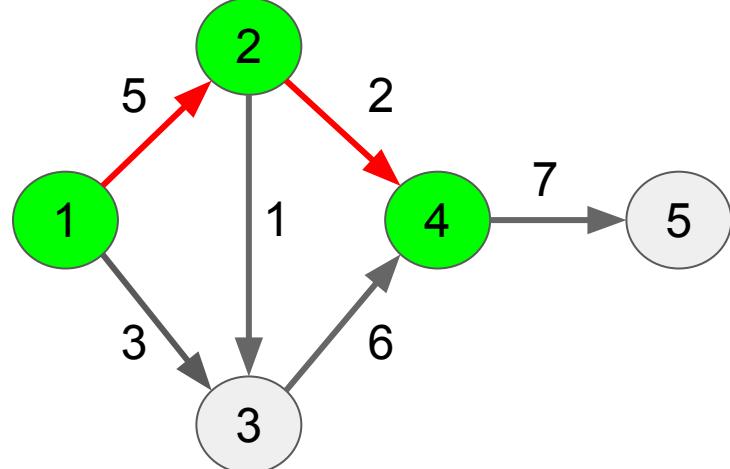
# Simulation of DP

Visiting

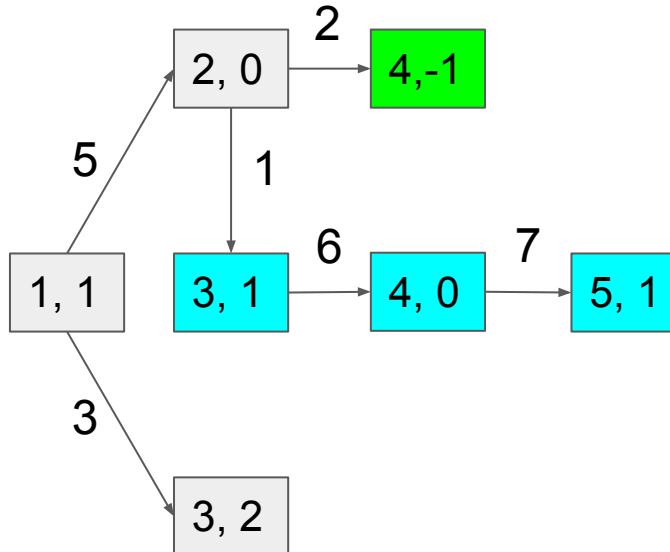
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

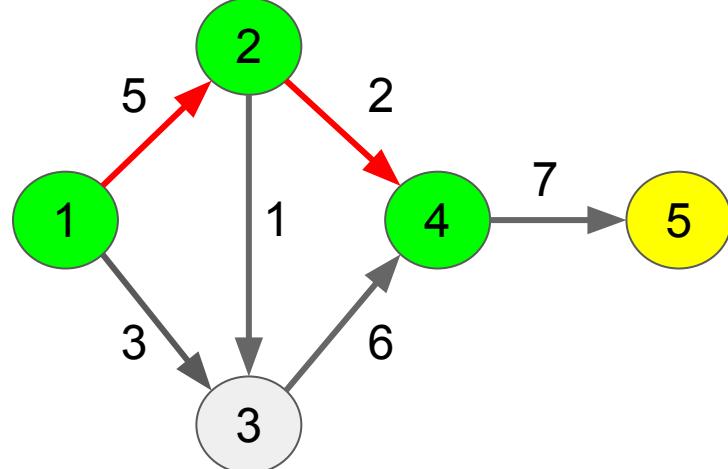
# Simulation of DP

Visiting

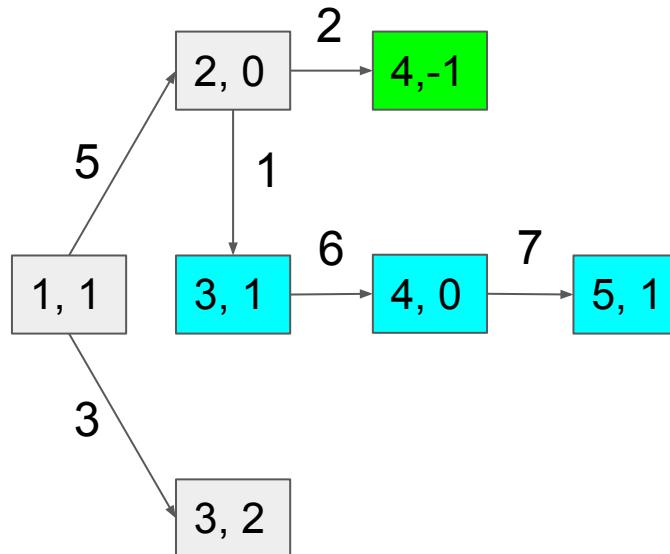
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

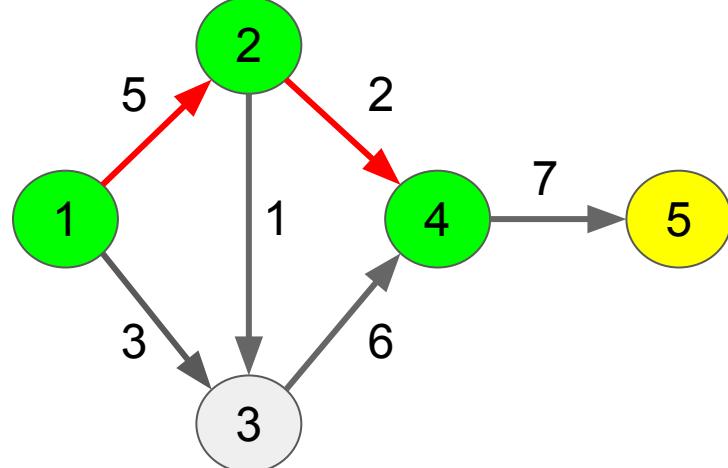
# Simulation of DP

Visiting

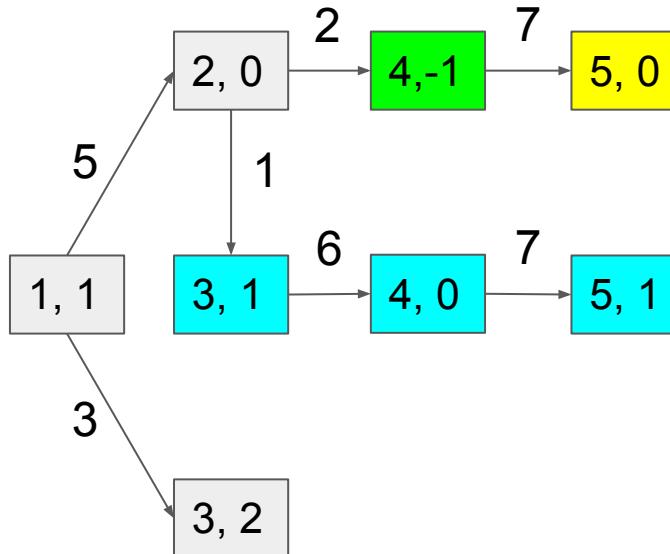
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

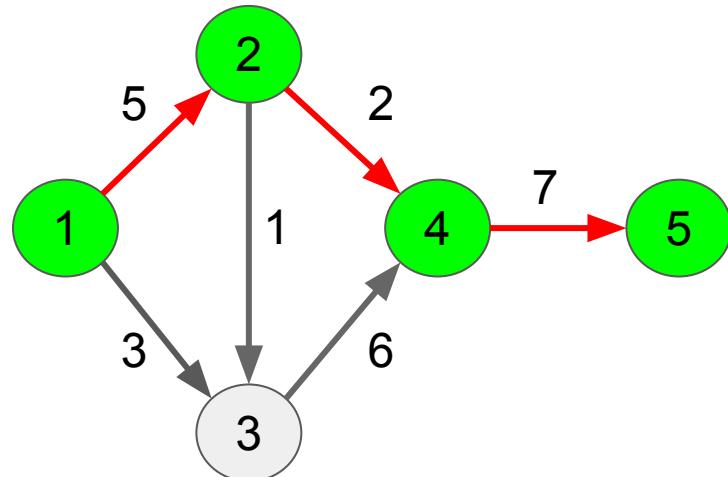
# Simulation of DP

Visiting

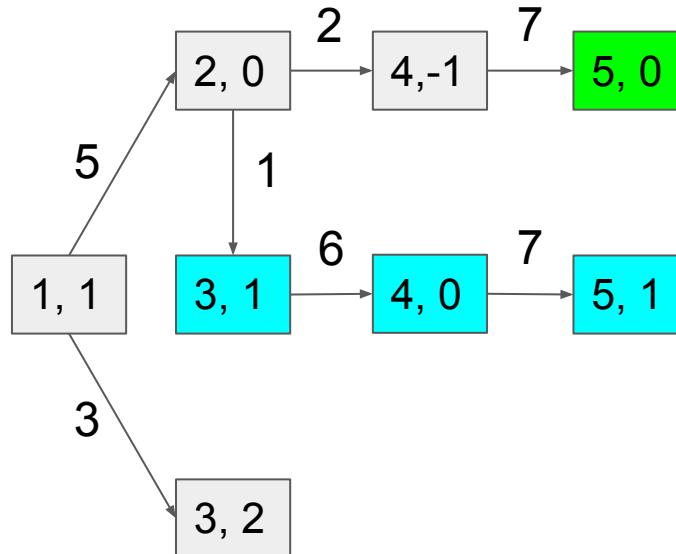
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

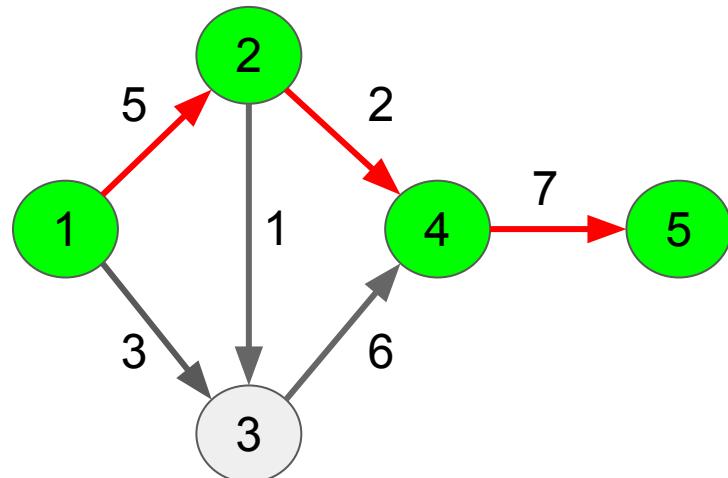
# Simulation of DP

Visiting

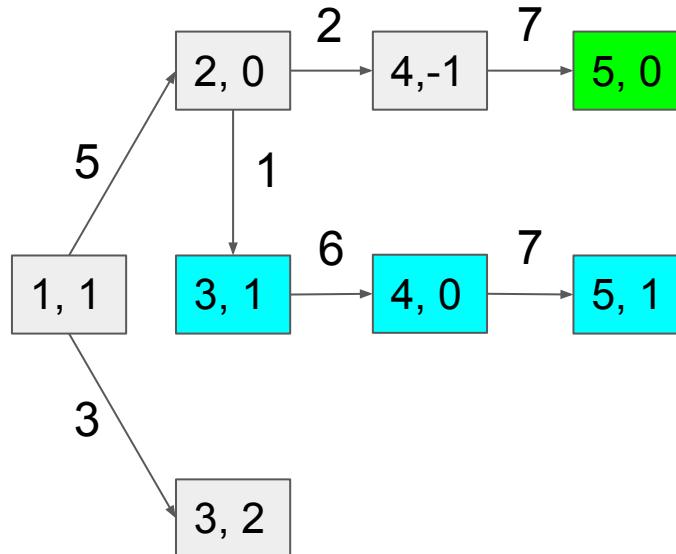
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

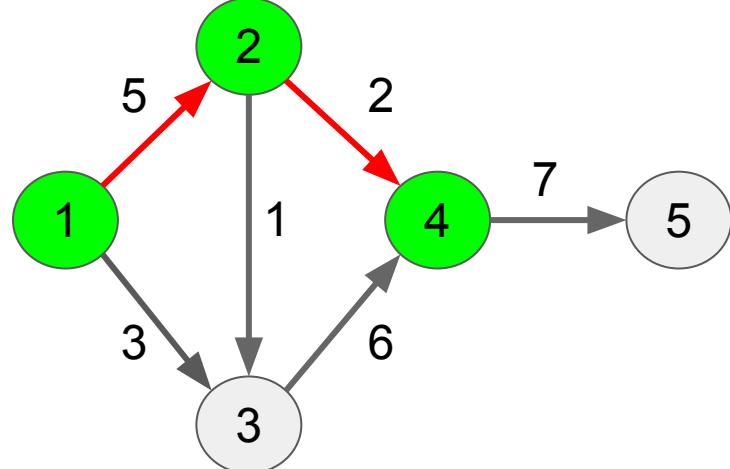
# Simulation of DP

Visiting

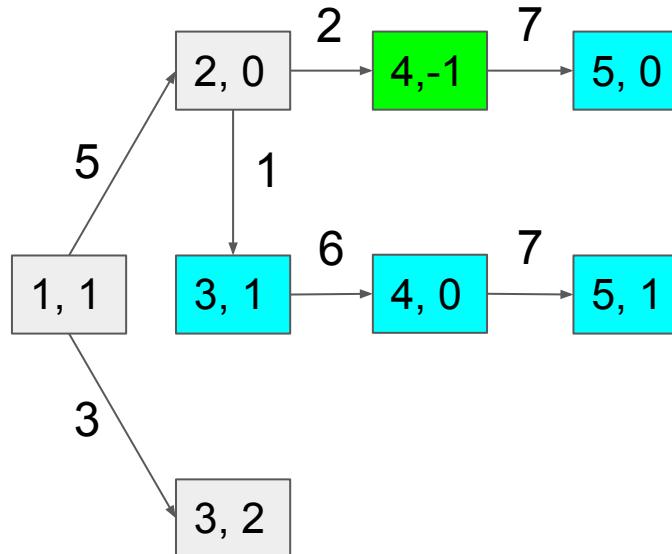
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

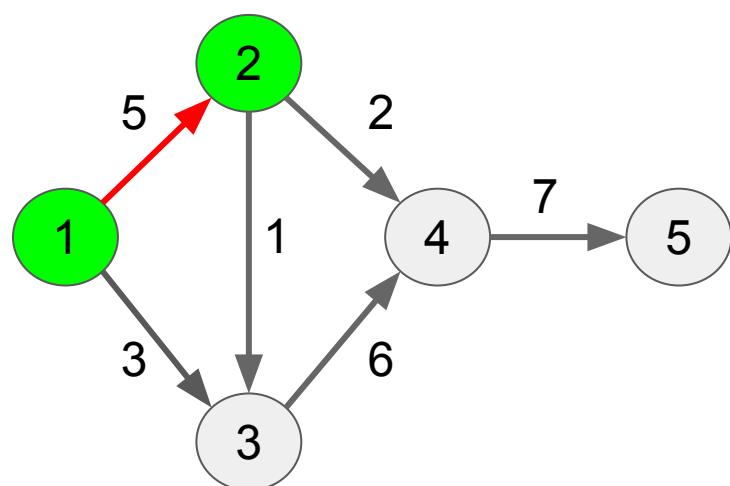
# Simulation of DP

Visiting

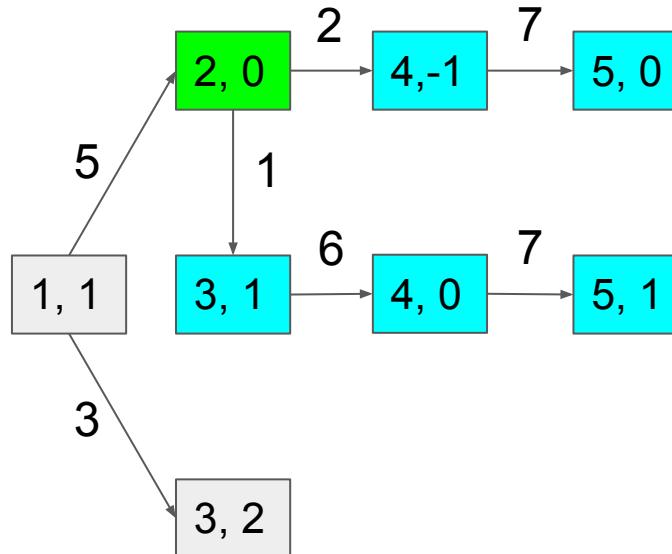
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	$13 + 1 = 14$

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

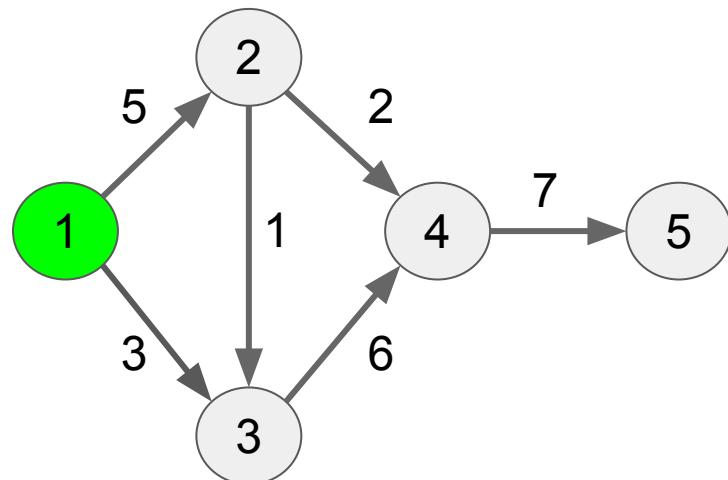
# Simulation of DP

Visiting

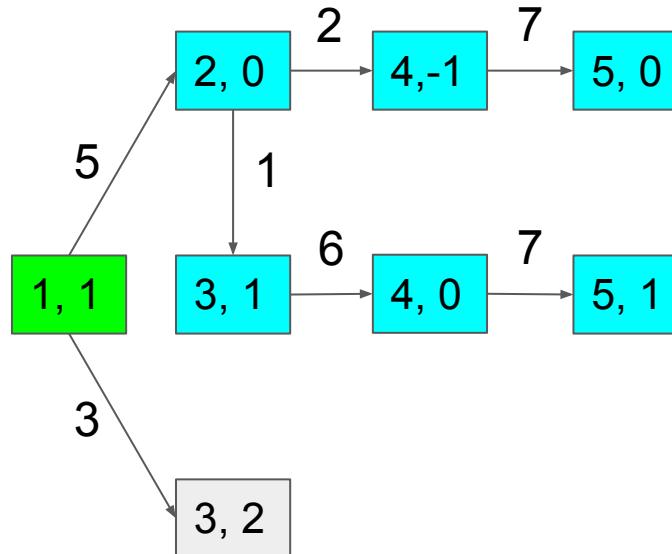
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

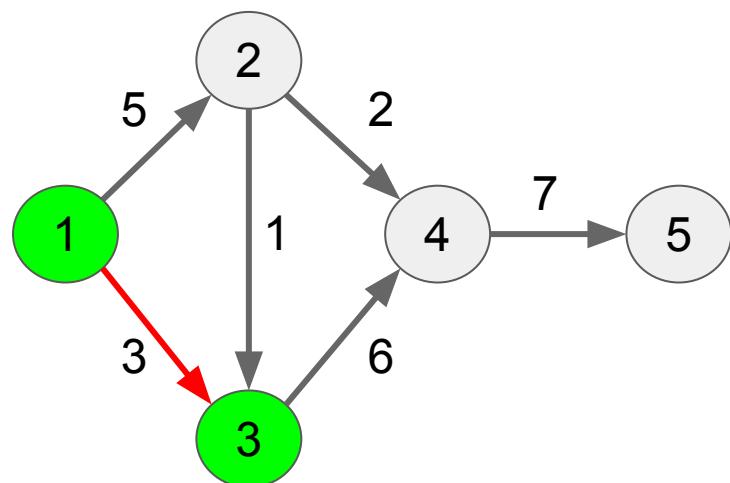
# Simulation of DP

Visiting

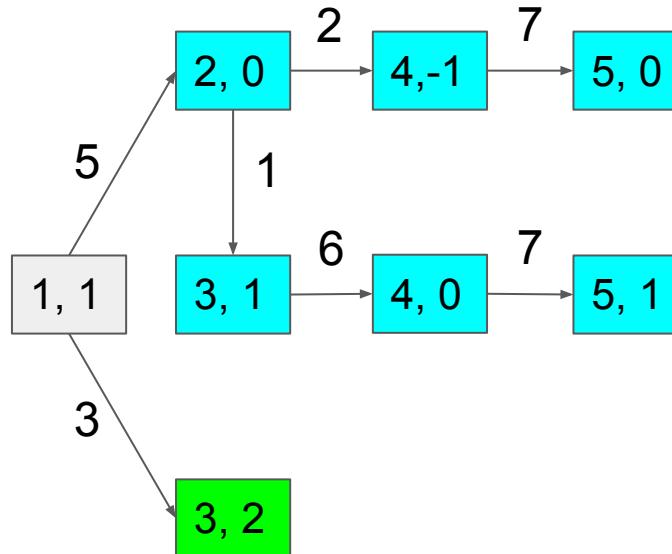
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

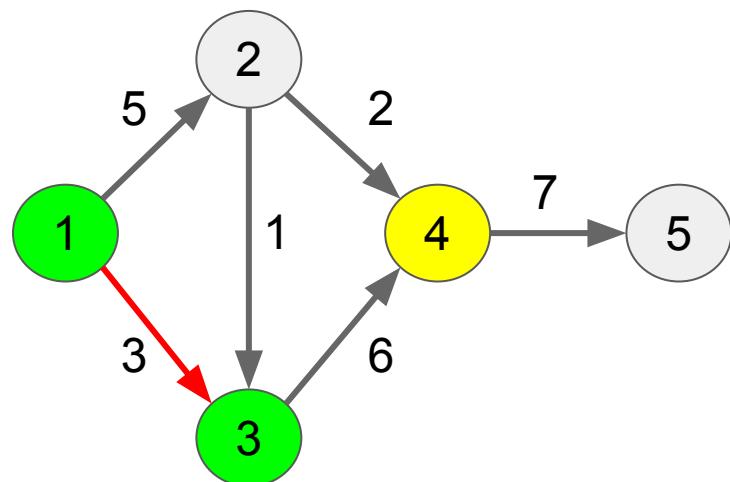
# Simulation of DP

Visiting

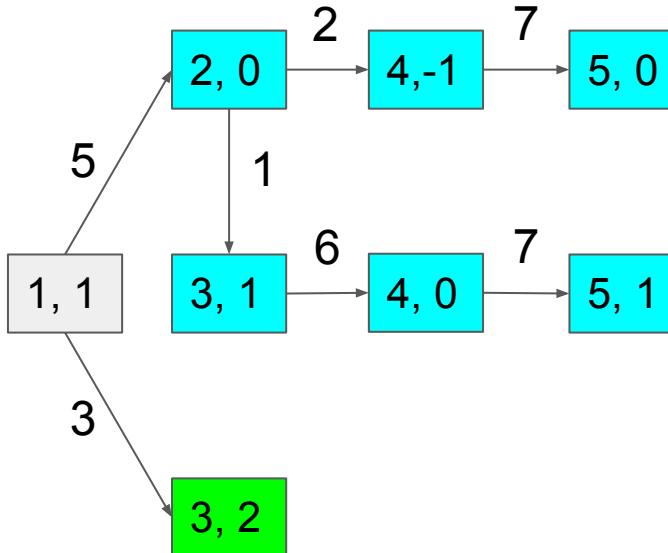
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

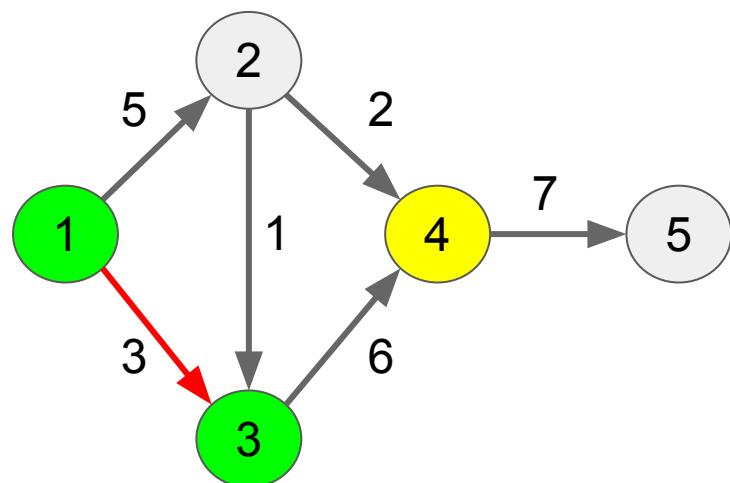
# Simulation of DP

Visiting

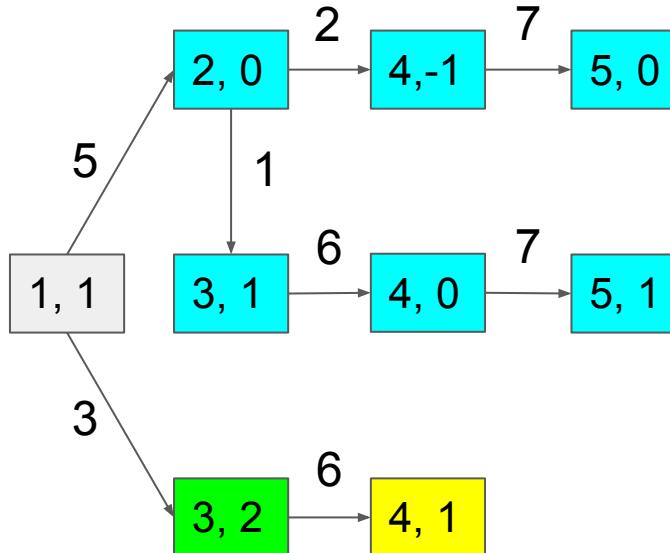
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

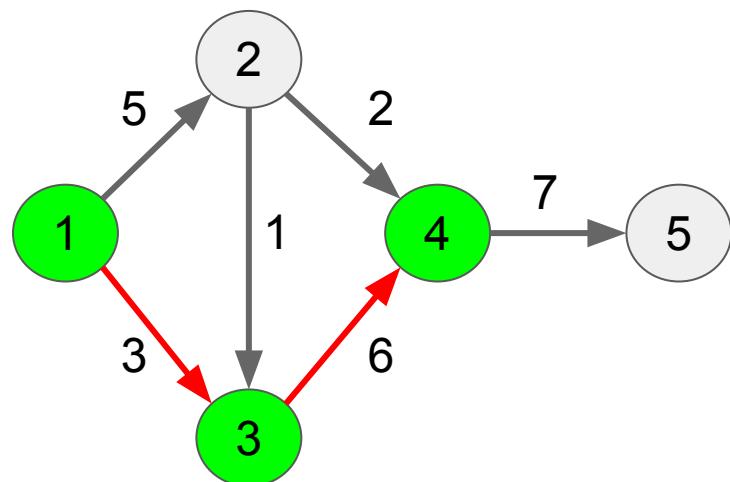
# Simulation of DP

Visiting

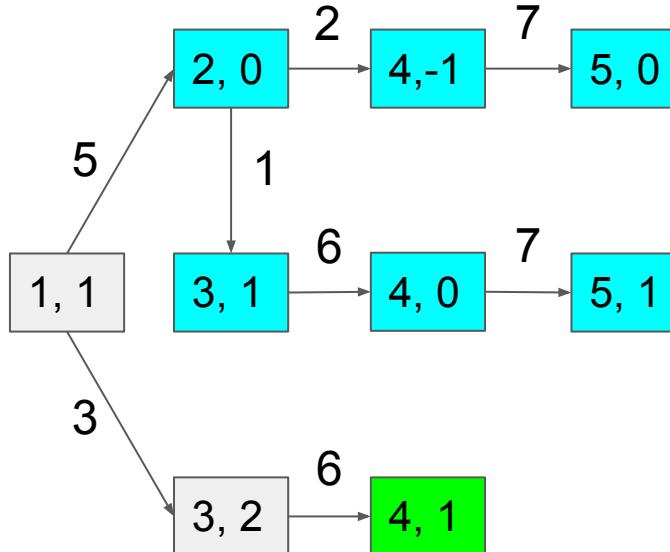
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

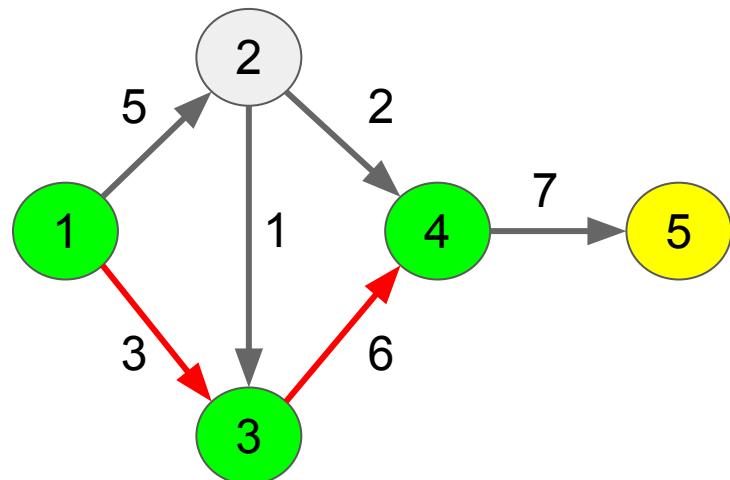
# Simulation of DP

Visiting

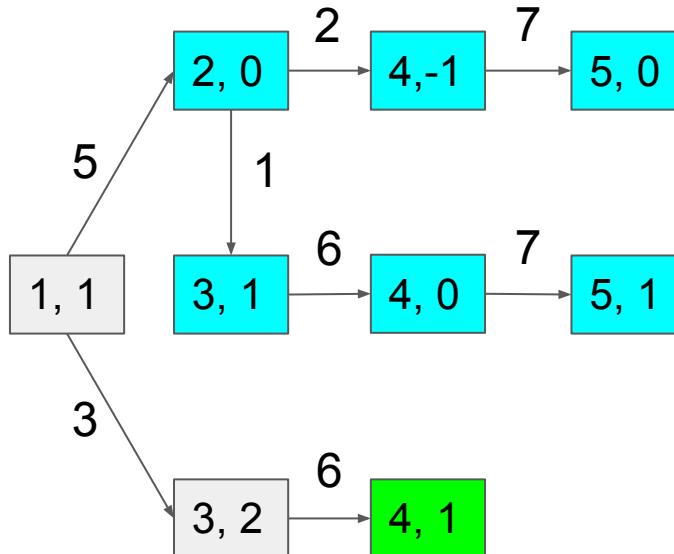
Successors

Completed

Regular Graph



State Graph



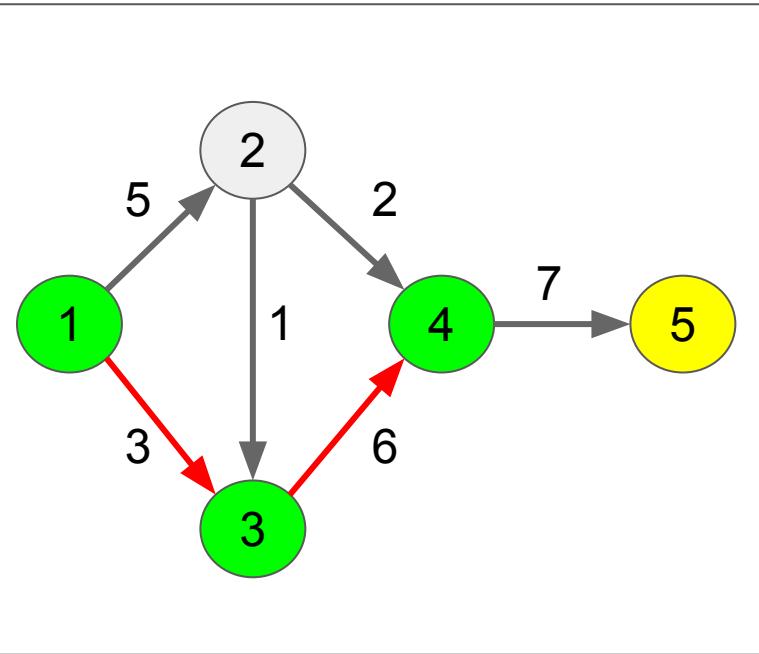
Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

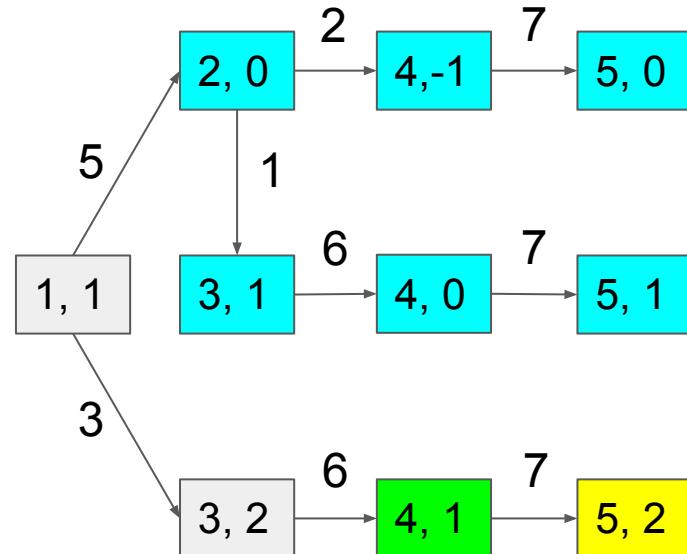
# Simulation of DP

Visiting	Successors	Completed
----------	------------	-----------

## Regular Graph



## State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

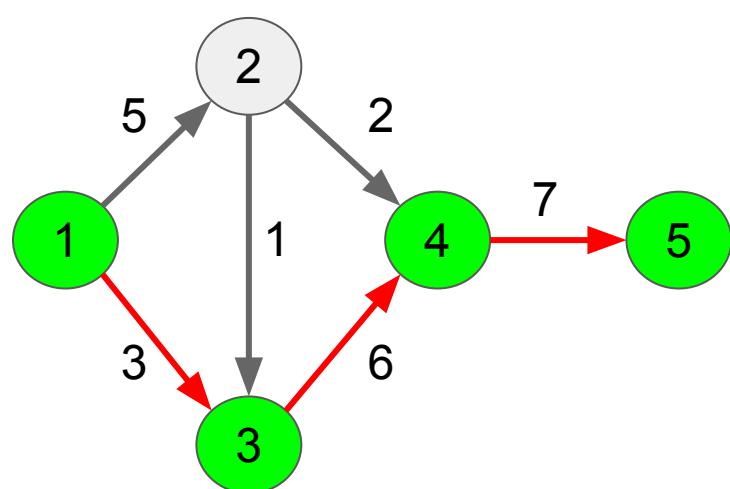
# Simulation of DP

Visiting

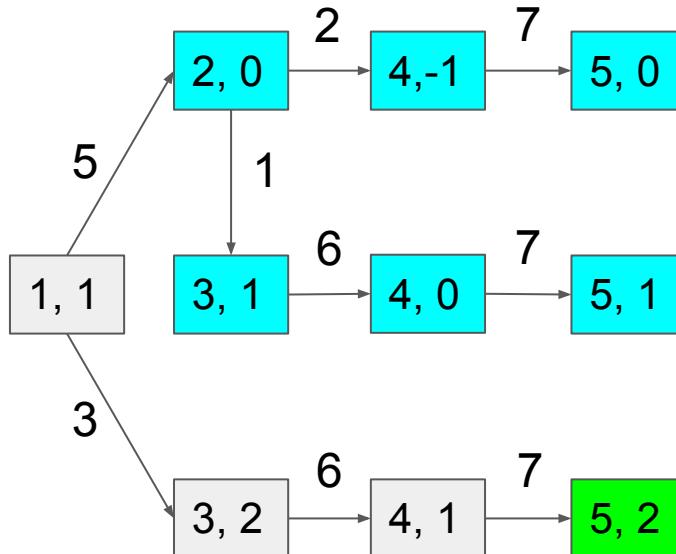
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14
(5, 2)	0

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

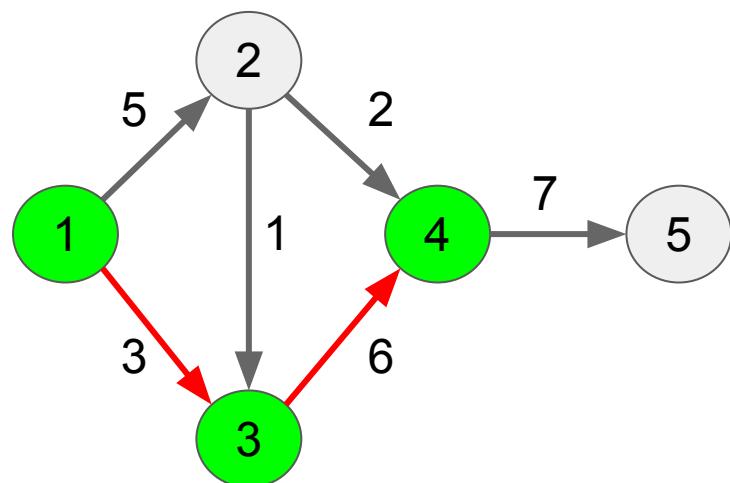
# Simulation of DP

Visiting

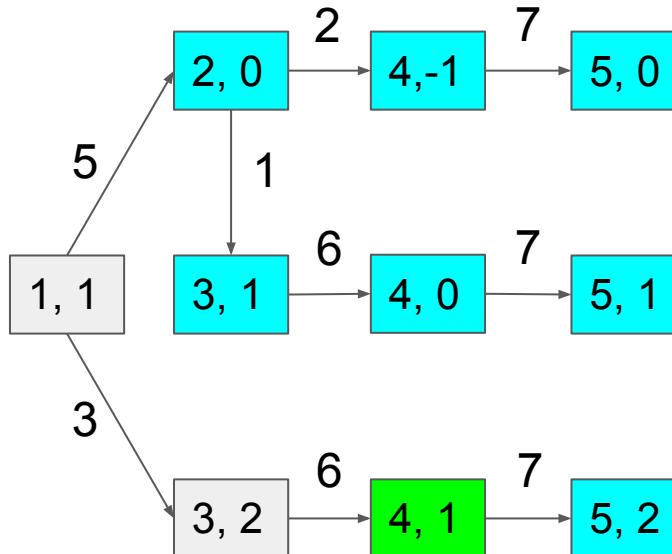
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14
(5, 2)	0
(4, 1)	7

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

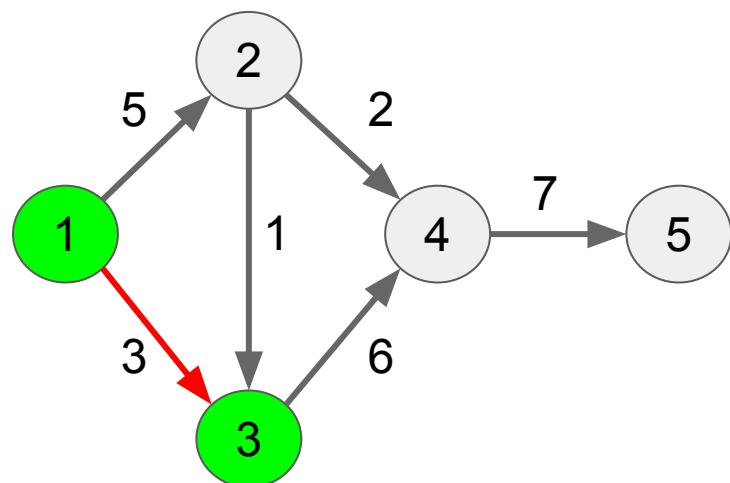
# Simulation of DP

Visiting

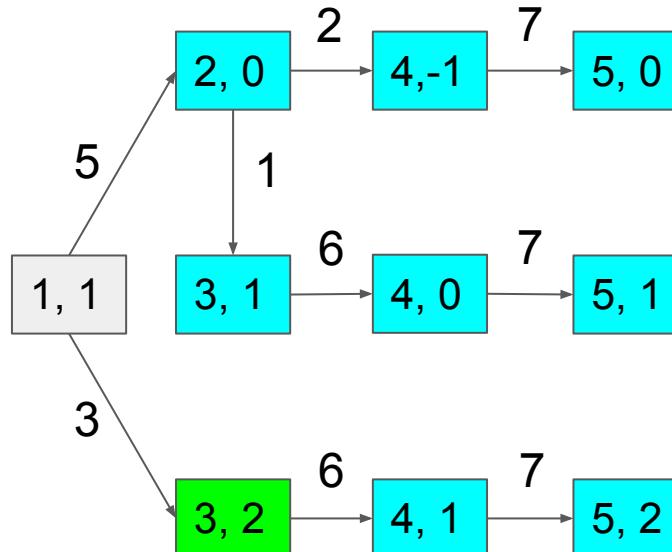
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14
(5, 2)	0
(4, 1)	7
(3, 2)	13

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

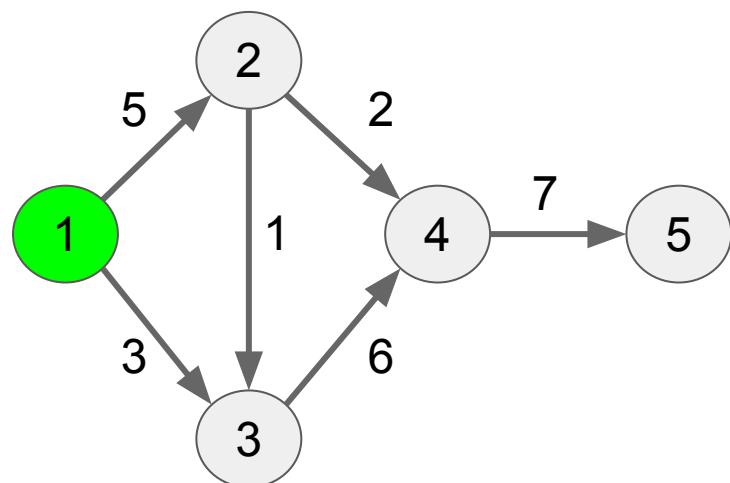
# Simulation of DP

Visiting

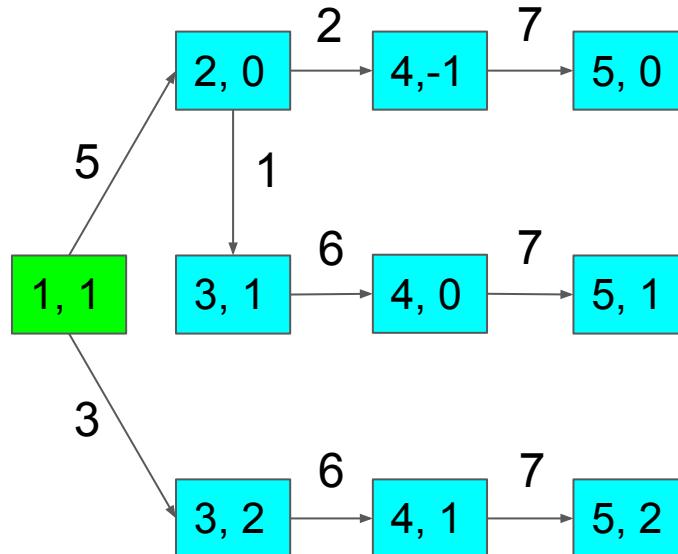
Successors

Completed

Regular Graph



State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14
(5, 2)	0
(4, 1)	7
(3, 2)	13
(1, 1)	16

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

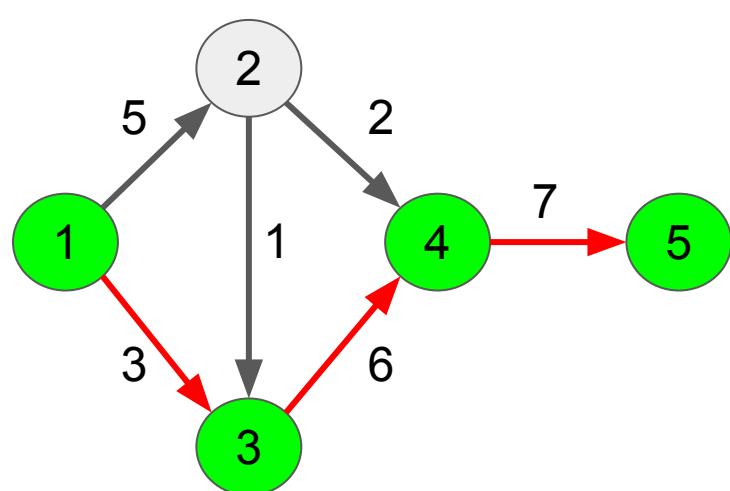
# Simulation of DP

Visiting

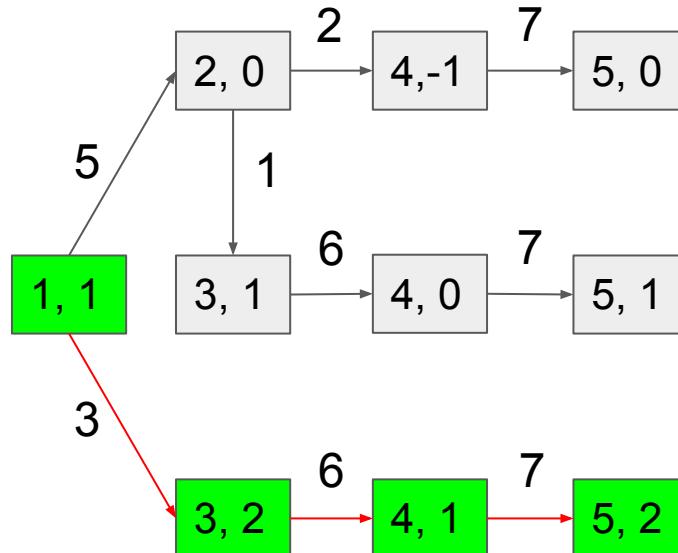
Successors

Completed

Regular Graph



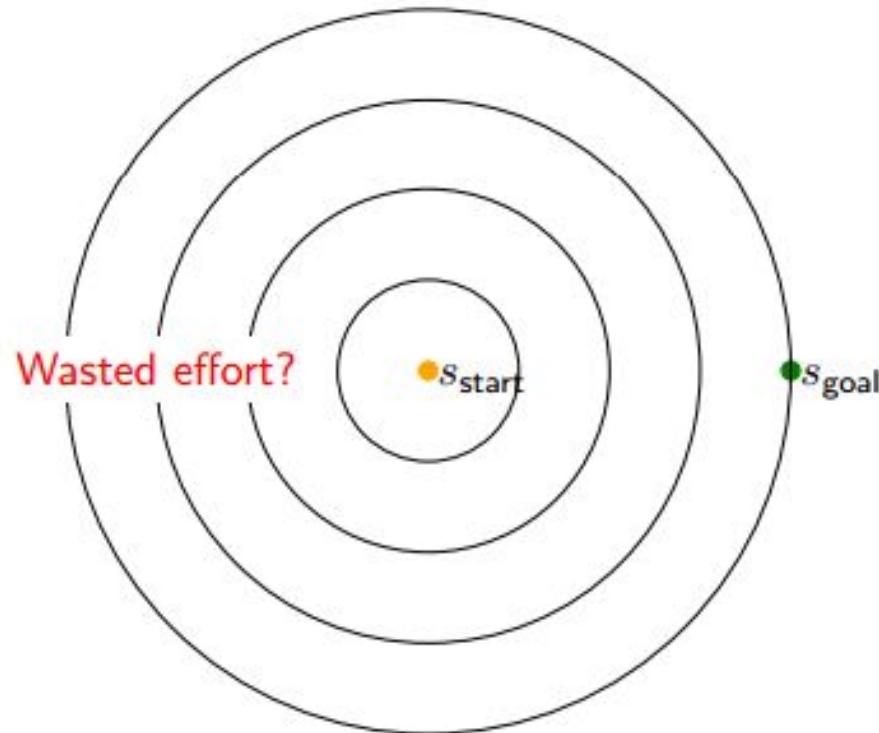
State Graph



Cache	
Key	Value
(5, 1)	0
(4, 0)	7
(3, 1)	13
(5, 0)	$\infty$
(4, -1)	$\infty$
(2, 0)	14
(5, 2)	0
(4, 1)	7
(3, 2)	13
(1, 1)	16

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsGoal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

# Improve UCS: A\* Search



# Contents

1. Uniform Cost Search
2. Defining States
3. Dynamic Programming
4. A\* Search

# Recap of A\* Search

- We want to avoid wasted effort (to go from SF to LA, we probably don't want to end up looking at roads to Seattle, for example).
- To do this, we can use a heuristic to estimate how far is left until we reach our goal.
- The heuristic **must be optimistic**. It must underestimate the true cost. Why?

# Recap of A\* Search

- Modify the cost of edges and run UCS on the new graph
  - New cost = Current cost + future cost
  - $\text{Cost}'(s, a) = \text{Cost}(s, a) + h(\text{Succ}(s, a)) - h(s)$
- You can find a good consistent  $h$  by performing relaxation.
- If  $c$  is min cost on original graph,  $c'$  is min cost on modified graph, then  $c' = c + h(s_{\text{goal}}) - h(s_{\text{start}})$

# Relaxation

A good way to come up with a reasonable heuristic is to solve an easier (less constrained) version of the problem

For example, we can use geographic distance as a heuristic for distance if we have the positions of nodes.

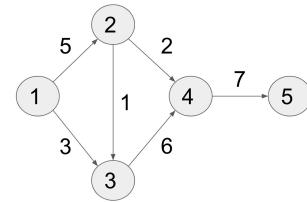
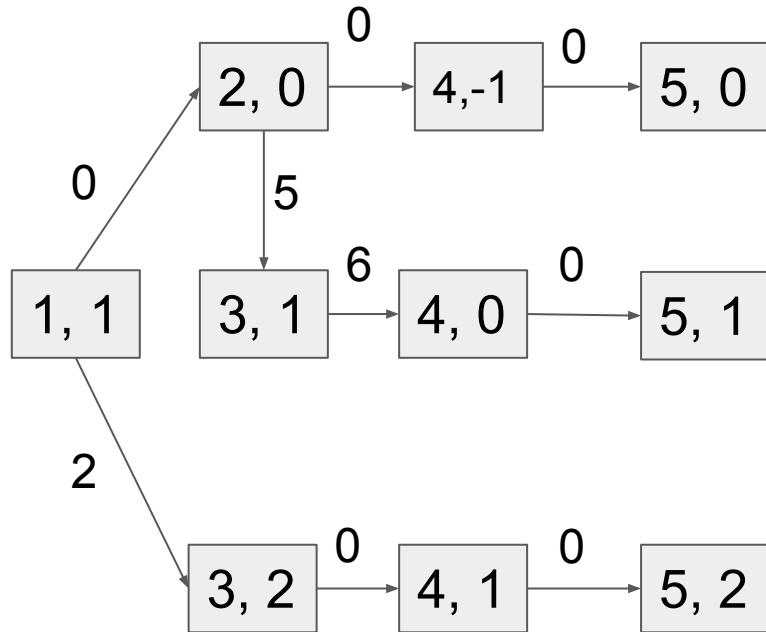
Note: The main point of relaxation is to attain a problem that **can be solved more efficiently**.

# How to compute h for our example?

Consider again our example from before. Suppose we ignore the constraint that there must be more odd cities visited. This is a relaxation of the problem. The following is h for our graph:

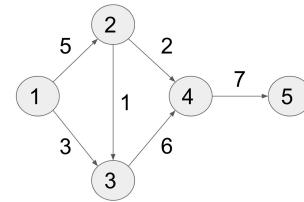
city	1	2	3	4	5
h	14	9	13	7	0

# Modified State Graph

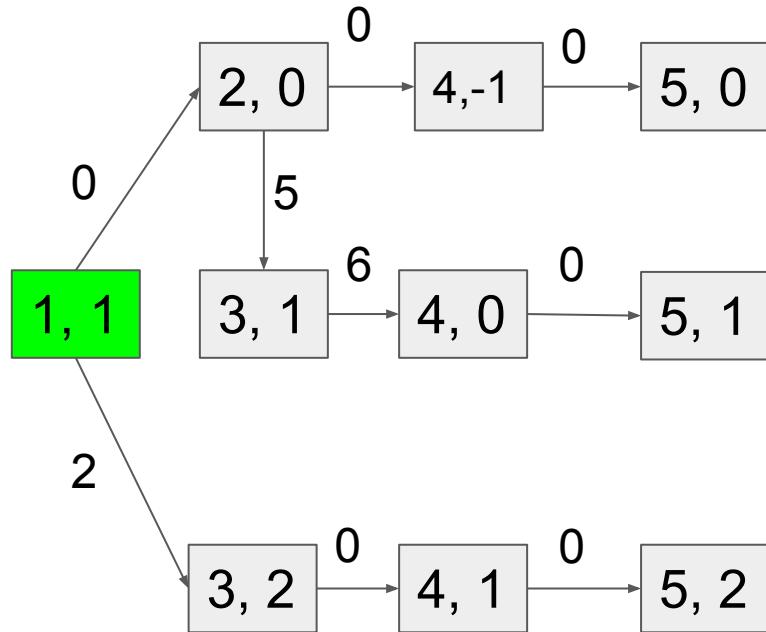


city	1	2	3	4	5
h	14	9	13	7	0

# Simulation of UCS (A\*)



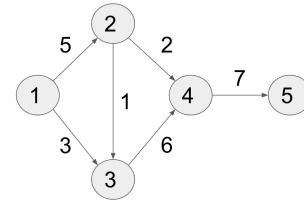
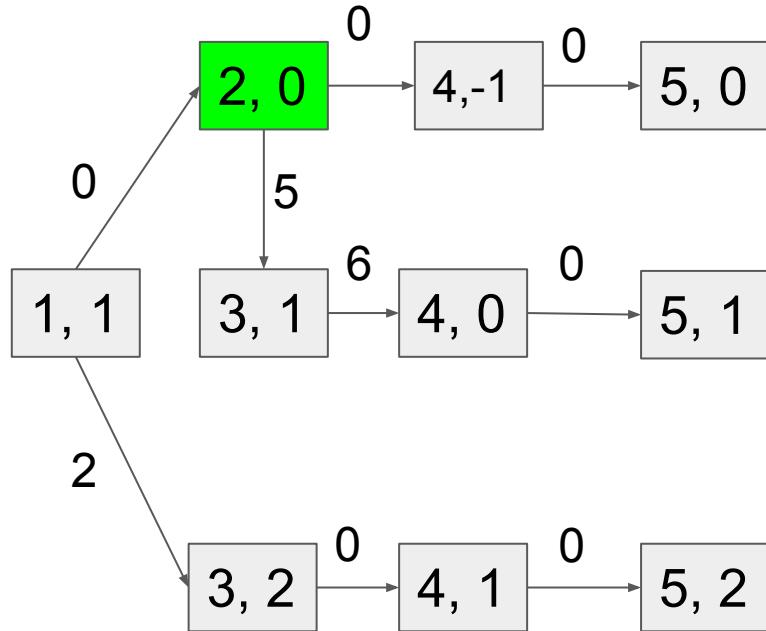
city	1	2	3	4	5
h	14	9	13	7	0



**Explored:**  
 $(1, 1) : 0$

**Frontier:**  
 $(2, 0) : 5 + 9 - 14 = 0$   
 $(3, 2) : 3 + 13 - 14 = 2$

# Simulation of UCS (A\*)



city	1	2	3	4	5
h	14	9	13	7	0

**Explored:**

(1, 1) : 0

(2, 0): 0

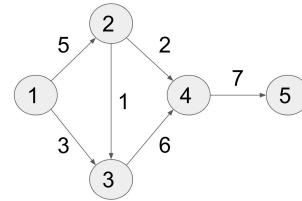
**Frontier:**

(3, 2) :  $3 + 13 - 14 = 2$

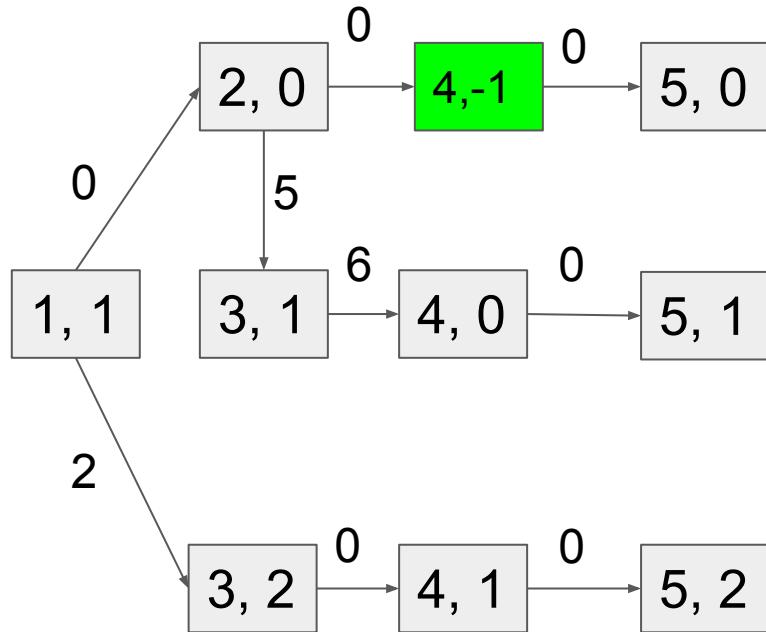
(3, 1):  $1 + 13 - 9 = 5$

(4, -1):  $2 + 7 - 9 = 0$

# Simulation of UCS (A\*)



city	1	2	3	4	5
h	14	9	13	7	0



**Explored:**

(1, 1) : 0

(2, 0): 0

(4, -1): 0

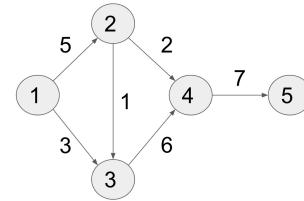
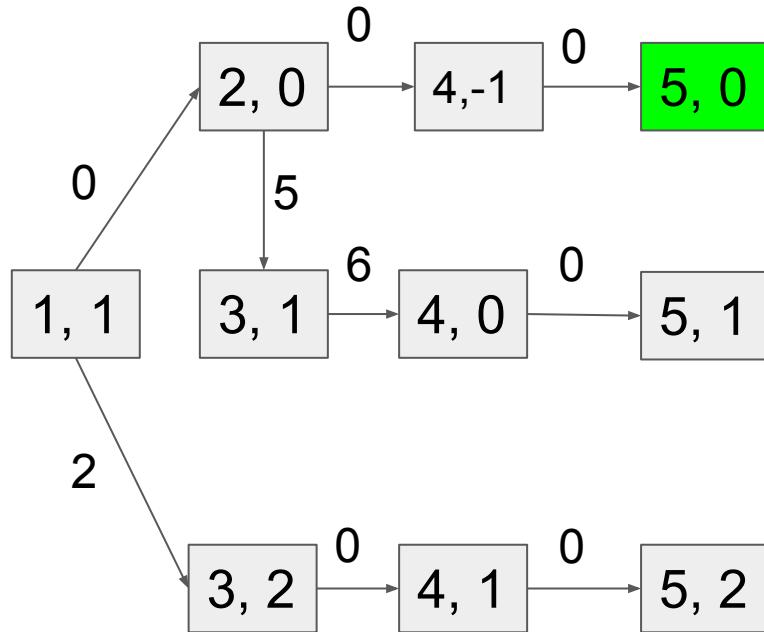
**Frontier:**

(3, 2) : 3 + 13 - 14 = 2

(3, 1): 1 + 13 - 9 = 5

(5, 0): 7 + 0 - 7 = 0

# Simulation of UCS (A\*)



city	1	2	3	4	5
h	14	9	13	7	0

**Explored:**

(1, 1) : 0

(2, 0): 0

(4, -1): 0

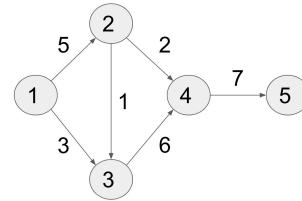
(5, 0): 0

**Frontier:**

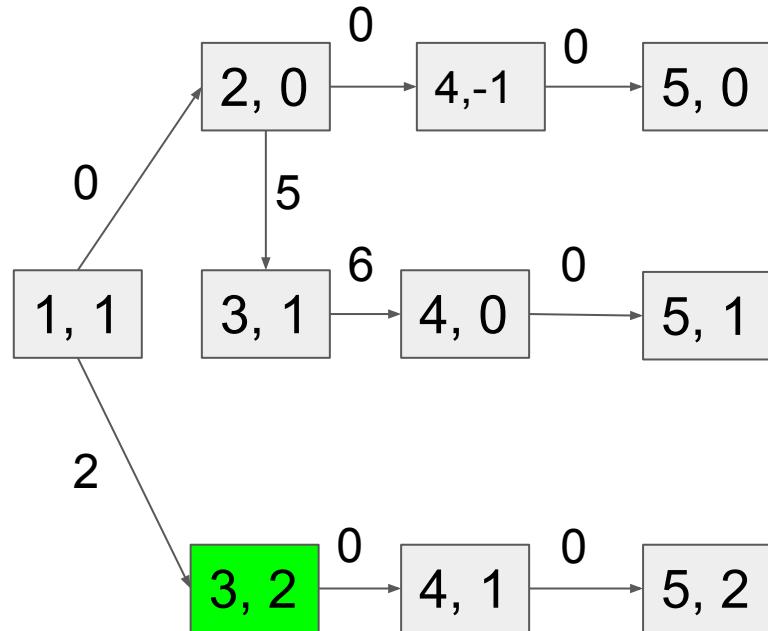
(3, 2) :  $3 + 13 - 14 = 2$

(3, 1):  $1 + 13 - 9 = 5$

# Simulation of UCS (A\*)



city	1	2	3	4	5
h	14	9	13	7	0



**Explored:**

(1, 1) : 0

(2, 0): 0

(4, -1): 0

(5, 0): 0

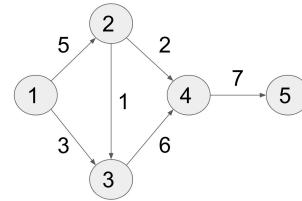
(3, 2): 2

**Frontier:**

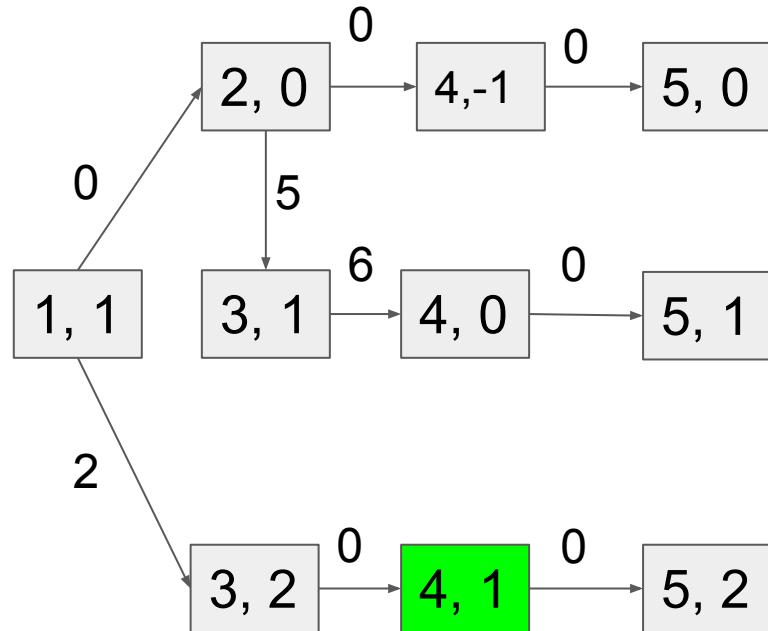
(3, 1):  $1 + 13 - 9 = 5$

(4, 1):  $6 + 7 - 13 = 0$

# Simulation of UCS (A\*)



city	1	2	3	4	5
h	14	9	13	7	0



# Explored:

(1, 1) : 0

(2, 0): 0

(4, -1): 0

(5, 0): 0

(3, 2): 2

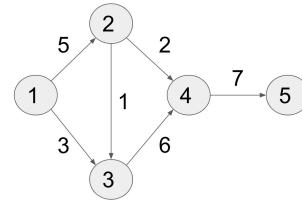
(4, 1): 0

# Frontier:

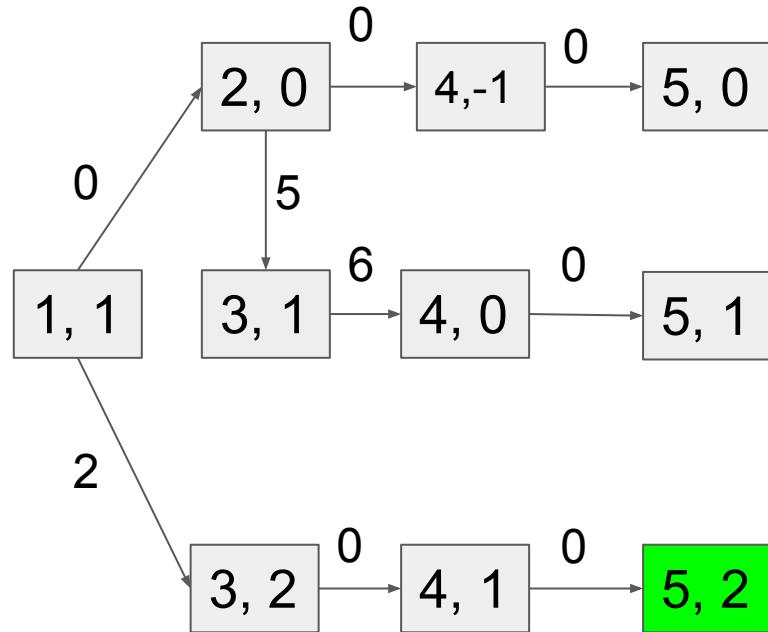
$$(3, 1): 1 + 13 - 9 = 5$$

$$(5, 2): 7 + 0 - 7 = 0$$

# Simulation of UCS (A\*)



city	1	2	3	4	5
h	14	9	13	7	0



**Explored:**

(1, 1) : 0

(2, 0): 0

(4, -1): 0

(5, 0): 0

(3, 2): 2

(4, 1): 0

(5, 2): 0

**Frontier:**

(3, 1):  $1 + 13 - 9 = 5$

**STOP!**

# Comparison of States visited

**UCS**

Explored:

(1, 1) : 0

(3, 2) : 3

(2, 0) : 5

(3, 1) : 6

(4, -1) : 7

(4, 1) : 9

(4, 0) : 12

(5, 0) : 14

(5, 2) : 16

Frontier:

(5, 1) : 19

**UCS(A\*)**

Explored:

(1, 1) : 0

(2, 0) : 0

(4, -1) : 0

(5, 0) : 0

(3, 2) : 2

(4, 1) : 0

(5, 2) : 0

Frontier:

(3, 1) : 5

# Summary

- States Representation/Modelling
  - make state representation as compact as possible, remove unnecessary information
- DP
  - underlying graph cannot have cycles
  - visit all reachable states, but no log overhead
- UCS
  - actions cannot have negative cost
  - visit only a subset of states, log overhead
- A\*
  - ensure that relaxed problem can be solved more efficiently



# Lecture 7: MDPs I





# Question

How would you get to Mountain View on Friday night in the least amount of time?

bike

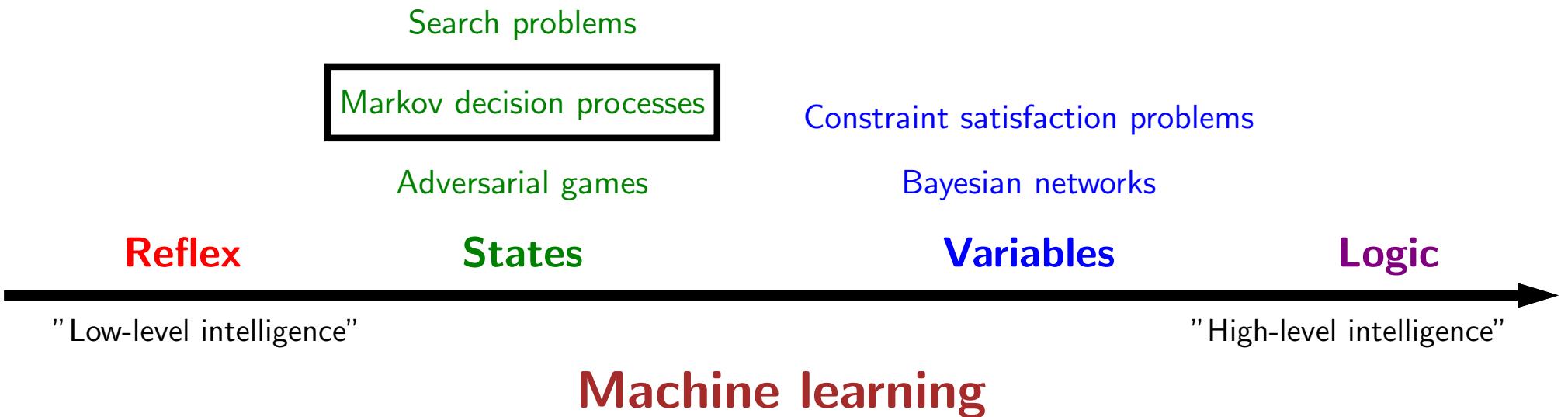
drive

Caltrain

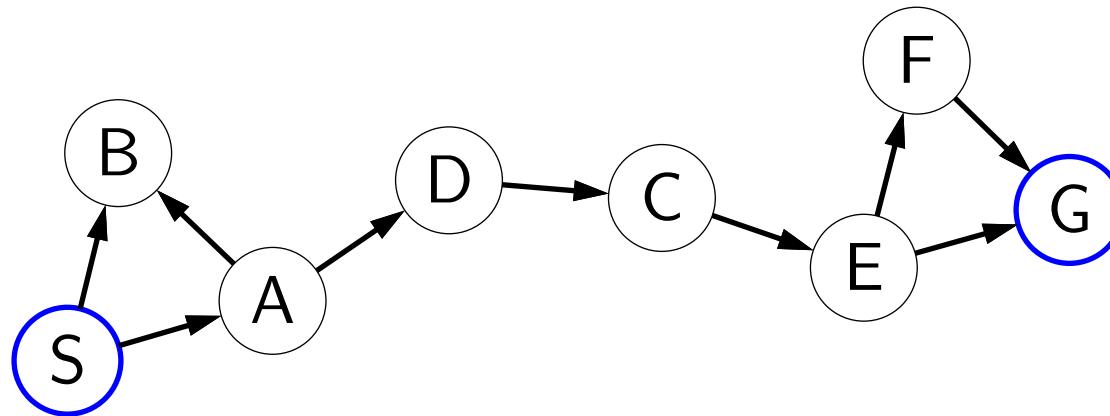
Uber/Lyft

fly

# Course plan



# So far: search problems



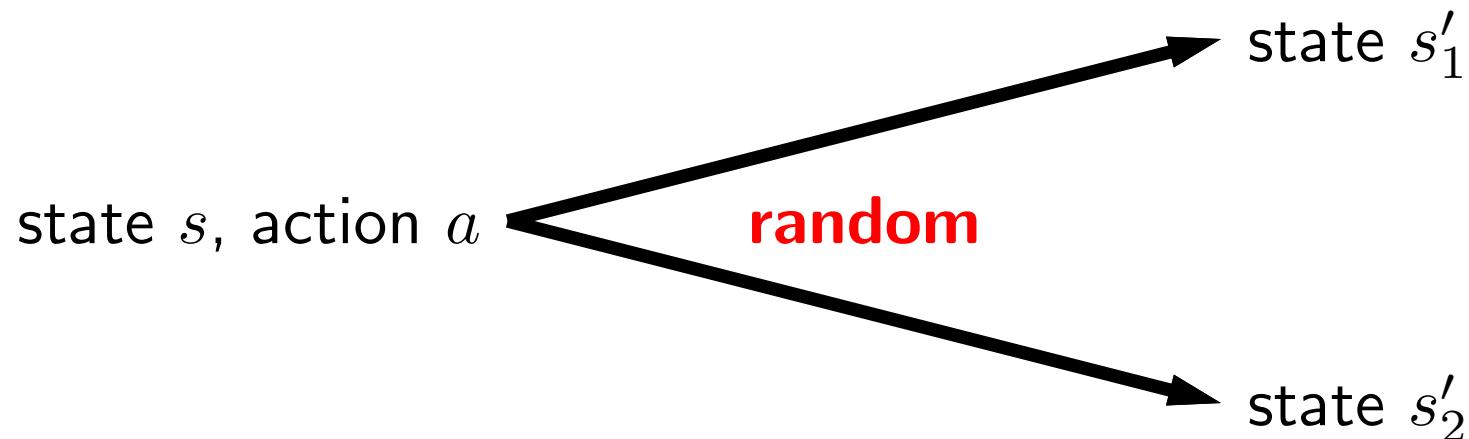
**deterministic**

state  $s$ , action  $a$   $\xrightarrow{\hspace{2cm}}$  state  $\text{Succ}(s, a)$



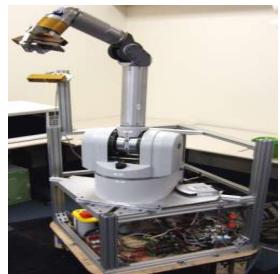
- Last week, we looked at search problems, a powerful paradigm that can be used to solve a diverse range of problems ranging from word segmentation to package delivery to route finding. The key was to cast whatever problem we were interested in solving into the problem of finding the minimum cost path in a graph.
- However, search problems assume that taking an action  $a$  from a state  $s$  results **deterministically** in a unique successor state  $\text{Succ}(s, a)$ .

# Uncertainty in the real world



- In the real world, the deterministic successor assumption is often unrealistic, for there is **randomness**: taking an action might lead to any one of many possible states.
- One deep question here is how we can even hope to act optimally in the face of randomness? Certainly we can't just have a single deterministic plan, and talking about a minimum cost path doesn't make sense.
- Today, we will develop tools to tackle this more challenging setting. We will fortunately still be able to reuse many of the intuitions about search problems, in particular the notion of a state.

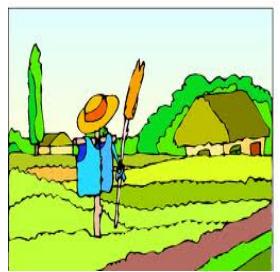
# Applications



**Robotics:** decide where to move, but actuators can fail, hit unseen obstacles, etc.



**Resource allocation:** decide what to produce, don't know the customer demand for various products



**Agriculture:** decide what to plant, but don't know weather and thus crop yield

- Randomness shows up in many places. They could be caused by limitations of the sensors and actuators of the robot (which we can control to some extent). Or they could be caused by market forces or nature, which we have no control over.
- We'll see that all of these sources of randomness can be handled in the same mathematical framework.

# Volcano crossing



**Run** (or press ctrl-enter)

		-50	20
		-50	
2			

- Let us consider an example. You are exploring a South Pacific island, which is modeled as a 3x4 grid of states. From each state, you can take one of four actions to move to an adjacent state: north (N), east (E), south (S), or west (W). If you try to move off the grid, you remain in the same state. You start at (2,1). If you end up in either of the green or red squares, your journey ends, either in a lava lake (reward of -50) or in a safe area with either no view (2) or a fabulous view of the island (20). What do you do?
- If we have a deterministic search problem, then the obvious thing will be to go for the fabulous view, which yields a reward of 20. You can set `numIters` to 10 and press Run. Each state is labeled with the maximum expected utility (sum of rewards) one can get from that state (analogue of FutureCost in a search problem). We will define this quantity formally later. For now, look at the arrows, which represent the best action to take from each cell. Note that in some cases, there is a tie for the best, where some of the actions seem to be moving in the wrong direction. This is because there is no penalty for moving around indefinitely. If you change `moveReward` to -0.1, then you'll see the arrows point in the right direction.
- In reality, we are dealing with treacherous terrain, and there is on each action a probability `slipProb` of slipping, which results in moving in a random direction. Try setting `slipProb` to various values. For small values (e.g., 0.1), the optimal action is to still go for the fabulous view. For large values (e.g., 0.3), then it's better to go for the safe and boring 2. Play around with the other reward values to get intuition for the problem.
- Important: note that we are only specifying the dynamics of the world, not directly specifying the best action to take. The best actions are computed automatically from the algorithms we'll see shortly.



# Roadmap

**Markov decision process**

Policy evaluation

Value iteration

# Dice game



## Example: dice game

For each round  $r = 1, 2, \dots$

- You choose **stay** or **quit**.
- If **quit**, you get \$10 and we end the game.
- If **stay**, you get \$4 and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.

Start

Stay

Quit

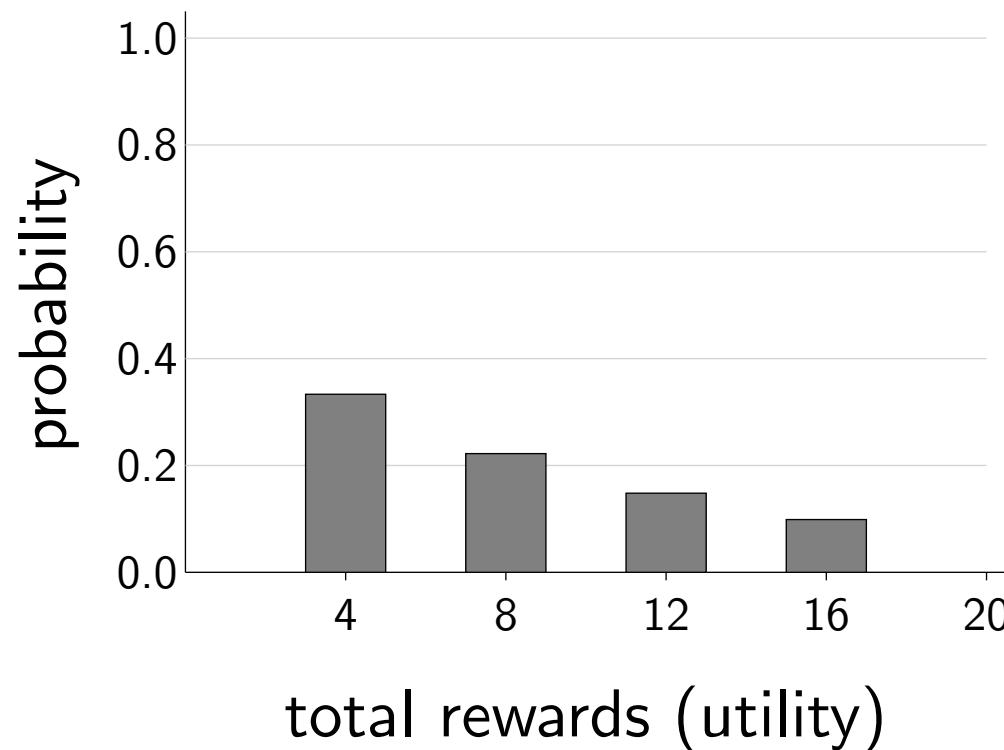
Dice:

Rewards:  
 0

- We'll see more volcanoes later, but let's start with a much simpler example: a dice game. What is the best strategy for this game?

# Rewards

If follow policy "stay":



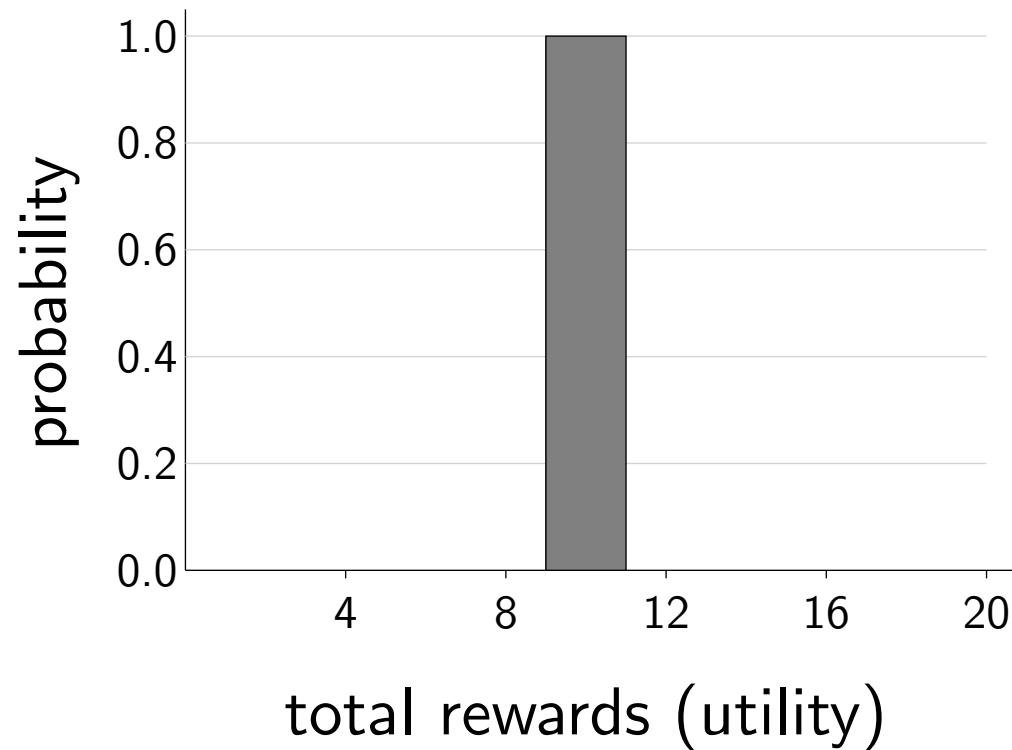
Expected utility:

$$\frac{1}{3}(4) + \frac{2}{3} \cdot \frac{1}{3}(8) + \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3}(12) + \dots = 12$$

- Let's suppose you always stay. Note that each outcome of the game will result in a different sequence of rewards, resulting in a **utility**, which is in this case just the sum of the rewards.
- We are interested in the **expected** utility, which you can compute to be 12.

# Rewards

If follow policy "quit":



Expected utility:

$$1(10) = 10$$

- If you quit, then you'll get a reward of 10 deterministically. Therefore, in expectation, the "stay" strategy is preferred, even though sometimes you'll get less than 10.

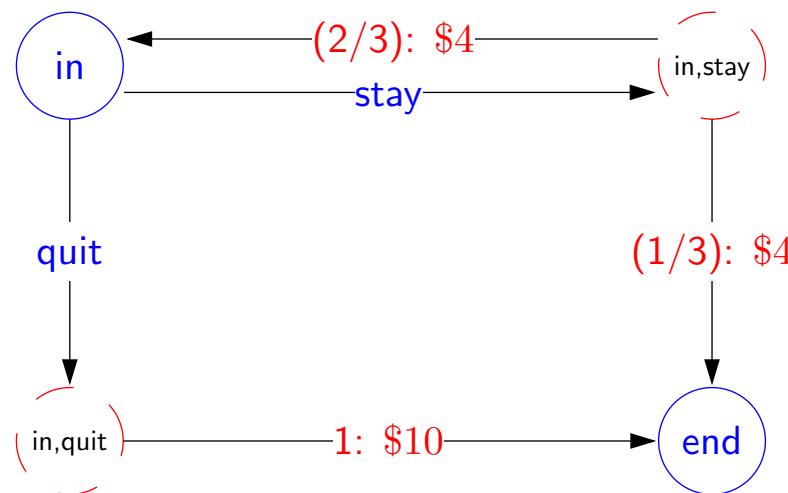
# MDP for dice game



## Example: dice game

For each round  $r = 1, 2, \dots$

- You choose **stay** or **quit**.
- If **quit**, you get \$10 and we end the game.
- If **stay**, you get \$4 and then I roll a 6-sided dice.
  - If the dice results in 1 or 2, we end the game.
  - Otherwise, continue to the next round.



- While we already solved this game directly, we'd like to develop a more general framework for thinking about not just this game, but also other problems such as the volcano crossing example. To that end, let us formalize the dice game as a **Markov decision process** (MDP).
- An MDP can be represented as a graph. The nodes in this graph include both **states** and **chance nodes**. Edges coming out of states are the possible actions from that state, which lead to chance nodes. Edges coming out of a chance nodes are the possible random outcomes of that action, which end up back in states. Our convention is to label these chance-to-state edges with the probability of a particular **transition** and the associated reward for traversing that edge.

# Markov decision process



## Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in \text{States}$ : starting state

$\text{Actions}(s)$ : possible actions from state  $s$

$T(s, a, s')$ : probability of  $s'$  if take action  $a$  in state  $s$

$\text{Reward}(s, a, s')$ : reward for the transition  $(s, a, s')$

$\text{IsEnd}(s)$ : whether at end of game

$0 \leq \gamma \leq 1$ : discount factor (default: 1)

- A **Markov decision process** has a set of states  $\text{States}$ , a starting state  $s_{\text{start}}$ , and the set of actions  $\text{Actions}(s)$  from each state  $s$ .
- It also has a **transition distribution**  $T$ , which specifies for each state  $s$  and action  $a$ , a distribution over possible successor states  $s'$ . Specifically, we have that  $\sum_{s'} T(s, a, s') = 1$  because  $T$  is a probability distribution (more on this later).
- Associated with each transition  $(s, a, s')$  is a reward, which could be either positive or negative.
- If we arrive in a state  $s$  for which  $\text{IsEnd}(s)$  is true, then the game is over.
- Finally, the discount factor  $\gamma$  is a quantity which specifies how much we value the future and will be discussed later.

# Search problems



## Definition: search problem

States: the set of states

$s_{\text{start}} \in \text{States}$ : starting state

$\text{Actions}(s)$ : possible actions from state  $s$

$\text{Succ}(s, a)$ : where we end up if take action  $a$  in state  $s$

$\text{Cost}(s, a)$ : cost for taking action  $a$  in state  $s$

$\text{IsEnd}(s)$ : whether at end

- $\text{Succ}(s, a) \Rightarrow T(s, a, s')$
- $\text{Cost}(s, a) \Rightarrow \text{Reward}(s, a, s')$

- MDPs share many similarities with search problems, but there are differences (one main difference and one minor one).
- The main difference is the move from a deterministic successor function  $\text{Succ}(s, a)$  to transition probabilities over  $s'$ . We can think of the successor function  $\text{Succ}(s, a)$  as a special case of transition probabilities:

$$T(s, a, s') = \begin{cases} 1 & \text{if } s' = \text{Succ}(s, a) \\ 0 & \text{otherwise} \end{cases}.$$

- A minor difference is that we've gone from minimizing costs to maximizing rewards. The two are really equivalent: you can negate one to get the other.

# Transitions



## Definition: transition probabilities

The **transition probabilities**  $T(s, a, s')$  specify the probability of ending up in state  $s'$  if taken action  $a$  in state  $s$ .



## Example: transition probabilities

$s$	$a$	$s'$	$T(s, a, s')$
in	quit	end	1
in	stay	in	2/3
in	stay	end	1/3

- Just to dwell on the major difference, transition probabilities, a bit more: for each state  $s$  and action  $a$ , the transition probabilities specifies a distribution over successor states  $s'$ .

# Probabilities sum to one



## Example: transition probabilities

$s$	$a$	$s'$	$T(s, a, s')$
in	quit	end	1
in	stay	in	2/3
in	stay	end	1/3

For each state  $s$  and action  $a$ :

$$\sum_{s' \in \text{States}} T(s, a, s') = 1$$

Successors:  $s'$  such that  $T(s, a, s') > 0$

- This means that for each given  $s$  and  $a$ , if we sum the transition probability  $T(s, a, s')$  over all possible successor states  $s'$ , we get 1.
- If a transition to a particular  $s'$  is not possible, then  $T(s, a, s') = 0$ . We refer to the  $s'$  for which  $T(s, a, s') > 0$  as the successors.
- Generally, the number of successors of a given  $(s, a)$  is much smaller than the total number of states. For instance, in a search problem, each  $(s, a)$  has exactly one successor.



# Transportation example



## Example: transportation

Street with blocks numbered 1 to  $n$ .

Walking from  $s$  to  $s + 1$  takes 1 minute.

Taking a magic tram from  $s$  to  $2s$  takes 2 minutes.

How to travel from 1 to  $n$  in the least time?

**Tram fails with probability 0.5.**

[semi-live solution]

- Let us revisit the transportation example. As we all know, magic trams aren't the most reliable forms of transportation, so let us assume that with probability  $\frac{1}{2}$ , it actually does as advertised, and with probability  $\frac{1}{2}$  it just leaves you in the same state.

# What is a solution?

Search problem: path (sequence of actions)

MDP:



## Definition: policy

A **policy**  $\pi$  is a mapping from each state  $s \in \text{States}$  to an action  $a \in \text{Actions}(s)$ .



## Example: volcano crossing

$s$	$\pi(s)$
(1,1)	S
(2,1)	E
(3,1)	N
...	...

- So we now know what an MDP is. What do we do with one? For search problems, we were trying to find the minimum cost **path**.
- However, fixed paths won't suffice for MDPs, because we don't know which states the random dice rolls are going to take us.
- Therefore, we define a **policy**, which specifies an action for every single state, not just the states along a path. This way, we have all our bases covered, and know what action to take no matter where we are.
- One might wonder if we ever need to take different actions from a given state. The answer is no, since like as in a search problem, the state contains all the information that we need to act optimally for the future. In more formal speak, the transitions and rewards satisfy the **Markov property**. Every time we end up in a state, we are faced with the exact same problem and therefore should take the same optimal action.



# Roadmap

Markov decision process

**Policy evaluation**

Value iteration

# Evaluating a policy

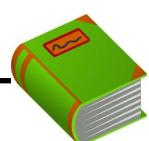


## Definition: utility

Following a policy yields a **random path**.

The **utility** of a policy is the (discounted) sum of the rewards on the path (this is a random quantity).

Path	Utility
[in; stay, 4, end]	4
[in; stay, 4, in; stay, 4, in; stay, 4, end]	12
[in; stay, 4, in; stay, 4, end]	8
[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]	16
...	...



## Definition: value (expected utility)

The **value** of a policy is the **expected utility**.

- Now that we've defined an MDP (the input) and a policy (the output), let's turn to defining the evaluation metric for a policy — there are many of them, which one should we choose?
- Recall that we'd like to maximize the total rewards (utility), but this is a random quantity, so we can't quite do that. Instead, we will instead maximize the **expected utility**, which we will refer to as **value** (of a policy).

# Evaluating a policy: volcano crossing

Run

(or press ctrl-enter)

$a$	$r$	$s$	
			(2,1)
E	-0.1	(2,2)	
S	-0.1	(3,2)	
E	-0.1	(3,3)	
E	-50.1	(2,3)	

2.4	-0.5	<b>-50</b>	<b>40</b>
3.7 →	5	<b>-50</b>	31
2	12.6 →	16.3 →	26.2

Value: 3.73

Utility: -36.79

- To get an intuitive feel for the relationship between a value and utility, consider the volcano example. If you press Run multiple times, you will get random paths shown on the right leading to different utilities. Note that there is considerable variation in what happens.
- The expectation of this utility is the **value**.
- You can run multiple simulations by increasing numEpisodes. If you set numEpisodes to 1000, then you'll see the average utility converging to the value.

# Discounting



## Definition: utility

Path:  $s_0, a_1 r_1 s_1, a_2 r_2 s_2, \dots$  (action, reward, new state).

The **utility** with discount  $\gamma$  is

$$u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \dots$$

Discount  $\gamma = 1$  (save for the future):

[stay, stay, stay, stay]:  $4 + 4 + 4 + 4 = 16$

Discount  $\gamma = 0$  (live in the moment):

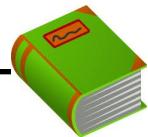
[stay, stay, stay, stay]:  $4 + 0 \cdot (4 + \dots) = 4$

Discount  $\gamma = 0.5$  (balanced life):

[stay, stay, stay, stay]:  $4 + \frac{1}{2} \cdot 4 + \frac{1}{4} \cdot 4 + \frac{1}{8} \cdot 4 = 7.5$

- There is an additional aspect to utility: **discounting**, which captures the fact that a reward today might be worth more than the same reward tomorrow. If the discount  $\gamma$  is small, then we favor the present more and downweight future rewards more.
- Note that the discounting parameter is applied exponentially to future rewards, so the distant future is always going to have a fairly small contribution to the utility (unless  $\gamma = 1$ ).
- The terminology, though standard, is slightly confusing: a larger value of the discount parameter  $\gamma$  actually means that the future is discounted less.

# Policy evaluation



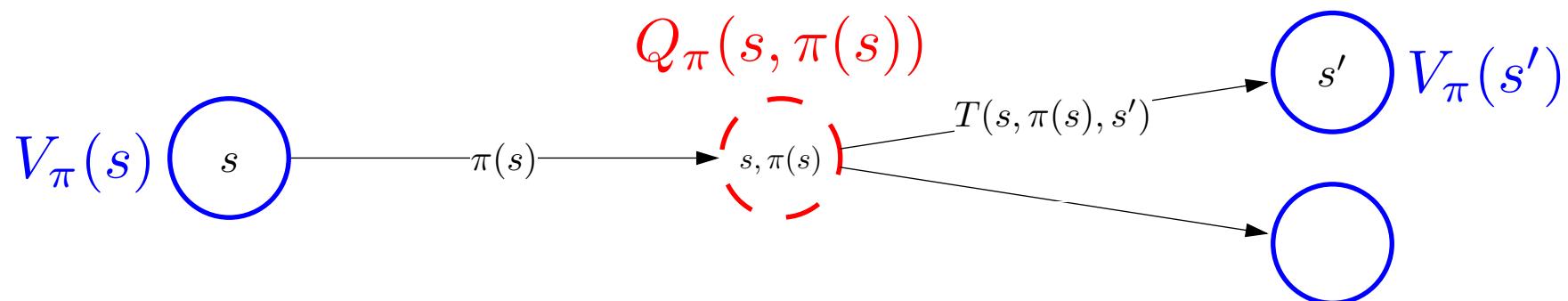
## Definition: value of a policy

Let  $V_\pi(s)$  be the expected utility received by following policy  $\pi$  from state  $s$ .



## Definition: Q-value of a policy

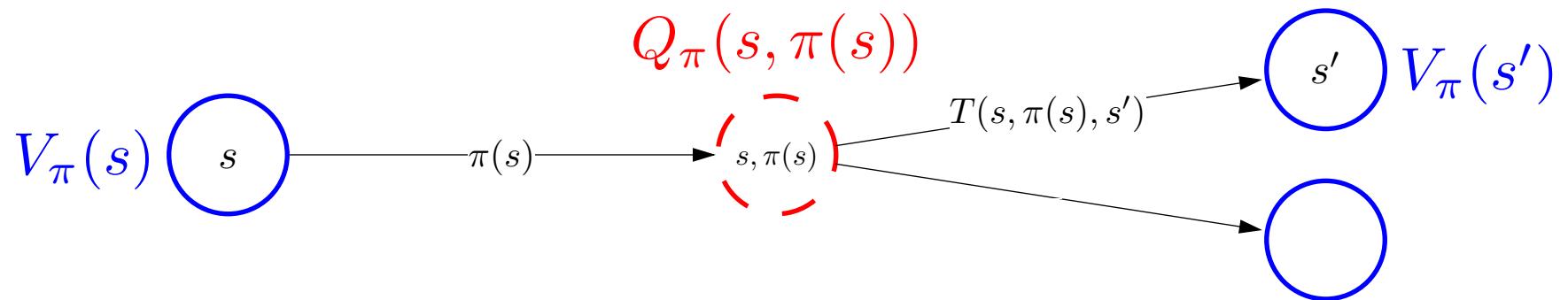
Let  $Q_\pi(s, a)$  be the expected utility of taking action  $a$  from state  $s$ , and then following policy  $\pi$ .



- Associated with any policy  $\pi$  are two important quantities, the value of the policy  $V_\pi(s)$  and the Q-value of a policy  $Q_\pi(s, a)$ .
- In terms of the MDP graph, one can think of the value  $V_\pi(s)$  as labeling the state nodes, and the Q-value  $Q_\pi(s, a)$  as labeling the chance nodes.
- This label refers to the expected utility if we were to start at that node and continue the dynamics of the game.

# Policy evaluation

Plan: define recurrences relating value and Q-value

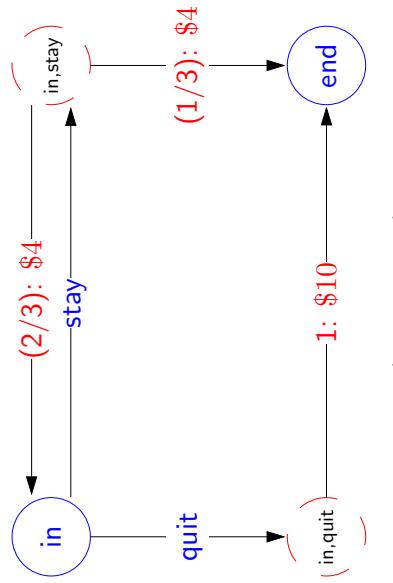


$$V_\pi(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

- We will now write down some equations relating value and Q-value. Our eventual goal is to get to an algorithm for computing these values, but as we will see, writing down the relationships gets us most of the way there, just as writing down the recurrence for FutureCost directly lead to a dynamic programming algorithm for acyclic search problems.
- First, we get  $V_\pi(s)$ , the value of a state  $s$ , by just following the action edge specified by the policy and taking the Q-value  $Q_\pi(s, \pi(s))$ . (There's also a base case where  $\text{IsEnd}(s)$ .)
- Second, we get  $Q_\pi(s, a)$  by considering all possible transitions to successor states  $s'$  and taking the expectation over the immediate reward  $\text{Reward}(s, a, s')$  plus the discounted future reward  $\gamma V_\pi(s')$ .
- While we've defined the recurrence for the expected utility directly, we can derive the recurrence by applying the law of total expectation and invoking the Markov property. To do this, we need to set up some random variables: Let  $s_0$  be the initial state,  $a_1$  be the action that we take,  $r_1$  be the reward we obtain, and  $s_1$  be the state we end up in. Also define  $u_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \dots$  to be the utility of following policy  $\pi$  from time step  $t$ . Then  $V_\pi(s) = \mathbb{E}[u_1 \mid s_0 = s]$ , which (assuming  $s$  is not an end state) in turn equals  $\sum_{s'} \mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)] \mathbb{E}[u_1 \mid s_1 = s', s_0 = s, a_1 = \pi(s)]$ . Note that  $\mathbb{P}[s_1 = s' \mid s_0 = s, a_1 = \pi(s)] = T(s, \pi(s), s')$ . Using the fact that  $u_1 = r_1 + \gamma u_2$  and taking expectations, we get that  $\mathbb{E}[u \mid s_1 = s', s_0 = s, a_1 = \pi(s)] = \text{Reward}(s, \pi(s), s') + \gamma V_\pi(s')$ . The rest follows from algebra.

## Dice game



Let  $\pi$  be the "stay" policy:  $\pi(\text{in}) = \text{stay}$ .

$$V_\pi(\text{end}) = 0$$

$$V_\pi(\text{in}) = \frac{1}{3}(4 + V_\pi(\text{end})) + \frac{2}{3}(4 + V_\pi(\text{in}))$$

In this case, can solve in closed form:

$$V_\pi(\text{in}) = 12$$

- As an example, let's compute the values of the nodes in the dice game for the policy "stay".
- Note that the recurrence involves both  $V_\pi(\text{in})$  on the left-hand side and the right-hand side. At least in this simple example, we can solve this recurrence easily to get the value.

# Policy evaluation



## Key idea: iterative algorithm

Start with arbitrary policy values and repeatedly apply recurrences to converge to true values.



## Algorithm: policy evaluation

Initialize  $V_{\pi}^{(0)}(s) \leftarrow 0$  for all states  $s$ .

For iteration  $t = 1, \dots, t_{PE}$ :

For each state  $s$ :

$$V_{\pi}^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_{\pi}^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$$

- But for a much larger MDP with 100000 states, how do we efficiently compute the value of a policy?
- One option is the following: observe that the recurrences define a system of linear equations, where the variables are  $V_\pi(s)$  for each state  $s$  and there is an equation for each state. So we could solve the system of linear equations by computing a matrix inverse. However, inverting a  $100000 \times 100000$  matrix is expensive in general.
- There is an even simpler approach called **policy evaluation**. We've already seen examples of iterative algorithms in machine learning: the basic idea is to start with something crude, and refine it over time.
- Policy iteration starts with a vector of all zeros for the initial values  $V_\pi^{(0)}$ . Each iteration, we loop over all the states and apply the two recurrences that we had before. The equations look hairier because of the superscript  $(t)$ , which simply denotes the value of at iteration  $t$  of the algorithm.

# Policy evaluation computation

$$V_{\pi}^{(t)}(s)$$

iteration  $t$

state $s$	0	-0.1	-0.2	0.7	1.1	1.6	1.9	2.2	2.4	2.6
0	0	-0.1	1.8	1.8	2.2	2.4	2.7	2.8	3	3.1
0	4	4	4	4	4	4	4	4	4	4
0	-0.1	1.8	1.8	2.2	2.4	2.7	2.8	3	3.1	
0	-0.1	-0.2	0.7	1.1	1.6	1.9	2.2	2.4	2.6	

- We can visualize the computation of policy evaluation on a grid, where column  $t$  denotes all the values  $V_{\pi}^{(t)}(s)$  for a given iteration  $t$ . The algorithm initializes the first column with 0 and then proceeds to update each subsequent column given the previous column.
- For those who are curious, the diagram shows policy evaluation on an MDP over 5 states where state 3 is a terminal state that delivers a reward of 4, and where there is a single action, MOVE, which transitions to an adjacent state (with wrap-around) with equal probability.

# Policy evaluation implementation

How many iterations ( $t_{PE}$ )? Repeat until values don't change much:

$$\max_{s \in \text{States}} |V_\pi^{(t)}(s) - V_\pi^{(t-1)}(s)| \leq \epsilon$$

Don't store  $V_\pi^{(t)}$  for each iteration  $t$ , need only last two:

$$V_\pi^{(t)} \text{ and } V_\pi^{(t-1)}$$

- Some implementation notes: a good strategy for determining how many iterations to run policy evaluation is based on how accurate the result is. Rather than set some fixed number of iterations (e.g, 100), we instead set an error tolerance (e.g.,  $\epsilon = 0.01$ ), and iterate until the maximum change between values of any state  $s$  from one iteration ( $t$ ) to the previous ( $t - 1$ ) is at most  $\epsilon$ .
- The second note is that while the algorithm is stated as computing  $V_{\pi}^{(t)}$  for each iteration  $t$ , we actually only need to keep track of the last two values. This is important for saving memory.

# Complexity



## Algorithm: policy evaluation

Initialize  $V_{\pi}^{(0)}(s) \leftarrow 0$  for all states  $s$ .

For iteration  $t = 1, \dots, t_{\text{PE}}$ :

For each state  $s$ :

$$V_{\pi}^{(t)}(s) \leftarrow \underbrace{\sum_{s'} T(s, \pi(s), s')[\text{Reward}(s, \pi(s), s') + \gamma V_{\pi}^{(t-1)}(s')]}_{Q^{(t-1)}(s, \pi(s))}$$

## MDP complexity

$S$  states

$A$  actions per state

$S'$  successors (number of  $s'$  with  $T(s, a, s') > 0$ )

Time:  $O(t_{\text{PE}} S S')$

- Computing the running time of policy evaluation is straightforward: for each of the  $t_{PE}$  iterations, we need to enumerate through each of the  $S$  states, and for each one of those, loop over the successors  $S'$ . Note that we don't have a dependence on the number of actions  $A$  because we have a fixed policy  $\pi(s)$  and we only need to look at the action specified by the policy.
- Advanced: Here, we have to iterate  $t_{PE}$  time steps to reach a target level of error  $\epsilon$ . It turns out that  $t_{PE}$  doesn't actually have to be very large for very small errors. Specifically, the error decreases exponentially fast as we increase the number of iterations. In other words, to cut the error in half, we only have to run a constant number of more iterations.
- Advanced: For acyclic graphs (for example, the MDP for Blackjack), we just need to do one iteration (not  $t_{PE}$ ) provided that we process the nodes in reverse topological order of the graph. This is the same setup as we had for dynamic programming in search problems, only the equations are different.

# Policy evaluation on dice game

Let  $\pi$  be the "stay" policy:  $\pi(\text{in}) = \text{stay}$ .

$$V_{\pi}^{(t)}(\text{end}) = 0$$

$$V_{\pi}^{(t)}(\text{in}) = \frac{1}{3}(4 + V_{\pi}^{(t-1)}(\text{end})) + \frac{2}{3}(4 + V_{\pi}^{(t-1)}(\text{in}))$$

$s$	end	in	$(t = 100 \text{ iterations})$
$V_{\pi}^{(t)}$	0.00	12.00	

Converges to  $V_{\pi}(\text{in}) = 12$ .

- Let us run policy evaluation on the dice game. The value converges very quickly to the correct answer.



# Summary so far

- MDP: graph with states, chance nodes, transition probabilities, rewards
- Policy: mapping from state to action (solution to MDP)
- Value of policy: expected utility over random paths
- Policy evaluation: iterative algorithm to compute value of policy

- Let's summarize: we have defined an MDP, which we should think of a graph where the nodes are states and chance nodes. Because of randomness, solving an MDP means generating policies, not just paths. A policy is evaluated based on its value: the expected utility obtained over random paths. Finally, we saw that policy evaluation provides a simple way to compute the value of a policy.



# Roadmap

Markov decision process

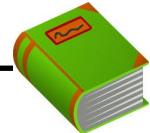
Policy evaluation

**Value iteration**

- If we are given a policy  $\pi$ , we now know how to compute its value  $V_\pi(s_{\text{start}})$ . So now, we could just enumerate all the policies, compute the value of each one, and take the best policy, but the number of policies is exponential in the number of states ( $A^S$  to be exact), so we need something a bit more clever.
- We will now introduce value iteration, which is an algorithm for finding the best policy. While evaluating a given policy and finding the best policy might seem very different, it turns out that value iteration will look a lot like policy evaluation.

# Optimal value and policy

Goal: try to get directly at maximum expected utility

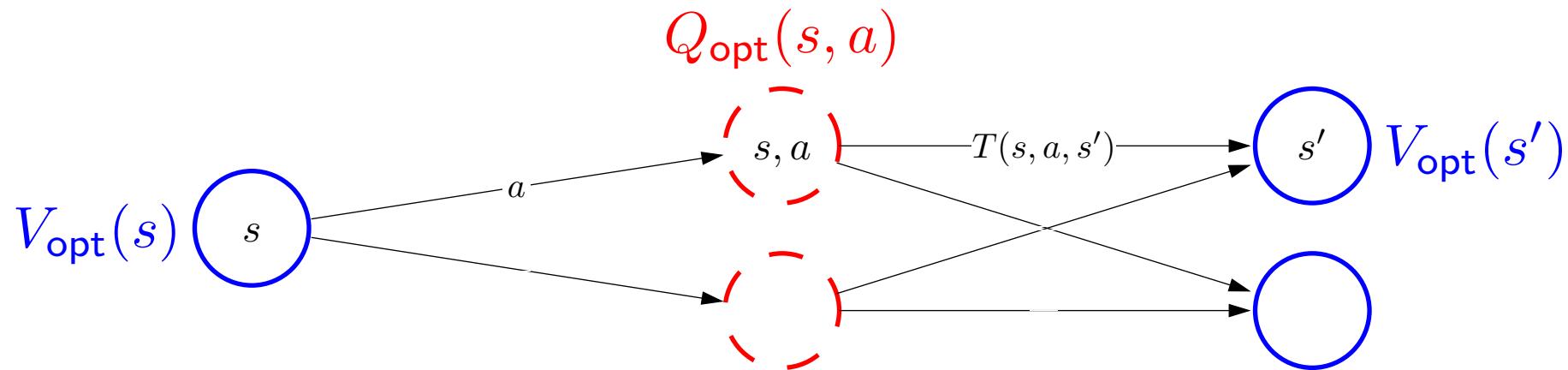


## Definition: optimal value

The **optimal value**  $V_{\text{opt}}(s)$  is the maximum value attained by any policy.

- We will write down a bunch of recurrences which look exactly like policy evaluation, but instead of having  $V_\pi$  and  $Q_\pi$  with respect to a fixed policy  $\pi$ , we will have  $V_{\text{opt}}$  and  $Q_{\text{opt}}$ , which are with respect to the optimal policy.

# Optimal values and Q-values



Optimal value if take action  $a$  in state  $s$ :

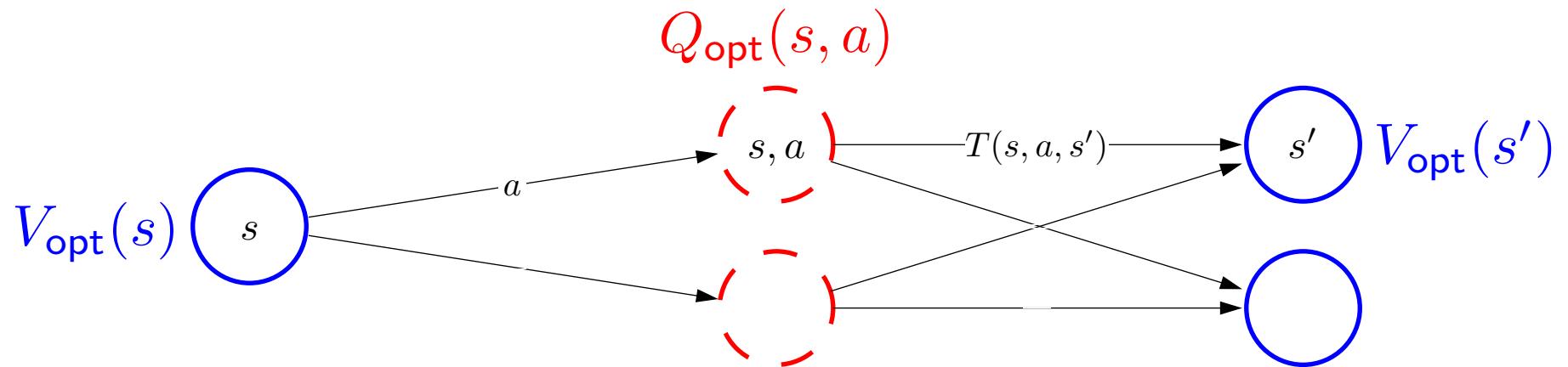
$$Q_{\text{opt}}(s, a) = \sum_{s'} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s')].$$

Optimal value from state  $s$ :

$$V_{\text{opt}}(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a) & \text{otherwise.} \end{cases}$$

- The recurrences for  $V_{\text{opt}}$  and  $Q_{\text{opt}}$  are identical to the ones for policy evaluation with one difference: in computing  $V_{\text{opt}}$ , instead of taking the action from the fixed policy  $\pi$ , we take the best action, the one that results in the largest  $Q_{\text{opt}}(s, a)$ .

# Optimal policies



Given  $Q_{\text{opt}}$ , read off the optimal policy:

$$\pi_{\text{opt}}(s) = \arg \max_{a \in \text{Actions}(s)} Q_{\text{opt}}(s, a)$$

- So far, we have focused on computing the value of the optimal policy, but what is the actual policy? It turns out that this is pretty easy to compute.
- Suppose you're at a state  $s$ .  $Q_{\text{opt}}(s, a)$  tells you the value of taking action  $a$  from state  $s$ . So the optimal action is simply to take the action  $a$  with the largest value of  $Q_{\text{opt}}(s, a)$ .

# Value iteration



## Algorithm: value iteration [Bellman, 1957]

Initialize  $V_{\text{opt}}^{(0)}(s) \leftarrow 0$  for all states  $s$ .

For iteration  $t = 1, \dots, t_{\text{VI}}$ :

For each state  $s$ :

$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \underbrace{\sum_{s'} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}^{(t-1)}(s')]}_{Q_{\text{opt}}^{(t-1)}(s, a)}$$

Time:  $O(t_{\text{VI}} S A S')$

[semi-live solution]

- By now, you should be able to go from recurrences to algorithms easily. Following the recipe, we simply iterate some number of iterations, go through each state  $s$  and then replace the equality in the recurrence with the assignment operator.
- Value iteration is also guaranteed to converge to the optimal value.
- What about the optimal policy? We get it as a byproduct. The optimal value  $V_{\text{opt}}(s)$  is computed by taking a max over actions. If we take the argmax, then we get the optimal policy  $\pi_{\text{opt}}(s)$ .

# Value iteration: dice game

$s$	end	in
$V_{\text{opt}}^{(t)}$	0.00	12.00 ( $t = 100$ iterations)
$\pi_{\text{opt}}(s)$	-	stay

- Let us demonstrate value iteration on the dice game. Initially, the optimal policy is "quit", but as we run value iteration longer, it switches to "stay".

# Value iteration: volcano crossing

Run

(or press ctrl-enter)

		-50	20
		-50	
2			

- As another example, consider the volcano crossing. Initially, the optimal policy and value correspond to going to the safe and boring 2. But as you increase numIters, notice how the value of the far away 20 propagates across the grid to the starting point.
- To see this propagation even more clearly, set slipProb to 0.

# Convergence



## Theorem: convergence

Suppose either

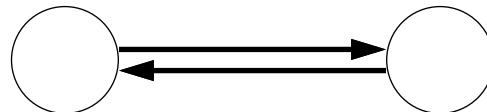
- discount  $\gamma < 1$ , or
- MDP graph is acyclic.

Then value iteration converges to the correct answer.



## Example: non-convergence

discount  $\gamma = 1$ , zero rewards



- Let us state more formally the conditions under which any of these algorithms that we talked about will work. A sufficient condition is that either the discount  $\gamma$  must be strictly less than 1 or the MDP graph is acyclic.
- We can reinterpret the discount  $\gamma < 1$  condition as introducing a new transition from each state to a special end state with probability  $(1 - \gamma)$ , multiplying all the other transition probabilities by  $\gamma$ , and setting the discount to 1. The interpretation is that with probability  $1 - \gamma$ , the MDP terminates at any state.
- In this view, we just need that a sampled path be finite with probability 1.
- We won't prove this theorem, but will instead give a counterexample to show that things can go badly if we have a cyclic graph and  $\gamma = 1$ . In the graph, whatever we initialize value iteration, value iteration will terminate immediately with the same value. In some sense, this isn't really the fault of value iteration, but it's because all paths are of infinite length. In some sense, if you were to simulate from this MDP, you would never terminate, so we would never find out what your utility was at the end.

# Summary of algorithms

- Policy evaluation:  $(\text{MDP}, \pi) \rightarrow V_\pi$
- Value iteration:  $\text{MDP} \rightarrow (V_{\text{opt}}, \pi_{\text{opt}})$

# Unifying idea

Algorithms:

- Search DP computes  $\text{FutureCost}(s)$
- Policy evaluation computes policy value  $V_\pi(s)$
- Value iteration computes optimal value  $V_{\text{opt}}(s)$

Recipe:

- Write down recurrence (e.g.,  $V_\pi(s) = \dots V_\pi(s') \dots$ )
- Turn into iterative algorithm (replace mathematical equality with assignment operator)

- There are two key ideas in this lecture. First, the policy  $\pi$ , value  $V_\pi$ , and Q-value  $Q_\pi$  are the three key quantities of MDPs, and they are related via a number of recurrences which can be easily gotten by just thinking about their interpretations.
- Second, given recurrences that depend on each other for the values you're trying to compute, it's easy to turn these recurrences into algorithms that iterate between those recurrences until convergence.



# Summary

- **Markov decision processes** (MDPs) cope with uncertainty
- Solutions are **policies** rather than paths
- **Policy evaluation** computes policy value (expected utility)
- **Value iteration** computes optimal value (maximum expected utility) and optimal policy
- Main technique: write recurrences → algorithm
- Next time: reinforcement learning — when we don't know rewards, transition probabilities



# Lecture 8: MDPs II





# Question

If you wanted to go from Orbisonia to Rockhill, how would you get there?

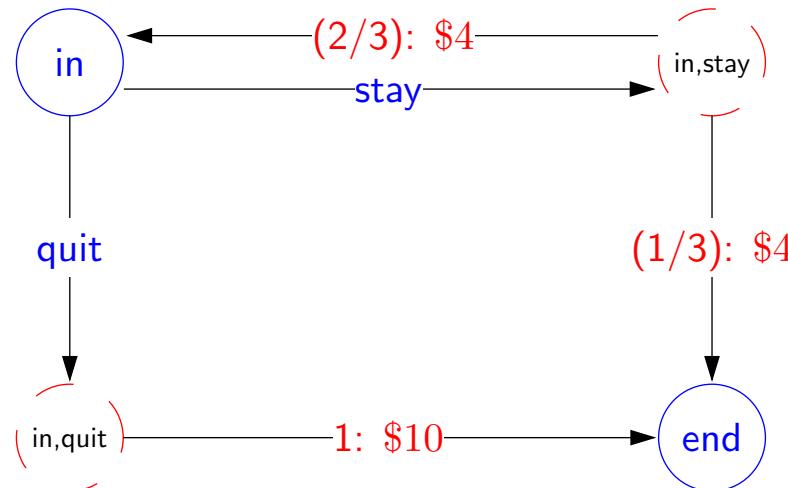
ride bus 1

ride bus 17

ride the magic tram

- In the previous lecture, you probably had some model of the world (how far Mountain View is, how long biking, driving, and Caltraining each take). But now, you should have no clue what's going on. This is the setting of **reinforcement learning**. Now, you just have to try things and learn from your experience - that's life!

# Review: MDPs



## Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in \text{States}$ : starting state

$\text{Actions}(s)$ : possible actions from state  $s$

$T(s, a, s')$ : probability of  $s'$  if take action  $a$  in state  $s$

$\text{Reward}(s, a, s')$ : reward for the transition  $(s, a, s')$

$\text{IsEnd}(s)$ : whether at end of game

$0 \leq \gamma \leq 1$ : discount factor (default: 1)

- Last time, we talked about MDPs, which we can think of as graphs, where each node is either a state  $s$  or a chance node  $(s, a)$ . Actions take us from states to chance nodes. This movement is something we can control. Transitions take us from chance nodes to states. This movement is random, and the various likelihoods are governed by transition probabilities.

# Review: MDPs

- Following a **policy**  $\pi$  produces a path (**episode**)

$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$

- **Value** function  $V_\pi(s)$ : expected utility if follow  $\pi$  from state  $s$

$$V_\pi(s) = \begin{cases} 0 & \text{if } \text{IsEnd}(s) \\ Q_\pi(s, \pi(s)) & \text{otherwise.} \end{cases}$$

- **Q-value** function  $Q_\pi(s, a)$ : expected utility if first take action  $a$  from state  $s$  and then follow  $\pi$

$$Q_\pi(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_\pi(s')]$$

- Given a policy  $\pi$  and an MDP, we can run the policy on the MDP yielding a sequence of states, action, rewards  $s_0; a_1, r_1, s_1; a_2, r_2, s_2; \dots$ . Formally, for each time step  $t$ ,  $a_t = \pi(s_{t-1})$ , and  $s_t$  is sampled with probability  $T(s_{t-1}, a_t, s_t)$ . We call such a sequence an **episode** (a path in the MDP graph). This will be a central notion in this lecture.
- Each episode (path) is associated with a **utility**, which is the discounted sum of rewards:  $u_1 = r_1 + \gamma r_2 + \gamma^2 r_3 + \dots$ . It's important to remember that the utility  $u_1$  is a **random variable** which depends on how the transitions were sampled.
- The value of the policy (from state  $s_0$ ) is  $V_\pi(s_0) = \mathbb{E}[u_1]$ , the expected utility. In the last lecture, we worked with the values directly without worrying about the underlying random variables (but that will soon no longer be the case). In particular, we defined recurrences relating the value  $V_\pi(s)$  and Q-value  $Q_\pi(s, a)$ , which represents the expected utility from starting at the corresponding nodes in the MDP graph.
- Given these mathematical recurrences, we produced algorithms: policy evaluation computes the value of a policy, and value iteration computes the optimal policy.

# Unknown transitions and rewards



## Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in \text{States}$ : starting state

$\text{Actions}(s)$ : possible actions from state  $s$

$\text{IsEnd}(s)$ : whether at end of game

$0 \leq \gamma \leq 1$ : discount factor (default: 1)

**reinforcement learning!**

- In this lecture, we assume that we have an MDP where we neither know the transitions nor the reward functions. We are still trying to maximize expected utility, but we are in a much more difficult setting called **reinforcement learning**.

# Mystery game



## Example: mystery buttons

For each round  $r = 1, 2, \dots$

- You choose A or B.
- You move to a new state and get some rewards.

Start

A

B

State: 5,0

Rewards: 0

- To put yourselves in the shoes of a reinforcement learner, try playing the game. You can either push the A button or the B button. Each of the two actions will take you to a new state and give you some reward.
- This simple game illustrates some of the challenges of reinforcement learning: we should take good actions to get rewards, but in order to know which actions are good, we need to explore and try different actions.



# Roadmap

**Reinforcement learning**

Monte Carlo methods

Bootstrapping methods

Covering the unknown

Summary

# From MDPs to reinforcement learning



## Markov decision process (offline)

- Have mental model of how the world works.
- Find policy to collect maximum rewards.

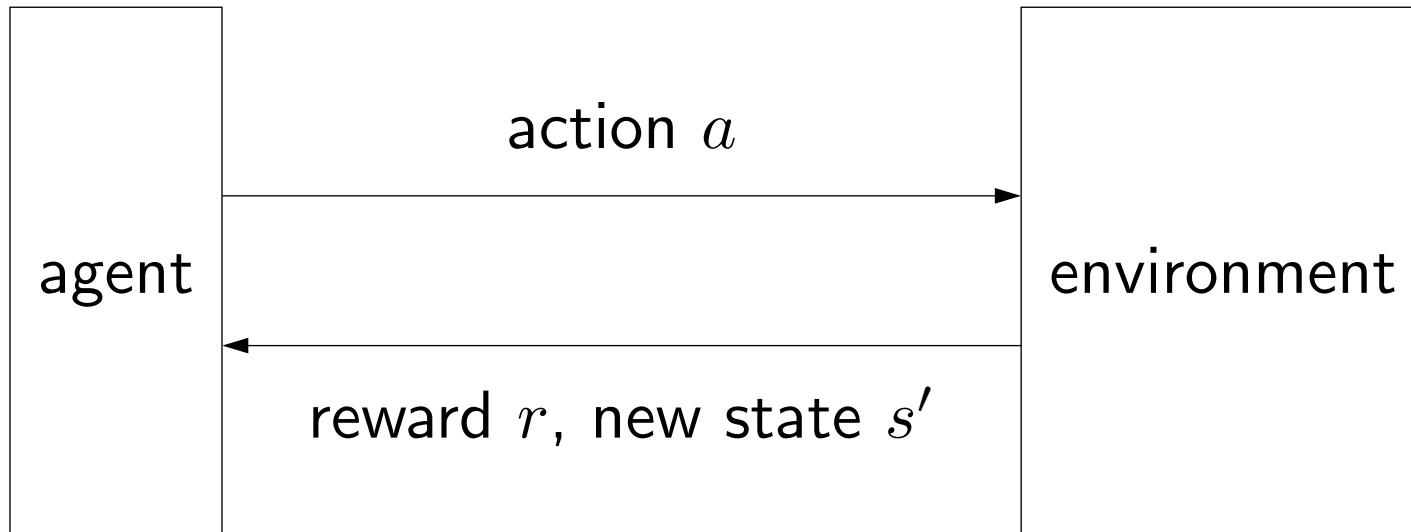


## Reinforcement learning (online)

- Don't know how the world works.
- Perform actions in the world to find out and collect rewards.

- An important distinction between solving MDPs (what we did before) and reinforcement learning (what we will do now) is that the former is **offline** and the latter is **online**.
- In the former case, you have a mental model of how the world works. You go lock yourself in a room, think really hard, come up with a policy. Then you come out and use it to act in the real world.
- In the latter case, you don't know how the world works, but you only have one life, so you just have to go out into the real world and learn how it works from experiencing it and trying to take actions that yield high rewards.
- At some level, reinforcement learning is really the way humans work: we go through life, taking various actions, getting feedback. We get rewarded for doing well and learn along the way.

# Reinforcement learning framework



## Algorithm: reinforcement learning template

For  $t = 1, 2, 3, \dots$

Choose action  $a_t = \pi_{\text{act}}(s_{t-1})$  (**how?**)

Receive reward  $r_t$  and observe new state  $s_t$

Update parameters (**how?**)

- To make the framework clearer, we can think of an **agent** (the reinforcement learning algorithm) that repeatedly chooses an action  $a_t$  to perform in the environment, and receives some reward  $r_t$ , and information about the new state  $s_t$ .
- There are two questions here: how to choose actions (what is  $\pi_{\text{act}}$ ) and how to update the parameters. We will first talk about updating parameters (the learning part), and then come back to action selection later.

# Volcano crossing



Run (or press ctrl-enter)

		-50	20
		-50	
2			

Utility: 2

*a r s*  
(2,1)  
W 0 (2,1)  
W 0 (2,1)  
N 0 (1,1)  
W 0 (1,1)  
N 0 (1,1)  
E 0 (1,2)  
S 0 (2,2)  
W 0 (2,1)  
N 0 (2,2)  
N 0 (3,2)  
S 0 (3,2)  
W 2 (3,1)

- Recall the volcano crossing example from the previous lecture. Each square is a state. From each state, you can take one of four actions to move to an adjacent state: north (N), east (E), south (S), or west (W). If you try to move off the grid, you remain in the same state. The starting state is (2,1), and the end states are the four marked with red or green rewards. Transitions from  $(s, a)$  lead where you expect with probability `1-slipProb` and to a random adjacent square with probability `slipProb`.
- If we solve the MDP using value iteration (by setting `numIters` to 10), we will find the best policy (which is to head for the 20). Of course, we can't solve the MDP if we don't know the transitions or rewards.
- If you set `numIters` to zero, we start off with a random policy. Try pressing the Run button to generate fresh episodes. How can we learn from this data and improve our policy?



# Roadmap

Reinforcement learning

**Monte Carlo methods**

Bootstrapping methods

Covering the unknown

Summary

# Model-based Monte Carlo

Data:  $s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$



**Key idea: model-based learning**

Estimate the MDP:  $T(s, a, s')$  and  $\text{Reward}(s, a, s')$

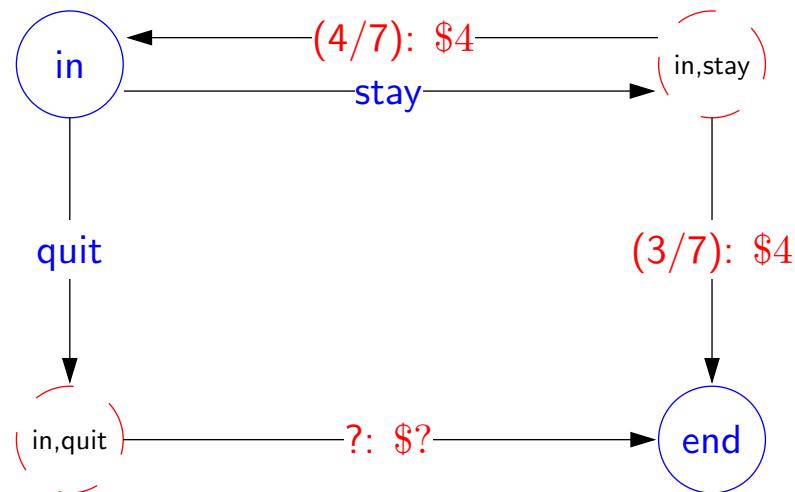
Transitions:

$$\hat{T}(s, a, s') = \frac{\# \text{ times } (s, a, s') \text{ occurs}}{\# \text{ times } (s, a) \text{ occurs}}$$

Rewards:

$$\widehat{\text{Reward}}(s, a, s') = r \text{ in } (s, a, r, s')$$

# Model-based Monte Carlo



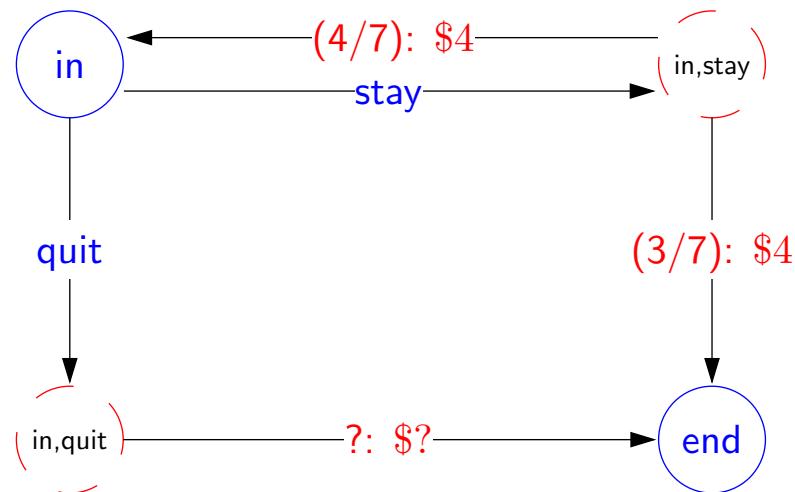
Data (following policy  $\pi(s) = \text{stay}$ ):

[in; stay, 4, end]

- Estimates converge to true values (under certain conditions)
- With estimated MDP  $(\hat{T}, \widehat{\text{Reward}})$ , compute policy using value iteration

- The first idea is called **model-based** Monte Carlo, where we try to estimate the model (transitions and rewards) using Monte Carlo simulation.
- Monte Carlo is a standard way to estimate the expectation of a random variable by taking an average over samples of that random variable.
- Here, the data used to estimate the model is the sequence of states, actions, and rewards in the episode. Note that the samples being averaged are not independent (because they come from the same episode), but they do come from a Markov chain, so it can be shown that these estimates converge to the expectations by the ergodic theorem (a generalization of the law of large numbers for Markov chains).
- But there is one important caveat...

# Problem



Problem: won't even see  $(s, a)$  if  $a \neq \pi(s)$  ( $a = \text{quit}$ )



**Key idea: exploration**

To do reinforcement learning, need to explore the state space.

Solution: need  $\pi$  to **explore** explicitly (more on this later)

- So far, our policies have been deterministic, mapping  $s$  always to  $\pi(s)$ . However, if we use such a policy to generate our data, there are certain  $(s, a)$  pairs that we will never see and therefore never be able to estimate their Q-value and never know what the effect of those actions are.
- This problem points at the most important characteristic of reinforcement learning, which is the need for **exploration**. This distinguishes reinforcement learning from supervised learning, because now we actually have to act to get data, rather than just having data poured over us.
- To close off this point, we remark that if  $\pi$  is a non-deterministic policy which allows us to explore each state and action infinitely often (possibly over multiple episodes), then the estimates of the transitions and rewards will converge.
- Once we get an estimate for the transitions and rewards, we can simply plug them into our MDP and solve it using standard value or policy iteration to produce a policy.
- Notation: we put hats on quantities that are estimated from data  $(\hat{Q}_{\text{opt}}, \hat{T})$  to distinguish from the true quantities  $(Q_{\text{opt}}, T)$ .

# From model-based to model-free

$$\hat{Q}_{\text{opt}}(s, a) = \sum_{s'} \hat{T}(s, a, s') [\widehat{\text{Reward}}(s, a, s') + \gamma \hat{V}_{\text{opt}}(s')]$$

All that matters for prediction is (estimate of)  $Q_{\text{opt}}(s, a)$ .



**Key idea: model-free learning**

Try to estimate  $Q_{\text{opt}}(s, a)$  directly.

- Taking a step back, if our goal is to just find good policies, all we need is to get a good estimate of  $\hat{Q}_{\text{opt}}$ . From that perspective, estimating the model (transitions and rewards) was just a means towards an end. Why not just cut to the chase and estimate  $\hat{Q}_{\text{opt}}$  directly? This is called **model-free** learning, where we don't explicitly estimate the transitions and rewards. Model-free learning is the equivalent of learning the expected utility at chance nodes instead of costs as edge weights like we were doing in the previous slide.

# Model-free Monte Carlo

Data (following policy  $\pi$ ):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

Recall:

$Q_\pi(s, a)$  is expected utility starting at  $s$ , first taking action  $a$ , and then following policy  $\pi$

Utility:

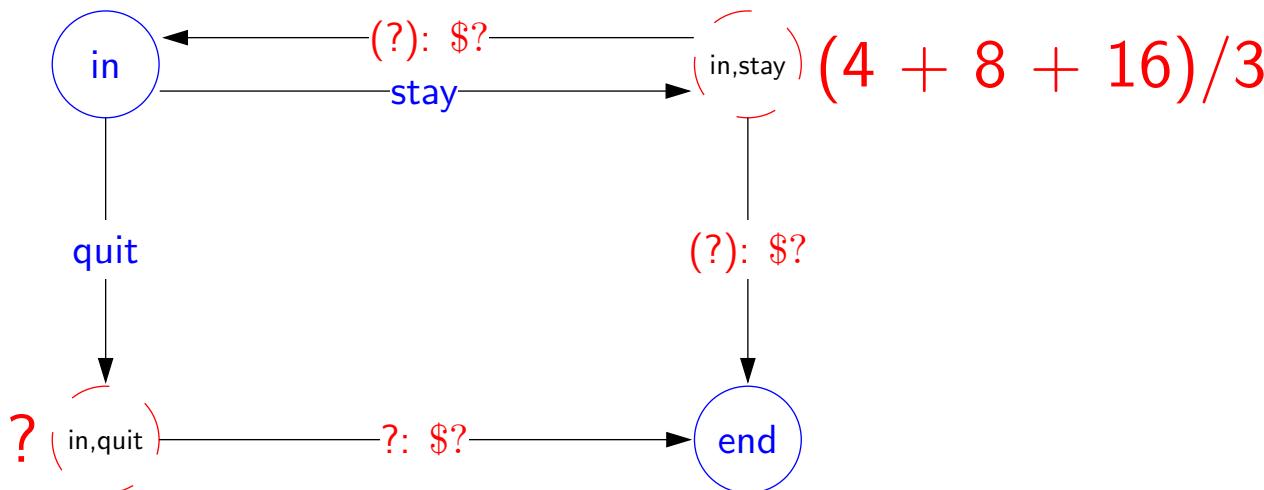
$$u_t = r_t + \gamma \cdot r_{t+1} + \gamma^2 \cdot r_{t+2} + \dots$$

Estimate:

$$\hat{Q}_\pi(s, a) = \text{average of } u_t \text{ where } s_{t-1} = s, a_t = a$$

(and  $s, a$  doesn't occur in  $s_0, \dots, s_{t-2}$ )

# Model-free Monte Carlo



Data (following policy  $\pi(s) = \text{stay}$ ):

[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]

Note: we are estimating  $Q_\pi$  now, not  $Q_{\text{opt}}$



## Definition: on-policy versus off-policy

On-policy: estimate the value of data-generating policy

Off-policy: estimate the value of another policy

- Recall that  $Q_\pi(s, a)$  is the expected utility starting at  $s$ , taking action  $a$ , and the following  $\pi$ .
- In terms of the data, define  $u_t$  to be the discounted sum of rewards starting with  $r_t$ .
- Observe that  $Q_\pi(s_{t-1}, a_t) = \mathbb{E}[u_t]$ ; that is, if we're at state  $s_{t-1}$  and take action  $a_t$ , the average value of  $u_t$  is  $Q_\pi(s_{t-1}, a_t)$ .
- But that particular state and action pair  $(s, a)$  will probably show up many times. If we take the average of  $u_t$  over all the times that  $s_{t-1} = s$  and  $a_t = a$ , then we obtain our Monte Carlo estimate  $\hat{Q}_\pi(s, a)$ . Note that nowhere do we need to talk about transitions or immediate rewards; the only thing that matters is total rewards resulting from  $(s, a)$  pairs.
- One technical note is that for simplicity, we only consider  $s_{t-1} = s, a_t = a$  for which the  $(s, a)$  doesn't show up later. This is not necessary for the algorithm to work, but it is easier to analyze and think about.
- Model-free Monte Carlo depends strongly on the policy  $\pi$  that is followed; after all it's computing  $Q_\pi$ . Because the value being computed is dependent on the policy used to generate the data, we call this an **on-policy** algorithm. In contrast, model-based Monte Carlo is **off-policy**, because the model we estimated did not depend on the exact policy (as long as it was able to explore all  $(s, a)$  pairs).

# Model-free Monte Carlo (equivalences)

Data (following policy  $\pi$ ):

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

Original formulation

$$\hat{Q}_\pi(s, a) = \text{average of } u_t \text{ where } s_{t-1} = s, a_t = a$$

Equivalent formulation (convex combination)

On each  $(s, a, u)$ :

$$\eta = \frac{1}{1 + (\# \text{ updates to } (s, a))}$$

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta u$$

[whiteboard:  $u_1, u_2, u_3$ ]

- Over the next few slides, we will interpret model-free Monte Carlo in several ways. This is the same algorithm, just viewed from different perspectives. This will give us some more intuition and allow us to develop other algorithms later.
- The first interpretation is thinking in terms of **interpolation**. Instead of thinking of averaging as a batch operation that takes a list of numbers (realizations of  $u_t$ ) and computes the mean, we can view it as an iterative procedure for building the mean as new numbers are coming in.
- In particular, it's easy to work out for a small example that averaging is equivalent to just interpolating between the old value  $\hat{Q}_\pi(s, a)$  (current estimate) and the new value  $u$  (data). The interpolation ratio  $\eta$  is set carefully so that  $u$  contributes exactly the right amount to the average.
- Advanced: in practice, we would constantly improve the policy  $\pi$  constantly over time. In this case, we might want to set  $\eta$  to something that doesn't decay as quickly (for example,  $\eta = 1/\sqrt{\#\text{ updates to } (s, a)}$ ). This rate implies that a new example contributes more than an old example, which is desirable so that  $\hat{Q}_\pi(s, a)$  reflects the more recent policy rather than the old policy.

# Model-free Monte Carlo (equivalences)

─ Equivalent formulation (convex combination) ─

On each  $(s, a, u)$ :

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta) \hat{Q}_\pi(s, a) + \eta u$$

─ Equivalent formulation (stochastic gradient) ─

On each  $(s, a, u)$ :

$$\hat{Q}_\pi(s, a) \leftarrow \hat{Q}_\pi(s, a) - \eta [\underbrace{\hat{Q}_\pi(s, a)}_{\text{prediction}} - \underbrace{u}_{\text{target}}]$$

Implied objective: least squares regression

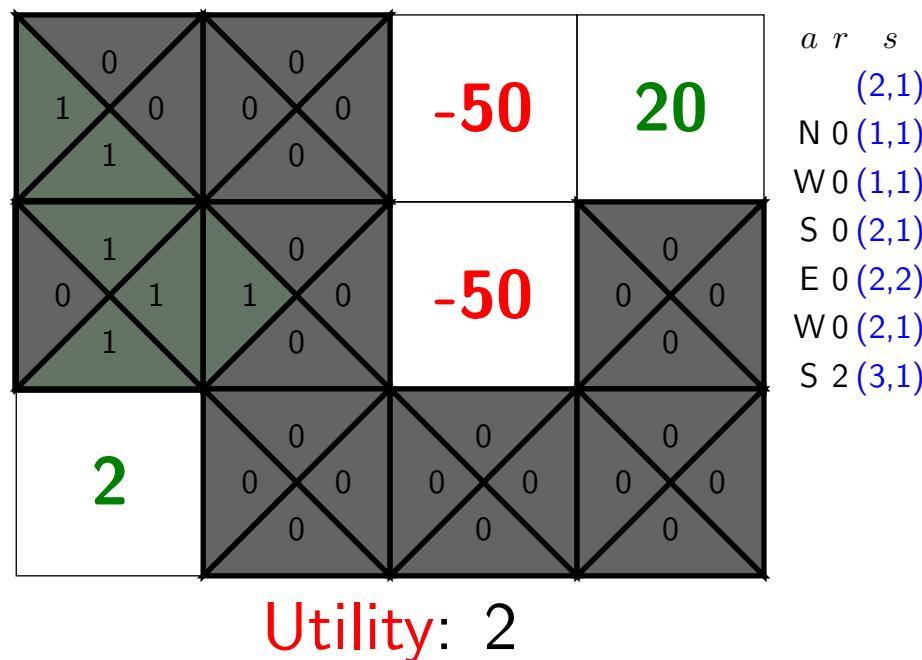
$$(\hat{Q}_\pi(s, a) - u)^2$$

- The second equivalent formulation is making the update look like a stochastic gradient update. Indeed, if we think about each  $(s, a, u)$  triple as an example (where  $(s, a)$  is the input and  $u$  is the output), then the model-free Monte Carlo is just performing stochastic gradient descent on a least squares regression problem, where the weight vector is  $\hat{Q}_\pi$  (which has dimensionality  $SA$ ) and there is one feature template " $(s, a)$  equals \_\_\_".
- The stochastic gradient descent view will become particularly relevant when we use non-trivial features on  $(s, a)$ .

# Volcanic model-free Monte Carlo

Run

(or press ctrl-enter)



- Let's run model-free Monte Carlo on the volcano crossing example. `slipProb` is zero to make things simpler. We are showing the Q-values: for each state, we have four values, one for each action.
- Here, our exploration policy is one that chooses an action uniformly at random.
- Try pressing "Run" multiple times to understand how the Q-values are set.
- Then try increasing `numEpisodes`, and seeing how the Q-values of this policy become more accurate.
- You will notice that a random policy has a very hard time reaching the 20.



# Roadmap

Reinforcement learning

Monte Carlo methods

**Bootstrapping methods**

Covering the unknown

Summary

# Using the utility

Data (following policy  $\pi(s) = \text{stay}$ ):

[in; <b>stay</b> , 4, end]	$u = 4$
[in; <b>stay</b> , 4, in; <b>stay</b> , 4, end]	$u = 8$
[in; <b>stay</b> , 4, in; <b>stay</b> , 4, in; <b>stay</b> , 4, end]	$u = 12$
[in; <b>stay</b> , 4, in; <b>stay</b> , 4, in; <b>stay</b> , 4, in; <b>stay</b> , 4, end]	$u = 16$



## Algorithm: model-free Monte Carlo

On each  $(s, a, u)$ :

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta \underbrace{u}_{\text{data}}$$

# Using the reward + Q-value

Current estimate:  $\hat{Q}_\pi(s, \text{stay}) = 11$

Data (following policy  $\pi(s) = \text{stay}$ ):

[in; stay, 4, end]  $4 + 0$

[in; stay, 4, in; stay, 4, end]  $4 + 11$

[in; stay, 4, in; stay, 4, in; stay, 4, end]  $4 + 11$

[in; stay, 4, in; stay, 4, in; stay, 4, in; stay, 4, end]  $4 + 11$



## Algorithm: SARSA

On each  $(s, a, r, s', a')$ :

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta [ \underbrace{r}_{\text{data}} + \gamma \underbrace{\hat{Q}_\pi(s', a')}_{\text{estimate}} ]$$

- Broadly speaking, reinforcement learning algorithms interpolate between new data (which specifies the **target** value) and the old estimate of the value (the **prediction**).
- Model-free Monte Carlo's target was  $u$ , the discounted sum of rewards after taking an action. However,  $u$  itself is just an estimate of  $Q_\pi(s, a)$ . If the episode is long,  $u$  will be a pretty lousy estimate. This is because  $u$  only corresponds to one episode out of a mind-blowing exponential (in the episode length) number of possible episodes, so as the episode lengthens, it becomes an increasingly less representative sample of what could happen. Can we produce better estimate of  $Q_\pi(s, a)$ ?
- An alternative to model-free Monte Carlo is SARSA, whose target is  $r + \gamma \hat{Q}_\pi(s', a')$ . Importantly, SARSA's target is a combination of the data (the first step) and the estimate (for the rest of the steps). In contrast, model-free Monte Carlo's  $u$  is taken purely from the data.

# Model-free Monte Carlo versus SARSA



## Key idea: bootstrapping

SARSA uses estimate  $\hat{Q}_\pi(s, a)$  instead of just raw data  $u$ .

$u$

based on one path

unbiased

large variance

wait until end to update

$r + \hat{Q}_\pi(s', a')$

based on estimate

biased

small variance

can update immediately

- The main advantage that SARSA offers over model-free Monte Carlo is that we don't have to wait until the end of the episode to update the Q-value.
- If the estimates are already pretty good, then SARSA will be more reliable since  $u$  is based on only one path whereas  $\hat{Q}_\pi(s', a')$  is based on all the ones that the learner has seen before.
- Advanced: We can actually interpolate between model-free Monte Carlo (all rewards) and SARSA (one reward). For example, we could update towards  $r_t + \gamma r_{t+1} + \gamma^2 \hat{Q}_\pi(s_{t+1}, a_{t+2})$  (two rewards). We can even combine all of these updates, which results in an algorithm called SARSA( $\lambda$ ), where  $\lambda$  determines the relative weighting of these targets. See the Sutton/Barto reinforcement learning book (chapter 7) for an excellent introduction.
- Advanced: There is also a version of these algorithms that estimates the value function  $V_\pi$  instead of  $Q_\pi$ . Value functions aren't enough to choose actions unless you actually know the transitions and rewards. Nonetheless, these are useful in game playing where we actually know the transition and rewards, but the state space is just too large to compute the value function exactly.



# Question

Which of the following algorithms allows you to estimate  $Q_{\text{opt}}(s, a)$  (select all that apply)?

model-based Monte Carlo

model-free Monte Carlo

SARSA

- Model-based Monte Carlo estimates the transitions and rewards, which fully specifies the MDP. With the MDP, you can estimate anything you want, including computing  $Q_{\text{opt}}(s, a)$
- Model-free Monte Carlo and SARSA are on-policy algorithms, so they only give you  $\hat{Q}_{\pi}(s, a)$ , which is specific to a policy  $\pi$ . These will not provide direct estimates of  $Q_{\text{opt}}(s, a)$ .

# Q-learning

**Problem:** model-free Monte Carlo and SARSA only estimate  $Q_\pi$ , but want  $Q_{\text{opt}}$  to act optimally

<b>Output</b>	<b>MDP</b>	<b>reinforcement learning</b>
$Q_\pi$	policy evaluation	model-free Monte Carlo, SARSA
$Q_{\text{opt}}$	value iteration	<b>Q-learning</b>

- Recall our goal is to get an optimal policy, which means estimating  $Q_{\text{opt}}$ .
- The situation is as follows: Our two methods (model-free Monte Carlo and SARSA) are model-free, but only produce estimates  $Q_{\pi}$ . We have one algorithm, model-based Monte Carlo, which can be used to produce estimates of  $Q_{\text{opt}}$ , but is model-based. Can we get an estimate of  $Q_{\text{opt}}$  in a model-free manner?
- The answer is yes, and Q-learning is an **off-policy** algorithm that accomplishes this.
- One can draw an analogy between reinforcement learning algorithms and the classic MDP algorithms. MDP algorithms are offline, RL algorithms are online. In both cases, algorithms either output the Q-values for a fixed policy or the optimal Q-values.

# Q-learning

MDP recurrence:

$$Q_{\text{opt}}(s, a) = \sum_{s'} T(s, a, s')[\text{Reward}(s, a, s') + \gamma V_{\text{opt}}(s')]$$



## Algorithm: Q-learning [Watkins/Dayan, 1992]

On each  $(s, a, r, s')$ :

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow (1 - \eta) \underbrace{\hat{Q}_{\text{opt}}(s, a)}_{\text{prediction}} + \eta \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}$$

$$\text{Recall: } \hat{V}_{\text{opt}}(s') = \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s', a')$$

- To derive Q-learning, it is instructive to look back at the MDP recurrence for  $Q_{\text{opt}}$ . There are several changes that take us from the MDP recurrence to Q-learning. First, we don't have an expectation over  $s'$ , but only have one sample  $s'$ .
- Second, because of this, we don't want to just replace  $\hat{Q}_{\text{opt}}(s, a)$  with the target value, but want to interpolate between the old value (prediction) and the new value (target).
- Third, we replace the actual reward  $\text{Reward}(s, a, s')$  with the observed reward  $r$  (when the reward function is deterministic, the two are the same).
- Finally, we replace  $V_{\text{opt}}(s')$  with our current estimate  $\hat{V}_{\text{opt}}(s')$ .
- Importantly, the estimated optimal value  $\hat{V}_{\text{opt}}(s')$  involves a maximum over actions rather than taking the action of the policy. This max over  $a'$  rather than taking the  $a'$  based on the current policy is the principle difference between Q-learning and SARSA.

# SARSA versus Q-learning



## Algorithm: SARSA

On each  $(s, a, r, s', a')$ :

$$\hat{Q}_\pi(s, a) \leftarrow (1 - \eta)\hat{Q}_\pi(s, a) + \eta(r + \gamma\hat{Q}_\pi(s', a'))$$



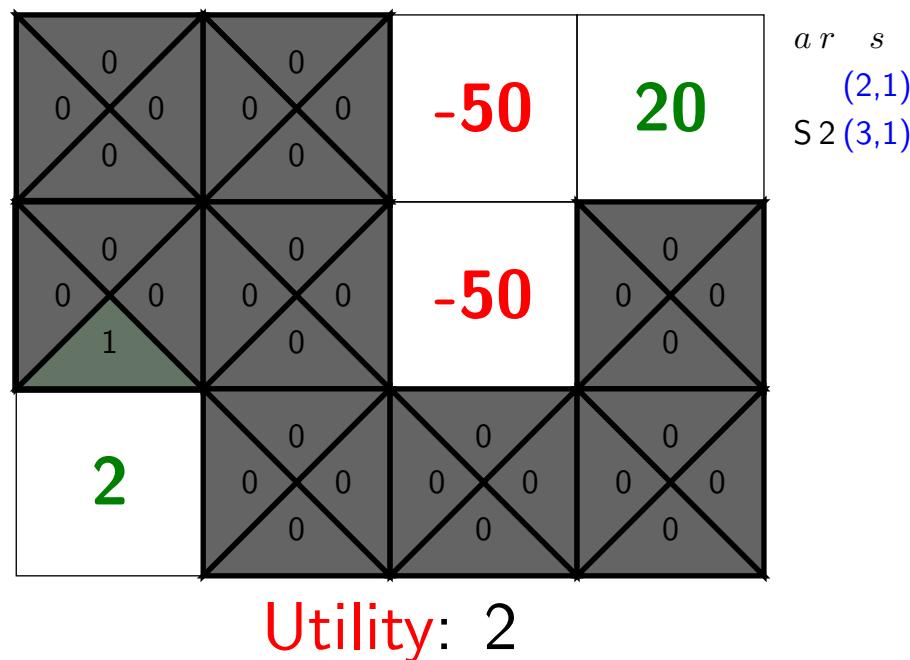
## Algorithm: Q-learning [Watkins/Dayan, 1992]

On each  $(s, a, r, s')$ :

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow (1 - \eta)\hat{Q}_{\text{opt}}(s, a) + \eta(r + \gamma \max_{a' \in \text{Actions}(s')} \hat{Q}_{\text{opt}}(s', a'))$$

# Volcanic SARSA and Q-learning

**Run** (or press ctrl-enter)



- Let us try SARSA and Q-learning on the volcanic example.
- If you increase `numEpisodes` to 1000, SARSA will behave very much like model-free Monte Carlo, computing the value of the random policy.
- However, note that Q-learning is computing an estimate of  $Q_{\text{opt}}(s, a)$ , so the resulting Q-values will be very different. The average utility will not change since we are still following and being evaluated on the same random policy. This is an important point for **off-policy** methods: the online performance (average utility) is generally a lot worse and not representative of what the model has learned, which is captured in the estimated Q-values.



# Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

**Covering the unknown**

Summary



# Exploration



## Algorithm: reinforcement learning template

For  $t = 1, 2, 3, \dots$

Choose action  $a_t = \pi_{\text{act}}(s_{t-1})$  (**how?**)

Receive reward  $r_t$  and observe new state  $s_t$

Update parameters (**how?**)

$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$

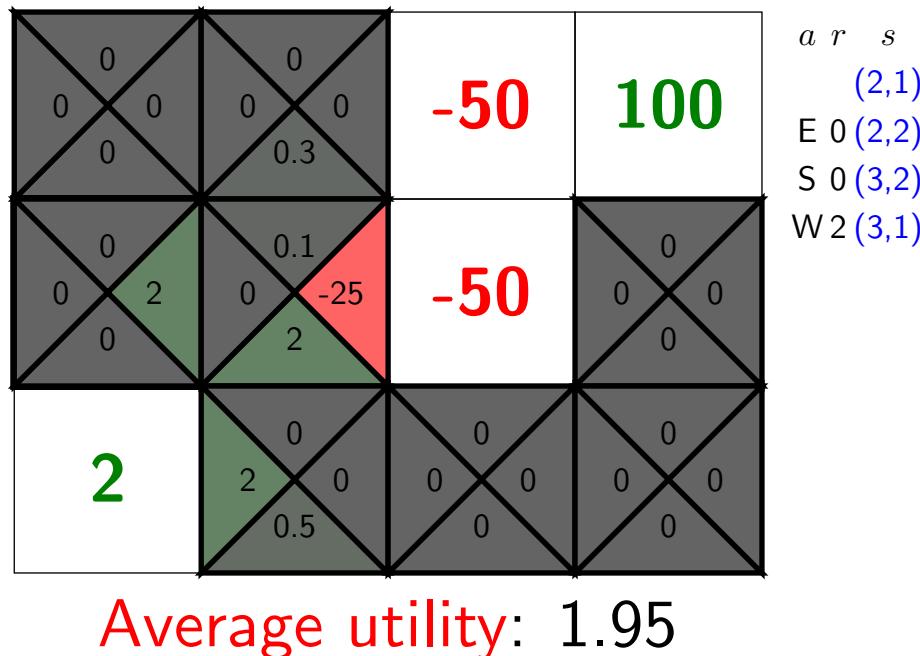
Which **exploration policy**  $\pi_{\text{act}}$  to use?

- We have so far given many algorithms for updating parameters (i.e.,  $\hat{Q}_\pi(s, a)$  or  $\hat{Q}_{\text{opt}}(s, a)$ ). If we were doing supervised learning, we'd be done, but in reinforcement learning, we need to actually determine our **exploration policy**  $\pi_{\text{act}}$  to collect data for learning. Recall that we need to somehow make sure we get information about each  $(s, a)$ .
- We will discuss two complementary ways to get this information: (i) explicitly explore  $(s, a)$  or (ii) explore  $(s, a)$  implicitly by actually exploring  $(s', a')$  with similar features and generalizing.
- These two ideas apply to many RL algorithms, but let us specialize to Q-learning.

# No exploration, all exploitation

Attempt 1: Set  $\pi_{\text{act}}(s) = \arg \max_{a \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s, a)$

**Run** (or press ctrl-enter)



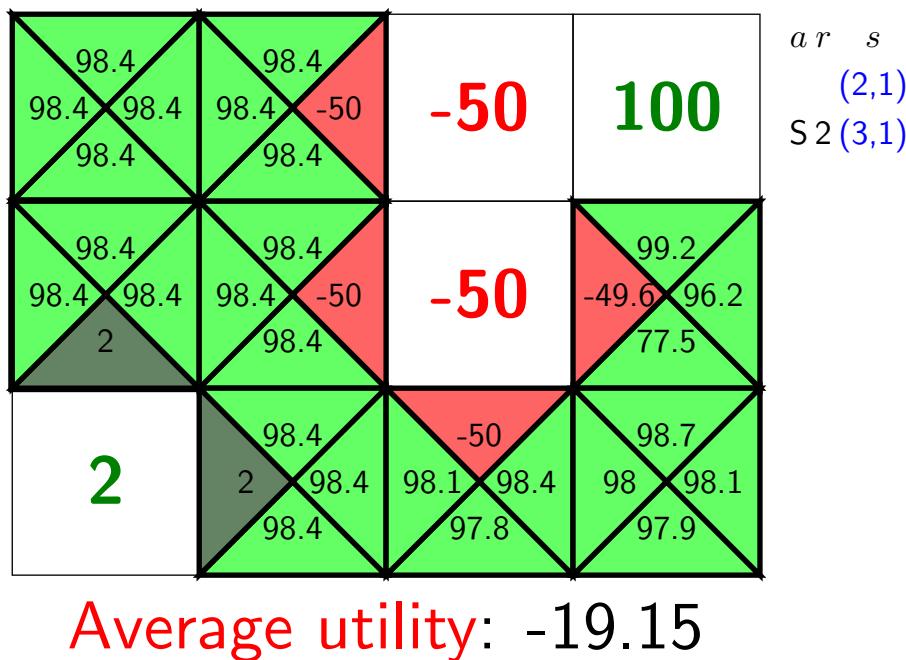
Problem:  $\hat{Q}_{\text{opt}}(s, a)$  estimates are inaccurate, **too greedy!**

- The naive solution is to explore using the optimal policy according to the estimated Q-value  $\hat{Q}_{\text{opt}}(s, a)$ .
- But this fails horribly. In the example, once the agent discovers that there is a reward of 2 to be gotten by going south that becomes its optimal policy and it will not try any other action. The problem is that the agent is being too greedy.
- In the demo, if multiple actions have the same maximum Q-value, we choose randomly. Try clicking "Run" a few times, and you'll end up with minor variations.
- Even if you increase numEpisodes to 10000, nothing new gets learned.

# No exploitation, all exploration

Attempt 2: Set  $\pi_{\text{act}}(s) = \text{random from Actions}(s)$

Run (or press ctrl-enter)



Problem: average utility is low because exploration is **not guided**

- We can go to the other extreme and use an exploration policy that always chooses a random action. It will do a much better job of exploration, but it doesn't exploit what it learns and ends up with a very low utility.
- It is interesting to note that the value (average over utilities across all the episodes) can be quite small and yet the Q-values can be quite accurate. Recall that this is possible because Q-learning is an off-policy algorithm.

# Exploration/exploitation tradeoff



**Key idea: balance**

Need to balance **exploration** and **exploitation**.



Examples from life: restaurants, routes, research

# Epsilon-greedy

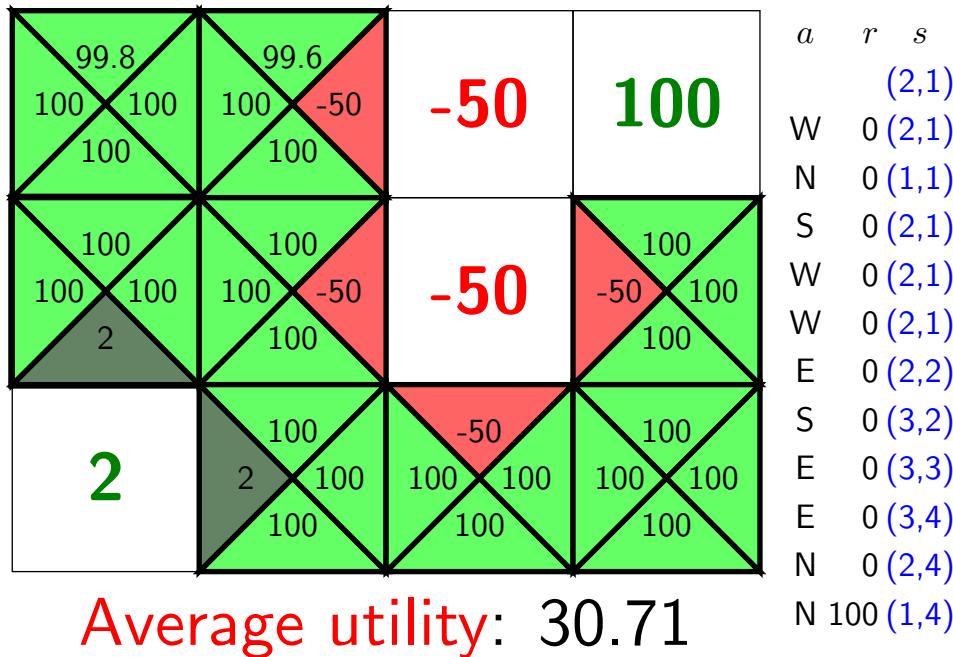


## Algorithm: epsilon-greedy policy

$$\pi_{\text{act}}(s) = \begin{cases} \arg \max_{a \in \text{Actions}} \hat{Q}_{\text{opt}}(s, a) & \text{probability } 1 - \epsilon, \\ \text{random from Actions}(s) & \text{probability } \epsilon. \end{cases}$$

Run

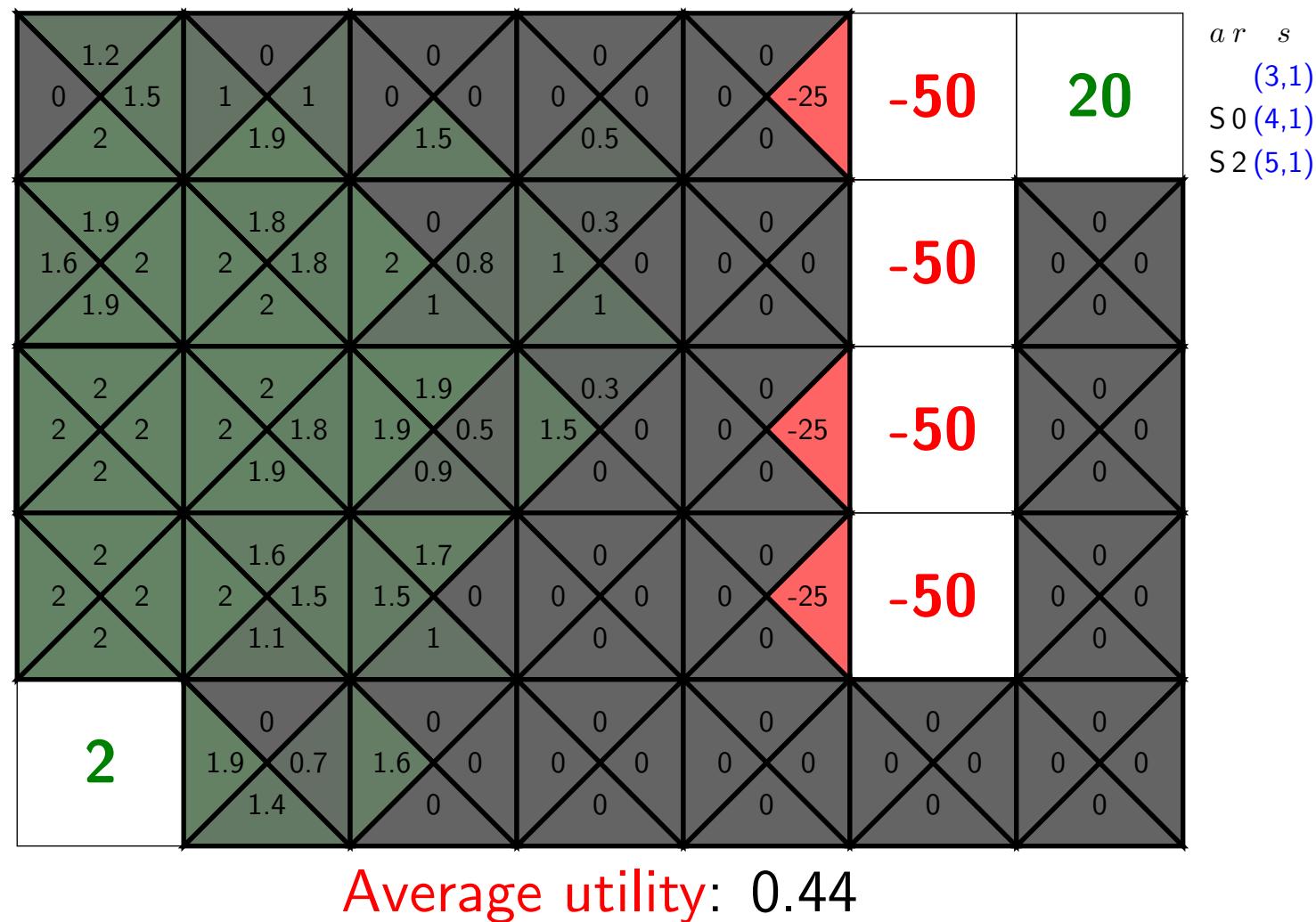
(or press ctrl-enter)



- The natural thing to do when you have two extremes is to interpolate between the two. The result is the **epsilon-greedy** algorithm which explores with probability  $\epsilon$  and exploits with probability  $1 - \epsilon$ .
- It is natural to let  $\epsilon$  decrease over time. When you're young, you want to explore a lot ( $\epsilon = 1$ ). After a certain point, when you feel like you've seen all there is to see, then you start exploiting ( $\epsilon = 0$ ).
- For example, we let  $\epsilon = 1$  for the first third of the episodes,  $\epsilon = 0.5$  for the second third, and  $\epsilon = 0$  for the final third. This is not the optimal schedule. Try playing around with other schedules to see if you can do better.

# Generalization

Problem: large state spaces, hard to explore



- Now we turn to another problem with vanilla Q-learning.
- In real applications, there can be millions of states, in which there's no hope for epsilon-greedy to explore everything in a reasonable amount of time.

# Q-learning

Stochastic gradient update:

$$\hat{Q}_{\text{opt}}(s, a) \leftarrow \hat{Q}_{\text{opt}}(s, a) - \eta \underbrace{\hat{Q}_{\text{opt}}(s, a)}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}}$$

This is **rote learning**: every  $\hat{Q}_{\text{opt}}(s, a)$  has a different value

Problem: doesn't generalize to unseen states/actions

- If we revisit the Q-learning algorithm, and think about it through the lens of machine learning, you'll find that we've just been memorizing Q-values for each  $(s, a)$ , treating each pair independently.
- In other words, we haven't been generalizing, which is actually one of the most important aspects of learning!

# Function approximation



**Key idea: linear regression model**

Define **features**  $\phi(s, a)$  and **weights**  $\mathbf{w}$ :

$$\hat{Q}_{\text{opt}}(s, a; \mathbf{w}) = \mathbf{w} \cdot \phi(s, a)$$



**Example: features for volcano crossing**

$$\phi_1(s, a) = \mathbf{1}[a = W] \quad \phi_7(s, a) = \mathbf{1}[s = (5, *)]$$

$$\phi_2(s, a) = \mathbf{1}[a = E] \quad \phi_8(s, a) = \mathbf{1}[s = (*, 6)]$$

...

...

- **Function approximation** fixes this by parameterizing  $\hat{Q}_{\text{opt}}$  by a weight vector and a feature vector, as we did in linear regression.
- Recall that features are supposed to be properties of the state-action  $(s, a)$  pair that are indicative of the quality of taking action  $a$  in state  $s$ .
- The ramification is that all the states that have similar features will have similar Q-values. For example, suppose  $\phi$  included the feature  $\mathbf{1}[s = (*, 4)]$ . If we were in state  $(1, 4)$ , took action E, and managed to get high rewards, then Q-learning with function approximation will propagate this positive signal to all positions in column 4 taking any action.
- In our example, we defined features on actions (to capture that moving east is generally good) and features on states (to capture the fact that the 6th column is best avoided, and the 5th row is generally a good place to travel to).

# Function approximation



## Algorithm: Q-learning with function approximation

On each  $(s, a, r, s')$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}} \phi(s, a)$$

Implied objective function:

$$(\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_{\text{target}})^2$$

- We now turn our linear regression into an algorithm. Here, it is useful to adopt the stochastic gradient view of RL algorithms, which we developed a while back.
- We just have to write down the least squares objective and then compute the gradient with respect to  $w$  now instead of  $\hat{Q}_{\text{opt}}$ . The chain rule takes care of the rest.

# Covering the unknown



Epsilon-greedy: balance the exploration/exploitation tradeoff

Function approximation: can generalize to unseen states



# Summary so far

- Online setting: learn and take actions in the real world!
- Exploration/exploitation tradeoff
- Monte Carlo: estimate transitions, rewards, Q-values from data
- Bootstrapping: update towards target that depends on estimate rather than just raw data

- This concludes the technical part of reinforcement learning.
- The first part is to understand the setup: we are taking good actions in the world both to get rewards under our current model, but also to collect information about the world so we can learn a better model. This exposes the fundamental exploration/exploitation tradeoff, which is the hallmark of reinforcement learning.
- We looked at several algorithms: model-based Monte Carlo, model-free Monte Carlo, SARSA, and Q-learning. There were two complementary ideas here: using Monte Carlo approximation (approximating an expectation with a sample) and bootstrapping (using the model predictions to update itself).



# Roadmap

Reinforcement learning

Monte Carlo methods

Bootstrapping methods

Covering the unknown

**Summary**

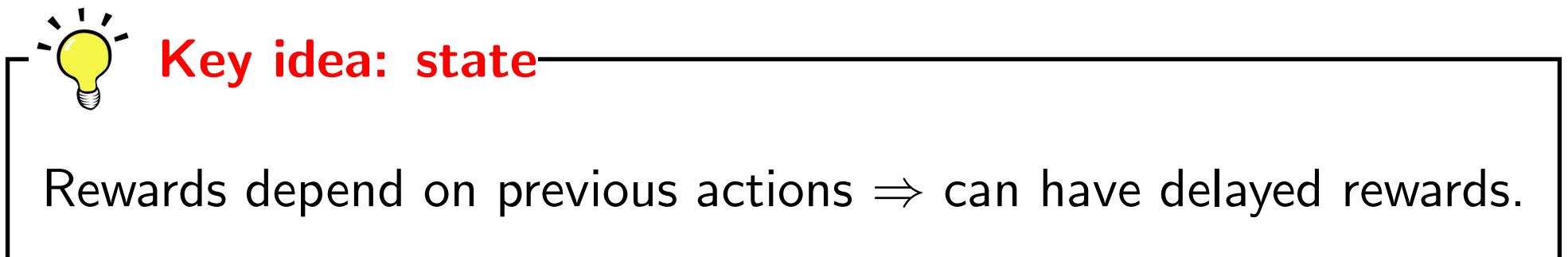
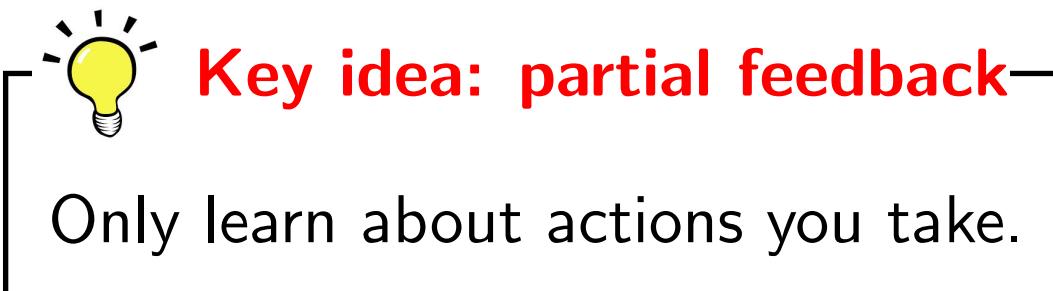
# Challenges in reinforcement learning

Binary classification (sentiment classification, SVMs):

- Stateless, full feedback

Reinforcement learning (flying helicopters, Q-learning):

- Stateful, partial feedback



# States and information

	<b>stateless</b>	<b>state</b>
<b>full feedback</b>	supervised learning (binary classification)	supervised learning (structured prediction)
<b>partial feedback</b>	multi-armed bandits	reinforcement learning

- If we compare simple supervised learning (e.g., binary classification) and reinforcement learning, we see that there are two main differences that make learning harder.
- First, reinforcement learning requires the modeling of state. State means that the rewards across time steps are related. This results in **delayed rewards**, where we take an action and don't see the ramifications of it until much later.
- Second, reinforcement learning requires dealing with partial feedback (rewards). This means that we have to actively explore to acquire the necessary feedback.
- There are two problems that move towards reinforcement learning, each on a different axis. Structured prediction introduces the notion of state, but the problem is made easier by the fact that we have full feedback, which means that for every situation, we know which action sequence is the correct one; there is no need for exploration; we just have to update our weights to favor that correct path.
- Multi-armed bandits require dealing with partial feedback, but do not have the complexities of state. One can think of a multi-armed bandit problem as an MDP with unknown random rewards and one state. Exploration is necessary, but there is no temporal depth to the problem.

# Deep reinforcement learning

just use a neural network for  $\hat{Q}_{\text{opt}}(s, a)$

Playing Atari [Google DeepMind, 2013]:

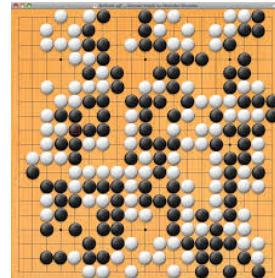


- last 4 frames (images)  $\Rightarrow$  3-layer NN  $\Rightarrow$  keystroke
- $\epsilon$ -greedy, train over 10M frames with 1M replay memory
- Human-level performance on some games (breakout), less good on others (space invaders)

- Recently, there has been a surge of interest in reinforcement learning due to the success of neural networks. If one is performing reinforcement learning in a simulator, one can actually generate tons of data, which is suitable for rich functions such as neural networks.
- A recent success story is DeepMind, who successfully trained a neural network to represent the  $\hat{Q}_{\text{opt}}$  function for playing Atari games. The impressive part was the lack of prior knowledge involved: the neural network simply took as input the raw image and outputted keystrokes.

# Deep reinforcement learning

- Policy gradient: train a policy  $\pi(a | s)$  (say, a neural network) to directly maximize expected reward
- Google DeepMind's AlphaGo (2016), AlphaZero (2017)



- Andrej Karpathy's blog post

<http://karpathy.github.io/2016/05/31/rl>

- One other major class of algorithms we will not cover in this class is **policy gradient**. Whereas Q-learning attempts to estimate the value of the optimal policy, policy gradient methods optimize the policy to maximize expected reward, which is what we care about. Recall that when we went from model-based methods (which estimated the transition and reward functions) to model-free methods (which estimated the Q function), we moved closer to the thing that we want. Policy gradient methods take this farther and just focus on the only object that really matters at the end of the day, which is the policy that an agent follows.
- Policy gradient methods have been quite successful. For example, it was one of the components of AlphaGo, Google DeepMind's program that beat the world champion at Go. One can also combine value-based methods with policy-based methods in actor-critic methods to get the best of both worlds.
- There is a lot more to say about deep reinforcement learning. If you wish to learn more, Andrej Karpathy's blog post offers a nice introduction.

# Applications



Autonomous helicopters: control helicopter to do maneuvers in the air



Backgammon: TD-Gammon plays 1-2 million games against itself, human-level performance



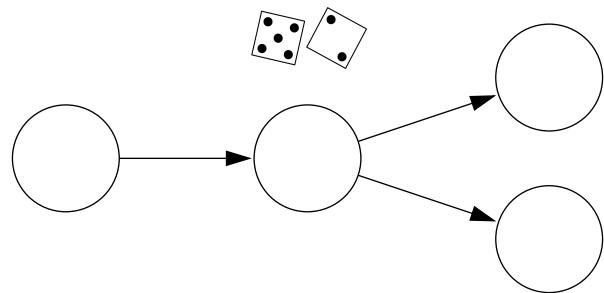
Elevator scheduling; send which elevators to which floors to maximize throughput of building



Managing datacenters; actions: bring up and shut down machine to minimize time/cost

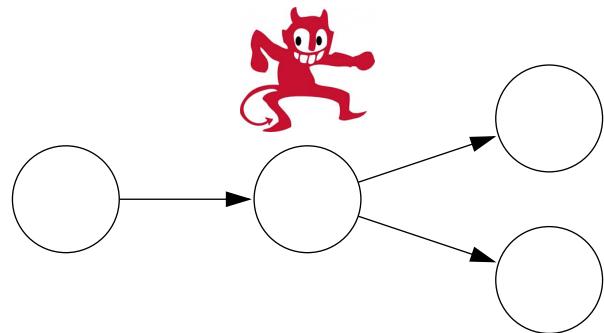
- There are many other applications of RL, which range from robotics to game playing to other infrastructural tasks. One could say that RL is so general that anything can be cast as an RL problem.
- For a while, RL only worked for small toy problems or settings where there were a lot of prior knowledge / constraints. Deep RL — the use of powerful neural networks with increased compute — has vastly expanded the realm of problems which are solvable by RL.

Markov decision processes: against nature (e.g., Blackjack)



**Next time...**

Adversarial games: against opponent (e.g., chess)



# Section 4: MDPs and Reinforcement Learning

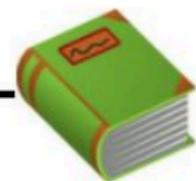
Will Deaderick, Chris Waites  
CS 221 - Autumn 2019

# Overview of today's section:

- Recap various policy learning algorithms (and when to use them) at a high-level
- SARSA vs. Q-Learning intuition
- Introduce deep reinforcement learning
- See deep RL in action on an Atari game!
- Cool RL extensions

# **Markov Decision Processes (MDPs)**

A MDP is a search problem where **transitions are random** and instead of minimizing cost, we are **maximizing reward**.



## Definition: Markov decision process

States: the set of states

$s_{\text{start}} \in \text{States}$ : starting state

Actions( $s$ ): possible actions from state  $s$

$T(s, a, s')$ : probability of  $s'$  if take action  $a$  in state  $s$

Reward( $s, a, s'$ ): reward for the transition  $(s, a, s')$

IsEnd( $s$ ): whether at end of game

$0 \leq \gamma \leq 1$ : discount factor (default: 1)

# Reward Specification

# Reward Specification

- For the majority of MDP applications, transition dynamics come from some real world process (e.g. stock market, traffic, plain old physics)

# Reward Specification

- For the majority of MDP applications, transition dynamics come from some real world process (e.g. stock market, traffic, plain old physics)
- But rewards are chosen by **you** - this is more art than science.

# Reward Specification

- For the majority of MDP applications, transition dynamics come from some real world process (e.g. stock market, traffic, plain old physics)
- But rewards are chosen by **you** - this is more art than science.

## Reward sparsity



# Reward Specification

- For the majority of MDP applications, transition dynamics come from some real world process (e.g. stock market, traffic, plain old physics)
- But rewards are chosen by **you** - this is more art than science.

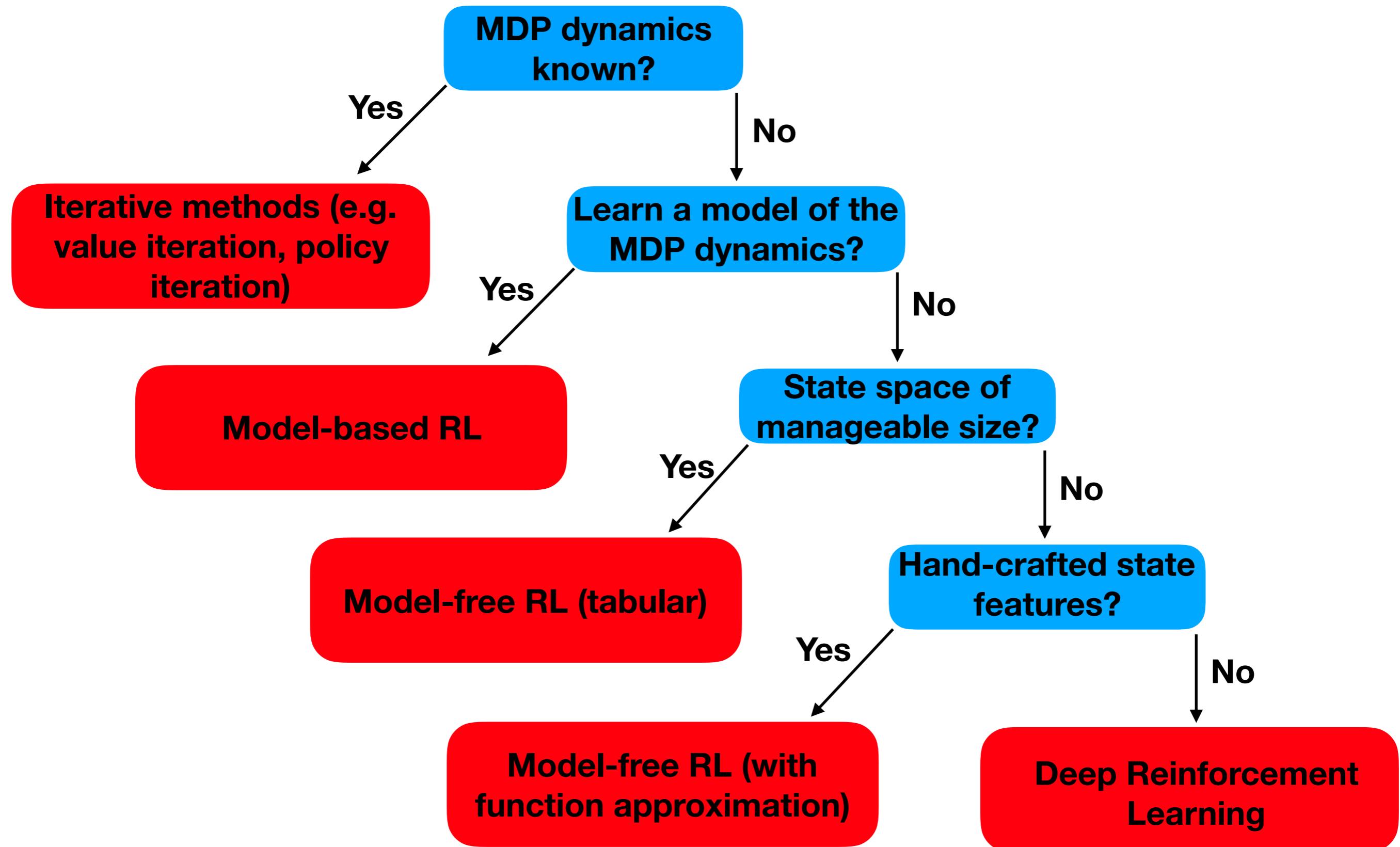
Reward sparsity

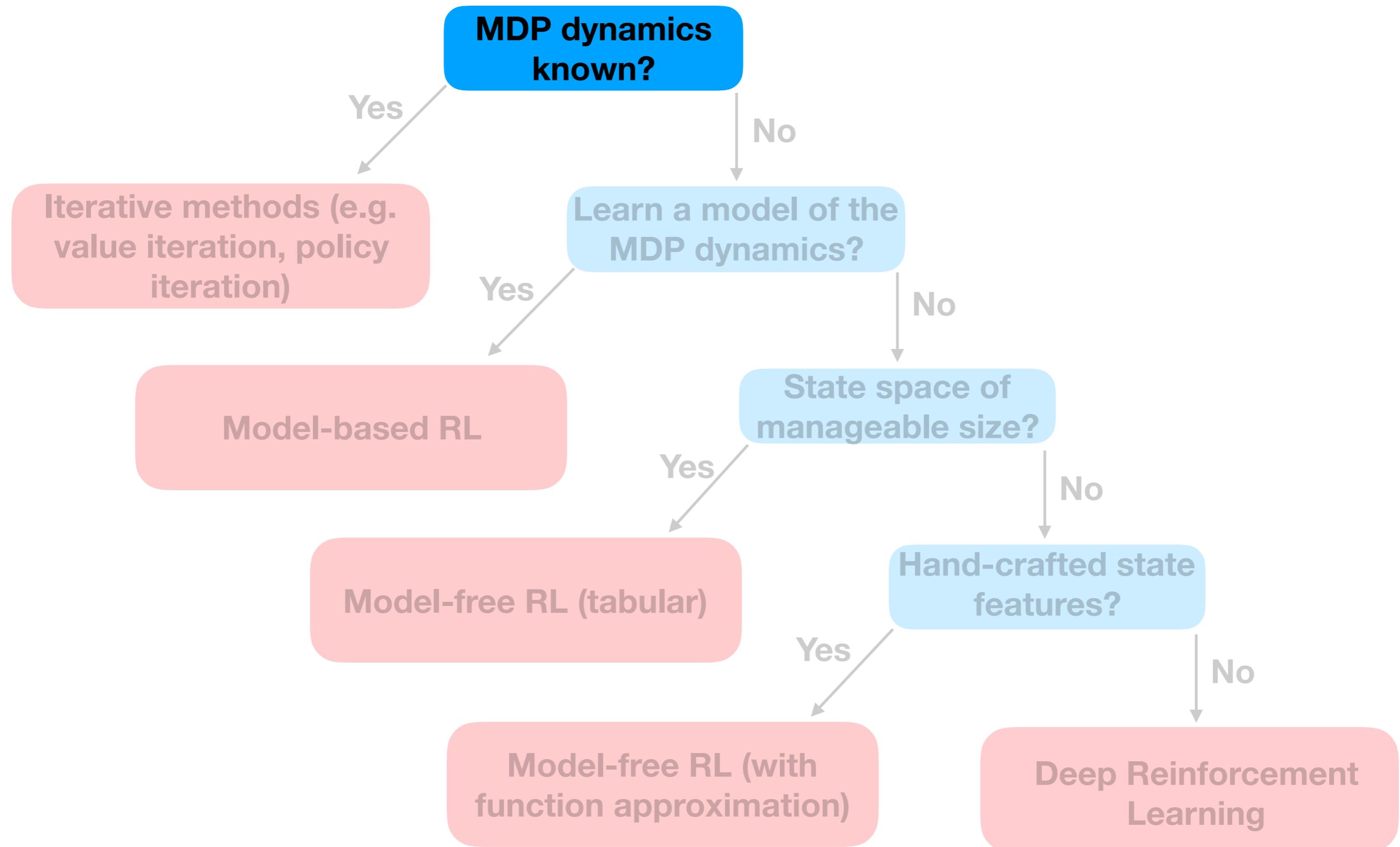


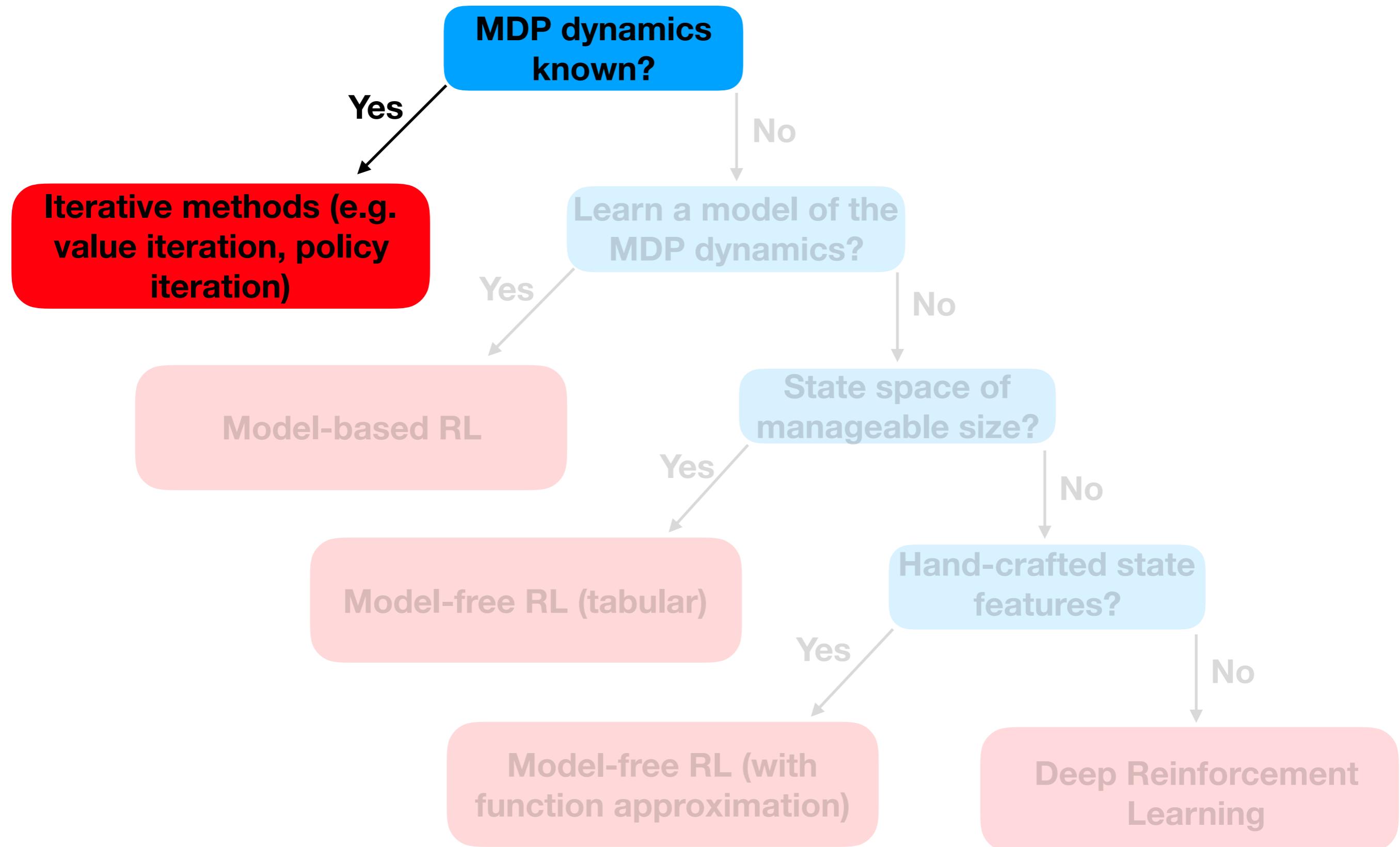
Goal alignment



# **Learning policies in MDPs**







# Iterative methods for when dynamics are known

- Pseudocode:
  1. Policy Evaluation:  $\pi \rightarrow V_\pi$
  2. Policy Improvement:  $V_\pi \rightarrow \pi'$

**Repeat until convergence  
to optimal policy**

# Iterative methods for when dynamics are known

- Pseudocode:

1. Policy Evaluation:  $\pi \rightarrow V_\pi$
  2. Policy Improvement:  $V_\pi \rightarrow \pi'$
- } Repeat until convergence  
to optimal policy

**“Generalized Policy Iteration”**

(See Sutton & Barto Ch. 4)

# Iterative methods for when dynamics are known

- Pseudocode:
  1. Policy Evaluation: iterate the value equations
  2. Policy Improvement: act optimally with respect to value estimate

# Iterative methods for when dynamics are known

- Pseudocode:
  1. Policy Evaluation: iterate the value equations
  2. Policy Improvement: act optimally with respect to value estimate

Iterative methods differ only  
in how they specify and  
interleave these two steps!

# Iterative methods for when dynamics are known

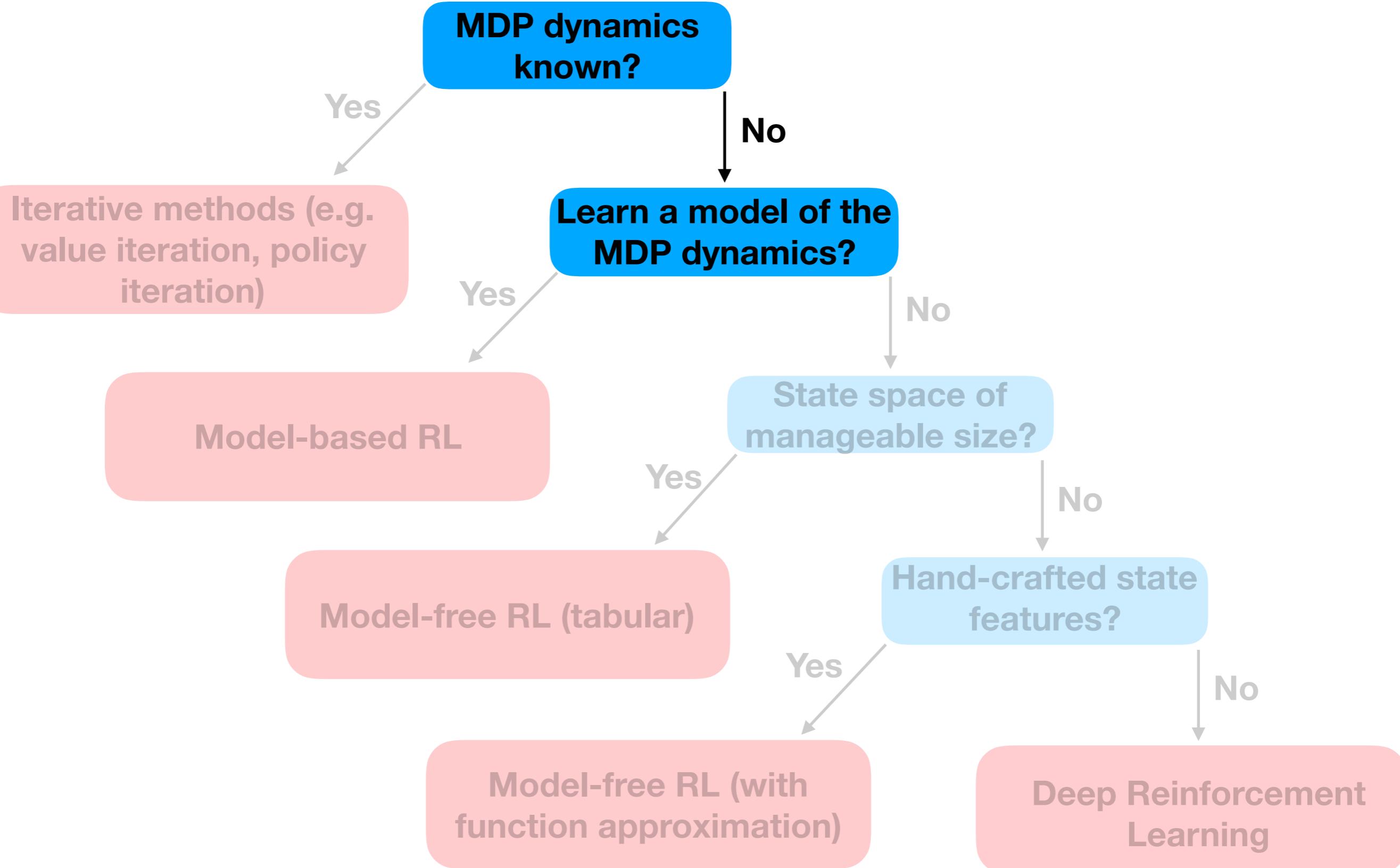
- Pseudocode:
  1. Policy Evaluation: iterate the value equations
  2. Policy Improvement: act optimally with respect to value estimate

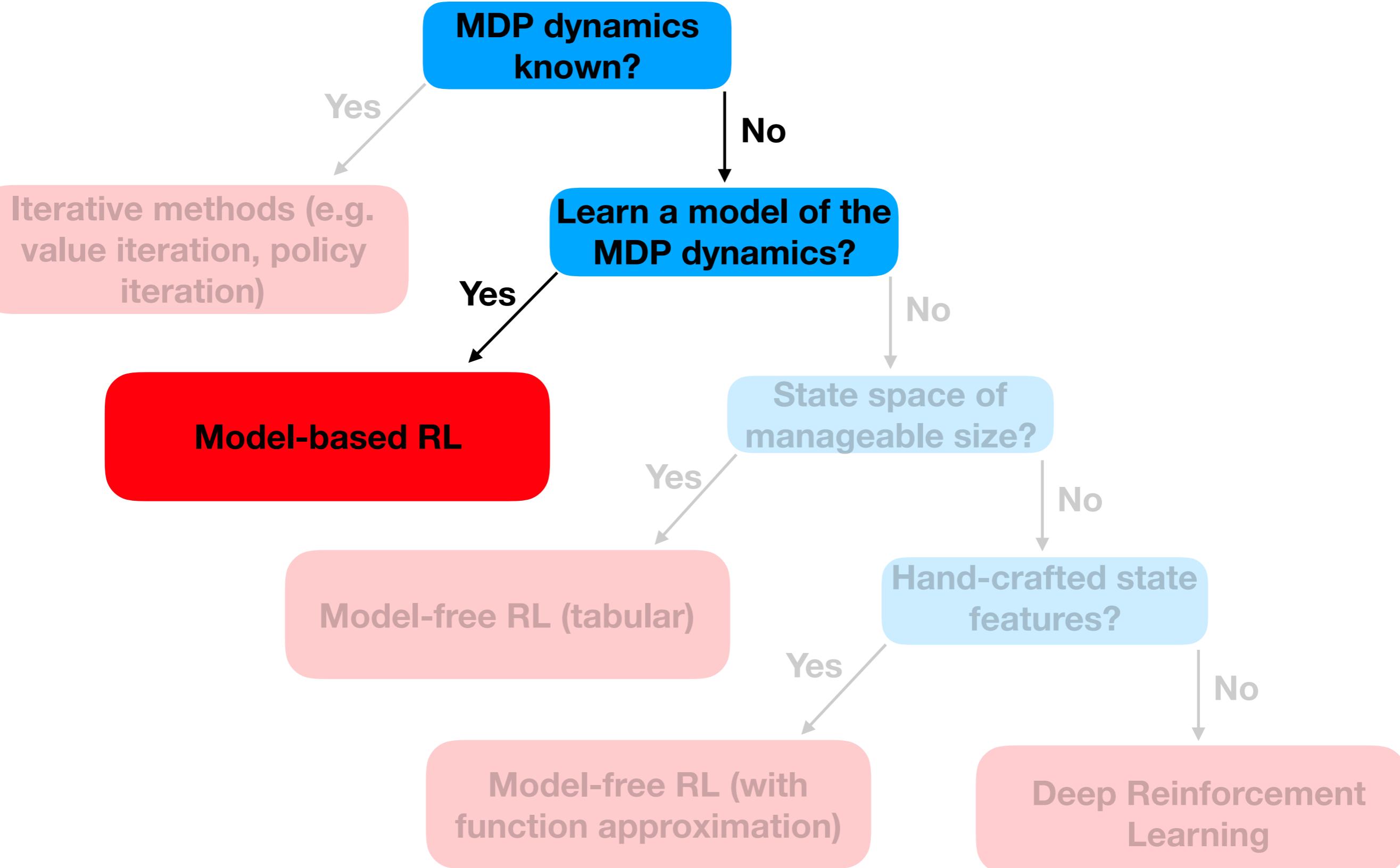
## Example: Value Iteration

$$V_{\text{opt}}^{(t)}(s) \leftarrow \max_{a \in \text{Actions}(s)} \sum_{s'} T(s, a, s') [\text{Reward}(s, a, s') + \gamma V_{\text{opt}}^{(t-1)}(s')]$$

Improvement

Evaluation





# Model-based RL

$$\hat{T}(s, a, s') = \frac{\text{\# times } (s, a, s') \text{ occurs}}{\text{\# times } (s, a) \text{ occurs}}$$

$\widehat{\text{Reward}}(s, a, s')$  = average of  $r$  in  $(s, a, r, s')$

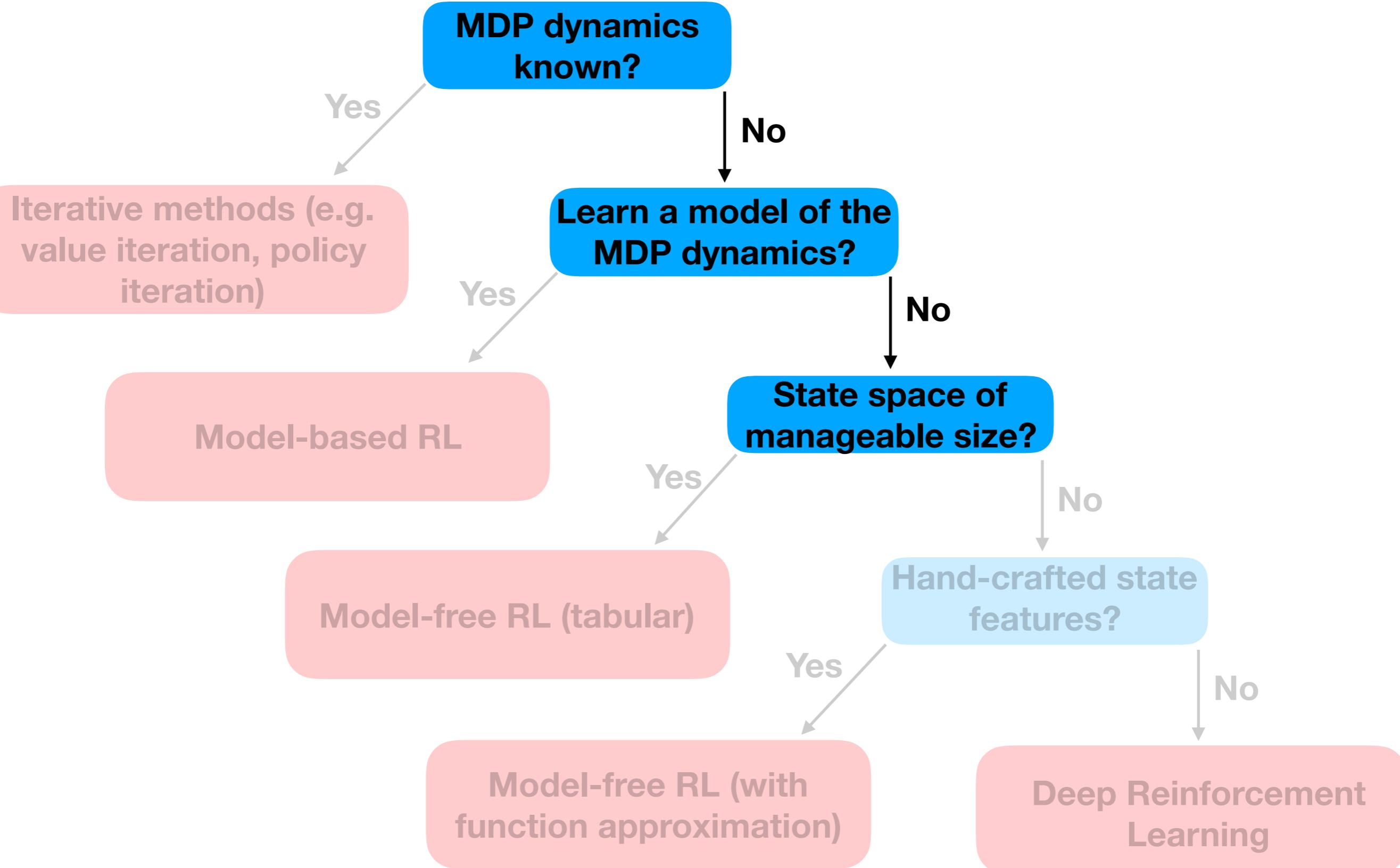
# Model-based RL

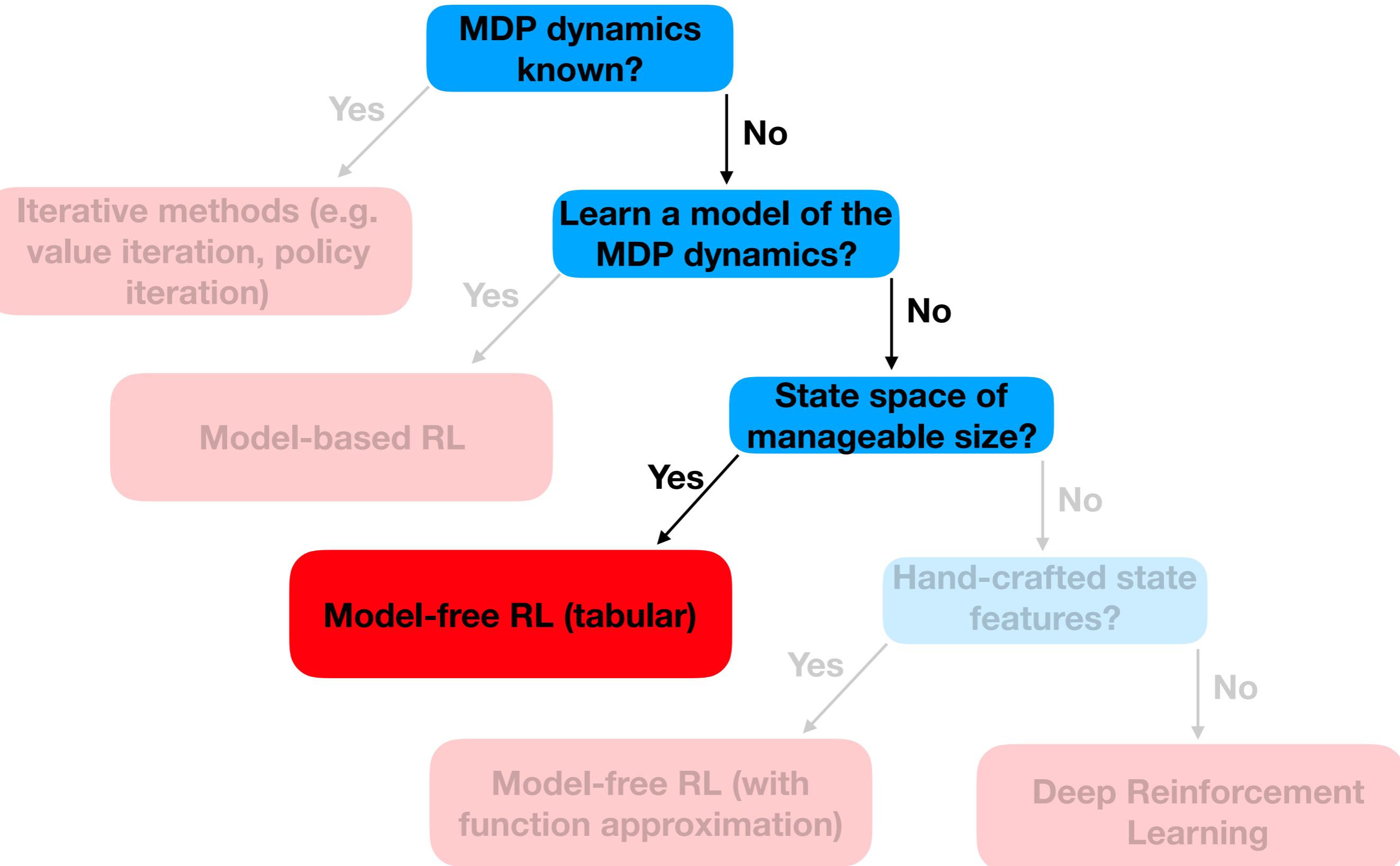
$$\hat{T}(s, a, s') = \frac{\text{\# times } (s, a, s') \text{ occurs}}{\text{\# times } (s, a) \text{ occurs}}$$

$\widehat{\text{Reward}}(s, a, s')$  = average of  $r$  in  $(s, a, r, s')$

**Now we have an estimate of the MDP dynamics - use an iterative method as if our estimate were exact!**

**But if our estimate is bad...**





# Model-free RL (tabular)

Starting in state  $s_0$ , apply policy  $\pi$

$s_0$

$a_1 \sim \pi(s_0)$

$s_1 \sim T(s_0, a_1, .)$

$r_1 \sim P_R(s_0, a_1, s_1)$

$a_2 \sim \pi(s_1)$

$s_2 \sim T(s_1, a_2, .)$

$r_2 \sim P_R(s_1, a_2, s_2)$

$\vdots$

# Model-free RL (tabular)

- For each  $(s, a)$  pair, maintain a value  $\hat{Q}(s, a)$
- Whenever we encounter  $(s, a)$ , update  $\hat{Q}(s, a)$  towards a “target” value computed using the rewards that follow
- Model-free RL algorithms primarily differ based on how they compute these “target” values.

Starting in state  $s_0$ , apply policy  $\pi$   
 $s_0$   
 $a_1 \sim \pi(s_0)$   
 $s_1 \sim T(s_0, a_1, \cdot)$   
 $r_1 \sim P_R(s_0, a_1, s_1)$   
 $a_2 \sim \pi(s_1)$   
 $s_2 \sim T(s_1, a_2, \cdot)$   
 $r_2 \sim P_R(s_1, a_2, s_2)$   
⋮

# Examples of target values

- **Monte Carlo:**
  - Sum of all (discounted) rewards that follow  $(s, a)$
- **SARSA**, for transition  $(s, a, r, s', a')$  :
  - $r + \gamma \hat{Q}(s', a')$
- **Q-Learning:**
  - $r + \gamma \max_{a'} \hat{Q}(s', a')$

# How are we selecting actions?

- Popular choice: **epsilon-greedy**

$$\pi(s) = \begin{cases} \operatorname{argmax}_a \hat{Q}(s, a) & \text{with prob } 1 - \epsilon \\ \text{random from Actions}(s) & \text{with prob } \epsilon \end{cases}$$

# How are we selecting actions?

- Popular choice: **epsilon-greedy**

$$\pi(s) = \begin{cases} \operatorname{argmax}_a \hat{Q}(s, a) & \text{with prob } 1 - \epsilon \\ \text{random from Actions}(s) & \text{with prob } \epsilon \end{cases}$$

- Policy  $\rightarrow$  Experience  $\rightarrow$  Value Function  $\rightarrow$  Policy  $\rightarrow \dots$

# How are we selecting actions?

- Popular choice: **epsilon-greedy**

$$\pi(s) = \begin{cases} \text{argmax}_a \hat{Q}(s, a) & \text{with prob } 1 - \epsilon \\ \text{random from Actions}(s) & \text{with prob } \epsilon \end{cases}$$

- Policy  $\rightarrow$  Experience  $\rightarrow$  Value Function  $\rightarrow$  Policy  $\rightarrow \dots$

**“Generalized Policy Iteration”**

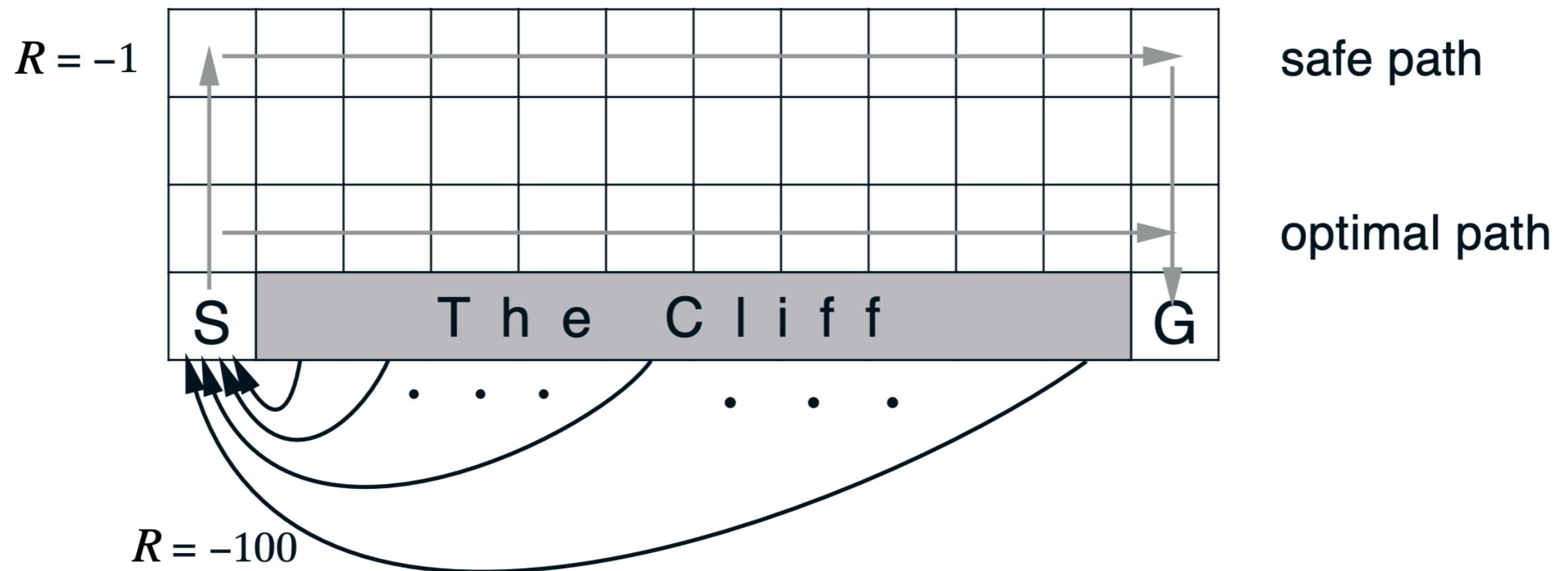
# Examples of target values

- **Monte Carlo:**
  - Sum of all (discounted) rewards that follow  $(s, a)$
- **SARSA**, for transition  $(s, a, r, s', a')$  :
  - $r + \gamma \hat{Q}(s', a')$
- **Q-Learning:**
  - $r + \gamma \max_{a'} \hat{Q}(s', a')$

Many other forms of target values exist!

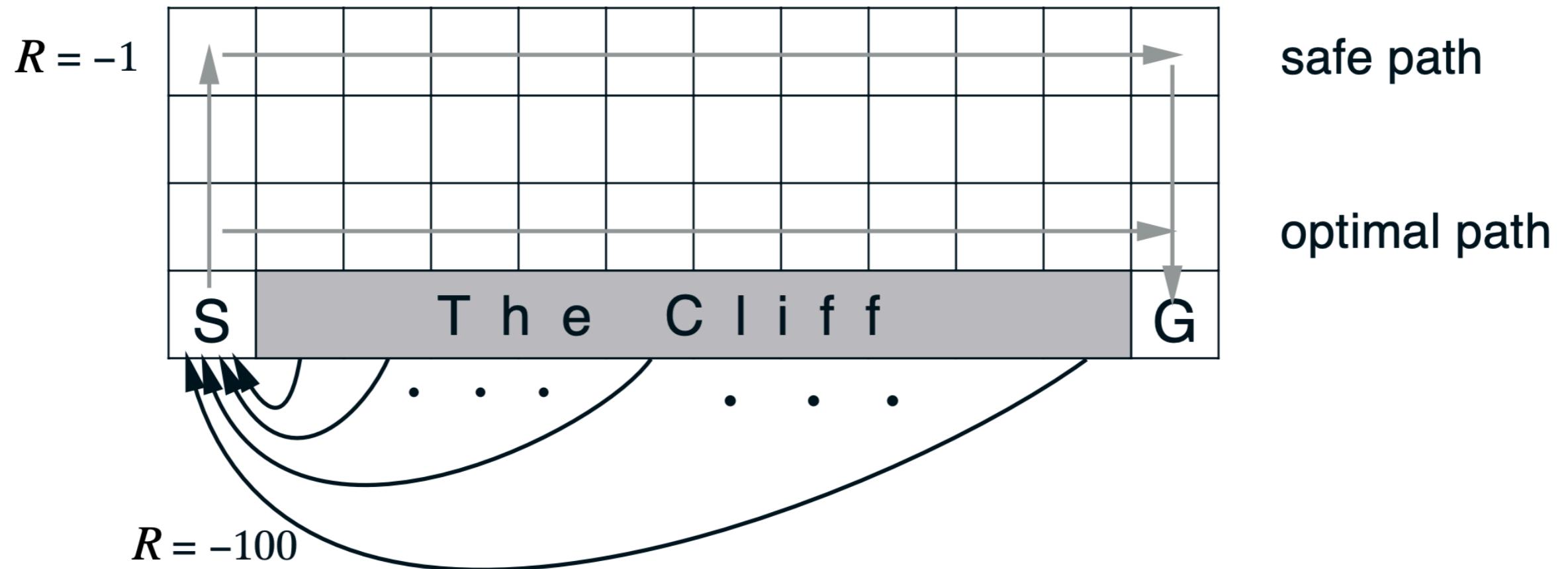
(See Sutton & Barto Ch. 5-7)

# SARSA v. Q-Learning



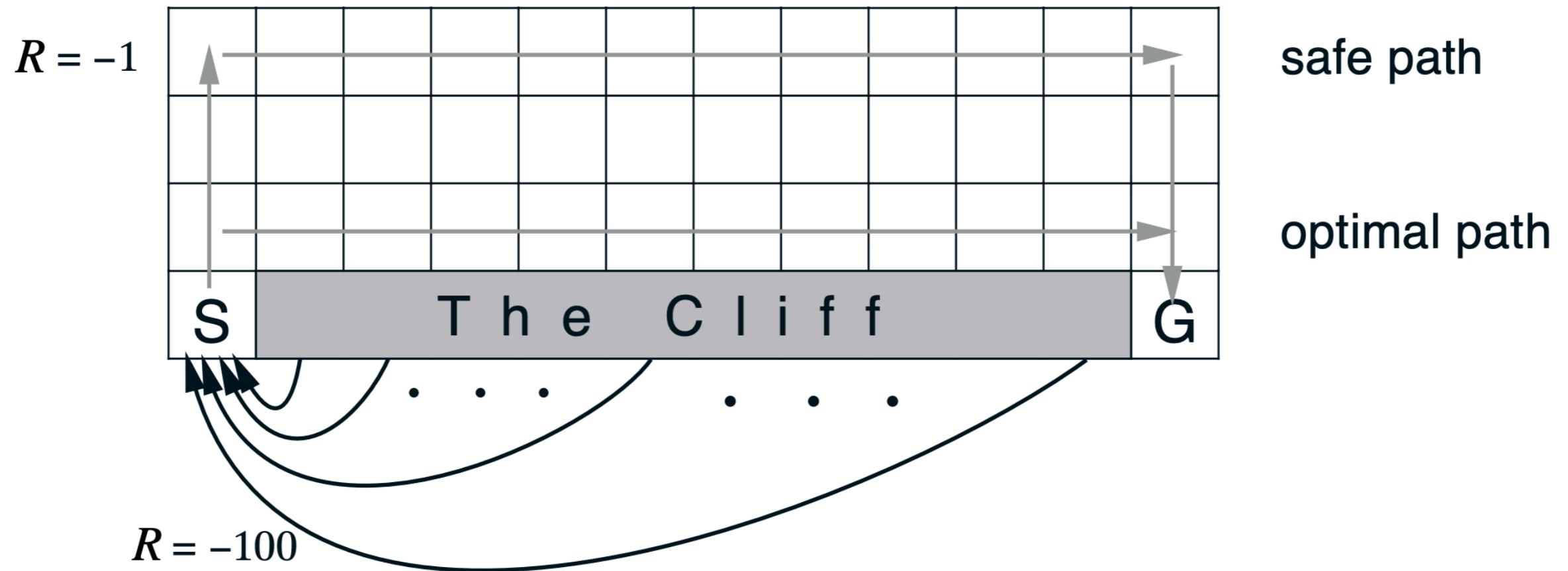
- Assume both SARSA and Q-learning use an epsilon-greedy policy

# SARSA v. Q-Learning

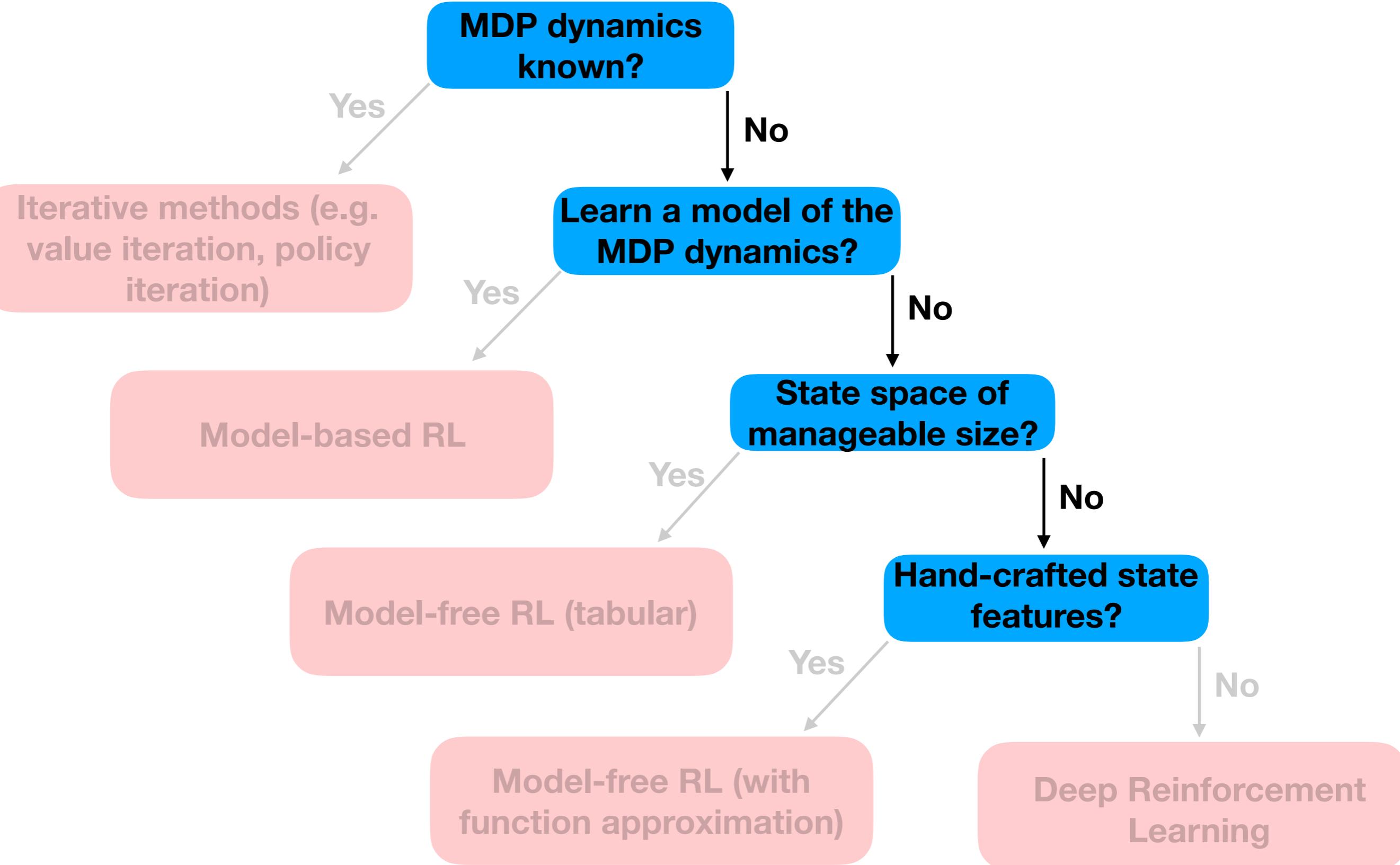


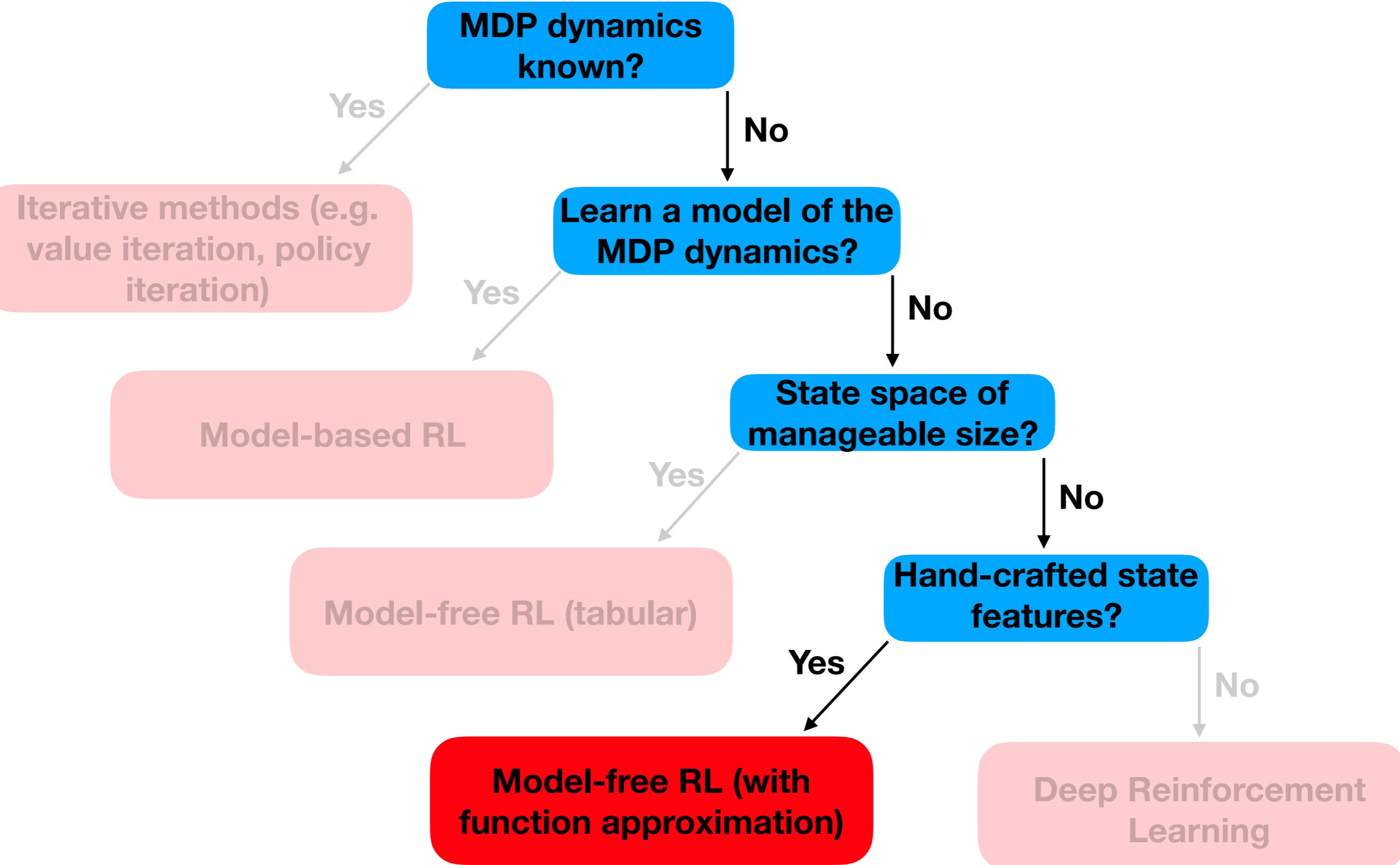
- Assume both SARSA and Q-learning use an epsilon-greedy policy
- SARSA outperforms Q-Learning in terms of online performance

# SARSA v. Q-Learning



- Assume both SARSA and Q-learning use an epsilon-greedy policy
- SARSA outperforms Q-Learning in terms of online performance
- Maybe epsilon-greedy isn't the smartest way to explore...





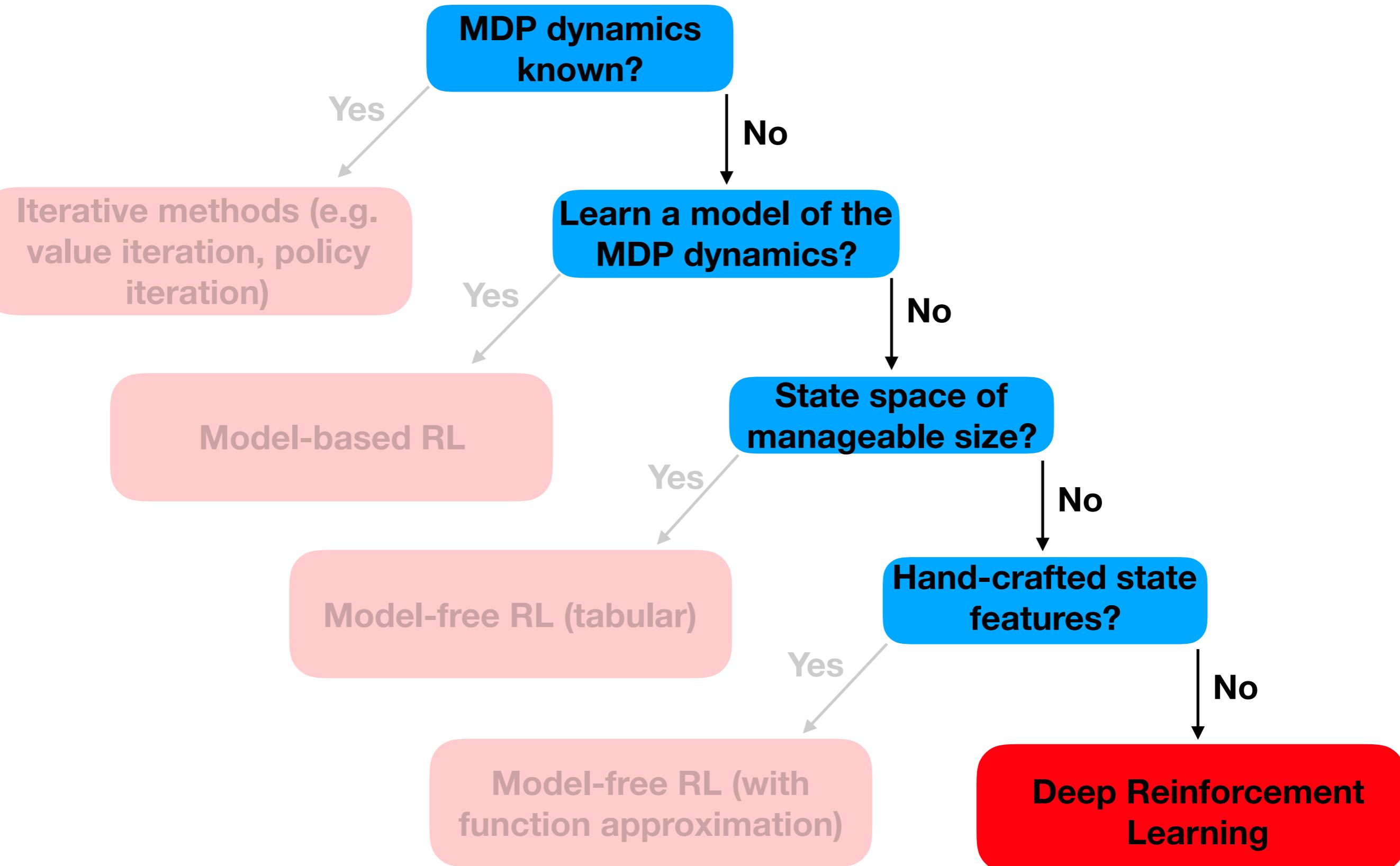
# Function Approximation

- Need to share value estimate information across states based on some notion of state similarity
- Use the same strategy from supervised learning - encode states to feature vectors, and learn a model with weights

# Function Approximation

- Need to share value estimate information across states based on some notion of state similarity
- Use the same strategy from supervised learning - encode states to feature vectors, and learn a model with weights
- Updating value estimates towards target values can now be done using gradient descent:

$$\min_{\mathbf{w}} \sum_{(s,a,r,s',a')} (\hat{Q}(s, a; \mathbf{w}) - \text{target})^2$$



# RL application: Breakout!

# Breakout Game Description

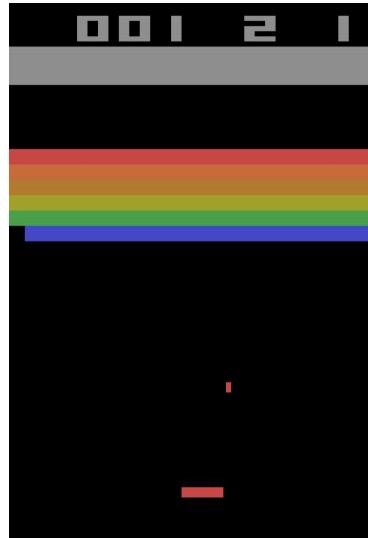
Formally:

- *Actions*
  - move\_paddle\_left
  - move\_paddle\_right
  - do\_not\_move\_paddle
- *Rewards*
  - If ball hits brick, reward = 1
  - Otherwise, reward = 0
- *End condition*
  - If ball falls off the screen,  
game ends

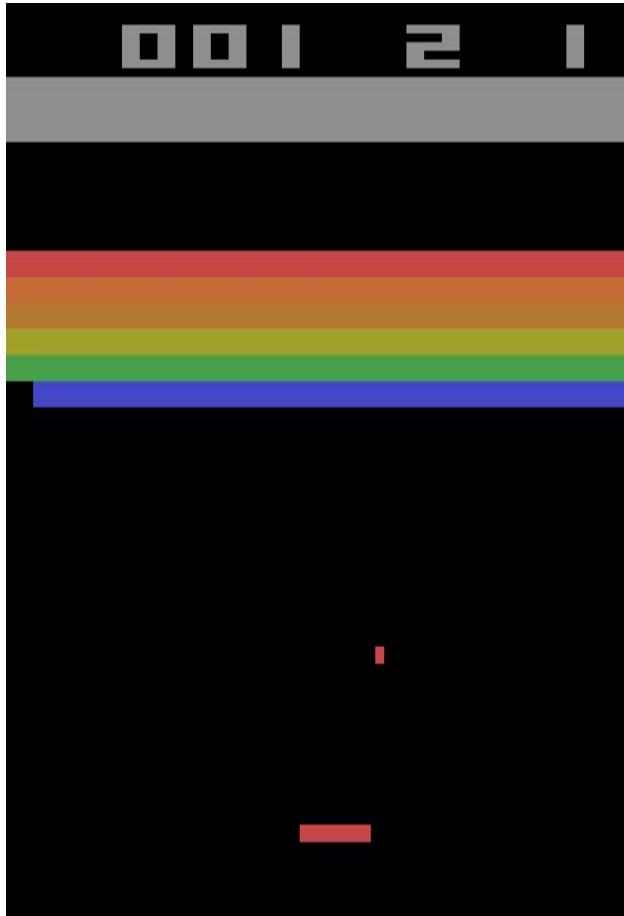


# Can we learn to control an agent directly from sensory input?

In Breakout, sensory input would be a game screen frame.



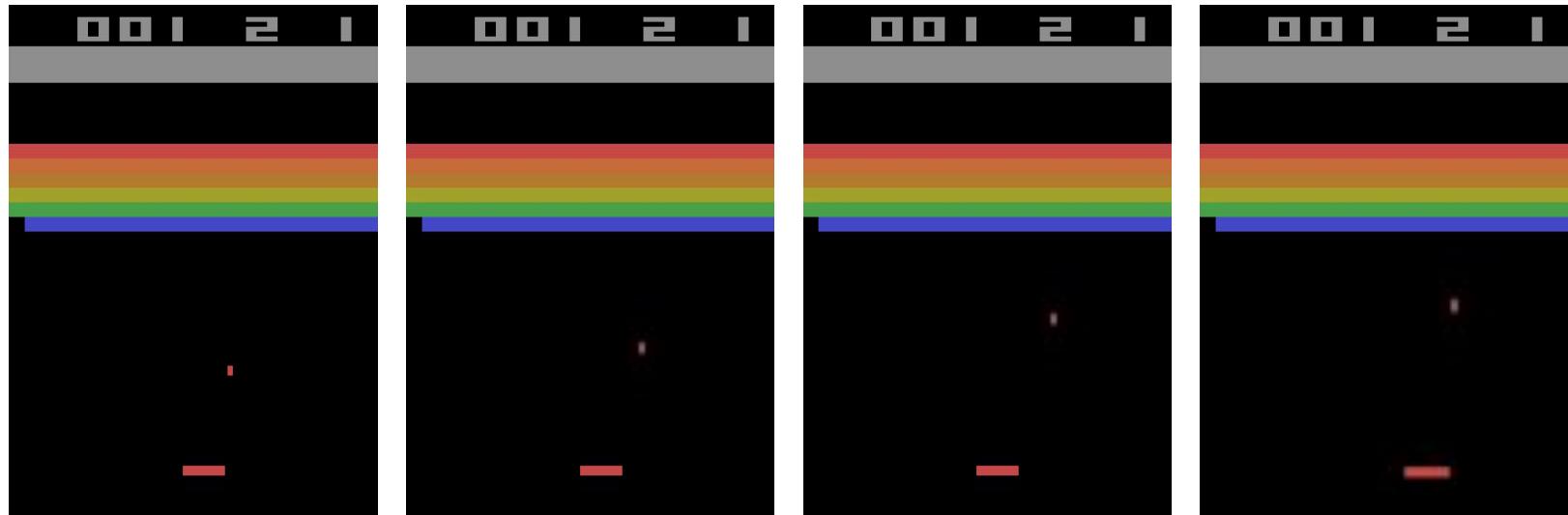
# Finding a state representation



Consider this frame.

- Can you capture information like direction of the ball?
- Can you capture velocity?

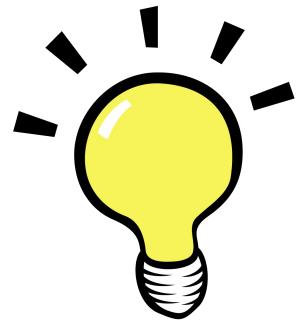
Use a small number of consecutive frames for each state.



# PROBLEM!!!

# states  $\approx 256^{84 \times 84 \times 4}$

(assume 84x84 pixels per screenshot, where each pixel can take on 256 values, and 4 screenshots per state)



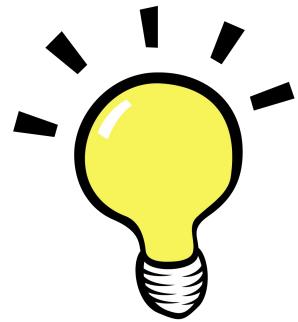
Use function approximation!

featurize our state space!

# 1st try: hand-designing features $\Phi(s, a)$

- Performance depends on the quality of features  $\Phi(s, a)$
- **Not generalized**
- Doesn't scale well with game complexity

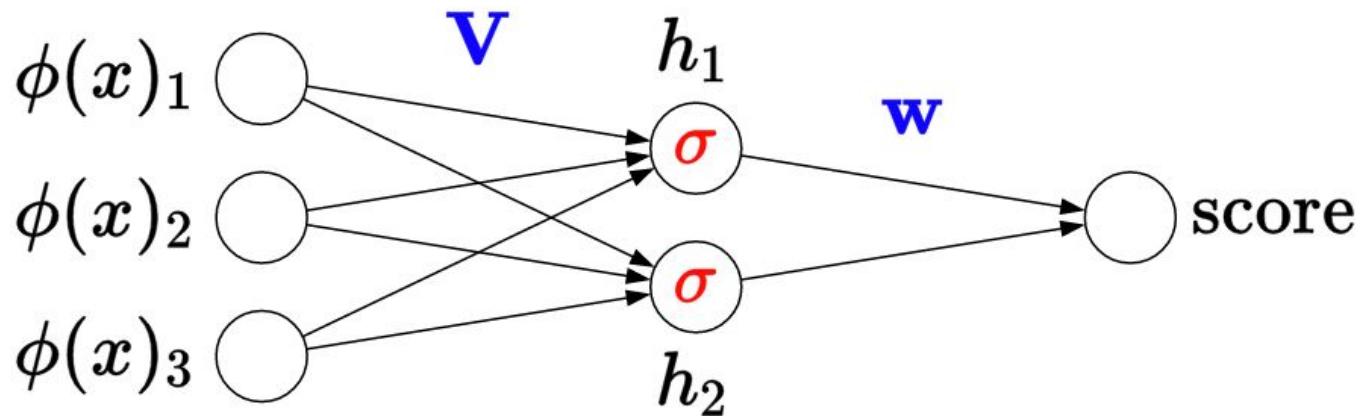
→ *Handcrafting features is very difficult!*



What if we automatically learned  
features from the pixels?

# Deep Neural Nets (Review)

Neural network:



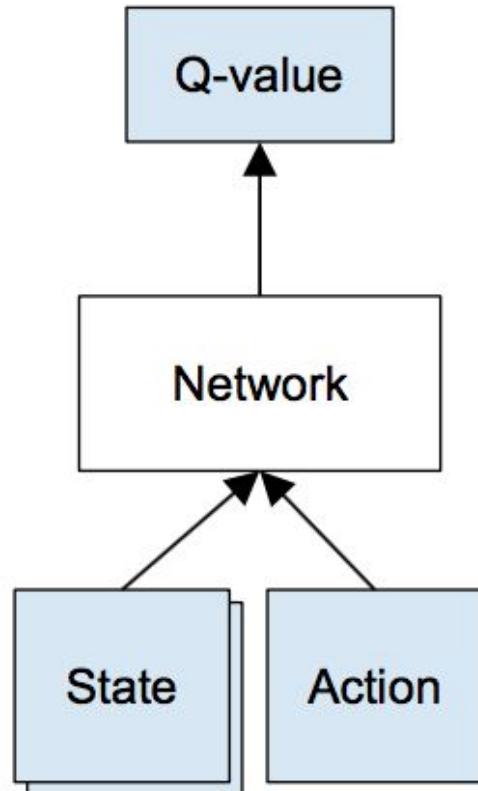
Intermediate hidden units:

$$h_j = \sigma(\mathbf{v}_j \cdot \phi(x)) \quad \sigma(z) = (1 + e^{-z})^{-1}$$

Output:

$$\text{score} = \mathbf{w} \cdot \mathbf{h}$$

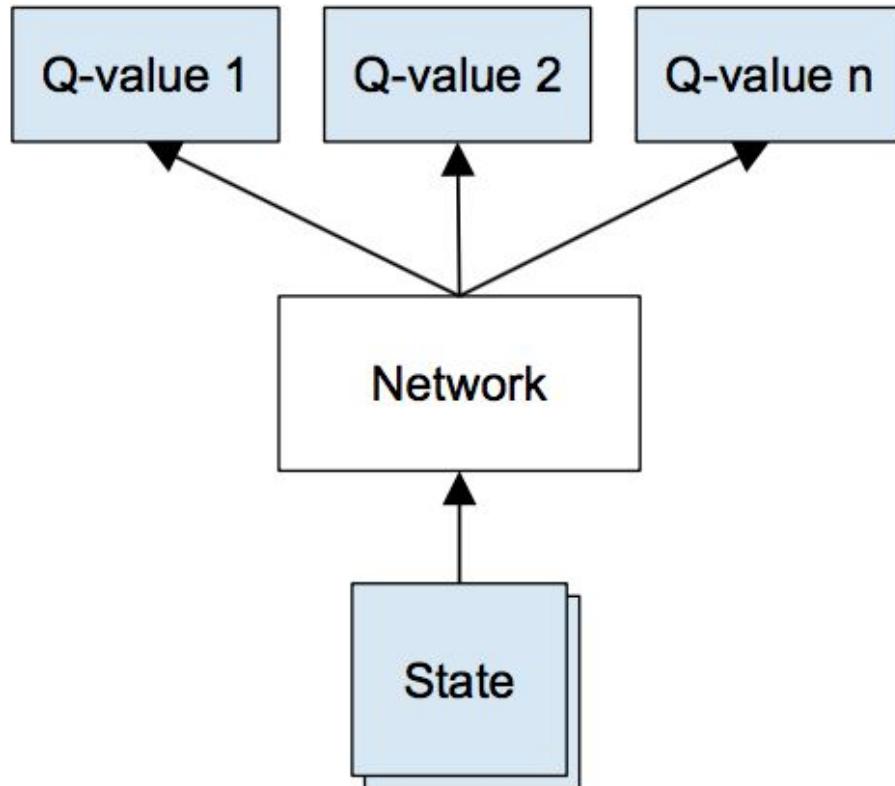
# Neural Networks as Q(s, a) approximators



- Input (s,a) pair to neural network.
- Neural network predicts the Q-value for (s,a) pair

Can we make this even more efficient?

# Neural Networks as Q(s, a) approximators



- State is the only input into the neural network.
- Network outputs a Q-value for every possible action.
- Action corresponding to the highest Q-value is chosen.

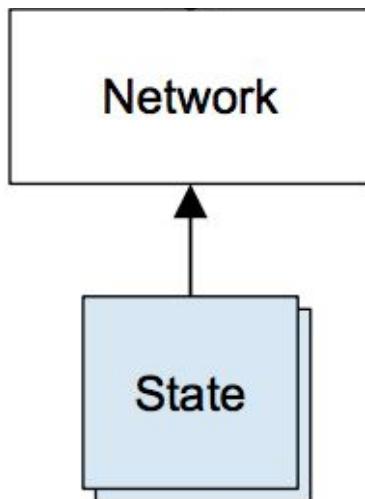
A single network to predict  $Q(s,a)$  for  
all possible states and actions!

# Training Deep-Q-Networks (DQN)

Network

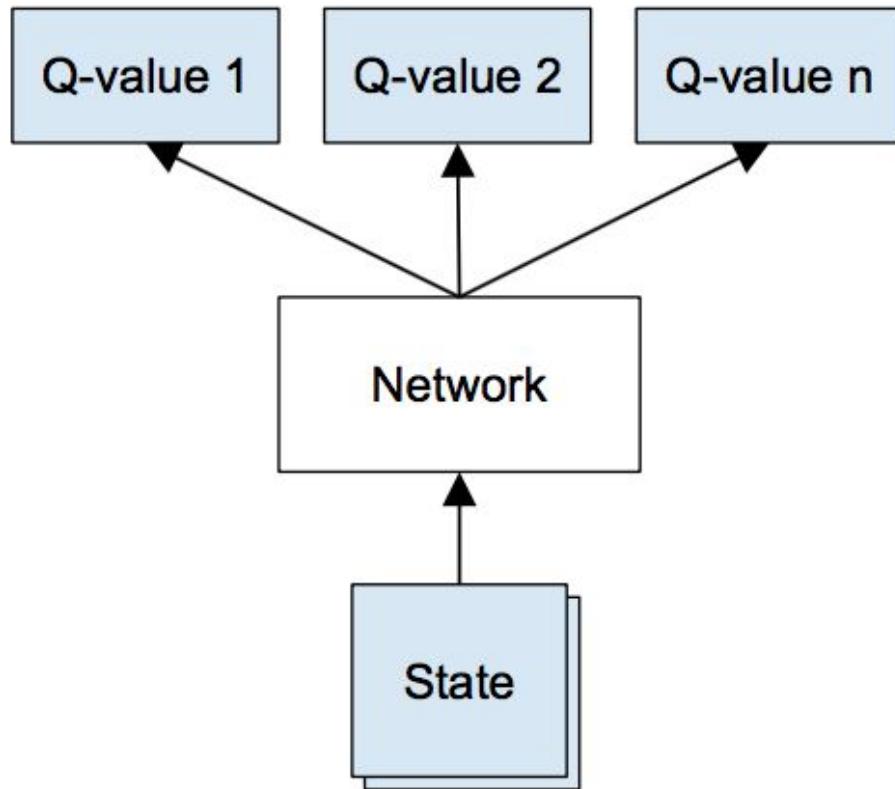
Initialize weights randomly!

# Training Deep-Q-Networks (DQN)



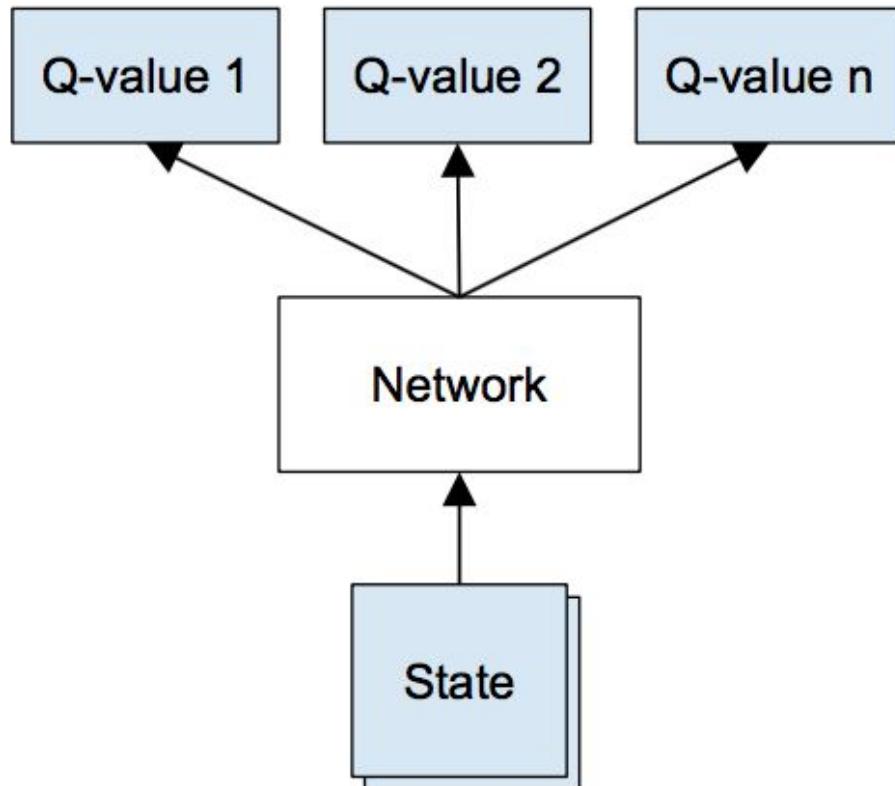
States are passed as input.

# Training Deep-Q-Networks (DQN)



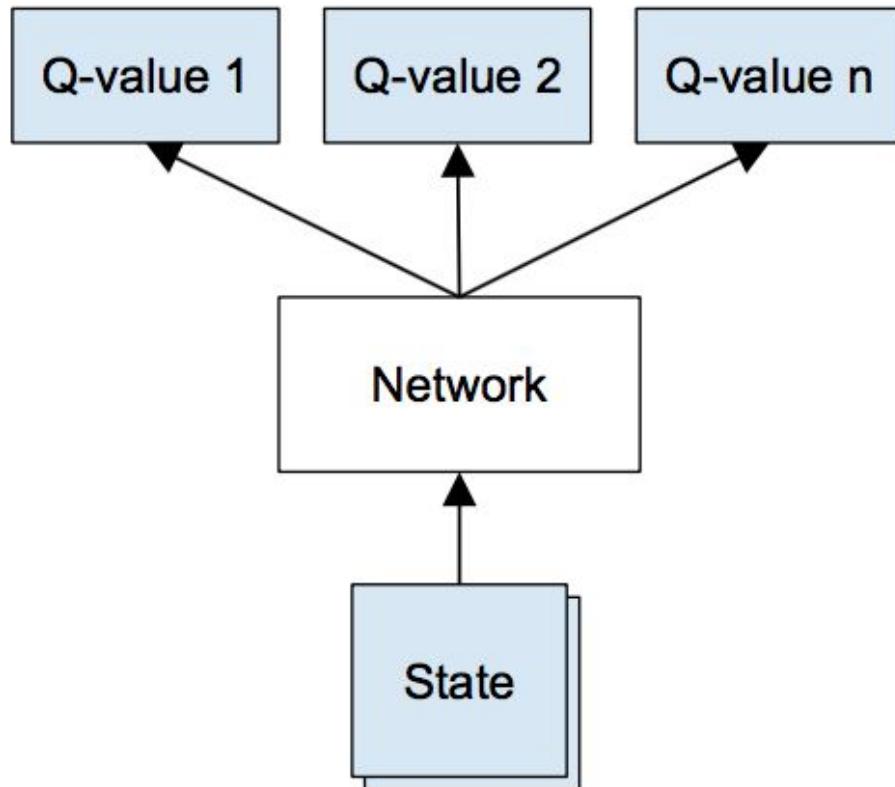
- Network outputs Q-values for each possible action.
- Action space for Breakout:  
[left, right, no-op]
- Since initial weights are random, Q-values are random.

# Training Deep-Q-Networks (DQN)



- Execute action that maximizes Q-value.
- Environment may or may not react to that action with a reward.
- Obtain next state.

# Training Deep-Q-Networks (DQN)



- Run gradient descent on Q-learning loss.

# Training Deep-Q-Networks

- Initialize weights randomly.
- Loop:
  - Obtain current state ( $s$ )
  - Run Neural Network on  $s$  to obtain Q-value for every action.
  - Execute action ( $a$ ) that maximizes Q-value.
  - Obtain reward ( $r$ ) and new state ( $s'$ ).
  - Perform gradient descent on Q-learning loss using  $(s, a, r, s')$

$$\min_{\mathbf{w}} \sum_{(s,a,r,s')} (\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_{\text{opt}}(s'))}_\text{target})^2$$

# Training Deep-Q-Networks - Additional Considerations

- Initialize weights randomly
- Initialize memory ( $D$ ) with capacity  $N$
- Loop:
  - Obtain current state ( $s$ )
  - Run Neural Network on  $s$  to obtain Q-value for every action
  - With probability  $\epsilon$ , execute random action ( $a$ )
  - Otherwise, execute action ( $a$ ) that maximizes Q-value
  - Obtain reward ( $r$ ) and new state ( $s'$ )
  - Store  $(s, a, r, s')$  in  $D$
  - Randomly sample  $(s, a, r, s')_D$  from  $D$
  - Perform gradient descent on Q-learning loss using  $(s, a, r, s')_D$

Let's watch Deep RL  
in action



ima...

-



02 | 3 |



<https://www.youtube.com/watch?v=V1eYniJ0Rnk>

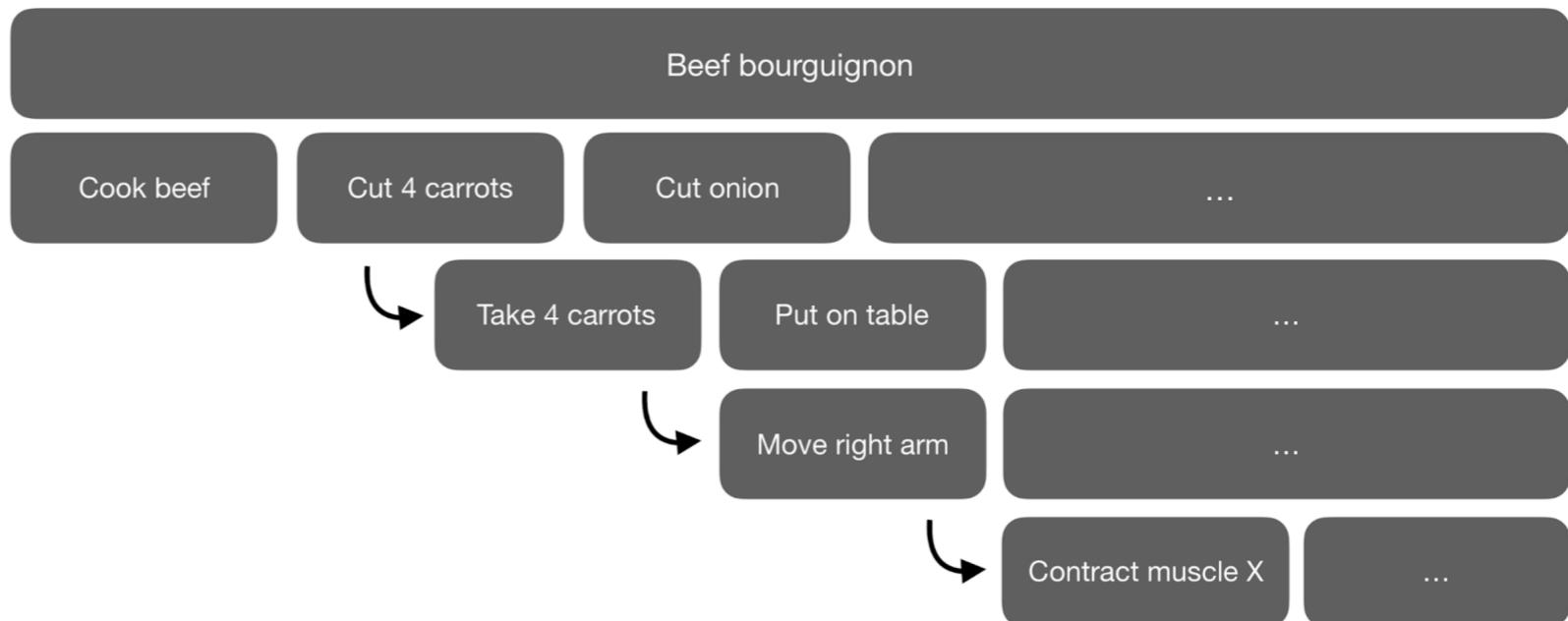
# Sources

- [1] Mnih, Volodymyr, et al. "Human-level control through deep reinforcement learning." *Nature* 518.7540 (2015): 529-533.
- [2] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." *arXiv preprint arXiv:1312.5602* (2013).

# **Different flavors of RL**

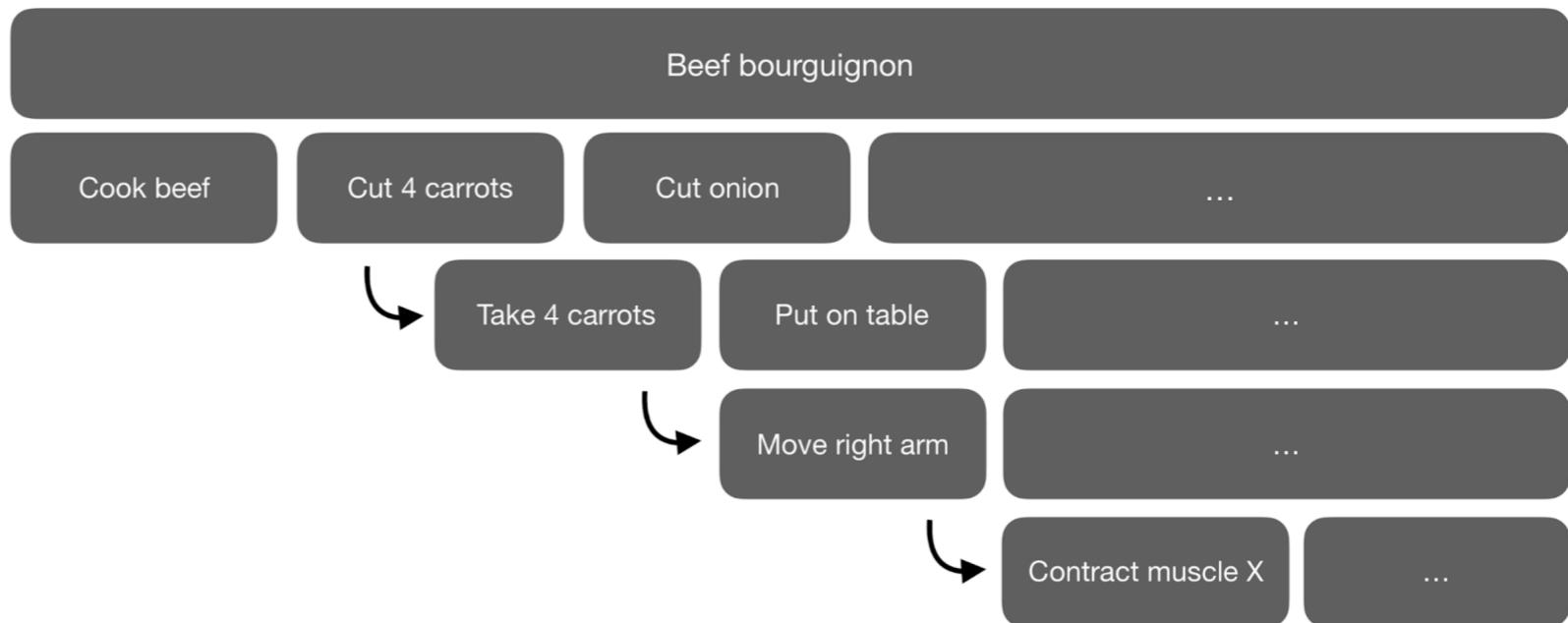
# Different flavors of RL

- **Hierarchical RL:**  
Compose low-level policies to form higher-level policies

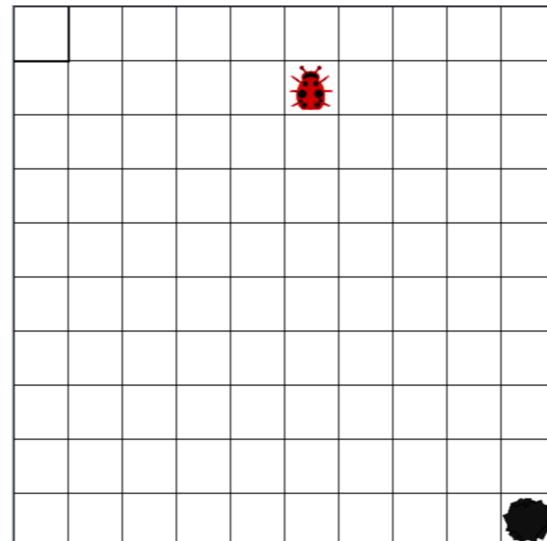


# Different flavors of RL

- **Hierarchical RL:** Compose low-level policies to form higher-level policies

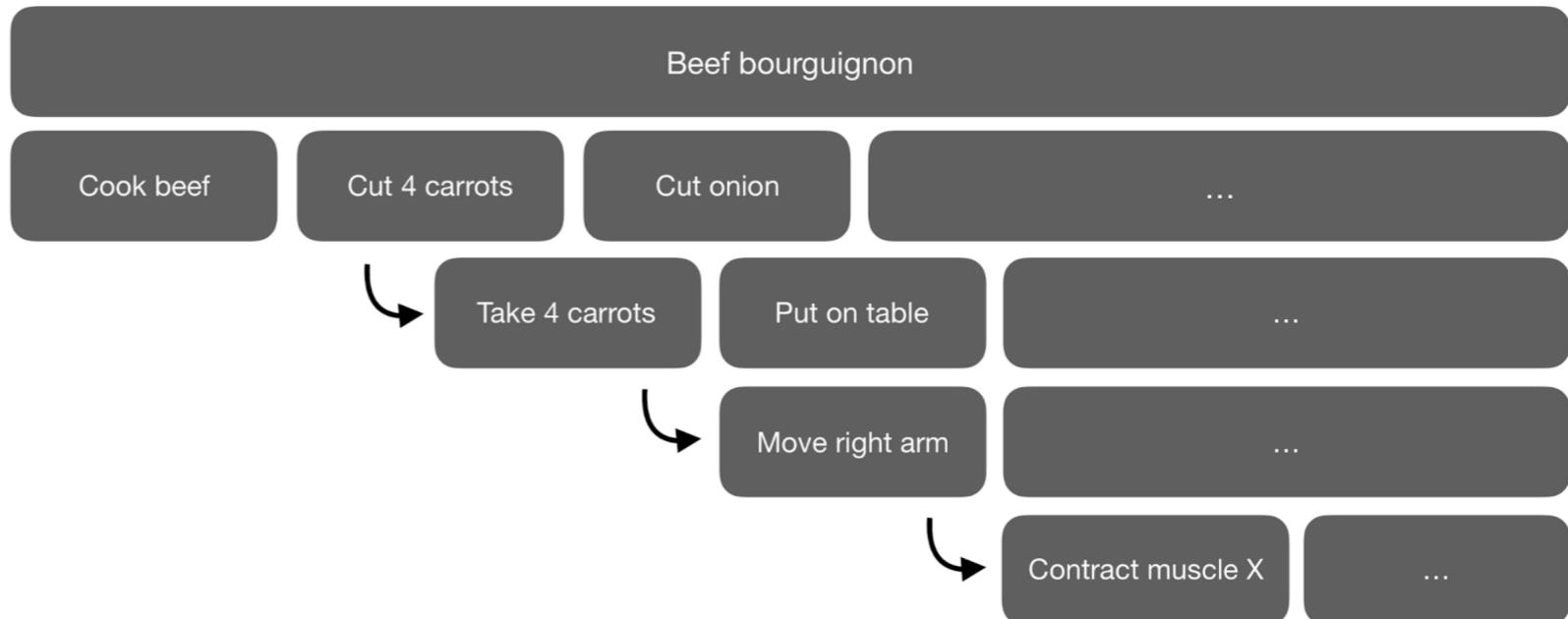


- **Meta RL:** Train an agent which can quickly adapt to small changes in the MDP

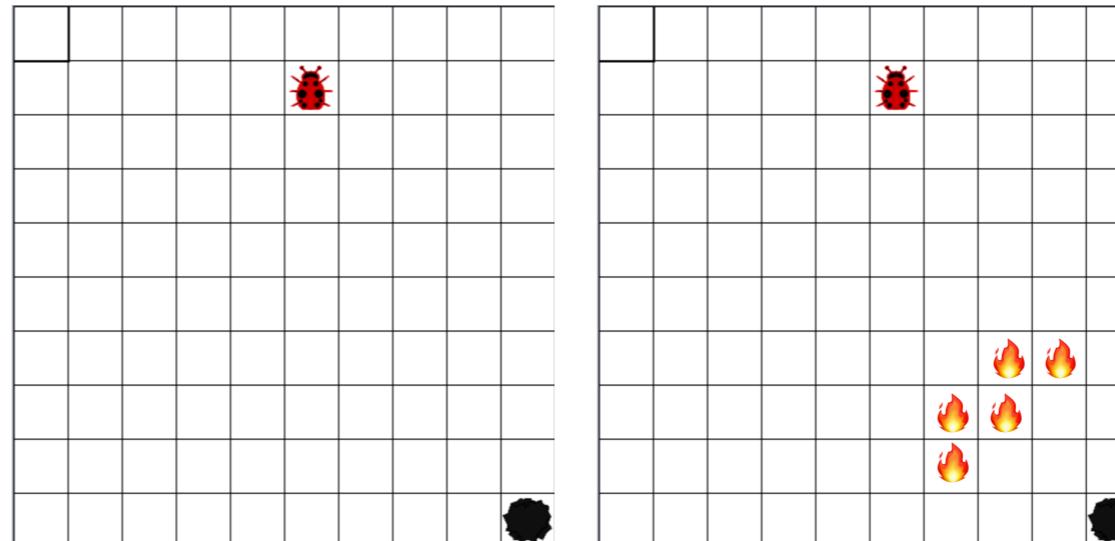


# Different flavors of RL

- **Hierarchical RL:** Compose low-level policies to form higher-level policies

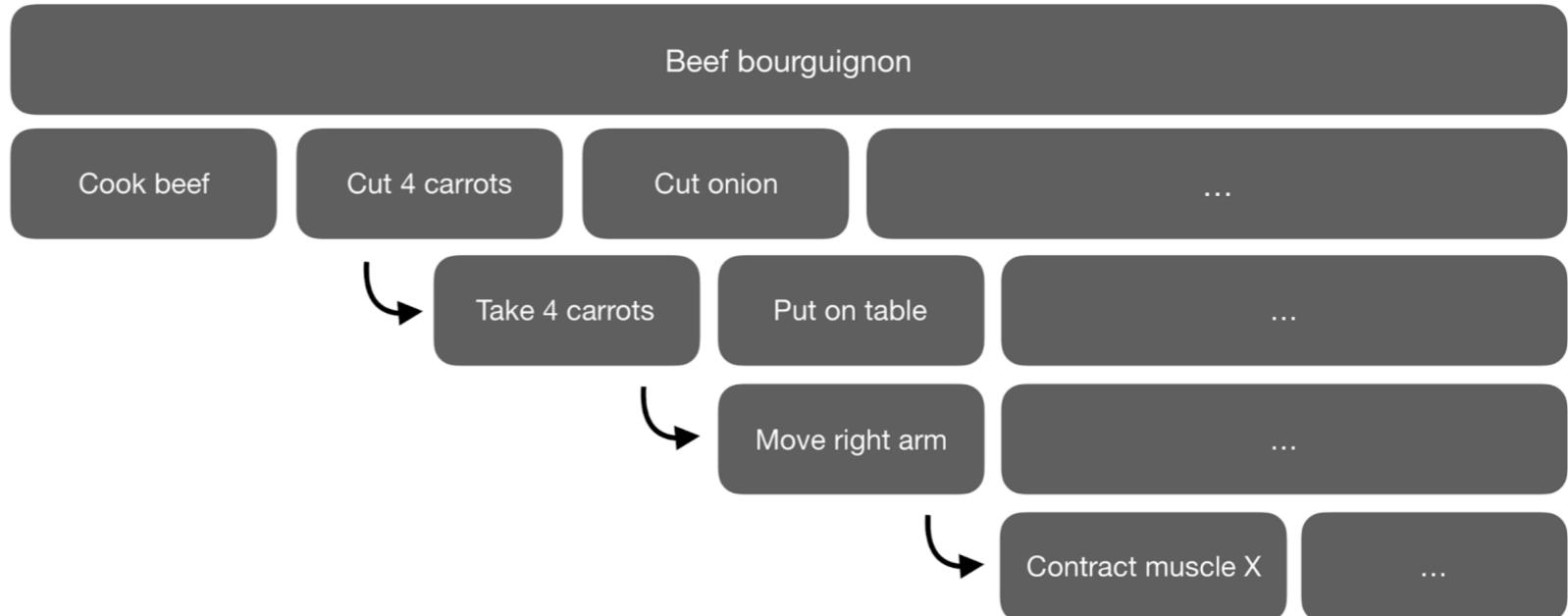


- **Meta RL:** Train an agent which can quickly adapt to small changes in the MDP

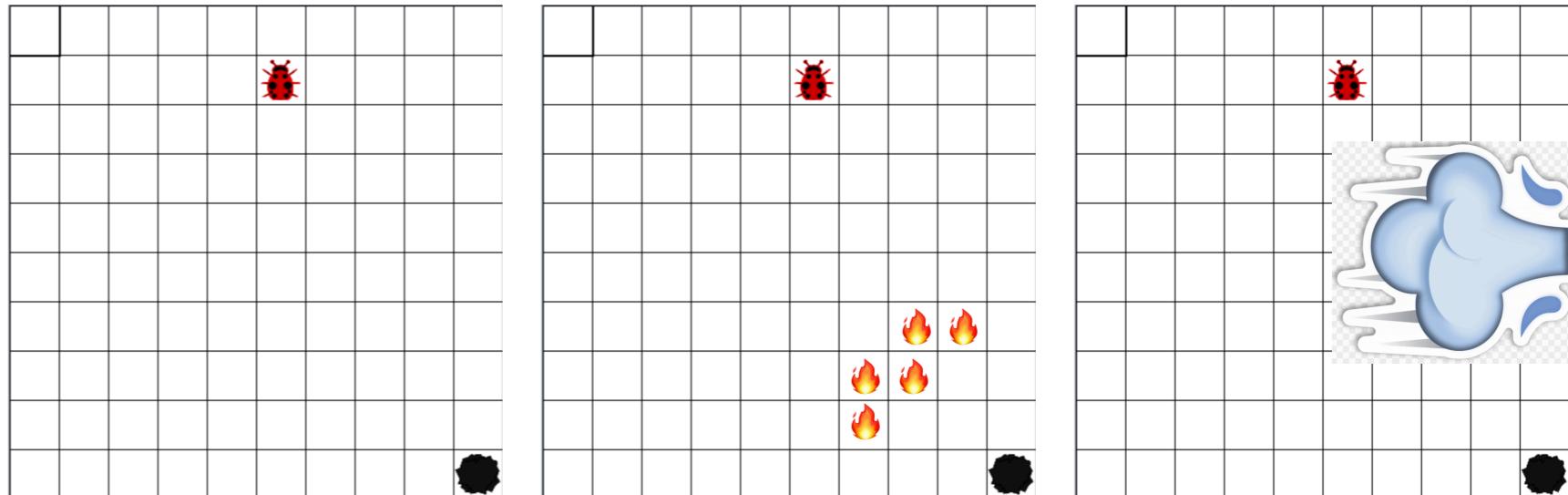


# Different flavors of RL

- **Hierarchical RL:**  
Compose low-level policies to form higher-level policies

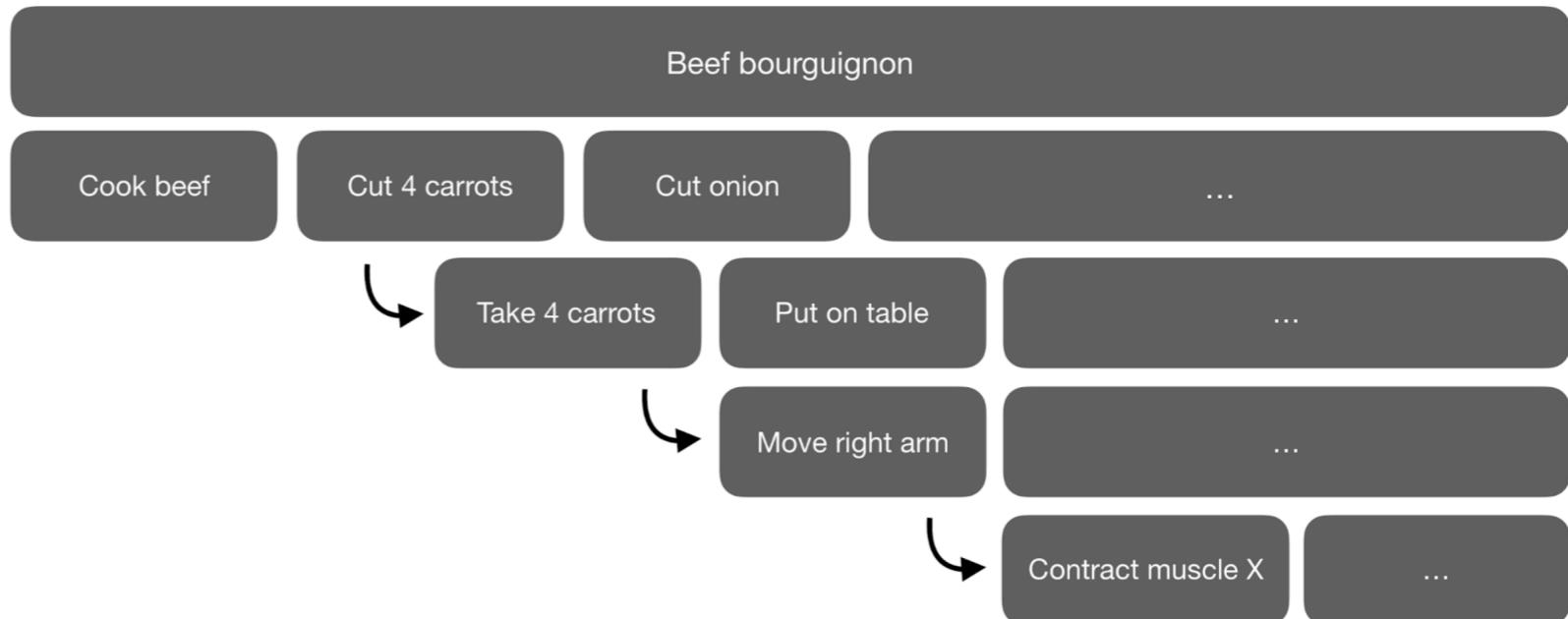


- **Meta RL:** Train an agent which can quickly adapt to small changes in the MDP

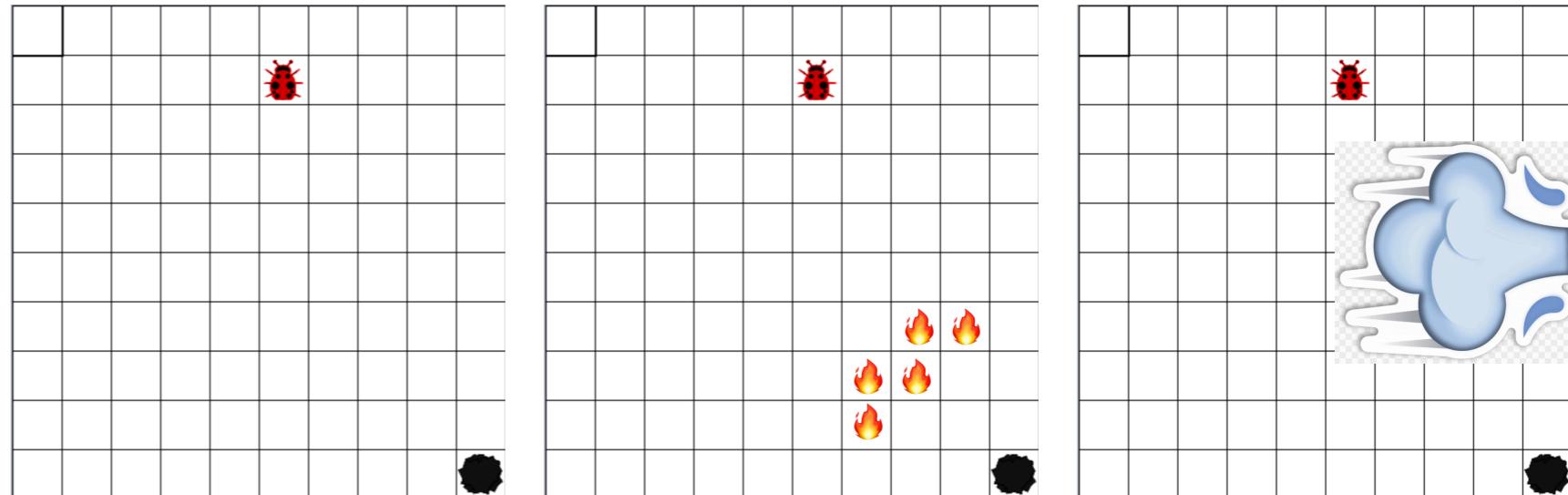


# Different flavors of RL

- **Hierarchical RL:** Compose low-level policies to form higher-level policies



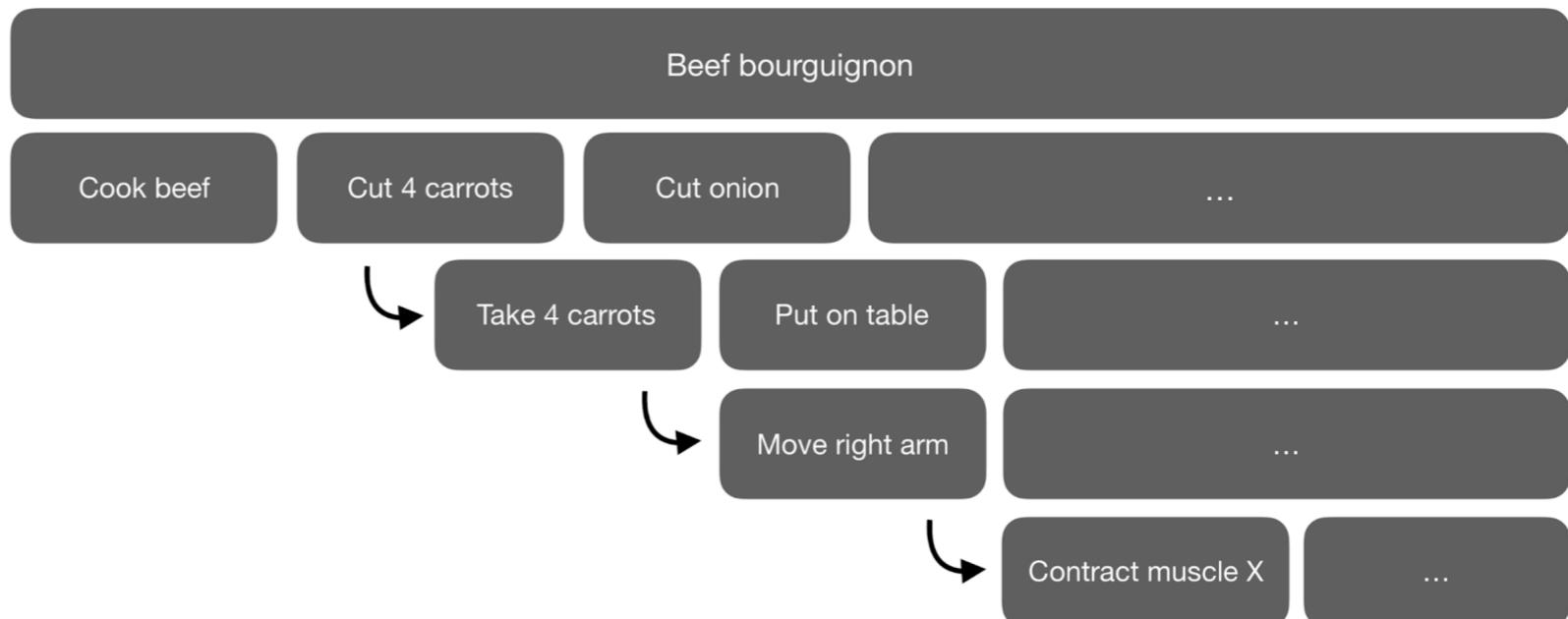
- **Meta RL:** Train an agent which can quickly adapt to small changes in the MDP



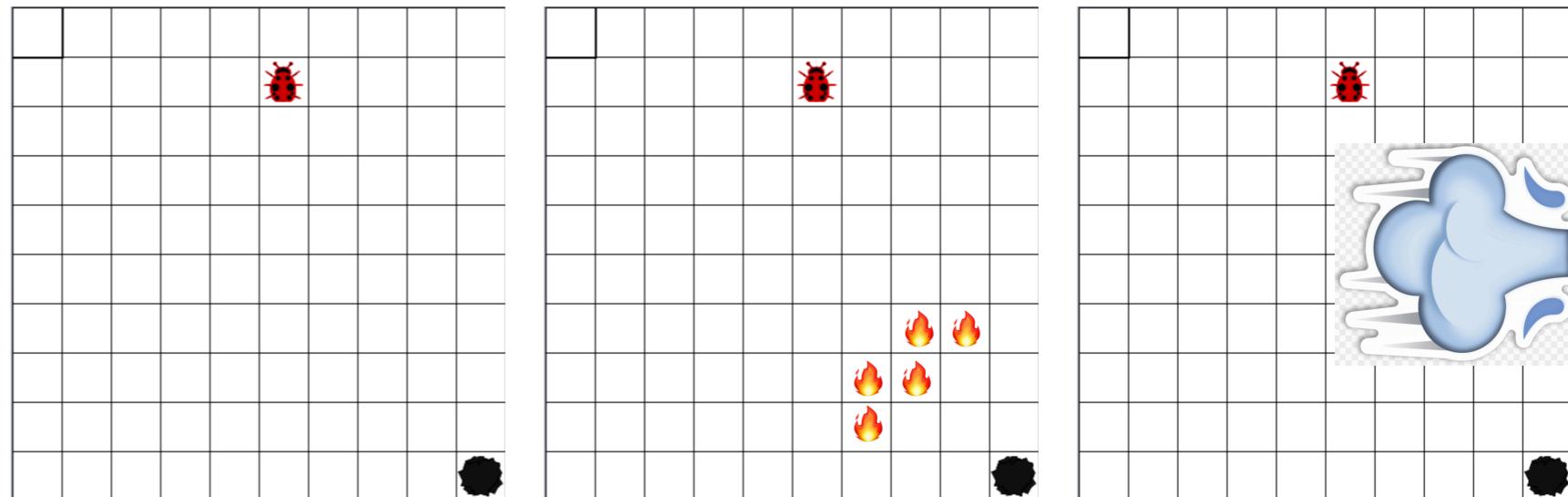
- **Inverse RL:** Infer agent preferences given experience

# Different flavors of RL

- **Hierarchical RL:** Compose low-level policies to form higher-level policies



- **Meta RL:** Train an agent which can quickly adapt to small changes in the MDP

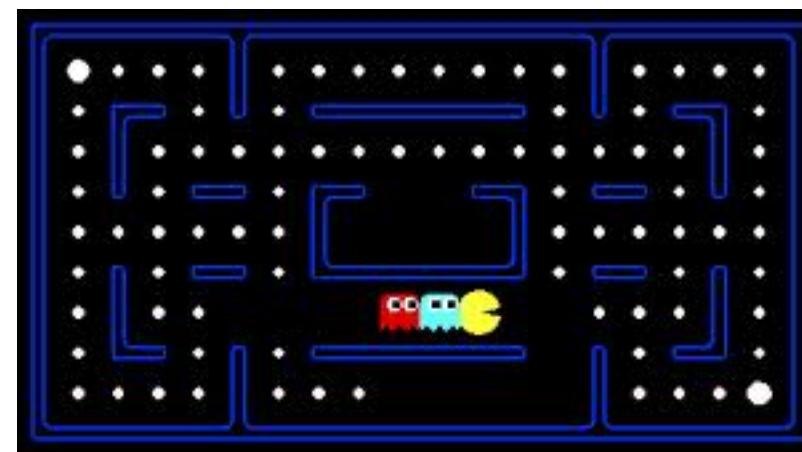


- **Inverse RL:** Infer agent preferences given experience

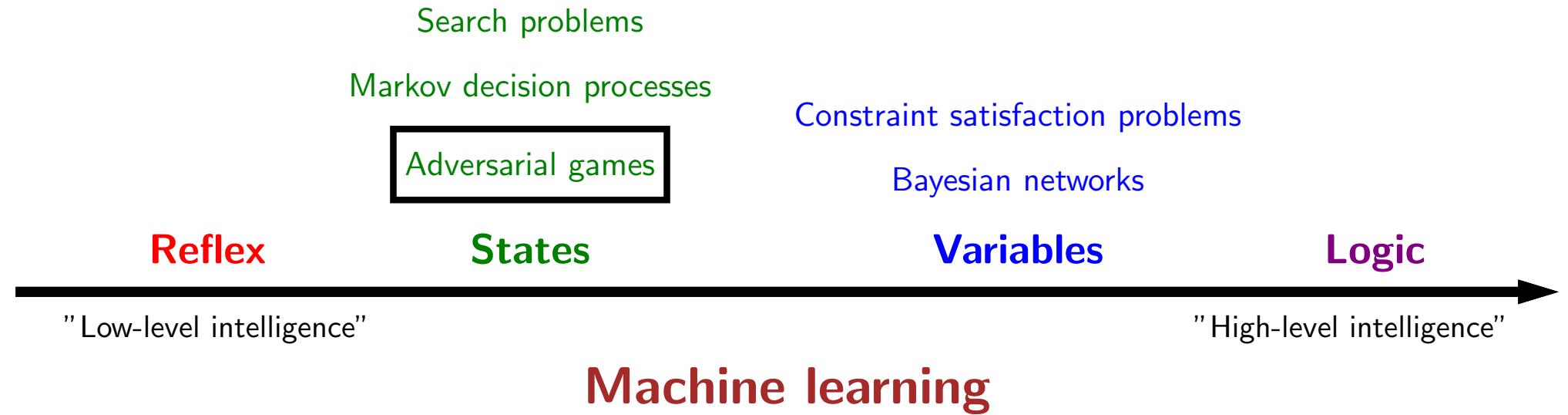
Reward  $\xrightarrow{\text{RL}}$  Behavior  
Behavior  $\xrightarrow{\text{IRL}}$  Reward



# Lecture 9: Games I

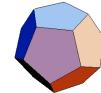


# Course plan



- This lecture will be about games, which have been one of the main testbeds for developing AI programs since the early days of AI. Games are distinguished from the other tasks that we've considered so far in this class in that they make explicit the presence of other agents, whose utility is not generally aligned with ours. Thus, the optimal strategy (policy) for us will depend on the strategies of these agents. Moreover, their strategies are often unknown and adversarial. How do we reason about this?

# A simple game



## Example: game 1

You choose one of the three bins.

I choose a number from that bin.

Your goal is to maximize the chosen number.

**A**

-50

50

**B**

1

3

**C**

-5

15

- Which bin should you pick? Depends on your mental model of the other player (me).
- If you think I'm working with you (unlikely), then you should pick A in hopes of getting 50. If you think I'm against you (likely), then you should pick B as to guard against the worst case (get 1). If you think I'm just acting uniformly at random, then you should pick C so that on average things are reasonable (get 5 in expectation).



# Roadmap

**Games, expectimax**

Minimax, expectiminimax

Evaluation functions

Alpha-beta pruning

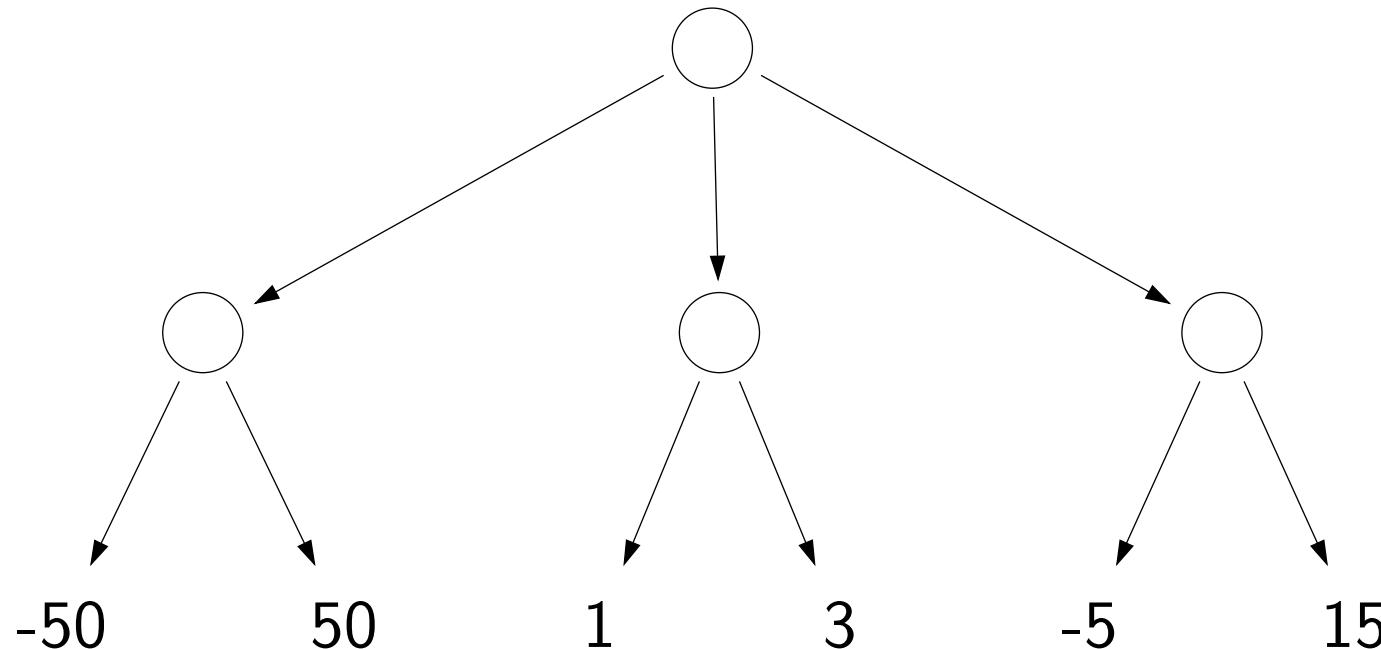
# Game tree



**Key idea: game tree**

Each node is a decision point for a player.

Each root-to-leaf path is a possible outcome of the game.



- Just as in search problems, we will use a tree to describe the possibilities of the game. This tree is known as a **game tree**.
- Note: We could also think of a game graph to capture the fact that there are multiple ways to arrive at the same game state. However, all our algorithms will operate on the tree rather than the graph since games generally have enormous state spaces, and we will have to resort to algorithms similar to backtracking search for search problems.

# Two-player zero-sum games

Players = {agent, opp}



## Definition: two-player zero-sum game

$s_{\text{start}}$ : starting state

$\text{Actions}(s)$ : possible actions from state  $s$

$\text{Succ}(s, a)$ : resulting state if choose action  $a$  in state  $s$

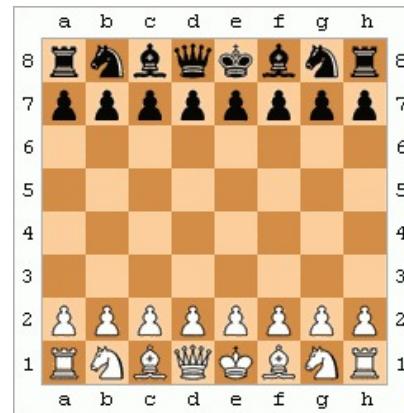
$\text{IsEnd}(s)$ : whether  $s$  is an end state (game over)

$\text{Utility}(s)$ : agent's utility for end state  $s$

$\text{Player}(s) \in \text{Players}$ : player who controls state  $s$

- In this lecture, we will specialize to **two-player zero-sum** games, such as chess. To be more precise, we will consider games in which people take turns (unlike rock-paper-scissors) and where the state of the game is fully-observed (unlike poker, where you don't know the other players' hands). By default, we will use the term **game** to refer to this restricted form.
- We will assume the two players are named agent (this is your program) and opp (the opponent). Zero-sum means that the utility of the agent is negative the utility of the opponent (equivalently, the sum of the two utilities is zero).
- Following our approach to search problems and MDPs, we start by formalizing a game. Since games are a type of state-based model, much of the skeleton is the same: we have a start state, actions from each state, a deterministic successor state for each state-action pair, and a test on whether a state is at the end.
- The main difference is that each state has a designated Player( $s$ ), which specifies whose turn it is. A player  $p$  only gets to choose the action for the states  $s$  such that  $\text{Player}(s) = p$ .
- Another difference is that instead of having edge costs in search problems or rewards in MDPs, we will instead have a utility function  $\text{Utility}(s)$  defined only at the end states. We could have used edge costs and rewards for games (in fact, that's strictly more general), but having all the utility at the end states emphasizes the all-or-nothing aspect of most games. You don't get utility for capturing pieces in chess; you only get utility if you win the game. This ultra-delayed utility makes games hard.

# Example: chess



Players = {white, black}

State  $s$ : (position of all pieces, whose turn it is)

Actions( $s$ ): legal chess moves that Player( $s$ ) can make

IsEnd( $s$ ): whether  $s$  is checkmate or draw

Utility( $s$ ):  $+\infty$  if white wins, 0 if draw,  $-\infty$  if black wins

- Chess is a canonical example of a two-player zero-sum game. In chess, the state must represent the position of all pieces, and importantly, whose turn it is (white or black).
- Here, we are assuming that white is the agent and black is the opponent. White moves first and is trying to maximize the utility, whereas black is trying to minimize the utility.
- In most games that we'll consider, the utility is degenerate in that it will be  $+\infty$ ,  $-\infty$ , or 0.

# Characteristics of games

- All the utility is at the end state



- Different players in control at different states



- There are two important characteristics of games which make them hard.
- The first is that the utility is only at the end state. In typical search problems and MDPs that we might encounter, there are costs and rewards associated with each edge. These intermediate quantities make the problem easier to solve. In games, even if there are cues that indicate how well one is doing (number of pieces, score), technically all that matters is what happens at the end. In chess, it doesn't matter how many pieces you capture, your goal is just to checkmate the opponent's king.
- The second is the recognition that there are other people in the world! In search problems, you (the agent) controlled all actions. In MDPs, we already hinted at the loss of control where nature controlled the chance nodes, but we assumed we knew what distribution nature was using to transition. Now, we have another player that controls certain states, who is probably out to get us.

# The halving game



## Problem: halving game

Start with a number  $N$ .

Players take turns either decrementing  $N$  or replacing it with  $\lfloor \frac{N}{2} \rfloor$ .

The player that is left with 0 wins.

[semi-live solution: HalvingGame]

# Policies

Deterministic policies:  $\pi_p(s) \in \text{Actions}(s)$

action that player  $p$  takes in state  $s$

Stochastic policies  $\pi_p(s, a) \in [0, 1]$ :

probability of player  $p$  taking action  $a$  in state  $s$

[semi-live solution: `humanPolicy`]

- Following our presentation of MDPs, we revisit the notion of a **policy**. Instead of having a single policy  $\pi$ , we have a policy  $\pi_p$  for each player  $p \in \text{Players}$ . We require that  $\pi_p$  only be defined when it's  $p$ 's turn; that is, for states  $s$  such that  $\text{Player}(s) = p$ .
- It will be convenient to allow policies to be stochastic. In this case, we will use  $\pi_p(s, a)$  to denote the probability of player  $p$  choosing action  $a$  in state  $s$ .
- We can think of an MDP as a game between the agent and nature. The states of the game are all MDP states  $s$  and all chance nodes  $(s, a)$ . It's the agent's turn on the MDP states  $s$ , and the agent acts according to  $\pi_{\text{agent}}$ . It's nature's turn on the chance nodes. Here, the actions are successor states  $s'$ , and nature chooses  $s'$  with probability given by the transition probabilities of the MDP:  $\pi_{\text{nature}}((s, a), s') = T(s, a, s')$ .

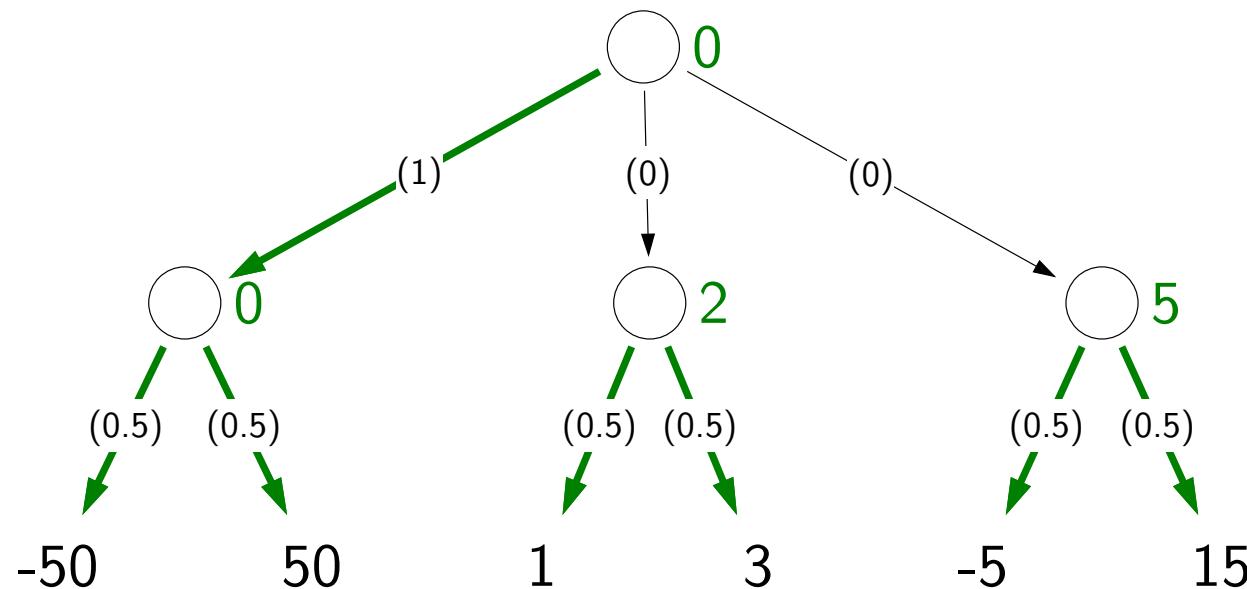
# Game evaluation example



## Example: game evaluation

$$\pi_{\text{agent}}(s) = A$$

$$\pi_{\text{opp}}(s, a) = \frac{1}{2} \text{ for } a \in \text{Actions}(s)$$

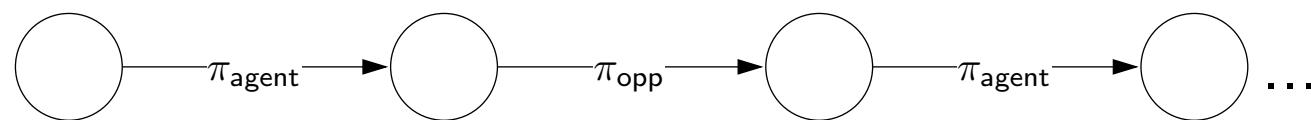


$$V_{\text{eval}}(s_{\text{start}}) = 0$$

- Given two policies  $\pi_{\text{agent}}$  and  $\pi_{\text{opp}}$ , what is the (agent's) expected utility? That is, if the agent and the opponent were to play their (possibly stochastic) policies a large number of times, what would be the average utility? Remember, since we are working with zero-sum games, the opponent's utility is the negative of the agent's utility.
- Given the game tree, we can recursively compute the value (expected utility) of each node in the tree. The value of a node is the weighted average of the values of the children where the weights are given by the probabilities of taking various actions given by the policy at that node.

# Game evaluation recurrence

Analogy: recurrence for policy evaluation in MDPs



Value of the game:

$$V_{\text{eval}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{agent}}(s, a) V_{\text{eval}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s, a) V_{\text{eval}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \end{cases}$$

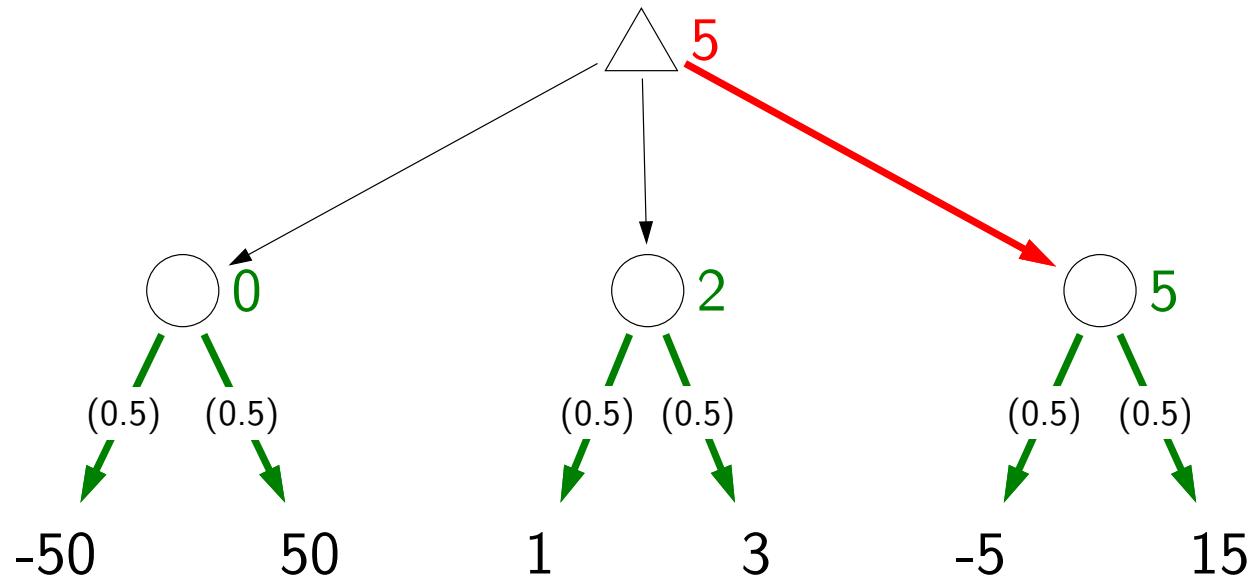
- More generally, we can write down a recurrence for  $V_{\text{eval}}(s)$ , which is the **value** (expected utility) of the game at state  $s$ .
- There are three cases: If the game is over ( $\text{IsEnd}(s)$ ), then the value is just the utility  $\text{Utility}(s)$ . If it's the agent's turn, then we compute the expectation over the value of the successor resulting from the agent choosing an action according to  $\pi_{\text{agent}}(s, a)$ . If it's the opponent's turn, we compute the expectation with respect to  $\pi_{\text{opp}}$  instead.

# Expectimax example



## Example: expectimax

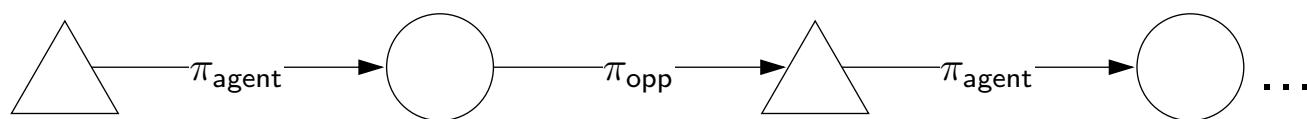
$$\pi_{\text{opp}}(s, a) = \frac{1}{2} \text{ for } a \in \text{Actions}(s)$$



- Game evaluation just gave us the value of the game with two fixed policies  $\pi_{\text{agent}}$  and  $\pi_{\text{opp}}$ . But we are not handed a policy  $\pi_{\text{agent}}$ ; we are trying to find the best policy. Expectimax gives us exactly that.
- In the game tree, we will now use an upward-pointing triangle to denote states where the player is maximizing over actions (we call them **max nodes**).
- At max nodes, instead of averaging with respect to a policy, we take the max of the values of the children.
- This computation produces the **expectimax value**  $V_{\text{exptmax}}(s)$  for a state  $s$ , which is the maximum expected utility of any agent policy when playing with respect to a fixed and known opponent policy  $\pi_{\text{opp}}$ .

# Expectimax recurrence

Analogy: recurrence for value iteration in MDPs



$$V_{\text{exptmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{opp}}(s, a) V_{\text{exptmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- The recurrence for the expectimax value  $V_{\text{exptmax}}$  is exactly the same as the one for the game value  $V_{\text{eval}}$ , except that we maximize over the agent's actions rather than following a fixed agent policy (which we don't know now).
- Where game evaluation was the analogue of policy evaluation for MDPs, expectimax is the analogue of value iteration.

Problem: don't know opponent's policy

Approach: assume the worst case





# Roadmap

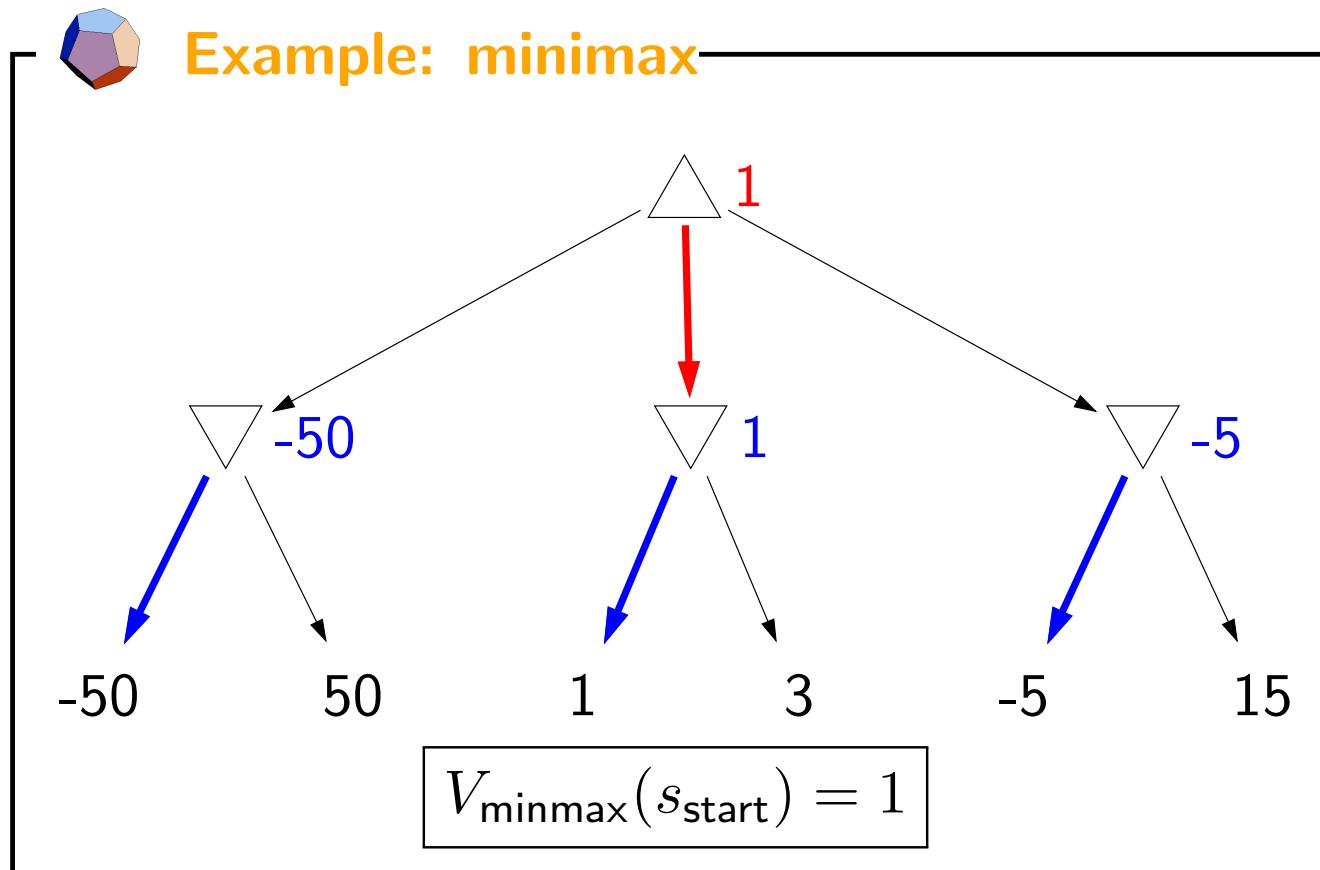
Games, expectimax

**Minimax, expectiminimax**

Evaluation functions

Alpha-beta pruning

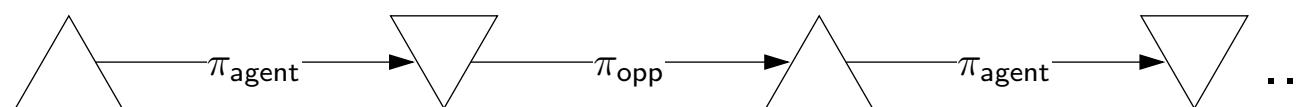
# Minimax example



- If we could perform some mind-reading and discover the opponent's policy, then we could maximally exploit it. However, in practice, we don't know the opponent's policy. So our solution is to assume the **worst case**, that is, the opponent is doing everything to minimize the agent's utility.
- In the game tree, we use an upside-down triangle to represent **min nodes**, in which the player minimizes the value over possible actions.
- Note that the policy for the agent changes from choosing the rightmost action (expectimax) to the middle action. Why is this?

# Minimax recurrence

No analogy in MDPs:



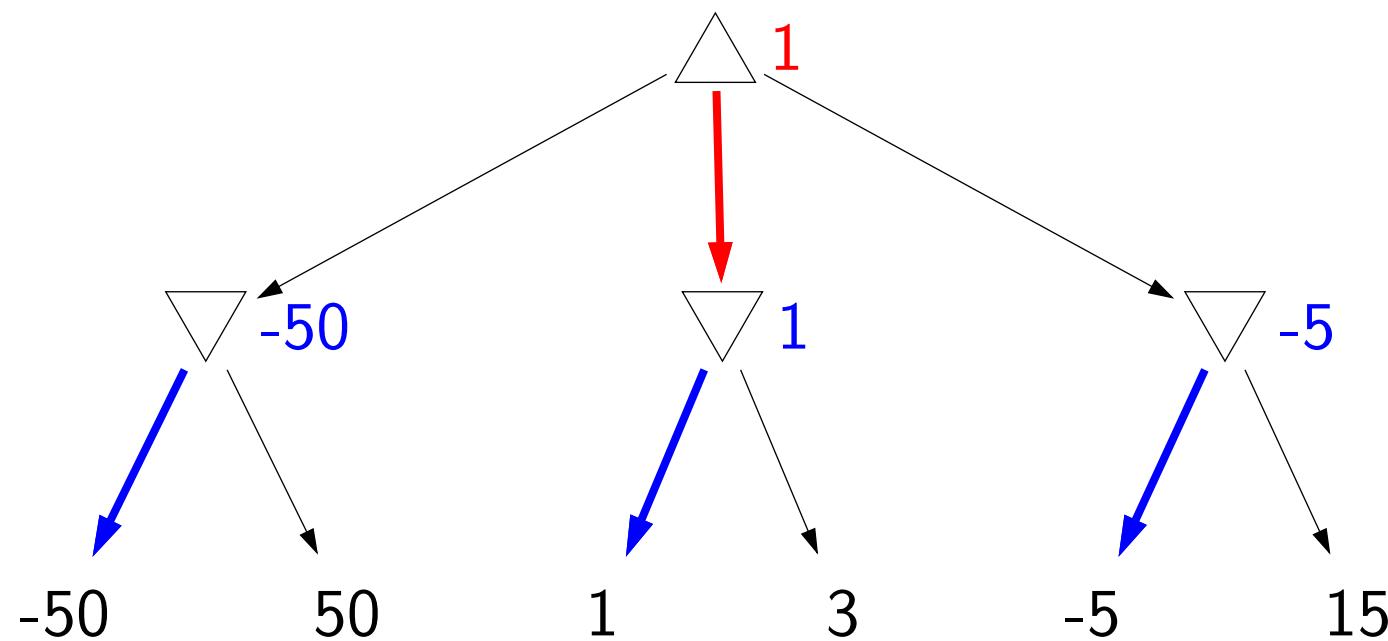
$$V_{\text{minmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{minmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \end{cases}$$

- The general recurrence for the minimax value is the same as expectimax, except that the expectation over the opponent's policy is replaced with a minimum over the opponent's possible actions. Note that the minimax value does not depend on any policies at all: it's just the agent and opponent playing optimally with respect to each other.

# Extracting minimax policies

$$\pi_{\max}(s) = \arg \max_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a))$$

$$\pi_{\min}(s) = \arg \min_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a))$$



- Having computed the minimax value  $V_{\text{minmax}}$ , we can extract the minimax policies  $\pi_{\text{max}}$  and  $\pi_{\text{min}}$  by just taking the action that leads to the state with the maximum (or minimum) value.
- In general, having a value function tells you which states are good, from which it's easy to set the policy to move to those states (provided you know the transition structure, which we assume we know here).

# The halving game



## Problem: halving game

Start with a number  $N$ .

Players take turns either decrementing  $N$  or replacing it with  $\lfloor \frac{N}{2} \rfloor$ .

The player that is left with 0 wins.

[semi-live solution: `minimaxPolicy`]

# Face off

Recurrences produces policies:

$$\begin{aligned} V_{\text{exptmax}} &\Rightarrow \pi_{\text{exptmax}(7)}, \pi_7 \text{ (some opponent)} \\ V_{\text{minmax}} &\Rightarrow \pi_{\text{max}}, \pi_{\text{min}} \end{aligned}$$

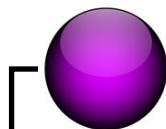
Play policies against each other:

	$\pi_{\text{min}}$	$\pi_7$
$\pi_{\text{max}}$	$V(\pi_{\text{max}}, \pi_{\text{min}})$	$V(\pi_{\text{max}}, \pi_7)$
$\pi_{\text{exptmax}(7)}$	$V(\pi_{\text{exptmax}(7)}, \pi_{\text{min}})$	$V(\pi_{\text{exptmax}(7)}, \pi_7)$

What's the relationship between these values?

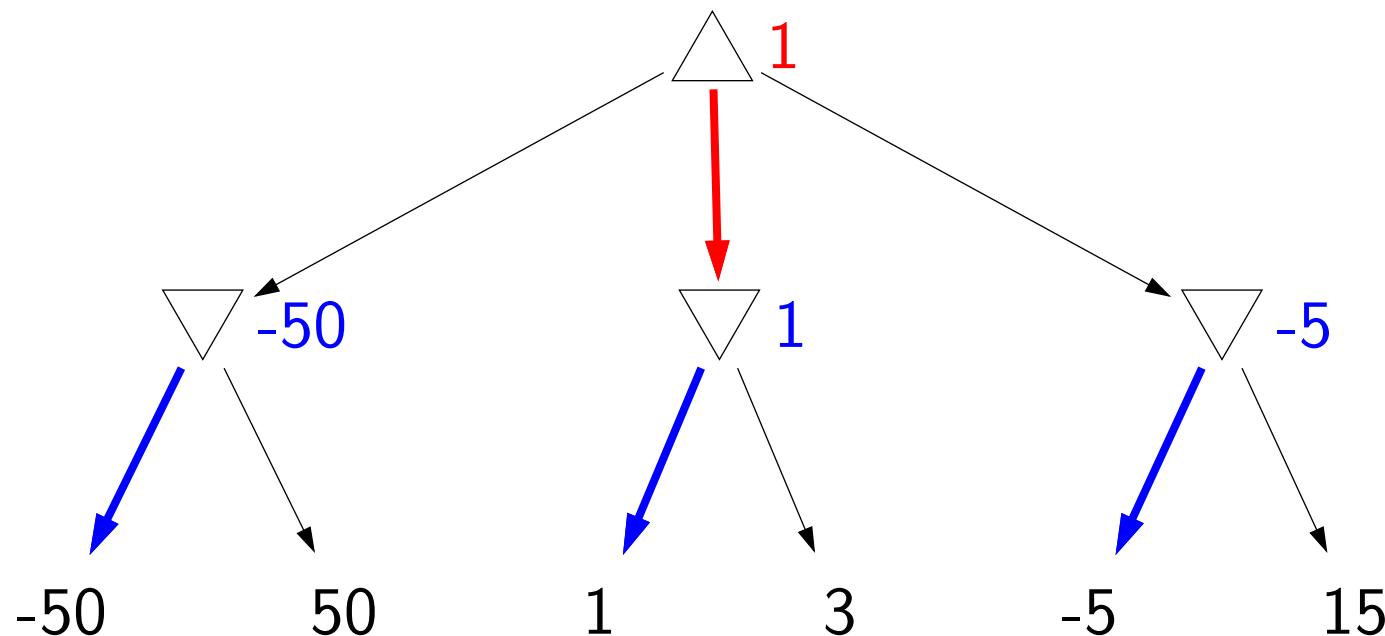
- So far, we have seen how expectimax and minimax recurrences produce policies.
- The expectimax recurrence computes the best policy  $\pi_{\text{exptmax}(7)}$  against a fixed opponent policy (say  $\pi_7$  for concreteness).
- The minimax recurrence computes the best policy  $\pi_{\text{max}}$  against the best opponent policy  $\pi_{\text{min}}$ .
- Now, whenever we take an agent policy  $\pi_{\text{agent}}$  and an opponent policy  $\pi_{\text{opp}}$ , we can play them against each other, which produces an expected utility via game evaluation, which we denote as  $V(\pi_{\text{agent}}, \pi_{\text{opp}})$ .
- How do the four game values of different combination of policies relate to each other?

# Minimax property 1



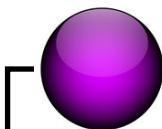
**Proposition: best against minimax opponent**

$$V(\pi_{\max}, \pi_{\min}) \geq V(\pi_{\text{agent}}, \pi_{\min}) \text{ for all } \pi_{\text{agent}}$$



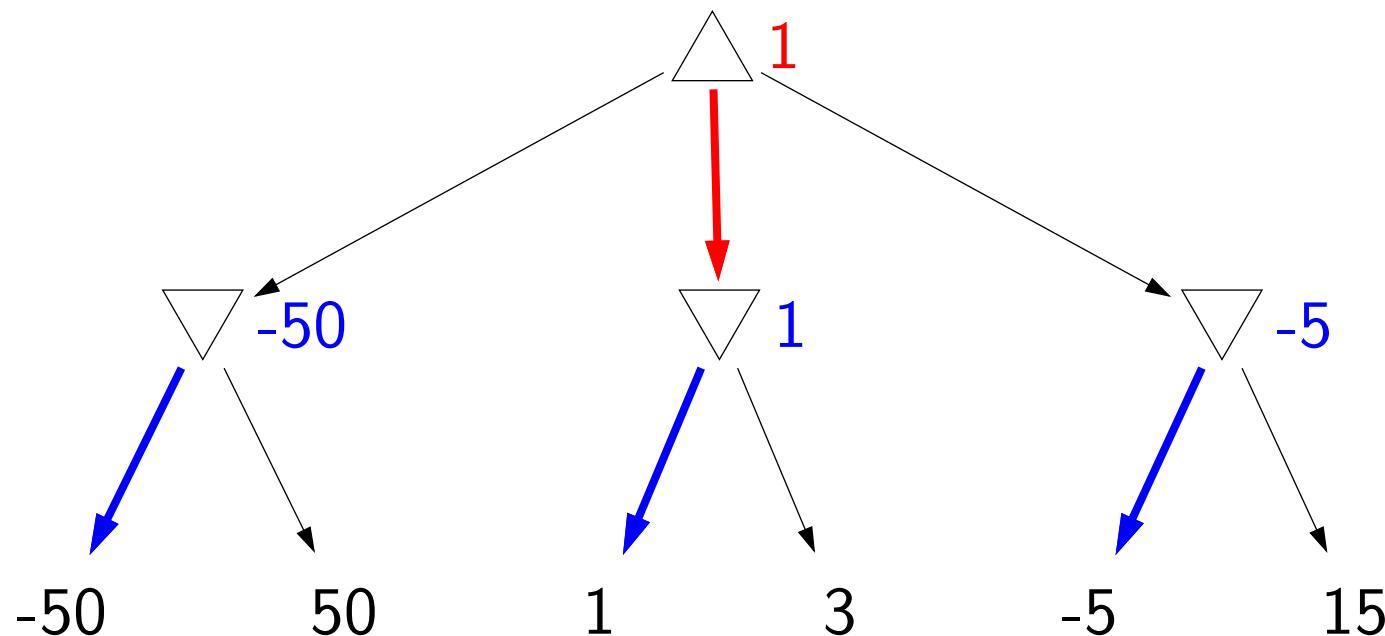
- Recall that  $\pi_{\max}$  and  $\pi_{\min}$  are the minimax agent and opponent policies, respectively. The first property is if the agent were to change her policy to any  $\pi_{\text{agent}}$ , then the agent would be no better off (and in general, worse off).
- From the example, it's intuitive that this property should hold. To prove it, we can perform induction starting from the leaves of the game tree, and show that the minimax value of each node is the highest over all possible policies.

# Minimax property 2



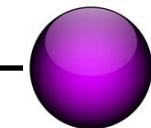
**Proposition: lower bound against any opponent**

$$V(\pi_{\max}, \pi_{\min}) \leq V(\pi_{\max}, \pi_{\text{opp}}) \text{ for all } \pi_{\text{opp}}$$



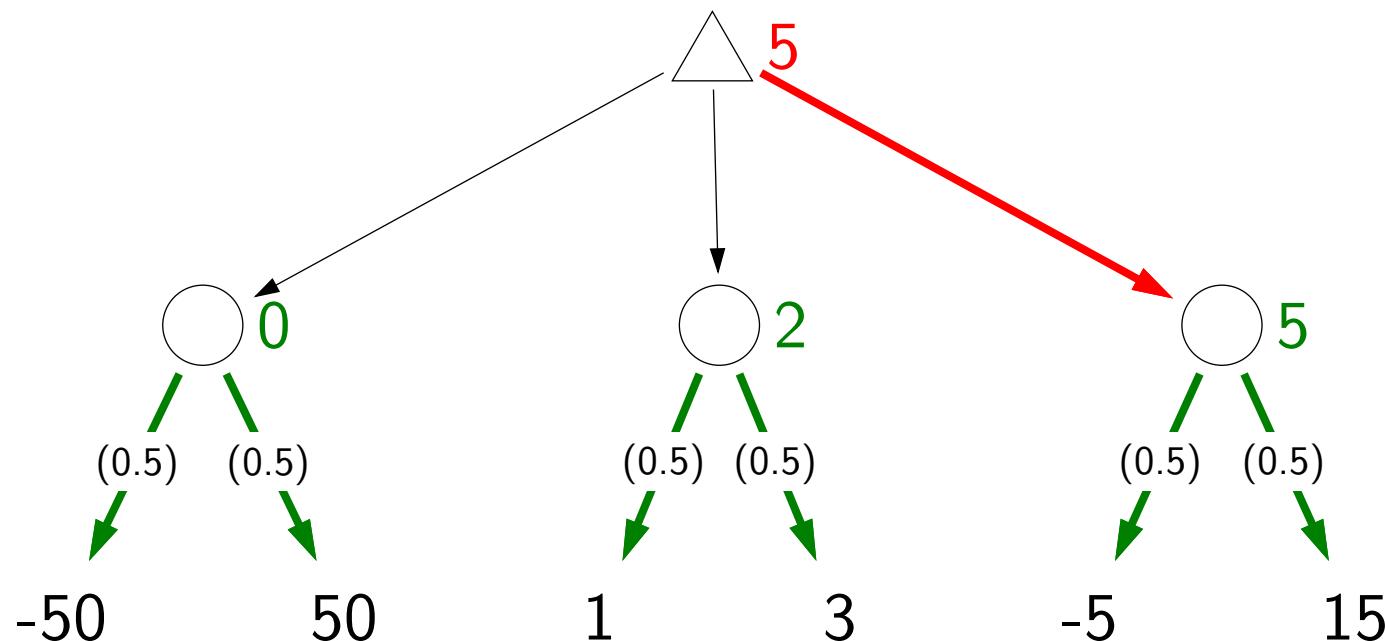
- The second property is the analogous statement for the opponent: if the opponent changes his policy from  $\pi_{\min}$  to  $\pi_{\text{opp}}$ , then he will be no better off (the value of the game can only increase).
- From the point of view of the agent, this can be interpreted as guarding against the worst case. In other words, if we get a minimax value of 1, that means no matter what the opponent does, the agent is guaranteed at least a value of 1. As a simple example, if the minimax value is  $+\infty$ , then the agent is guaranteed to win, provided it follows the minimax policy.

# Minimax property 3



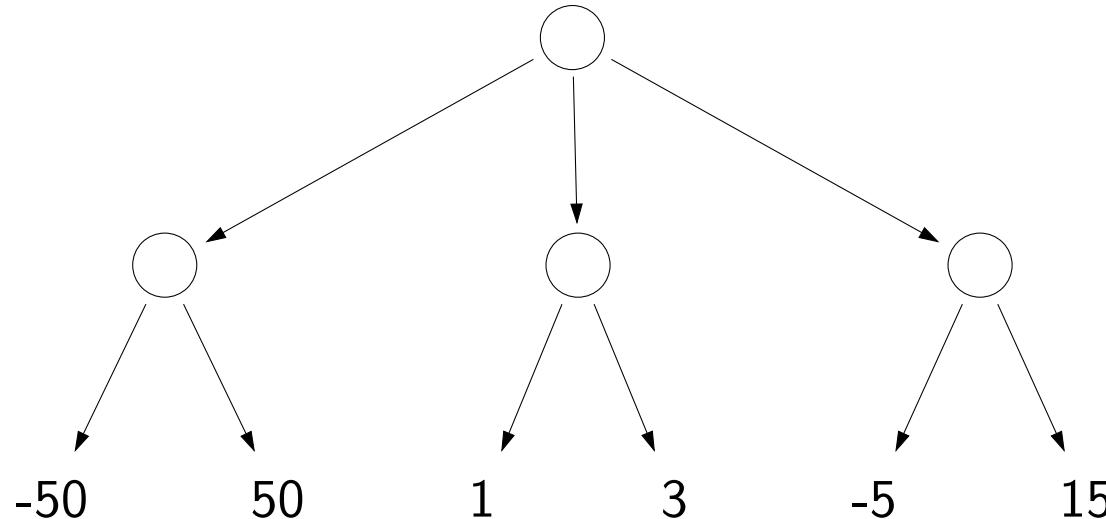
**Proposition: not optimal if opponent is known**

$$V(\pi_{\max}, \pi_7) \leq V(\pi_{\text{exptmax}(7)}, \pi_7) \text{ for opponent } \pi_7$$



- However, following the minimax policy might not be optimal for the agent if the opponent is known to be not playing the adversarial (minimax) policy.
- Consider the running example where the agent chooses A, B, or C and the opponent chooses a bin. Suppose the agent is playing  $\pi_{\max}$ , but the opponent is playing a stochastic policy  $\pi_7$  corresponding to choosing an action uniformly at random.
- Then the game value here would be 2 (which is larger than the minimax value 1, as guaranteed by property 2). However, if we followed the expectimax  $\pi_{\text{exptmax}(7)}$ , then we would have gotten a value of 5, which is even higher.

# Relationship between game values



$$\pi_{\max} \quad \begin{matrix} \pi_{\min} \\ V(\pi_{\max}, \pi_{\min}) \\ 1 \\ \vee \end{matrix} \leq \begin{matrix} \pi_7 \\ V(\pi_{\max}, \pi_7) \\ 2 \\ \wedge \end{matrix}$$

$$\pi_{\text{exptmax}(7)} \quad \begin{matrix} V(\pi_{\text{exptmax}(7)}, \pi_{\min}) \\ -5 \end{matrix} \quad \begin{matrix} V(\pi_{\text{exptmax}(7)}, \pi_7) \\ 5 \end{matrix}$$

- Putting the three properties together, we obtain a chain of inequalities that allows us to relate all four game values.
- We can also compute these values concretely for the running example.

# A modified game



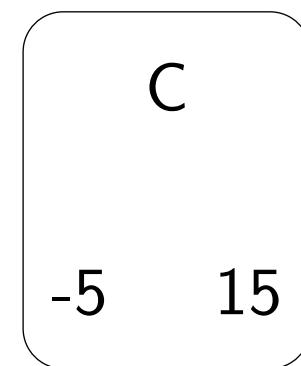
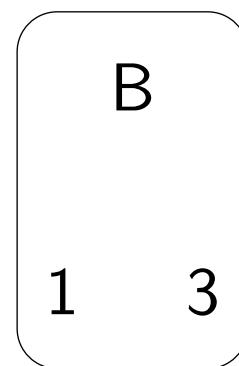
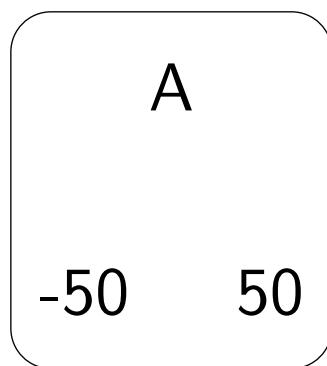
## Example: game 2

You choose one of the three bins.

Flip a coin; if heads, then move one bin to the left (with wrap around).

I choose a number from that bin.

Your goal is to maximize the chosen number.



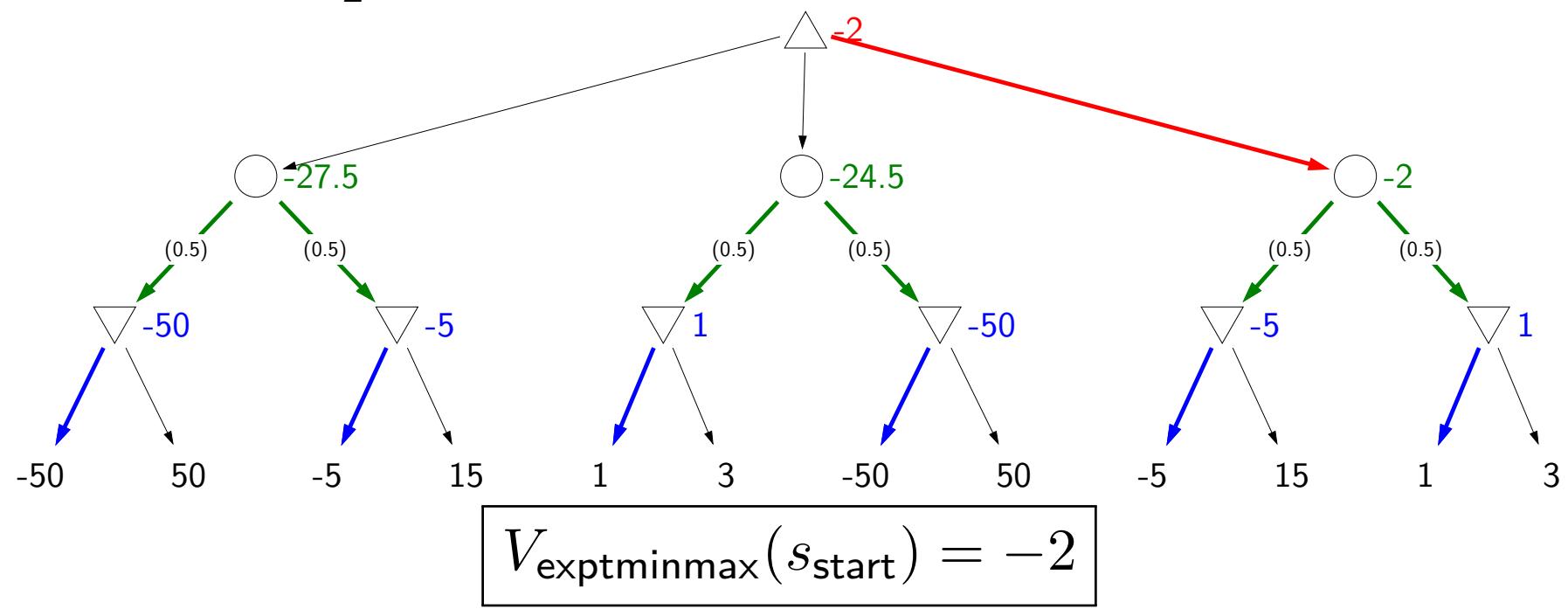
- Now let us consider games that have an element of chance that does not come from the agent or the opponent. Or in the simple modified game, the agent picks, a coin is flipped, and then the opponent picks.
- It turns out that handling games of chance is just a straightforward extension of the game framework that we have already.

# Expectiminimax example



## Example: expectiminimax

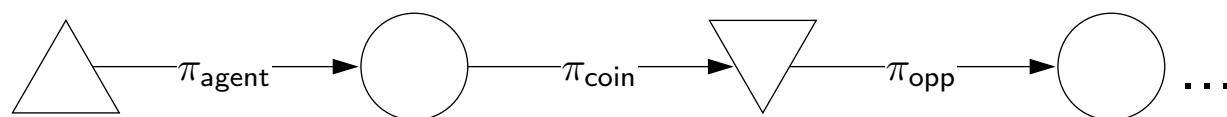
$$\pi_{\text{coin}}(s, a) = \frac{1}{2} \text{ for } a \in \{0, 1\}$$



- In the example, notice that the minimax optimal policy has shifted from the middle action to the rightmost action, which guards against the effects of the randomness. The agent really wants to avoid ending up on A, in which case the opponent could deliver a deadly  $-50$  utility.

# Expectiminimax recurrence

Players = {agent, opp, coin}



$$V_{\text{exptminmax}}(s) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \max_{a \in \text{Actions}(s)} V_{\text{exptminmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\text{exptminmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{opp} \\ \sum_{a \in \text{Actions}(s)} \pi_{\text{coin}}(s, a) V_{\text{exptminmax}}(\text{Succ}(s, a)) & \text{Player}(s) = \text{coin} \end{cases}$$

- The resulting game is modeled using **expectiminimax**, where we introduce a third player (called coin), which always follows a known stochastic policy. We are using the term *coin* as just a metaphor for any sort of natural randomness.
- To handle coin, we simply add a line into our recurrence that sums over actions when it's coin's turn.



# Summary so far

Primitives: **max** nodes, **chance** nodes, **min** nodes

Composition: alternate nodes according to model of game

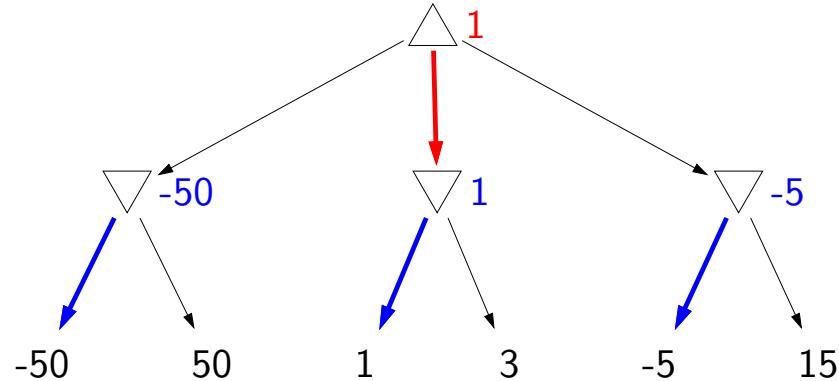
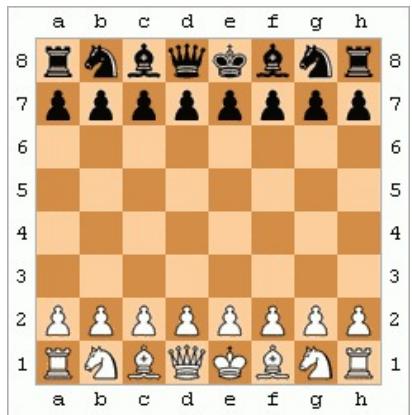
Value function  $V_{\dots}(s)$ : recurrence for expected utility

Scenarios to think about:

- What if you are playing against multiple opponents?
- What if you and your partner have to take turns (table tennis)?
- Some actions allow you to take an extra turn?

- In summary, so far, we've shown how to model a number of games using game trees, where each node of the game tree is either a max, chance, or min node depending on whose turn it is at that node and what we believe about that player's policy.
- Using these primitives, one can model more complex turn-taking games involving multiple players with heterogeneous strategies and where the turn-taking doesn't have to strictly alternate. The only restriction is that there are two parties: one that seeks to maximize utility and the other that seeks to minimize utility, along with other players who have known fixed policies (like coin).

# Computation



Approach: tree search

Complexity:

- branching factor  $b$ , depth  $d$  ( $2d$  plies)
- $O(d)$  space,  $O(b^{2d})$  time

Chess:  $b \approx 35$ ,  $d \approx 50$

25515520672986852924121150151425587630190414488161019324176778440771467258239937365843732987043555789782336195637736653285543297897675074636936187744140625

- Thus far, we've only touched on the modeling part of games. The rest of the lecture will be about how to actually compute (or approximately compute) the values of games.
- The first thing to note is that we cannot avoid exhaustive search of the game tree in general. Recall that a state is a summary of the past actions which is sufficient to act optimally in the future. In most games, the future depends on the exact position of all the pieces, so we cannot forget much and exploit dynamic programming.
- Second, game trees can be enormous. Chess has a branching factor of around 35 and go has a branching factor of up to 361 (the number of moves to a player on his/her turn). Games also can last a long time, and therefore have a depth of up to 100.
- A note about terminology specific to games: A game tree of depth  $d$  corresponds to a tree where each player has moved  $d$  times. Each level in the tree is called a **ply**. The number of plies is the depth times the number of players.

# Speeding up minimax

- Evaluation functions: use domain-specific knowledge, compute approximate answer
- Alpha-beta pruning: general-purpose, compute exact answer



- The rest of the lecture will be about how to speed up the basic minimax search using two ideas: evaluation functions and alpha-beta pruning.



# Roadmap

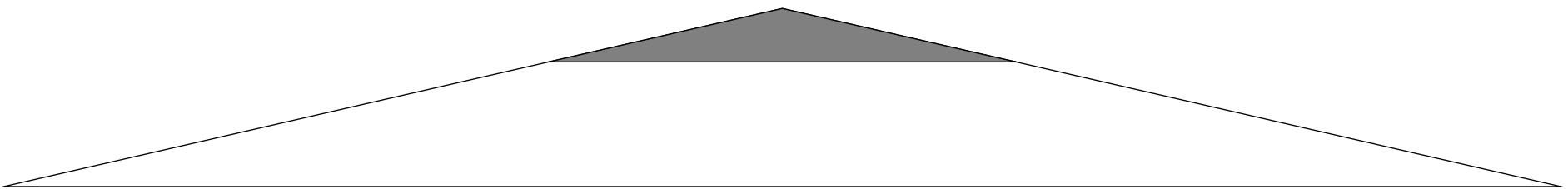
Games, expectimax

Minimax, expectiminimax

**Evaluation functions**

Alpha-beta pruning

# Depth-limited search



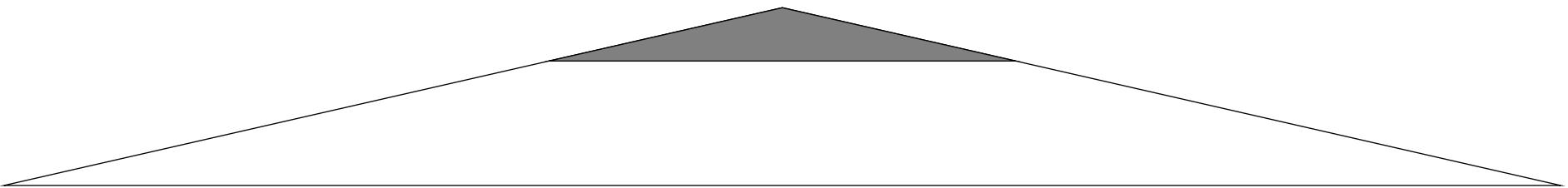
Limited depth tree search (stop at maximum depth  $d_{\max}$ ):

$$V_{\min\max}(s, \textcolor{red}{d}) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a), \textcolor{red}{d}) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a), \textcolor{red}{d - 1}) & \text{Player}(s) = \text{opp} \end{cases}$$

Use: at state  $s$ , call  $V_{\min\max}(s, d_{\max})$

Convention: decrement depth at last player's turn

# Evaluation functions



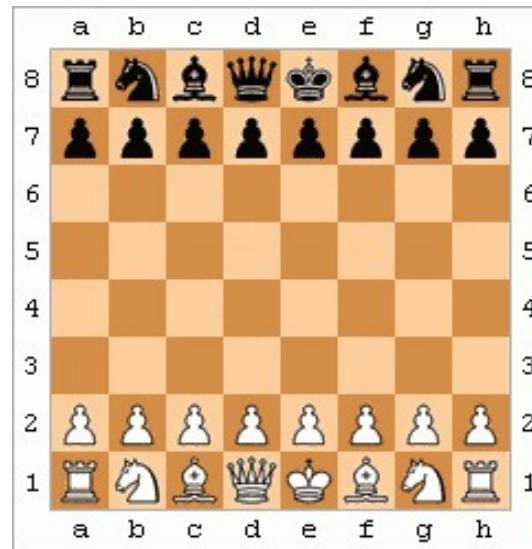
## Definition: Evaluation function

An evaluation function  $\text{Eval}(s)$  is a (possibly very weak) estimate of the value  $V_{\text{minmax}}(s)$ .

Analogy: FutureCost( $s$ ) in search problems

- The first idea on how to speed up minimax is to search only the tip of the game tree, that is down to depth  $d_{\max}$ , which is much smaller than the total depth of the tree  $D$  (for example,  $d_{\max}$  might be 4 and  $D = 50$ ).
- We modify our minimax recurrence from before by adding an argument  $d$ , which is the maximum depth that we are willing to descend from state  $s$ . If  $d = 0$ , then we don't do any more search, but fall back to an **evaluation function**  $\text{Eval}(s)$ , which is supposed to approximate the value of  $V_{\min\max}(s)$  (just like the heuristic  $h(s)$  approximated  $\text{FutureCost}(s)$  in A\* search).
- If  $d > 0$ , we recurse, decrementing the allowable depth by one at only min nodes, not the max nodes. This is because we are keeping track of the depth rather than the number of plies.

# Evaluation functions



## Example: chess

$\text{Eval}(s) = \text{material} + \text{mobility} + \text{king-safety} + \text{center-control}$

$$\begin{aligned}\text{material} = & 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + \\ & 3(B - B' + N - N') + 1(P - P')\end{aligned}$$

$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

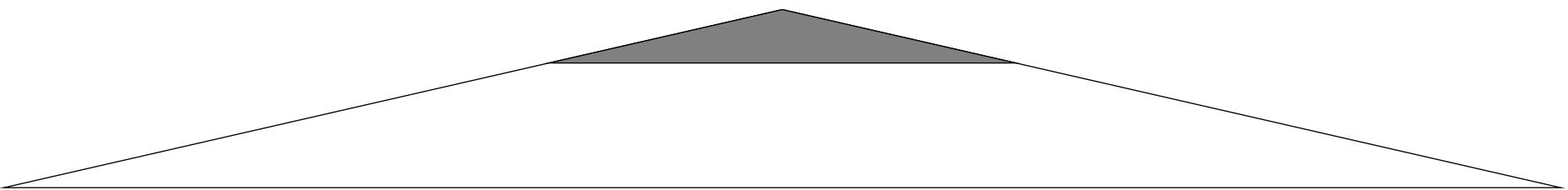
...

- Now what is this mysterious evaluation function  $\text{Eval}(s)$  that serves as a substitute for the horrendously hard  $V_{\min\max}$  that we can't compute?
- Just as in A\*, there is no free lunch, and we have to use domain knowledge about the game. Let's take chess for example. While we don't know who's going to win, there are some features of the game that are likely indicators. For example, having more pieces is good (material), being able to move them is good (mobility), keeping the king safe is good, and being able to control the center of the board is also good. We can then construct an evaluation function which is a weighted combination of the different properties.
- For example,  $K - K'$  is the difference in the number of kings that the agent has over the number that the opponent has (losing kings is really bad since you lose then),  $Q - Q'$  is the difference in queens,  $R - R'$  is the difference in rooks,  $B - B'$  is the difference in bishops,  $N - N'$  is the difference in knights, and  $P - P'$  is the difference in pawns.



# Summary: evaluation functions

Depth-limited exhaustive search:  $O(b^{2d})$  time



- $\text{Eval}(s)$  attempts to estimate  $V_{\text{minmax}}(s)$  using domain knowledge
- No guarantees (unlike A\*) on the error from approximation

- To summarize, this section has been about how to make naive exhaustive search over the game tree to compute the minimax value of a game faster.
- The methods so far have been focused on taking shortcuts: only searching up to depth  $d$  and relying on an **evaluation function**, and using a cheaper mechanism for estimating the value at a node rather than search its entire subtree.



# Roadmap

Games, expectimax

Minimax, expectiminimax

Evaluation functions

**Alpha-beta pruning**

# Pruning principle

Choose A or B with maximum value:

A: [3, **5**]

B: [**5**, 100]



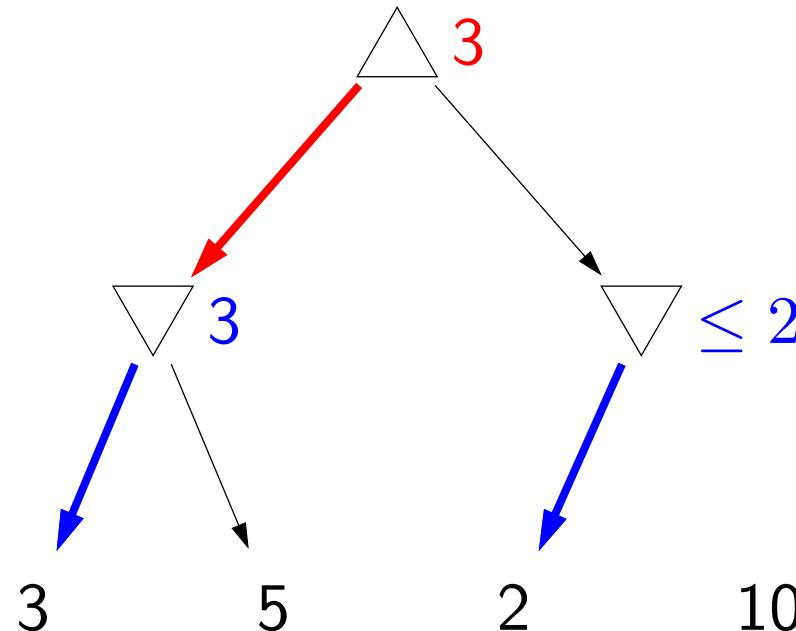
**Key idea: branch and bound**

Maintain lower and upper bounds on values.

If intervals don't overlap non-trivially, then can choose optimally without further work.

- We continue on our quest to make minimax run faster based on **pruning**. Unlike evaluation functions, these are general purpose and have theoretical guarantees.
- The core idea of pruning is based on the branch and bound principle. As we are searching (branching), we keep lower and upper bounds on each value we're trying to compute. If we ever get into a situation where we are choosing between two options A and B whose intervals don't overlap or just meet at a single point (in other words, they do not **overlap non-trivially**), then we can choose the interval containing larger values (B in the example). The significance of this observation is that we don't have to do extra work to figure out the precise value of A.

# Pruning game trees



Once see 2, we know that value of right node must be  $\leq 2$

Root computes  $\max(3, \leq 2) = 3$

Since branch doesn't affect root value, can safely prune

- In the context of minimax search, we note that the root node is a max over its children.
- Once we see the left child, we know that the root value must be at least 3.
- Once we get the 2 on the right, we know the right child has to be at most 2.
- Since those two intervals are non-overlapping, we can prune the rest of the right subtree and not explore it.

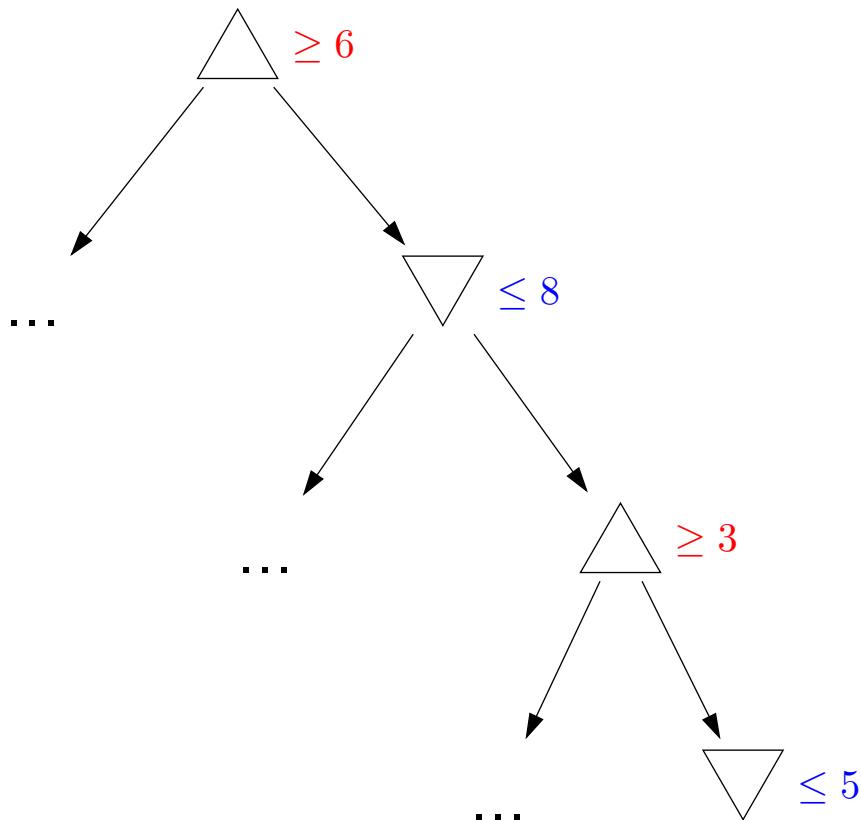
# Alpha-beta pruning



## Key idea: optimal path

The optimal path is path that minimax policies take.

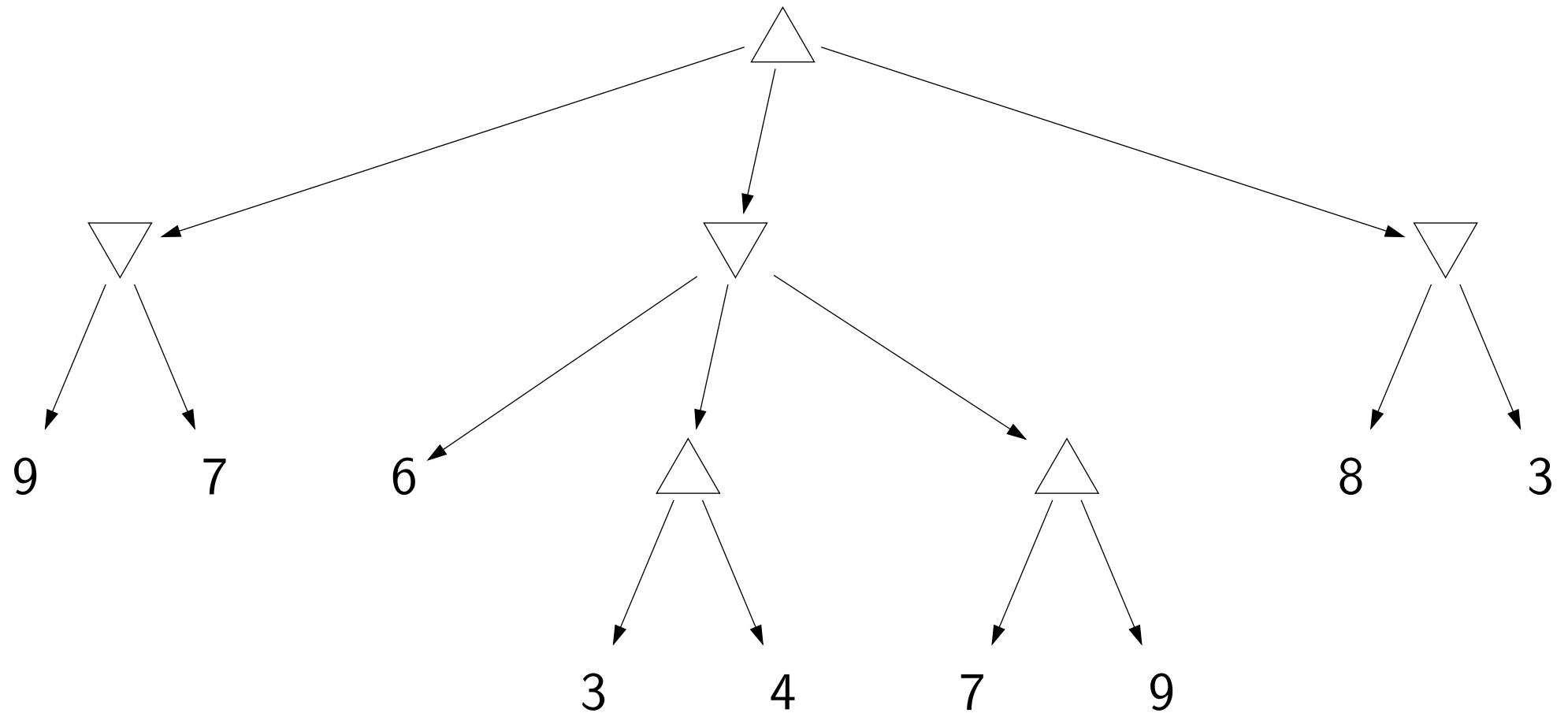
Values of all nodes on path are the same.



- $a_s$ : lower bound on value of max node  $s$
- $b_s$ : upper bound on value of min node  $s$
- Prune a node if its interval doesn't have non-trivial overlap with every ancestor (store  $\alpha_s = \max_{s' \preceq s} a_s$  and  $\beta_s = \min_{s' \preceq s} b_s$ )

- In general, let's think about the minimax values in the game tree. The value of a node is equal to the utility of at least one of its leaf nodes (because all the values are just propagated from the leaves with min and max applied to them). Call the first path (ordering by children left-to-right) that leads to the first such leaf node the **optimal path**. An important observation is that the values of all nodes on the optimal path are the same (equal to the minimax value of the root).
- Since we are interested in computing the value of the root node, if we can certify that a node is not on the optimal path, then we can prune it and its subtree.
- To do this, during the depth-first exhaustive search of the game tree, we think about maintaining a lower bound ( $\geq a_s$ ) for all the max nodes  $s$  and an upper bound ( $\leq b_s$ ) for all the min nodes  $s$ .
- If the interval of the current node does not non-trivially overlap the interval of every one of its ancestors, then we can prune the current node. In the example, we've determined the root's node must be  $\geq 6$ . Once we get to the node on at ply 4 and determine that node is  $\leq 5$ , we can prune the rest of its children since it is impossible that this node will be on the optimal path ( $\leq 5$  and  $\geq 6$  are incompatible). Remember that all the nodes on the optimal path have the same value.
- Implementation note: for each max node  $s$ , rather than keeping  $a_s$ , we keep  $\alpha_s$ , which is the maximum value of  $a_{s'}$  over  $s$  and all its max node ancestors. Similarly, for each min node  $s$ , rather than keeping  $b_s$ , we keep  $\beta_s$ , which is the minimum value of  $b_{s'}$  over  $s$  and all its min node ancestors. That way, at any given node, we can check interval overlap in constant time regardless of how deep we are in the tree.

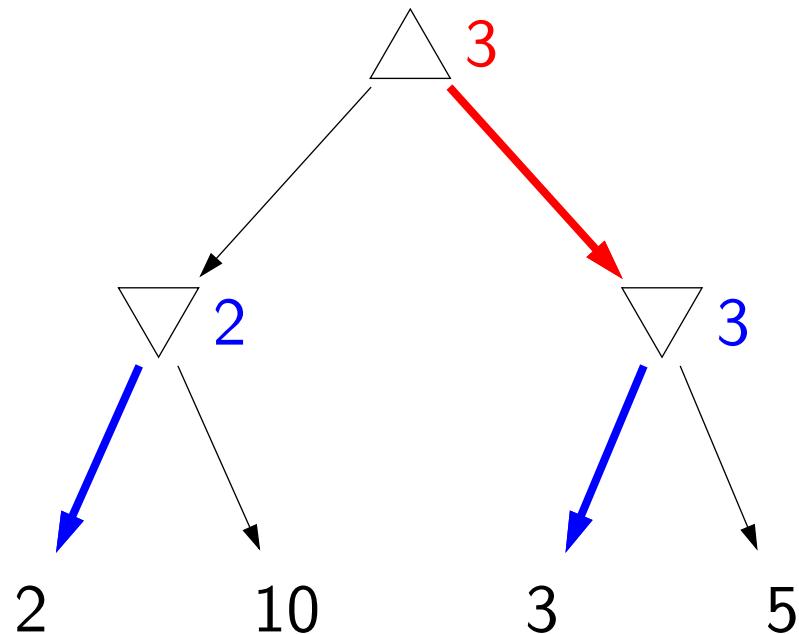
# Alpha-beta pruning example



# Move ordering

Pruning depends on order of actions.

Can't prune the 5 node:



- We have so far shown that alpha-beta pruning correctly computes the minimax value at the root, and seems to save some work by pruning subtrees. But how much of a savings do we get?
- The answer is that it depends on the order in which we explore the children. This simple example shows that with one ordering, we can prune the final leaf, but in the second, we can't.

# Move ordering

Which ordering to choose?

- Worst ordering:  $O(b^{2 \cdot d})$  time
- Best ordering:  $O(b^{2 \cdot 0.5d})$  time
- Random ordering:  $O(b^{2 \cdot 0.75d})$  time

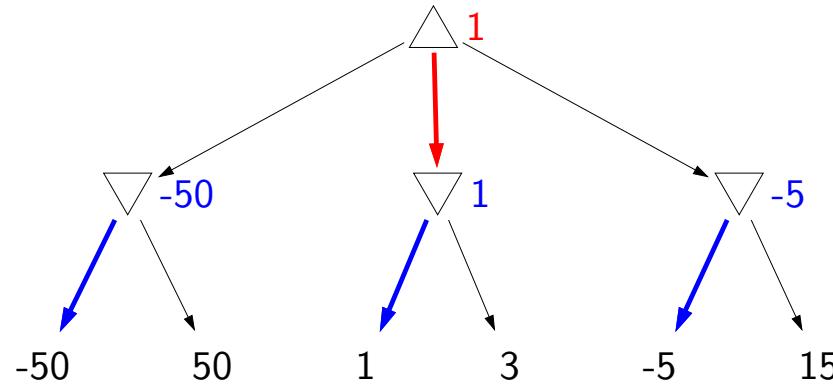
In practice, can use evaluation function  $\text{Eval}(s)$ :

- Max nodes: order successors by decreasing  $\text{Eval}(s)$
- Min nodes: order successors by increasing  $\text{Eval}(s)$

- In the worst case, we don't get any savings.
- If we use the best possible ordering, then we save half the exponent, which is *significant*. This means that if we could search to depth 10 before, we can now search to depth 20, which is truly remarkable given that the time increases exponentially with the depth.
- In practice, of course we don't know the best ordering. But interestingly, if we just use a random ordering, that allows us to search 33 percent deeper.
- We could also use a heuristic ordering based on a simple evaluation function. Intuitively, we want to search children that are going to give us the largest lower bound for max nodes and the smallest upper bound for min nodes.



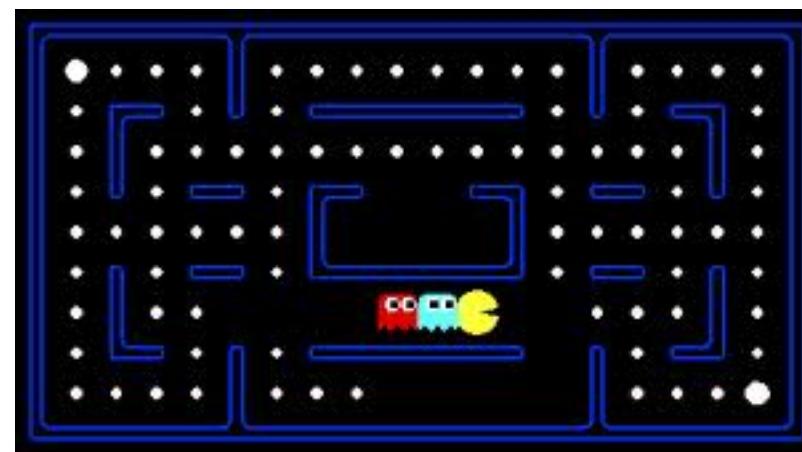
# Summary



- Game trees: model opponents, randomness
- Minimax: find optimal policy against an adversary
- Evaluation functions: domain-specific, approximate
- Alpha-beta pruning: domain-general, exact



# Lecture 10: Games II





# Question

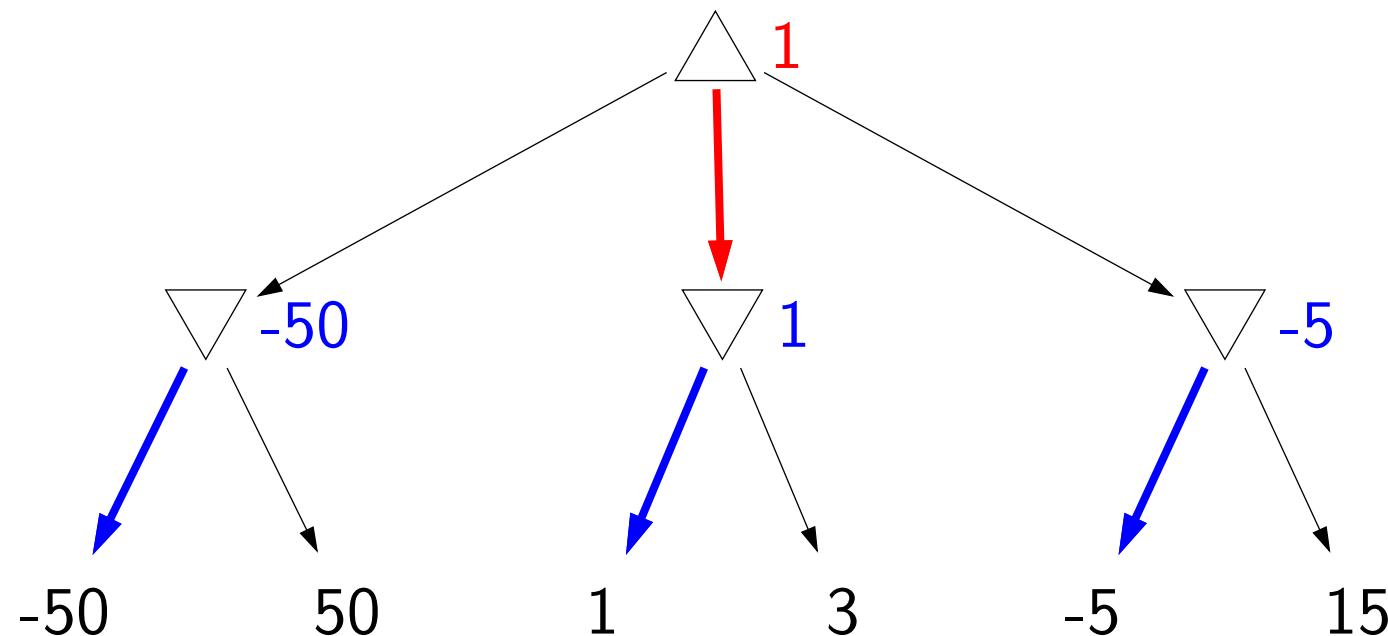
For a simultaneous two-player zero-sum game (like rock-paper-scissors), can you still be optimal if you reveal your strategy?

yes

no

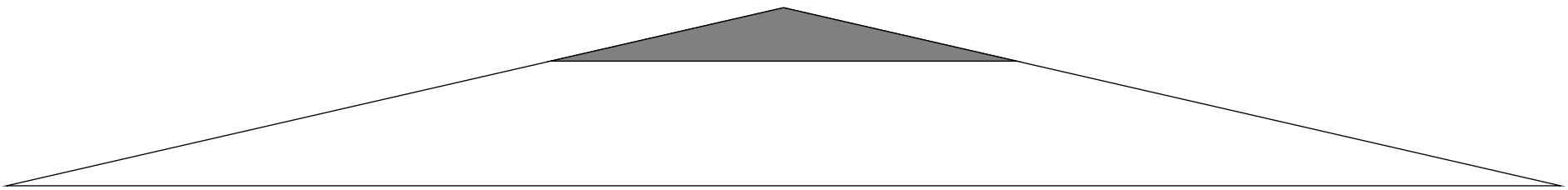
# Review: minimax

agent (max) versus opponent (min)



- Recall that the central object of study is the game tree. Game play starts at the root (starting state) and descends to a leaf (end state), where at each node  $s$  (state), the player whose turn it is ( $\text{Player}(s)$ ) chooses an action  $a \in \text{Actions}(s)$ , which leads to one of the children  $\text{Succ}(s, a)$ .
- The **minimax principle** provides one way for the agent (your computer program) to compute a pair of minimax policies for both the agent and the opponent ( $\pi_{\text{agent}}^*, \pi_{\text{opp}}^*$ ).
- For each node  $s$ , we have the minimax value of the game  $V_{\text{minmax}}(s)$ , representing the expected utility if both the agent and the opponent play optimally. Each node where it's the agent's turn is a max node (right-side up triangle), and its value is the maximum over the children's values. Each node where it's the opponent's turn is a min node (upside-down triangle), and its value is the minimum over the children's values.
- Important properties of the minimax policies: The agent can only decrease the game value (do worse) by changing his/her strategy, and the opponent can only increase the game value (do worse) by changing his/her strategy.

# Review: depth-limited search



$$V_{\min\max}(s, d) = \begin{cases} \text{Utility}(s) & \text{IsEnd}(s) \\ \text{Eval}(s) & d = 0 \\ \max_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a), d) & \text{Player}(s) = \text{agent} \\ \min_{a \in \text{Actions}(s)} V_{\min\max}(\text{Succ}(s, a), d - 1) & \text{Player}(s) = \text{opp} \end{cases}$$

Use: at state  $s$ , choose action resulting in  $V_{\min\max}(s, d_{\max})$

- In order to approximately compute the minimax value, we used a **depth-limited search**, where we compute  $V_{\text{minmax}}(s, d_{\text{max}})$ , the approximate value of  $s$  if we are only allowed to search to at most depth  $d_{\text{max}}$ .
- Each time we hit  $d = 0$ , we invoke an evaluation function  $\text{Eval}(s)$ , which provides a fast reflex way to assess the value of the game at state  $s$ .

# Evaluation function

Old: hand-crafted



## Example: chess

$\text{Eval}(s) = \text{material} + \text{mobility} + \text{king-safety} + \text{center-control}$

$$\begin{aligned}\text{material} &= 10^{100}(K - K') + 9(Q - Q') + 5(R - R') + \\ &\quad 3(B - B' + N - N') + 1(P - P')\end{aligned}$$

$$\text{mobility} = 0.1(\text{num-legal-moves} - \text{num-legal-moves}')$$

...

New: learn from data

$$\text{Eval}(s) = V(s; \mathbf{w})$$

- Having a good evaluation function is one of the most important components of game playing. So far we've shown how one can manually specify the evaluation function by hand. However, this can be quite tedious, and moreover, how does one figure out to weigh the different factors? In this lecture, we will consider a method for learning this evaluation function automatically from data.
- The three ingredients in any machine learning approach are to determine the (i) model family (in this case, what is  $V(s; \mathbf{w})$ ?), (ii) where the data comes from, and (iii) the actual learning algorithm. We will go through each of these in turn.



# Roadmap

**TD learning**

Simultaneous games

Non-zero-sum games

State-of-the-art

# Model for evaluation functions

Linear:

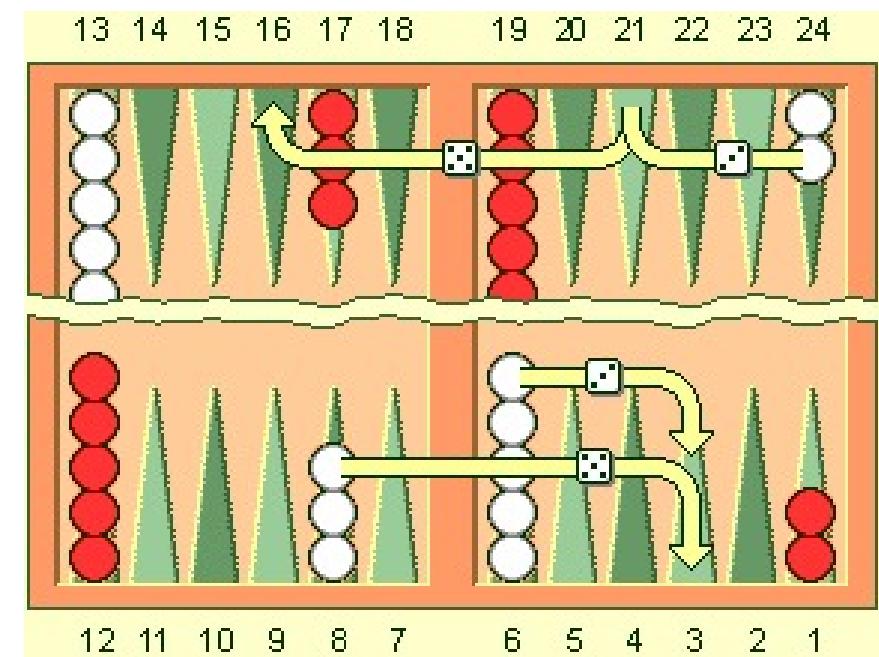
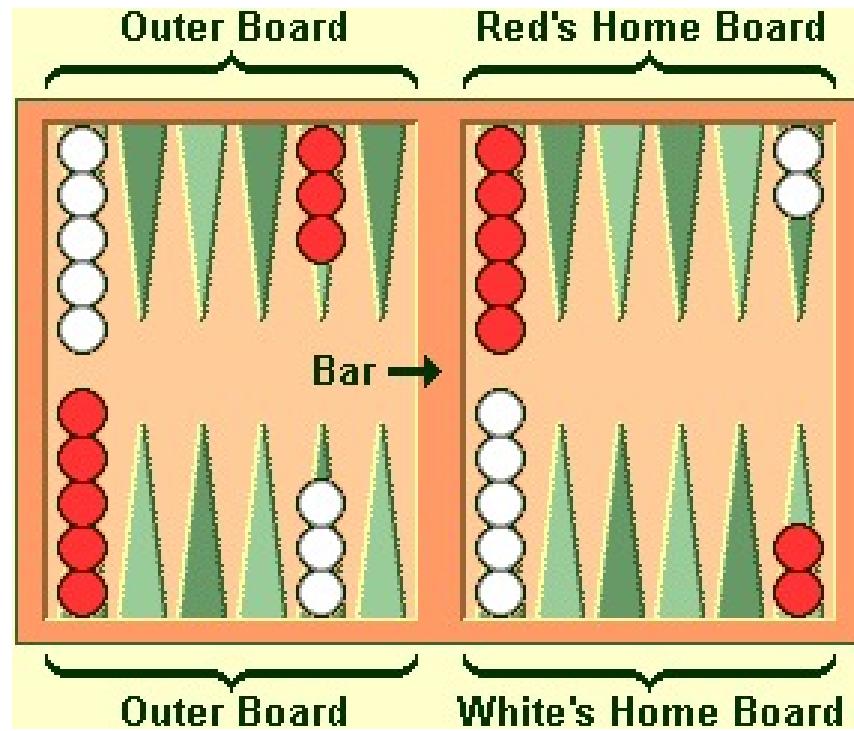
$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

Neural network:

$$V(s; \mathbf{w}, \mathbf{v}_{1:k}) = \sum_{j=1}^k w_j \sigma(\mathbf{v}_j \cdot \phi(s))$$

- When we looked at Q-learning, we considered linear evaluation functions (remember, linear in the weights  $w$ ). This is the simplest case, but it might not be suitable in some cases.
- But the evaluation function can really be any parametrized function. For example, the original TD-Gammon program used a neural network, which allows us to represent more expressive functions that capture the non-linear interactions between different features.
- Any model that you could use for regression in supervised learning you could also use here.

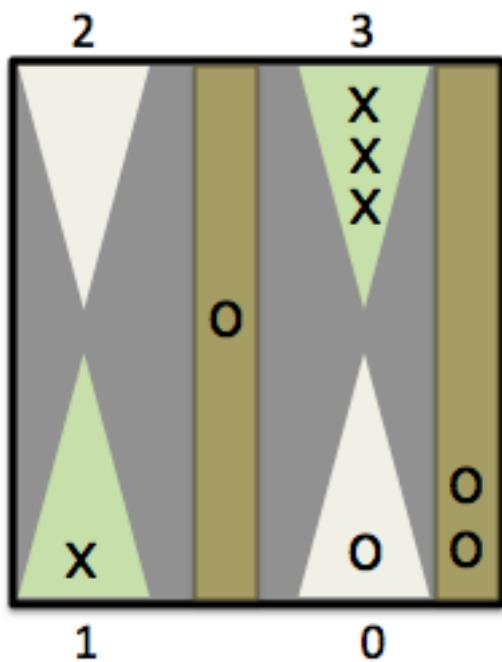
# Example: Backgammon



- As an example, let's consider the classic game of backgammon. Backgammon is a two-player game of strategy and chance in which the objective is to be the first to remove all your pieces from the board.
- The simplified version is that on your turn, you roll two dice, and choose two of your pieces to move forward that many positions.
- You cannot land on a position containing more than one opponent piece. If you land on exactly one opponent piece, then that piece goes on the bar and has start over from the beginning. (See the Wikipedia article for the full rules.).

# Features for Backgammon

state  $s$



Features  $\phi(s)$ :

$[(\# \text{ o in column 0}) = 1] : 1$

$[(\# \text{ o on bar})] : 1$

$[(\text{fraction o removed})] : \frac{1}{2}$

$[(\# \text{ x in column 1}) = 1] : 1$

$[(\# \text{ x in column 3}) = 3] : 1$

$[(\text{is it o's turn})] : 1$

- As an example, we can define the following features for Backgammon, which are inspired by the ones used by TD-Gammon.
- Note that the features are pretty generic; there is no explicit modeling of strategies such as trying to avoid having singleton pieces (because it could get clobbered) or preferences for how the pieces are distributed across the board.
- On the other hand, the features are mostly **indicator** features, which is a common trick to allow for more expressive functions using the machinery of linear regression. For example, instead of having one feature whose value is the number of pieces in a particular column, we can have multiple features for indicating whether the number of pieces is over some threshold.

# Generating data

Generate using policies based on current  $V(s; \mathbf{w})$ :

$$\pi_{\text{agent}}(s; \mathbf{w}) = \arg \max_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$$

$$\pi_{\text{opp}}(s; \mathbf{w}) = \arg \min_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$$

Note: don't need to randomize ( $\epsilon$ -greedy) because game is already stochastic (backgammon has dice) and there's function approximation

Generate episode:

$$s_0; a_1, r_1, s_1; a_2, r_2, s_2; a_3, r_3, s_3; \dots; a_n, r_n, s_n$$

- The second ingredient of doing learning is generating the data. As in reinforcement learning, we will generate a sequence of states, actions, and rewards by simulation — that is, by playing the game.
- In order to play the game, we need two exploration policies: one for the agent, one for the opponent. The policy of the dice is fixed to be uniform over  $\{1, \dots, 6\}$  as expected.
- A natural policy to use is one that uses our current estimate of the value  $V(s; \mathbf{w})$ . Specifically, the agent's policy will consider all possible actions from a state, use the value function to evaluate how good each of the successor states are, and then choose the action leading to the highest value. Generically, we would include  $\text{Reward}(s, a, \text{Succ}(s, a))$ , but in games, all the reward is at the end, so  $r_t = 0$  for  $t < n$  and  $r_n = \text{Utility}(s_n)$ . Symmetrically, the opponent's policy will choose the action that leads to the lowest possible value.
- Given this choice of  $\pi_{\text{agent}}$  and  $\pi_{\text{opp}}$ , we generate the actions  $a_t = \pi_{\text{Player}(s_{t-1})}(s_{t-1})$ , successors  $s_t = \text{Succ}(s_{t-1}, a_t)$ , and rewards  $r_t = \text{Reward}(s_{t-1}, a_t, s_t)$ .
- In reinforcement learning, we saw that using an exploration policy based on just the current value function is a bad idea, because we can get stuck exploiting local optima and not exploring. In the specific case of Backgammon, using deterministic exploration policies for the agent and opponent turns out to be fine, because the randomness from the dice naturally provides exploration.

# Learning algorithm

Episode:

$s_0; a_1, r_1, s_1; a_2, r_2, s_2, a_3, r_3, s_3; \dots, a_n, r_n, s_n$

A small piece of experience:

$(s, a, r, s')$

Prediction:

$V(s; \mathbf{w})$

Target:

$r + \gamma V(s'; \mathbf{w})$

- With a model family  $V(s; \mathbf{w})$  and data  $s_0, a_1, r_1, s_1, \dots$  in hand, let's turn to the learning algorithm.
- A general principle in learning is to figure out the **prediction** and the **target**. The prediction is just the value of the current function at the current state  $s$ , and the target uses the data by looking at the immediate reward  $r$  plus the value of the function applied to the successor state  $s'$  (discounted by  $\gamma$ ). This is analogous to the SARSA update for Q-values, where our target actually depends on a one-step lookahead prediction.

# General framework

Objective function:

$$\frac{1}{2}(\text{prediction}(\mathbf{w}) - \text{target})^2$$

Gradient:

$$(\text{prediction}(\mathbf{w}) - \text{target})\nabla_{\mathbf{w}}\text{prediction}(\mathbf{w})$$

Update:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \underbrace{(\text{prediction}(\mathbf{w}) - \text{target})\nabla_{\mathbf{w}}\text{prediction}(\mathbf{w})}_{\text{gradient}}$$

- Having identified a prediction and target, the next step is to figure out how to update the weights. The general strategy is to set up an objective function that encourages the prediction and target to be close (by penalizing their squared distance).
- Then we just take the gradient with respect to the weights  $w$ .
- Note that even though technically the target also depends on the weights  $w$ , we treat this as constant for this derivation. The resulting learning algorithm by no means finds the global minimum of this objective function. We are simply using the objective function to motivate the update rule.

# Temporal difference (TD) learning



## Algorithm: TD learning

On each  $(s, a, r, s')$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \underbrace{V(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma V(s'; \mathbf{w}))}_{\text{target}} \nabla_{\mathbf{w}} V(s; \mathbf{w})$$

For linear functions:

$$V(s; \mathbf{w}) = \mathbf{w} \cdot \phi(s)$$

$$\nabla_{\mathbf{w}} V(s; \mathbf{w}) = \phi(s)$$

- Plugging in the prediction and the target in our setting yields the TD learning algorithm. For linear functions, recall that the gradient is just the feature vector.

# Example of TD learning

Step size  $\eta = 0.5$ , discount  $\gamma = 1$ , reward is end utility

Example: TD learning						
<b>S1</b>	<b>r:0</b>	<b>S4</b>	<b>r:0</b>	<b>S8</b>	<b>r:1</b>	<b>S9</b>
$\phi: \binom{0}{1}$		$\phi: \binom{1}{0}$		$\phi: \binom{1}{2}$		$\phi: \binom{1}{0}$
$w: \binom{0}{0}$	p:0	$w: \binom{0}{0}$	p:0	$w: \binom{0}{0}$	p:0	$w: \binom{0.5}{1}$
t:0		t:0			t:1	
p-t:0		p-t:0			p-t:-1	
<b>S1</b>	<b>r:0</b>	<b>S2</b>	<b>r:0</b>	<b>S6</b>	<b>r:0</b>	<b>S10</b>
$\phi: \binom{0}{1}$		$\phi: \binom{1}{0}$		$\phi: \binom{0}{0}$		$\phi: \binom{1}{0}$
$w: \binom{0.5}{1}$	p:1	$w: \binom{0.5}{0.75}$	p:0.5	$w: \binom{0.25}{0.75}$	p:0	$w: \binom{0.25}{0.75}$
t:0.5		t:0			t:0.25	
p-t:0.5		p-t:0.5			p-t:-0.25	

- Here's an example of TD learning in action. We have two episodes: [S1, 0, S4, 0, S8, 1, S9] and [S1, 0, S2, 0, S6, 0, S10].
- In games, all the reward comes at the end and the discount is 1. We have omitted the action because TD learning doesn't depend on the action.
- Under each state, we have written its feature vector, and the weight vector before updating on that state. Note that no updates are made until the first non-zero reward. Our prediction is 0, and the target is  $1 + 0$ , so we subtract  $-0.5[1, 2]$  from the weights to get  $[0.5, 1]$ .
- In the second row, we have our second episode, and now notice that even though all the rewards are zero, we are still making updates to the weight vectors since the prediction and targets computed based on adjacent states are different.

# Comparison



## Algorithm: TD learning

On each  $(s, a, r, s')$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta [\underbrace{\hat{V}_\pi(s; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \hat{V}_\pi(s'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} \hat{V}_\pi(s; \mathbf{w})$$



## Algorithm: Q-learning

On each  $(s, a, r, s')$ :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta [\underbrace{\hat{Q}_{\text{opt}}(s, a; \mathbf{w})}_{\text{prediction}} - \underbrace{(r + \gamma \max_{a' \in \text{Actions}(s)} \hat{Q}_{\text{opt}}(s', a'; \mathbf{w}))}_{\text{target}}] \nabla_{\mathbf{w}} \hat{Q}_{\text{opt}}(s, a; \mathbf{w})$$

# Comparison

## Q-learning:

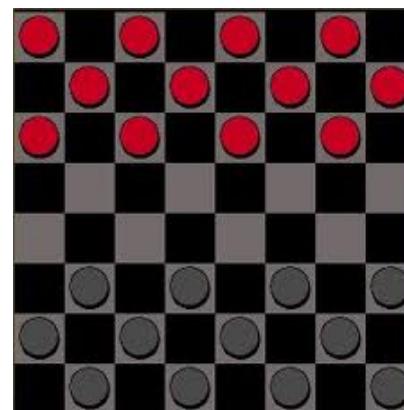
- Operate on  $\hat{Q}_{\text{opt}}(s, a; \mathbf{w})$
- Off-policy: value is based on estimate of optimal policy
- To use, don't need to know MDP transitions  $T(s, a, s')$

## TD learning:

- Operate on  $\hat{V}_\pi(s; \mathbf{w})$
- On-policy: value is based on exploration policy (usually based on  $\hat{V}_\pi$ )
- To use, need to know rules of the game  $\text{Succ}(s, a)$

- TD learning is very similar to Q-learning. Both algorithms learn from the same data and are based on gradient-based weight updates.
- The main difference is that Q-learning learns the Q-value, which measures how good an action is to take in a state, whereas TD learning learns the value function, which measures how good it is to be in a state.
- Q-learning is an off-policy algorithm, which means that it tries to compute  $Q_{\text{opt}}$ , associated with the optimal policy (not  $Q_\pi$ ), whereas TD learning is on-policy, which means that it tries to compute  $V_\pi$ , the value associated with a fixed policy  $\pi$ . Note that the action  $a$  does not show up in the TD updates because  $a$  is given by the fixed policy  $\pi$ . Of course, we usually are trying to optimize the policy, so we would set  $\pi$  to be the current guess of optimal policy  $\pi(s) = \arg \max_{a \in \text{Actions}(s)} V(\text{Succ}(s, a); \mathbf{w})$ .
- When we don't know the transition probabilities and in particular the successors, the value function isn't enough, because we don't know what effect our actions will have. However, in the game playing setting, we do know the transitions (the rules of the game), so using the value function is sufficient.

# Learning to play checkers

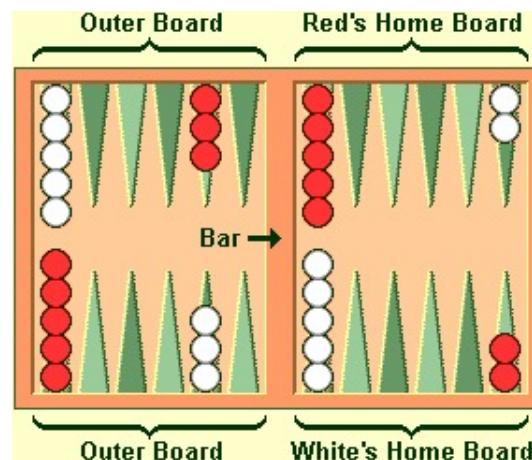


Arthur Samuel's checkers program [1959]:

- Learned by playing itself repeatedly (self-play)
- Smart features, linear evaluation function, use intermediate rewards
- Used alpha-beta pruning + search heuristics
- Reach human amateur level of play
- IBM 701: 9K of memory!

- The idea of using machine learning for game playing goes as far back as Arthur Samuel's checkers program. Many of the ideas (using features, alpha-beta pruning) were employed, resulting in a program that reached a human amateur level of play. Not bad for 1959!

# Learning to play Backgammon

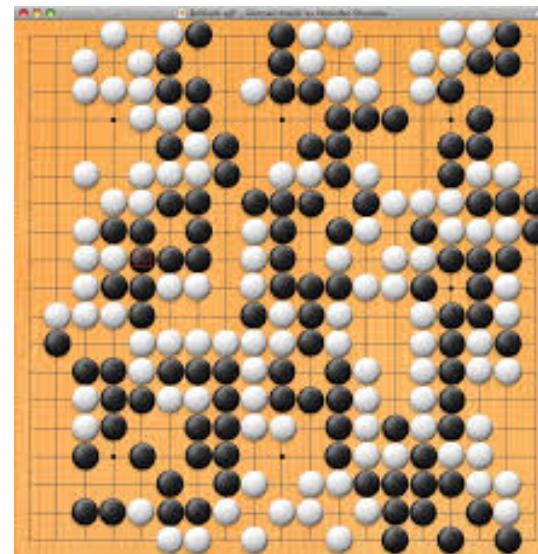


Gerald Tesauro's TD-Gammon [1992]:

- Learned weights by playing itself repeatedly (1 million times)
- Dumb features, neural network, no intermediate rewards
- Reached human expert level of play, provided new insights into opening

- Tesauro refined some of the ideas from Samuel with his famous TD-Gammon program provided the next advance, using a variant of TD learning called  $\text{TD}(\lambda)$ . It had dumber features, but a more expressive evaluation function (neural network), and was able to reach an expert level of play.

# Learning to play Go



AlphaGo Zero [2017]:

- Learned by self play (4.9 million games)
- Dumb features (stone positions), neural network, no intermediate rewards, Monte Carlo Tree Search
- Beat AlphaGo, which beat Le Sedol in 2016
- Provided new insights into the game

- Very recently, self-play reinforcement learning has been applied to the game of Go. AlphaGo Zero uses a single neural network to predict winning probability and actions to be taken, using raw board positions as inputs. Starting from random weights, the network is trained to gradually improve its predictions and match the results of an approximate (Monte Carlo) tree search algorithm.



# Summary so far

- Parametrize evaluation functions using features
- TD learning: learn an evaluation function

$$(\text{prediction}(\mathbf{w}) - \text{target})^2$$

Up next:

Turn-based



Simultaneous

Zero-sum



Non-zero-sum



# Roadmap

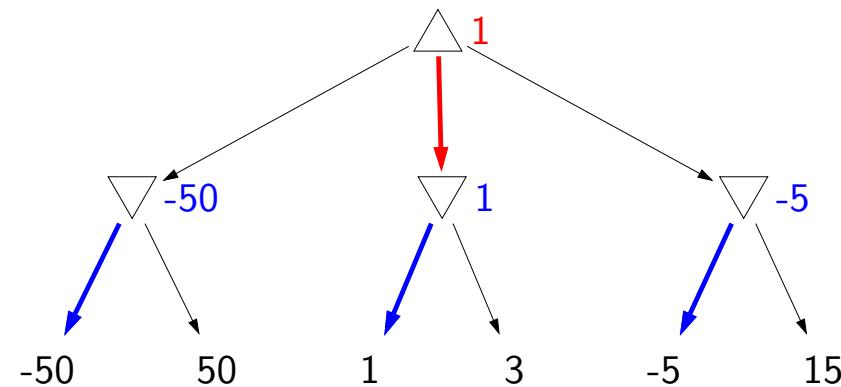
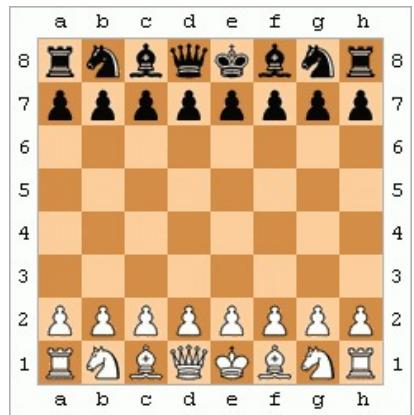
TD learning

**Simultaneous games**

Non-zero-sum games

State-of-the-art

## Turn-based games:



## Simultaneous games:



?

- Game trees were our primary tool to model turn-based games. However, in simultaneous games, there is no ordering on the player's moves, so we need to develop new tools to model these games. Later, we will see that game trees will still be valuable in understanding simultaneous games.



# Two-finger Morra



## Example: two-finger Morra

Players A and B each show 1 or 2 fingers.

If both show 1, B gives A 2 dollars.

If both show 2, B gives A 4 dollars.

Otherwise, A gives B 3 dollars.

[play with a partner]



# Question

What was the outcome?

player A chose 1, player B chose 1

player A chose 1, player B chose 2

player A chose 2, player B chose 1

player A chose 2, player B chose 2



# Payoff matrix



**Definition: single-move simultaneous game**

Players = {A, B}

Actions: possible actions

$V(a, b)$ : **A's utility** if A chooses action  $a$ , B chooses  $b$   
(let  $V$  be **payoff matrix**)



**Example: two-finger Morra payoff matrix**

A \ B	1 finger	2 fingers
1 finger	2	-3
2 fingers	-3	4

- In this lecture, we will consider only single move games. There are two players, A and B who both select from one of the available actions. The value or utility of the game is captured by a payoff matrix  $V$  whose dimensionality is  $|\text{Actions}| \times |\text{Actions}|$ . We will be analyzing everything from A's perspective, so entry  $V(a, b)$  is the utility that A gets if he/she chooses action  $a$  and player B chooses  $b$ .

# Strategies (policies)



## Definition: pure strategy

A pure strategy is a single action:

$$a \in \text{Actions}$$



## Definition: mixed strategy

A mixed strategy is a probability distribution

$$0 \leq \pi(a) \leq 1 \text{ for } a \in \text{Actions}$$



## Example: two-finger Morra strategies

Always 1:  $\pi = [1, 0]$

Always 2:  $\pi = [0, 1]$

Uniformly random:  $\pi = [\frac{1}{2}, \frac{1}{2}]$

- Each player has a **strategy** (or a policy). A pure strategy (deterministic policy) is just a single action. Note that there's no notion of state since we are only considering single-move games.
- More generally, we will consider **mixed strategies** (randomized policy), which is a probability distribution over actions. We will represent a mixed strategy  $\pi$  by the vector of probabilities.

# Game evaluation



## Definition: game evaluation

The **value** of the game if player A follows  $\pi_A$  and player B follows  $\pi_B$  is

$$V(\pi_A, \pi_B) = \sum_{a,b} \pi_A(a)\pi_B(b)V(a, b)$$



## Example: two-finger Morra

Player A always chooses 1:  $\pi_A = [1, 0]$

Player B picks randomly:  $\pi_B = [\frac{1}{2}, \frac{1}{2}]$

Value:

$$-\frac{1}{2}$$

[whiteboard: matrix]

- Given a game (payoff matrix) and the strategies for the two players, we can define the value of the game.
- For pure strategies, the value of the game by definition is just reading out the appropriate entry from the payoff matrix.
- For mixed strategies, the value of the game (that is, the expected utility for player  $A$ ) is gotten by summing over the possible actions that the players choose:  $V(\pi_A, \pi_B) = \sum_{a \in \text{Actions}} \sum_{b \in \text{Actions}} \pi_A(a)\pi_B(b)V(a, b)$ . We can also write this expression concisely using matrix-vector multiplications:  $\pi_A^\top V \pi_B$ .

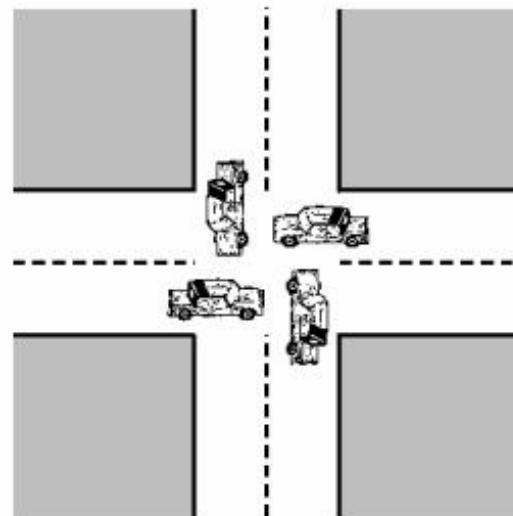
# How to optimize?

Game value:

$$V(\pi_A, \pi_B)$$

Challenge: player A wants to maximize, player B wants to minimize...

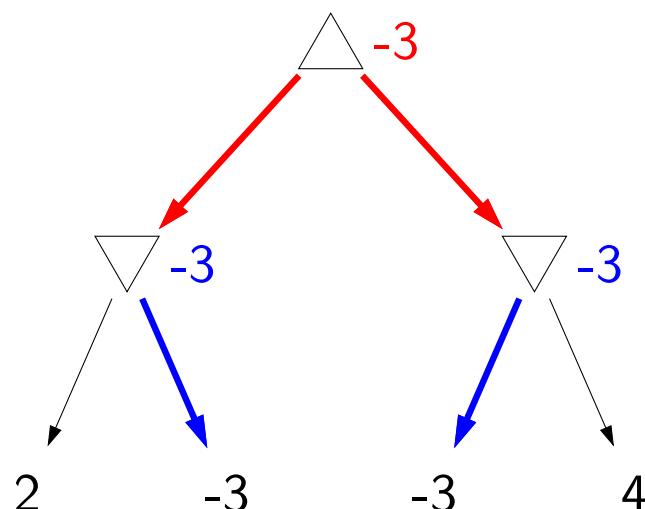
**simultaneously**



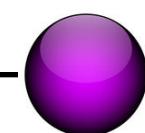
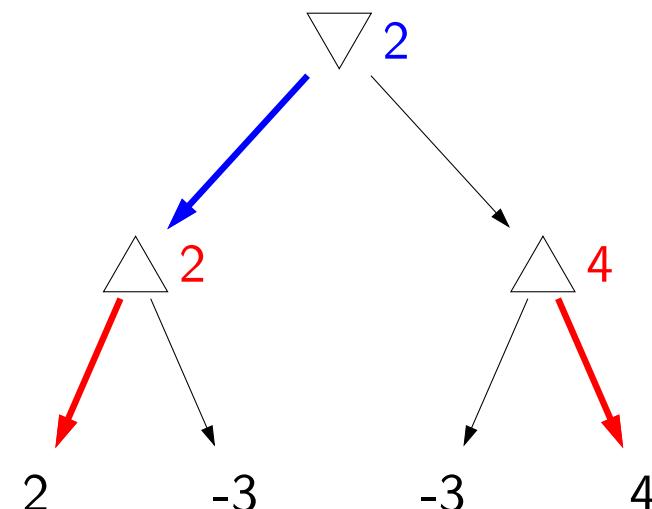
- Having established the values of fixed policies, let's try to optimize the policies themselves. Here, we run into a predicament: player A wants to maximize  $V$  but player B wants to minimize  $V$  **simultaneously**.
- Unlike turn-based games, we can't just consider one at a time. But let's consider the turn-based variant anyway to see where it leads us.

# Pure strategies: who goes first?

Player A goes first:



Player B goes first:



**Proposition: going second is no worse**

$$\max_a \min_b V(a, b) \leq \min_b \max_a V(a, b)$$

- Let us first consider pure strategies, where each player just chooses one action. The game can be modeled by using the standard minimax game trees that we're used to.
- The main point is that if player A goes first, he gets  $-3$ , but if he goes second, he gets  $2$ . In general, it's at least as good to go second, and often it is strictly better. This is intuitive, because seeing what the first player does gives more information.

# Mixed strategies



## Example: two-finger Morra

Player A reveals:  $\pi_A = [\frac{1}{2}, \frac{1}{2}]$

Value  $V(\pi_A, \pi_B) = \pi_B(1)(-\frac{1}{2}) + \pi_B(2)(+\frac{1}{2})$

Optimal strategy for player B is  $\pi_B = [1, 0]$  (**pure!**)



## Proposition: second player can play pure strategy

For any fixed mixed strategy  $\pi_A$ :

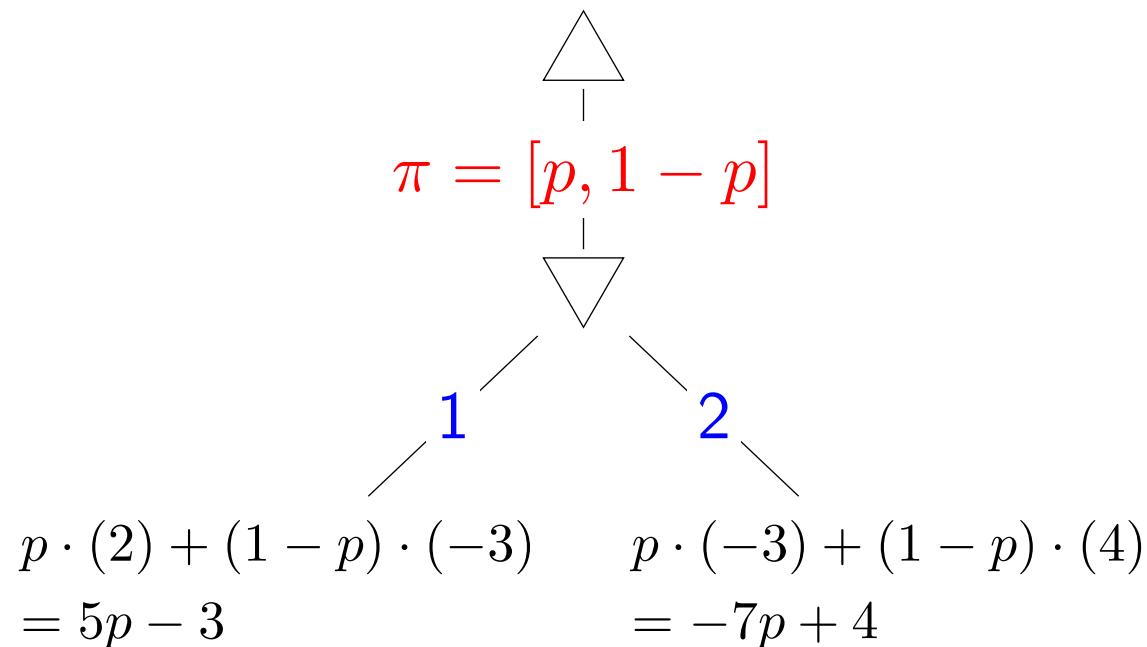
$$\min_{\pi_B} V(\pi_A, \pi_B)$$

can be attained by a pure strategy.

- Now let us consider mixed strategies. First, let's be clear on what playing a mixed strategy means. If player A chooses a mixed strategy, he reveals to player B the full probability distribution over actions, but importantly not a particular action (because that would be the same as choosing a pure strategy).
- As a warmup, suppose that player A reveals  $\pi_A = [\frac{1}{2}, \frac{1}{2}]$ . If we plug this strategy into the definition for the value of the game, we will find that the value is a convex combination between  $\frac{1}{2}(2) + \frac{1}{2}(-3) = -\frac{1}{2}$  and  $\frac{1}{2}(-3) + \frac{1}{2}(4) = \frac{1}{2}$ . The value of  $\pi_B$  that minimizes this value is  $[1, 0]$ . The important part is that this is a **pure strategy**.
- It turns out that no matter what the payoff matrix  $V$  is, as soon as  $\pi_A$  is fixed, then the optimal choice for  $\pi_B$  is a pure strategy. This is useful because it will allow us to analyze games with mixed strategies more easily.

# Mixed strategies

Player A first reveals his/her mixed strategy



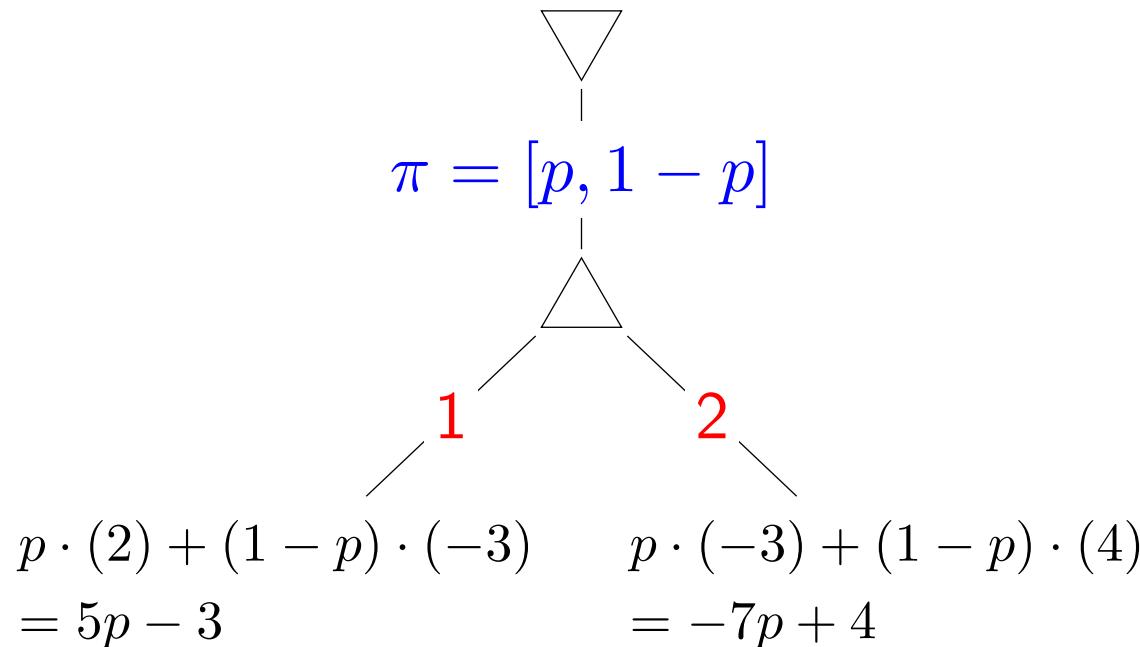
Minimax value of game:

$$\max_{0 \leq p \leq 1} \min\{5p - 3, -7p + 4\} = \boxed{-\frac{1}{12}} \text{ (with } p = \frac{7}{12} \text{)}$$

- Now let us try to draw the minimax game tree where the player A first chooses a mixed strategy, and then player B chooses a pure strategy.
- There are an uncountably infinite number of mixed strategies for player A, but we can summarize all of these actions by writing a single action template  $\pi = [p, 1 - p]$ .
- Given player A's action, we can compute the value if player B either chooses 1 or 2. For example, if player B chooses 1, then the value of the game is  $5p - 3$  (with probability  $p$ , player A chooses 1 and the value is 2; with probability  $1 - p$  the value is  $-3$ ). If player B chooses action 2, then the value of the game is  $-7p + 4$ .
- The value of the min node is  $F(p) = \min\{5p - 3, -7p + 4\}$ . The value of the max node (and thus the minimax value of the game) is  $\max_{0 \leq 1 \leq p} F(p)$ .
- What is the best strategy for player A then? We just have to find the  $p$  that maximizes  $F(p)$ , which is the minimum over two linear functions of  $p$ . If we plot this function, we will see that the maximum of  $F(p)$  is attained when  $5p - 3 = -7p + 4$ , which is when  $p = \frac{7}{12}$ . Plugging that value of  $p$  back in yields  $F(p) = -\frac{1}{12}$ , the minimax value of the game if player A goes first and is allowed to choose a mixed strategy.
- Note that if player A decides on  $p = \frac{7}{12}$ , it doesn't matter whether player B chooses 1 or 2; the payoff will be the same:  $-\frac{1}{12}$ . This also means that whatever mixed strategy (over 1 and 2) player B plays, the payoff would also be  $-\frac{1}{12}$ .

# Mixed strategies

Player B first reveals his/her mixed strategy



Minimax value of game:

$$\min_{p \in [0,1]} \max \{5p - 3, -7p + 4\} = \boxed{-\frac{1}{12}} \text{ (with } p = \frac{7}{12})$$

- Now let us consider the case where player B chooses a mixed strategy  $\pi = [p, 1 - p]$  first. If we perform the analogous calculations, we'll find that we get that the minimax value of the game is exactly the same ( $-\frac{1}{12}$ )!
- Recall that for pure strategies, there was a gap between going first and going second, but here, we see that for mixed strategies, there is no such gap, at least in this example.
- Here, we have been computed minimax values in the conceptually same manner as we were doing it for turn-based games. The only difference is that our actions are mixed strategies (represented by a probability distribution) rather than discrete choices. We therefore introduce a variable (e.g.,  $p$ ) to represent the actual distribution, and any game value that we compute below that variable is a function of  $p$  rather than a specific number.

# General theorem



## Theorem: minimax theorem [von Neumann, 1928]

For every simultaneous two-player zero-sum game with a finite number of actions:

$$\max_{\pi_A} \min_{\pi_B} V(\pi_A, \pi_B) = \min_{\pi_B} \max_{\pi_A} V(\pi_A, \pi_B),$$

where  $\pi_A, \pi_B$  range over **mixed strategies**.

Upshot: revealing your optimal mixed strategy doesn't hurt you!

Proof: linear programming duality

Algorithm: compute policies using linear programming

- It turns out that having no gap is not a coincidence, and is actually one of the most celebrated mathematical results: the von Neumann minimax theorem. The theorem states that for any simultaneous two-player zero-sum game with a finite set of actions (like the ones we've been considering), we can just swap the min and the max: it doesn't matter which player reveals his/her strategy first, as long as their strategy is optimal. This is significant because we were stressing out about how to analyze the game when two players play simultaneously, but now we find that both orderings of the players yield the same answer. It is important to remember that this statement is true only for mixed strategies, not for pure strategies.
- This theorem can be proved using linear programming duality, and policies can be computed also using linear programming. The sketch of the idea is as follows: recall that the optimal strategy for the second player is always deterministic, which means that the  $\max_{\pi_A} \min_{\pi_B} \dots$  turns into  $\max_{\pi_A} \min_b \dots$ . The min is now over  $n$  actions, and can be rewritten as  $n$  linear constraints, yielding a linear program.
- As an aside, recall that we also had a minimax result for turn-based games, where the max and the min were over agent and opponent policies, which map states to actions. In that case, optimal policies were always deterministic because at each state, there is only one player choosing.



# Summary

- **Challenge:** deal with simultaneous min/max moves
- **Pure strategies:** going second is better
- **Mixed strategies:** doesn't matter (von Neumann's minimax theorem)



# Roadmap

TD learning

Simultaneous games

**Non-zero-sum games**

State-of-the-art

# Utility functions

Competitive games: minimax (linear programming)



Collaborative games: pure maximization (plain search)



Real life: ?

- So far, we have focused on competitive games, where the utility of one player is the exact opposite of the utility of the other. The minimax principle is the appropriate tool for modeling these scenarios.
- On the other extreme, we have collaborative games, where the two players have the same utility function. This case is less interesting, because we are just doing pure maximization (e.g., finding the largest element in the payoff matrix or performing search).
- In many practical real life scenarios, games are somewhere in between pure competition and pure collaboration. This is where things get interesting...

# Prisoner's dilemma



## Example: Prisoner's dilemma

Prosecutor asks A and B individually if each will testify against the other.

If both testify, then both are sentenced to 5 years in jail.

If both refuse, then both are sentenced to 1 year in jail.

If only one testifies, then he/she gets out for free; the other gets a 10-year sentence.

[play with a partner]



# Question

What was the outcome?

player A testified, player B testified

player A refused, player B testified

player A testified, player B refused

player A refused, player B refused

# Prisoner's dilemma



## Example: payoff matrix

B \ A	testify	refuse
testify	$A = -5, B = -5$	$A = -10, B = 0$
refuse	$A = 0, B = -10$	$A = -1, B = -1$



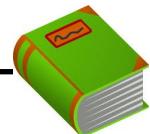
## Definition: payoff matrix

Let  $V_p(\pi_A, \pi_B)$  be the utility for player  $p$ .

- In the prisoner's dilemma, the players get both penalized only a little bit if they both refuse to testify, but if one of them defects, then the other will get penalized a huge amount. So in practice, what tends to happen is that both will testify and both get sentenced to 5 years, which is clearly worse than if they both had cooperated.

# Nash equilibrium

Can't apply von Neumann's minimax theorem (not zero-sum), but get something weaker:



## Definition: Nash equilibrium

A **Nash equilibrium** is  $(\pi_A^*, \pi_B^*)$  such that no player has an incentive to change his/her strategy:

$$V_A(\pi_A^*, \pi_B^*) \geq V_A(\pi_A, \pi_B^*) \text{ for all } \pi_A$$

$$V_B(\pi_A^*, \pi_B^*) \geq V_B(\pi_A^*, \pi_B) \text{ for all } \pi_B$$



## Theorem: Nash's existence theorem [1950]

In any finite-player game with finite number of actions, there exists **at least one** Nash equilibrium.

- Since we no longer have a zero-sum game, we cannot apply the minimax theorem, but we can still get a weaker result.
- A Nash equilibrium is kind of a state point, where no player has an incentive to change his/her policy unilaterally. Another major result in game theory is Nash's existence theorem, which states that any game with a finite number of players (importantly, not necessarily zero-sum) has at least one Nash equilibrium (a stable point). It turns out that finding one is hard, but we can be sure that one exists.

# Examples of Nash equilibria



## Example: Two-finger Morra

Nash equilibrium: A and B both play  $\pi = [\frac{7}{12}, \frac{5}{12}]$ .



## Example: Collaborative two-finger Morra

Two Nash equilibria:

- A and B both play 1 (value is 2).
- A and B both play 2 (value is 4).



## Example: Prisoner's dilemma

Nash equilibrium: A and B both testify.

- Here are three examples of Nash equilibria. The minimax strategies for zero-sum are also equilibria (and they are global optima).
- For purely collaborative games, the equilibria are simply the entries of the payoff matrix for which no other entry in the row or column are larger. There are often multiple local optima here.
- In the Prisoner's dilemma, the Nash equilibrium is when both players testify. This is of course not the highest possible reward, but it is stable in the sense that neither player would want to change his/her strategy. If both players had refused, then one of the players could testify to improve his/her payoff (from -1 to 0).



# Summary so far

## Simultaneous zero-sum games:

- von Neumann's minimax theorem
- Multiple minimax strategies, single game value

## Simultaneous non-zero-sum games:

- Nash's existence theorem
- Multiple Nash equilibria, multiple game values

Huge literature in game theory / economics

- For simultaneous zero-sum games, all minimax strategies have the same game value (and thus it makes sense to talk about the value of a game). For non-zero-sum games, different Nash equilibria could have different game values (for example, consider the collaborative version of two-finger Morra).



# Roadmap

TD learning

Simultaneous games

Non-zero-sum games

**State-of-the-art**



# State-of-the-art: chess

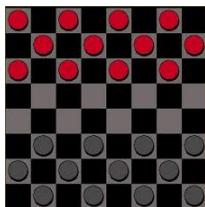
1997: IBM's Deep Blue defeated world champion Gary Kasparov

## Fast computers:

- Alpha-beta search over 30 billion positions, depth 14
- Singular extensions up to depth 20

## Domain knowledge:

- Evaluation function: 8000 features
- 4000 "opening book" moves, all endgames with 5 pieces
- 700,000 grandmaster games
- Null move heuristic: opponent gets to move twice



## State-of-the-art: checkers

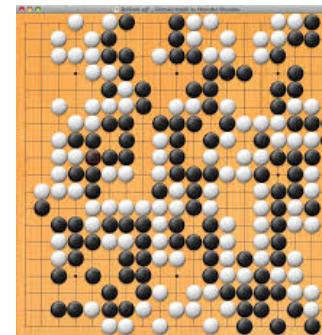
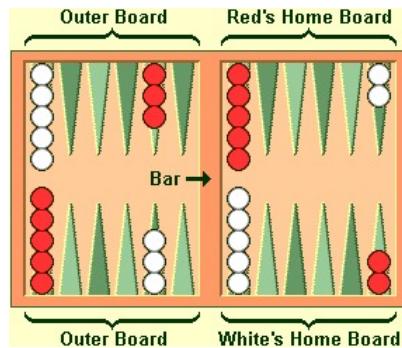
1990: Jonathan Schaeffer's **Chinook** defeated human champion; ran on standard PC

Closure:

- 2007: Checkers solved in the minimax sense (outcome is draw), but doesn't mean you can't win
- Alpha-beta search + 39 trillion endgame positions

# Backgammon and Go

Alpha-beta search isn't enough...



Challenge: large branching factor

- Backgammon: randomness from dice (can't prune!)
- Go: large board size (361 positions)

Solution: learning

- For games such as checkers and chess with a manageable branching factor, one can rely heavily on minimax search along with alpha-beta pruning and a lot of computation power. A good amount of domain knowledge can be employed as to attain or surpass human-level performance.
- However, games such as Backgammon and Go require more due to the large branching factor. Backgammon does not intrinsically have a larger branching factor, but much of this branching is due to the randomness from the dice, which cannot be pruned (it doesn't make sense to talk about the most promising dice move).
- As a result, programs for these games have relied a lot on TD learning to produce good evaluation functions without searching the entire space.

# AlphaGo



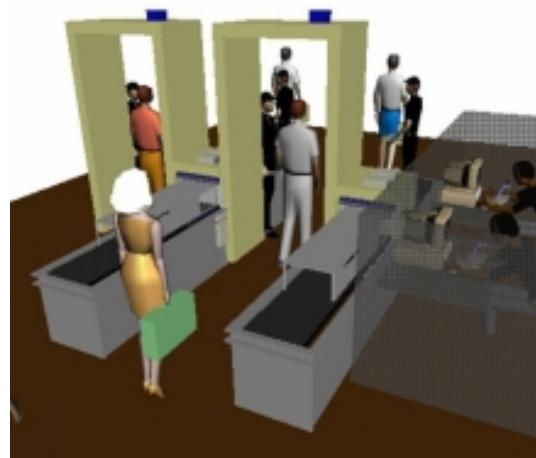
- Supervised learning: on human games
- Reinforcement learning: on self-play games
- Evaluation function: convolutional neural network (value network)
- Policy: convolutional neural network (policy network)
- Monte Carlo Tree Search: search / lookahead

Section: AlphaGo Zero

- The most recent visible advance in game playing was March 2016, when Google DeepMind's AlphaGo program defeated Le Sedol, one of the best professional Go players 4-1.
- AlphaGo took the best ideas from game playing and machine learning. DeepMind executed these ideas well with lots of computational resources, but these ideas should already be familiar to you.
- The learning algorithm consisted of two phases: a supervised learning phase, where a policy was trained on games played by humans (30 million positions) from the KGS Go server; and a reinforcement learning phase, where the algorithm played itself in attempt to improve, similar to what we say with Backgammon.
- The model consists of two pieces: a value network, which is used to evaluate board positions (the evaluation function); and a policy network, which predicts which move to make from any given board position (the policy). Both are based on convolutional neural networks, which we'll discuss later in the class.
- Finally, the policy network is not used directly to select a move, but rather to guide the search over possible moves in an algorithm similar to Monte Carlo Tree Search.

# Other games

**Security games:** allocate limited resources to protect a valuable target.  
Used by TSA security, Coast Guard, protect wildlife against poachers, etc.



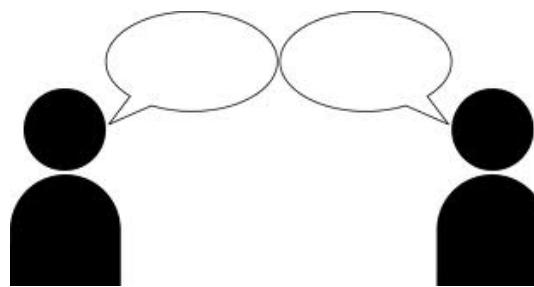
- The techniques that we've developed for game playing go far beyond recreational uses. Whenever there are multiple parties involved with conflicting interests, game theory can be employed to model the situation.
- For example, in a security game a defender needs to protect a valuable target from a malicious attacker. Game theory can be used to model these scenarios and devise optimal (randomized) strategies. Some of these techniques are used by TSA security at airports, to schedule patrol routes by the Coast Guard, and even to protect wildlife from poachers.

# Other games

**Resource allocation:** users share a resource (e.g., network bandwidth); selfish interests leads to volunteer's dilemma



**Language:** people have speaking and listening strategies, mostly collaborative, applied to dialog systems



- For example, in resource allocation, we might have  $n$  people wanting to access some Internet resource. If all of them access the resource, then all of them suffer because of congestion. Suppose that if  $n - 1$  connect, then those people can access the resource and are happy, but the one person left out suffers. Who should volunteer to step out (this is the volunteer's dilemma)?
- Another interesting application is modeling communication. There are two players, the speaker and the listener, and the speaker's actions are to choose what words to use to convey a message. Usually, it's a collaborative game where utility is high when communication is successful and efficient. These game-theoretic techniques have been applied to building dialog systems.



# Summary

- Main challenge: not just one objective
- Minimax principle: guard against adversary in turn-based games
- Simultaneous non-zero-sum games: mixed strategies, Nash equilibria
- Strategy: **search game tree + learned evaluation function**

- Games are an extraordinary rich topic of study, and we have only seen the tip of the iceberg. Beyond simultaneous non-zero-sum games, which are already complex, there are also games involving partial information (e.g., poker).
- But even if we just focus on two-player zero-sum games, things are quite interesting. To build a good game-playing agent involves integrating the two main thrusts of AI: search and learning, which are really symbiotic. We can't possibly search an exponentially large number of possible futures, which means we fall back to an evaluation function. But in order to learn an evaluation function, we need to search over enough possible futures to build an accurate model of the likely outcome of the game.

# CS 221 Section 5: Games

Jerry Qu & Will Deaderick

# Section Contents

- Minimax Search, w/ example
- Evaluation Functions
- Alpha-Beta Pruning, w/ example
- Alpha-Beta practice problem
- Game Theory practice problem
- MCTS & AlphaZero (time permitting)

# Minimax

- Applicable to two-player zero-sum games (and other adversarial games)
- Characteristics:
  - Players take turns
  - Utility comes from end state, no intermediate utility
- Overall goal is to model outcomes when both players play optimally
- When state transitions are not deterministic, we use *expectimax*.
- Complexity:  $O(d)$  space,  $O(b^{(2d)})$  time, for depth  $d$ , branching factor  $b$

# Evaluation Functions

- Goal is to provide some estimate of  $V_{\text{minimax}}(s)$  for state  $s$ .
- Oftentimes requires domain knowledge since we can't make additional moves.
- Because of large time complexity for searching games, evaluation functions are often necessary.
- Pitfalls: in some cases where utility can fluctuate (e.g. a piece capturing sequence in chess), evaluation functions may become inaccurate.

# Alpha-Beta Pruning: Interpretation

- At each “**Max**” state, the “**Max**” player keeps track of **alpha**, a lower bound on her value for that state.

# Alpha-Beta Pruning: Interpretation

- At each “**Max**” state, the “**Max**” player keeps track of **alpha**, a lower bound on her value for that state.
  - Values lower than **alpha** don’t interest “**Max**”

# Alpha-Beta Pruning: Interpretation

- At each “**Max**” state, the “**Max**” player keeps track of **alpha**, a lower bound on her value for that state.
  - Values lower than **alpha** don’t interest “**Max**”
- At each “**Min**” state, the “**Min**” player keeps track of **beta**, an upper bound on her value for that state.

# Alpha-Beta Pruning: Interpretation

- At each “**Max**” state, the “**Max**” player keeps track of **alpha**, a lower bound on her value for that state.
  - Values lower than **alpha** don’t interest “**Max**”
- At each “**Min**” state, the “**Min**” player keeps track of **beta**, an upper bound on her value for that state.
  - Values larger than **beta** don’t interest “**Min**”

# Alpha-Beta Pruning: Rules

- Start with **alpha** as negative infinity, and **beta** as infinity
- Propagate **alpha** and **beta** down the search tree
- At each “**Max**” node, update **alpha** if we find a larger (than **alpha**) child leaf value, child alpha, or child beta
- At each “**Min**” node, update **beta** if we find a smaller (than **beta**) child leaf value, child alpha, and child beta
- If **alpha**  $\geq$  **beta**, prune!

# Alpha-Beta Pruning: Intuition

- We can prune beneath a “**Max**” node if that “**Max**” node’s parent is already guaranteed a better (i.e. smaller) value than any of the “**Max**” node’s children.

# Alpha-Beta Pruning: Intuition

- We can prune beneath a “**Max**” node if that “**Max**” node’s parent is already guaranteed a better (i.e. smaller) value than any of the “**Max**” node’s children.
- We can prune beneath a “**Min**” node if that “**Min**” node’s parent is already guaranteed a better (i.e. larger) value than any of the “**Min**” node’s children.

# Extensions

- Beam search (“forward pruning”) - use an evaluation function to only expand the most promising child nodes, effectively lowering the branching factor

# Extensions

- Beam search (“forward pruning”) - use an evaluation function to only expand the most promising child nodes, effectively lowering the branching factor
  - Pros: similar to human intelligence, computational efficiency

# Extensions

- Beam search (“forward pruning”) - use an evaluation function to only expand the most promising child nodes, effectively lowering the branching factor
  - Pros: similar to human intelligence, computational efficiency
  - Cons: similar to human intelligence, no more guarantees

# Extensions

- Beam search (“forward pruning”) - use an evaluation function to only expand the most promising child nodes, effectively lowering the branching factor
  - Pros: similar to human intelligence, computational efficiency
  - Cons: similar to human intelligence, no more guarantees
- Dynamic move ordering - Manually specify which moves to expand first, or use an evaluation function, or keep track of which moves have been most successful in the past

# Game Theory

- Applicable to single-move simultaneous games (covered already) and more.
- Defined by a set of players, actions, and utility matrix.
- We can have *pure strategies* or *mixed strategies*.
- Nash Equilibrium: a (pure or mixed) strategy for all players such that no player can gain utility by changing their strategy.
  - In any finite-player game with finite actions, there's ALWAYS at least one Nash Equilibrium.



# Lecture 11: CSPs I

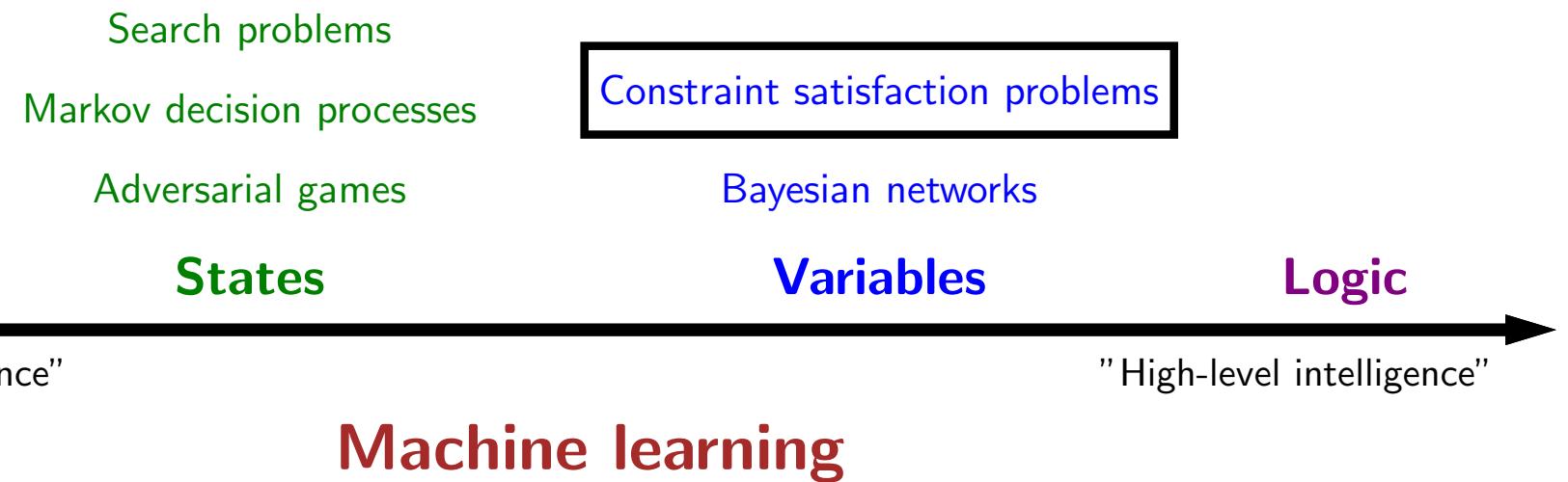
2		5	1	9
5		3		6
6	4			
			1 3 7	
	6		9	
5	9	3		
			4	8
8		5		2
1	7	8		4



# Question

Find two neighboring countries, one that begins with an A and the other that speaks Hungarian.

# Course plan



- We've finished our tour of machine learning and state-based models, which brings us to the midpoint of this course. Let's reflect a bit on what we've covered so far.

# Paradigm

Modeling

Inference

Learning

# State-based models

## [Modeling]

<b>Framework</b>	search problems	MDPs/games
<b>Objective</b>	minimum cost paths	maximum value policies

## [Inference]

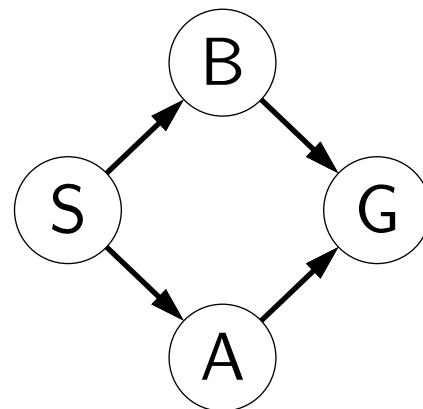
<b>Tree-based</b>	backtracking	minimax/expectimax
<b>Graph-based</b>	DP, UCS, A*	value/policy iteration

## [Learning]

<b>Methods</b>	structured Perceptron	Q-learning, TD learning
----------------	-----------------------	-------------------------

- **Modeling:** In the context of state-based models, we seek to find minimum cost paths (for search problems) or maximum value policies (for MDPs and games).
- **Inference:** To compute these solutions, we can either work on the search/game tree or on the state graph. In the former case, we end up with recursive procedures which take exponential time but require very little memory (generally linear in the size of the solution). In the latter case, where we are fortunate to have few enough states to fit into memory, we can work directly on the graph, which can often yield an exponential savings in time.
- Given that we can find the optimal solution with respect to a fixed model, the final question is where this model actually comes from. **Learning** provides the answer: from data. You should think of machine learning as not just a way to do binary classification, but more as a way of life, which can be used to support a variety of different models.
- In the rest of the course, modeling, inference, and learning will continue to be the three pillars of all techniques we will develop.

# State-based models: takeaway 1



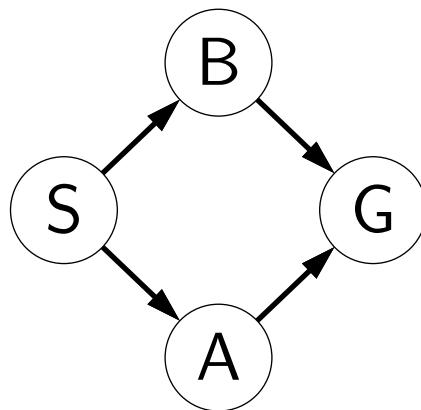
**Key idea: specify locally, optimize globally**

Modeling: specifies local interactions

Inference: find globally optimal solutions

- One high-level takeaway is the motto: specify locally, optimize globally. When we're building a search problem, we only need to specify how the states are connected through actions and what the local action costs are; we need not specify the long-term consequences of taking an action. It is the job of the inference to take all of this local information into account and produce globally optimal solutions (minimum cost paths).
- This separation is quite powerful in light of modeling and inference: having to worry only about local interactions makes modeling easier, but we still get the benefits of a globally optimal solution via inference which are constructed independent of the domain-specific details.
- We will see this local specification + global optimization pattern again in the context of variable-based models.

# State-based models: takeaway 2



## Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

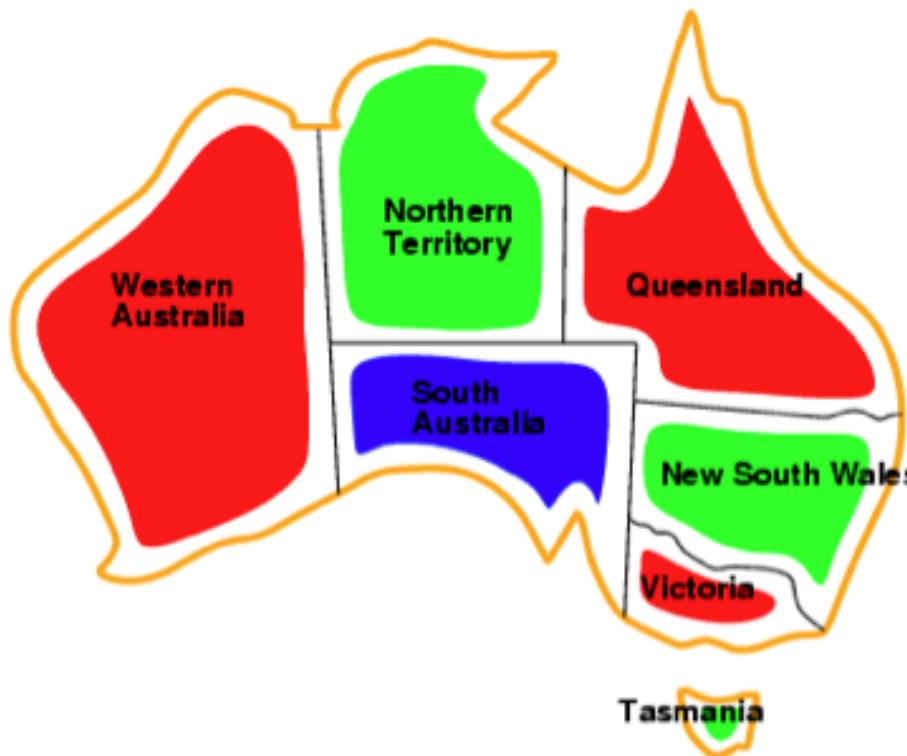
Mindset: move through states (nodes) via actions (edges)

- The second high-level takeaway which is core to state-based models is the notion of **state**. The state, which summarizes previous actions, is one of the key tools that allows us to manage the exponential search problems frequently encountered in AI. We will see the notion of state appear again in the context of conditional independence in variable-based models.
- With states, we were in the mindset of thinking about taking a sequence of actions (where order is important) to reach a goal. However, in some tasks, order is irrelevant. In these cases, maybe search isn't the best way to model the task. Let's see an example.



Question: how can we color each of the 7 provinces {red,green,blue} so that no two neighboring provinces have the same color?

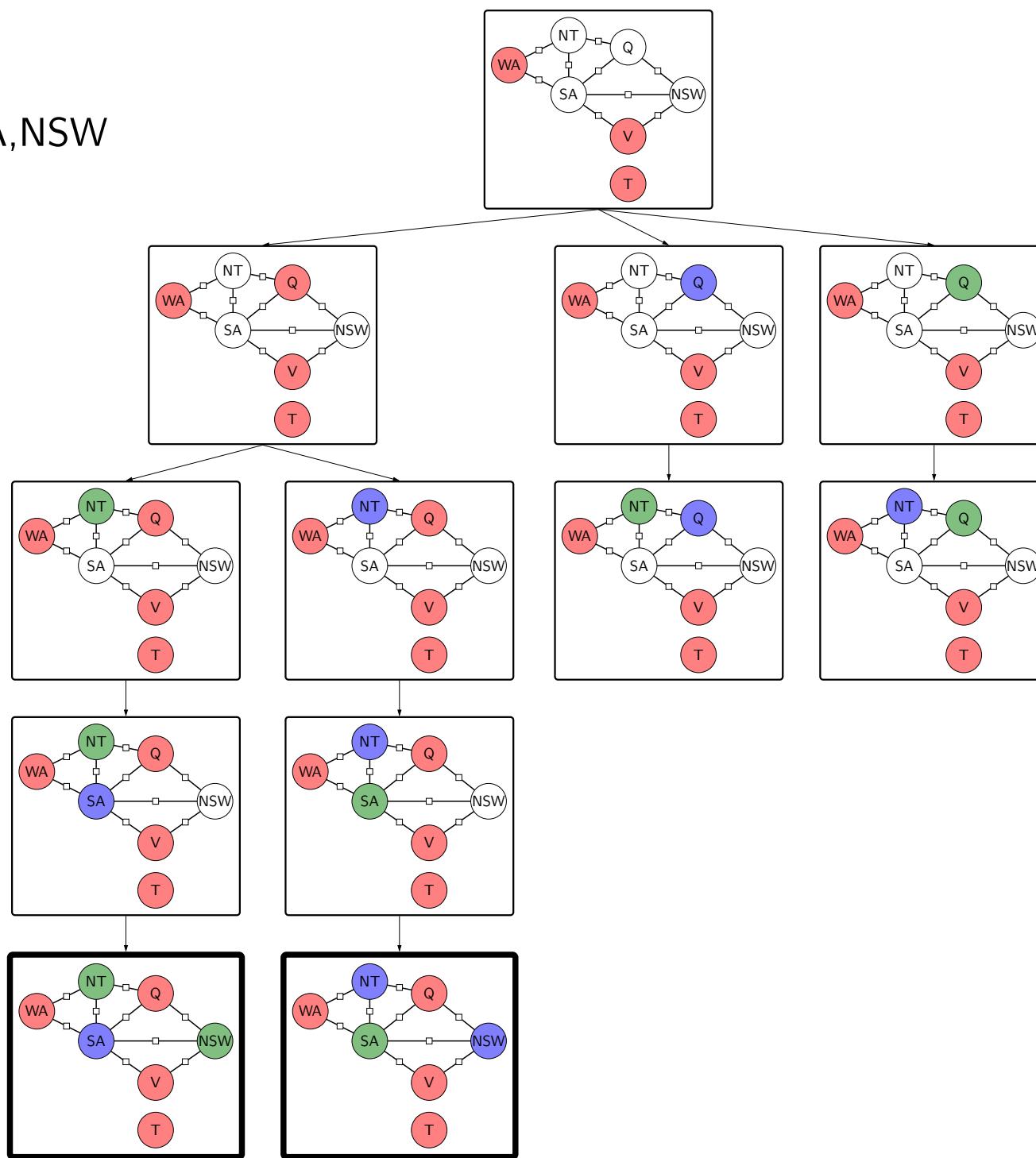
# Map coloring



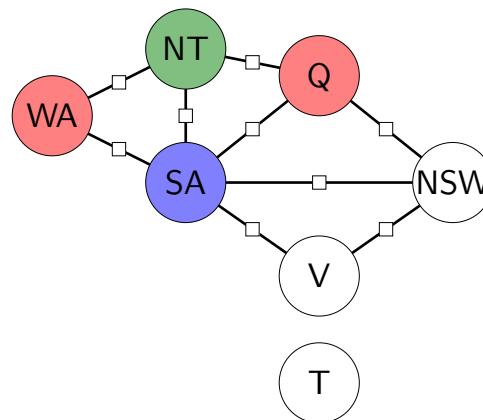
(one possible solution)

# Search

WA,V,T,Q,NT,SA,NSW



# As a search problem



- **State:** partial assignment of colors to provinces
- **Action:** assign next uncolored province a compatible color

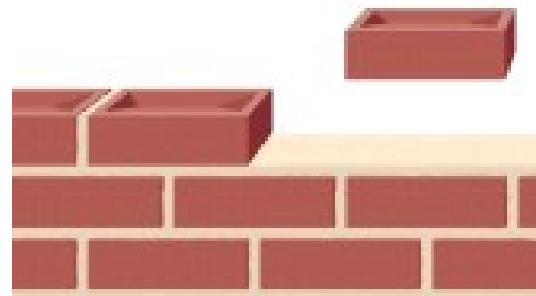
What's missing? There's more problem structure!

- Variable ordering doesn't affect correctness
- Variables are interdependent in a local way

- We can certainly use search to find an assignment of colors to the provinces of Australia. Let's fix an arbitrary ordering of the provinces. Each state contains an assignment of colors to a subset of the provinces (**a partial assignment**), and each action chooses a color for the next unassigned province as long as the color isn't already assigned to one of its neighbors. In this way, all the leaves of the search tree are solutions (18 of them). (In the slide, in the interest of space, we've only shown the subtree rooted at a partial assignment to 3 variables.)
- This is a fine way to solve this problem, and in general, it shows how powerful search is: we don't actually need any new machinery to solve this problem. But the question is: can we do better?
- First, the particular search tree that we drew had several dead ends; it would be better if we could detect these earlier. We will see in this lecture that the fact that **the order in which we assign variables doesn't matter for correctness** gives us the flexibility to dynamically choose a better ordering of the variables. That, with a bit of lookahead will allow us to dramatically improve the efficiency over naive tree search.
- Second, it's clear that Tasmania's color can be any of the three colors regardless of the colors on the mainland. This is an instance of **independence**, and next time, we'll see how to exploit these observations systematically.

# Variable-based models

A new framework...



## Key idea: variables

- Solutions to problems  $\Rightarrow$  assignments to variables (**modeling**).
- Decisions about variable ordering, etc. chosen by **inference**.

- With that motivation in mind, we now embark on our journey into variable-based models. Variable-based models is an umbrella term that includes constraint satisfaction problems (CSPs), Markov networks, Bayesian networks, hidden Markov models (HMMs), conditional random fields (CRFs), etc., which we'll get to later in the course. The term graphical models can be used interchangeably with variable-based models, and the term probabilistic graphical models (PGMs) generally encompasses both Markov networks (also called undirected graphical models) and Bayesian networks (directed graphical models).
- The unifying theme is the idea of thinking about solutions to problems as assignments of values to variables (this is the modeling part). All the details about how to find the assignment (in particular, which variables to try first) are delegated to inference. So the advantage of using variable-based models over state-based models is that it's making the algorithms do more of the work, freeing up more time for modeling.
- An apt analogy is programming languages. Solving a problem directly by implementing an ad-hoc program is like using assembly language. Solving a problem using state-based models is like using C. Solving a problem using variable-based models is like using Python. By moving to a higher language, you might forgo some amount of ability to optimize manually, but the advantage is that (i) you can think at a higher level and (ii) there are more opportunities for optimizing automatically.
- Once the different modeling frameworks become second nature, it is almost as if they are invisible. It's like when you master a language, you can "think" in it without constantly referring to the framework.



# Roadmap

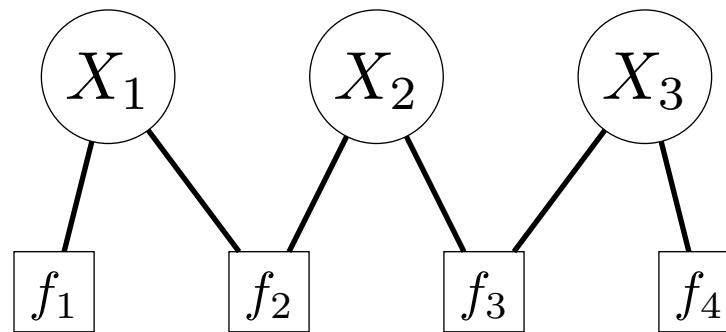
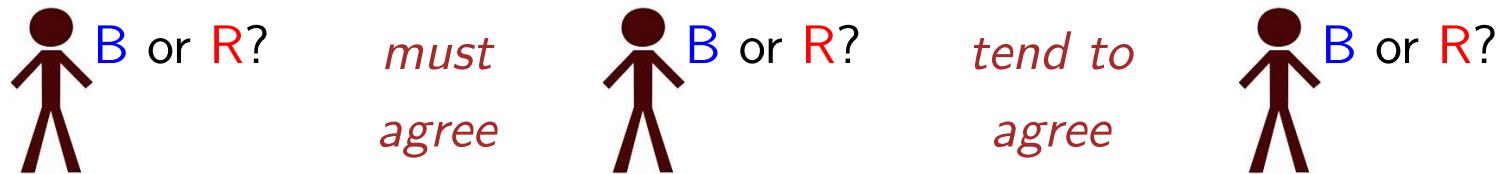
Factor graphs

Dynamic ordering

Arc consistency

Modeling

# Factor graph (example)



$x_1$	$f_1(x_1)$
R	0
B	1

$x_1$	$x_2$	$f_2(x_1, x_2)$
R	R	1
R	B	0
B	R	0
B	B	1

$x_2$	$x_3$	$f_3(x_2, x_3)$
R	R	3
R	B	2
B	R	2
B	B	3

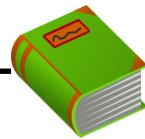
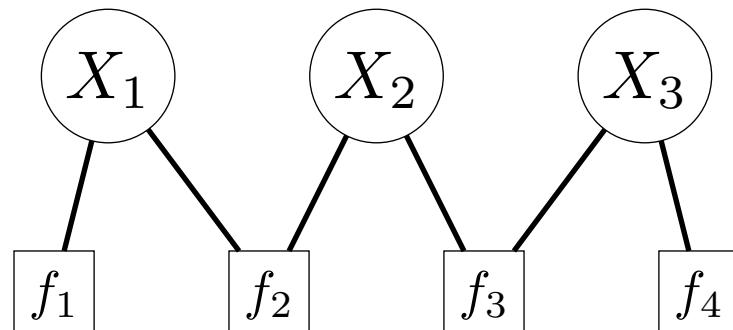
$x_3$	$f_4(x_3)$
R	2
B	1

$$f_2(x_1, x_2) = [x_1 = x_2] \quad f_3(x_2, x_3) = [x_2 = x_3] + 2$$

[demo]

- The most important concept for the next three weeks will be that of a **factor graph**. But before we define it formally, let us consider a simple example.
- Suppose there are three people, each of which will vote for a color, red or blue. We know that Person 1 is leaning pretty set on blue, and Person 3 is leaning red. Person 1 and Person 2 must have the same color, while Person 2 and Person 3 would weakly prefer to have the same color.
- We can model this as a factor graph consisting of three **variables**,  $X_1, X_2, X_3$ , each of which must be assigned red (**R**) or blue (**B**).
- We encode each of the constraints/preferences as a **factor**, which assigns a non-negative number based on the assignment to a subset of the variables. We can either describe the factor as an explicit table, or via a function (e.g.,  $[x_1 = x_2]$ ).

# Factor graph



## Definition: factor graph

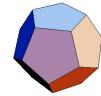
Variables:

$$X = (X_1, \dots, X_n), \text{ where } X_i \in \text{Domain}_i$$

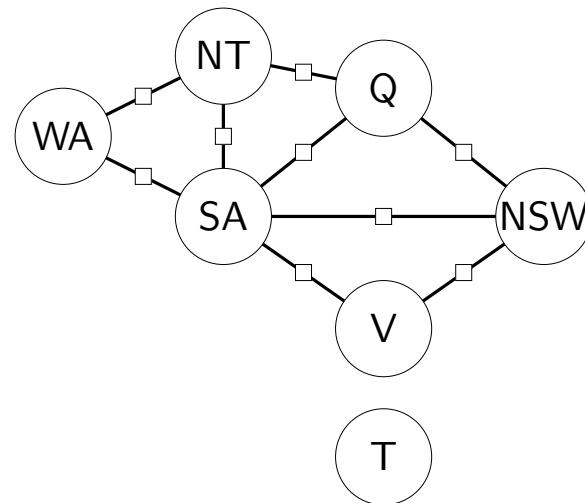
Factors:

$$f_1, \dots, f_m, \text{ with each } f_j(X) \geq 0$$

- Now we proceed to the general definition. a factor graph consists of a set of variables and a set of factors:  
 (i)  $n$  variables  $X_1, \dots, X_n$ , which are represented as circular nodes in the graphical notation; and (ii)  
 $m$  factors (also known as potentials)  $f_1, \dots, f_m$ , which are represented as square nodes in the graphical  
 notation.
- Each variable  $X_i$  can take on values in its **domain**  $\text{Domain}_i$ . Each factor  $f_j$  is a function that takes an  
 assignment  $x$  to all the variables and returns a non-negative number representing how good that assignment  
 is (from the factor's point of view). Usually, each factor will depend only on a small subset of the variables.



## Example: map coloring



Variables:

$$X = (\text{WA}, \text{NT}, \text{SA}, \text{Q}, \text{NSW}, \text{V}, \text{T})$$

$$\text{Domain}_i \in \{\text{R}, \text{G}, \text{B}\}$$

Factors:

$$f_1(X) = [\text{WA} \neq \text{NT}]$$

$$f_2(X) = [\text{NT} \neq \text{Q}]$$

...

- Notation: we use  $[condition]$  to represent the indicator function which is equal to 1 if the condition is true and 0 if not. Normally, this is written  $\mathbf{1}[condition]$ , but we drop the  $\mathbf{1}$  for succinctness.

# Factors



## Definition: scope and arity

**Scope** of a factor  $f_j$ : set of variables it depends on.

**Arity** of  $f_j$  is the number of variables in the scope.

**Unary** factors (arity 1); **Binary** factors (arity 2).



## Example: map coloring

- Scope of  $f_1(X) = [\text{WA} \neq \text{NT}]$  is  $\{\text{WA}, \text{NT}\}$
- $f_1$  is a binary factor

- The key aspect that makes factor graphs useful is that each factor  $f_j$  only depends on a subset of variables, called the **scope**. The arity of the factors is generally small (think 1 or 2).

# Assignment weights (example)

$x_1$	$f_1(x_1)$
R	0
B	1

$x_1$	$x_2$	$f_2(x_1, x_2)$
R	R	1
R	B	0
B	R	0
B	B	1

$x_2$	$x_3$	$f_3(x_2, x_3)$
R	R	3
R	B	2
B	R	2
B	B	3

$x_1$	$x_2$	$x_3$	Weight
R	R	R	$0 \cdot 1 \cdot 3 \cdot 2 = 0$
R	R	B	$0 \cdot 1 \cdot 2 \cdot 1 = 0$
R	B	R	$0 \cdot 0 \cdot 2 \cdot 2 = 0$
R	B	B	$0 \cdot 0 \cdot 3 \cdot 1 = 0$
B	R	R	$1 \cdot 0 \cdot 3 \cdot 2 = 0$
B	R	B	$1 \cdot 0 \cdot 2 \cdot 1 = 0$
B	B	R	$1 \cdot 1 \cdot 2 \cdot 2 = 4$
B	B	B	$1 \cdot 1 \cdot 3 \cdot 1 = 3$

- A factor graph specifies all the local interactions between variables. We wish to find a global solution. A solution is called an **assignment**, which specifies a value for each variable.
- Each assignment is associated with a weight, which is just the product over each factor evaluated on that assignment. Intuitively, each factor contributes to the weight. Note that any factor has veto power: if it returns zero, then the entire weight is irrecoverably zero.
- In this setting, the maximum weight assignment is  $(B, B, R)$ , which has a weight of 4. You can think of this as the optimal configuration or the most likely outcome.

# Assignment weights



## Definition: assignment weight

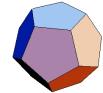
Each **assignment**  $x = (x_1, \dots, x_n)$  has a **weight**:

$$\text{Weight}(x) = \prod_{j=1}^m f_j(x)$$

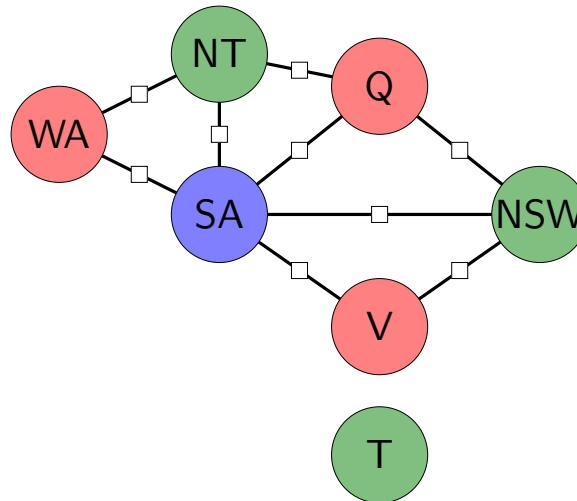
**Objective:** find the maximum weight assignment

$$\arg \max_x \text{Weight}(x)$$

- Formally, the **weight** of an assignment  $x$  is the product of all the factors applied to that assignment ( $\prod_{j=1}^m f_j(x)$ ). Think of all the factors chiming in on their opinion of  $x$ . We multiply all these opinions together to get the global opinion.
- Our objective will be to find the **maximum weight assignment**.
- Note: do not confuse the term "weight" in the context of factor graphs with the "weight vector" in machine learning.



## Example: map coloring



Assignment:

$$x = \{\text{WA : R, NT : G, SA : B, Q : R, NSW : G, V : R, T : G}\}$$

Weight:

$$\text{Weight}(x) = 1 \cdot 1 = 1$$

Assignment:

$$x' = \{\text{WA : R, NT : R, SA : B, Q : R, NSW : G, V : R, T : G}\}$$

Weight:

$$\text{Weight}(x') = 0 \cdot 0 \cdot 1 = 0$$

- In the map coloring example, each factor only looks at the variables of two adjacent provinces and checks if the colors are different (returning 1) or the same (returning 0). From a modeling perspective, this allows us to specify local interactions in a modular way. A global notion of consistency is achieved by multiplying together all the factors.
- Again note that the factors are multiplied (not added), which means that any factor has veto power: a single zero causes the entire weight to be zero.

# Constraint satisfaction problems



## Definition: constraint satisfaction problem (CSP)

A CSP is a factor graph where all factors are **constraints**:

$$f_j(x) \in \{0, 1\} \text{ for all } j = 1, \dots, m$$

The constraint is satisfied iff  $f_j(x) = 1$ .



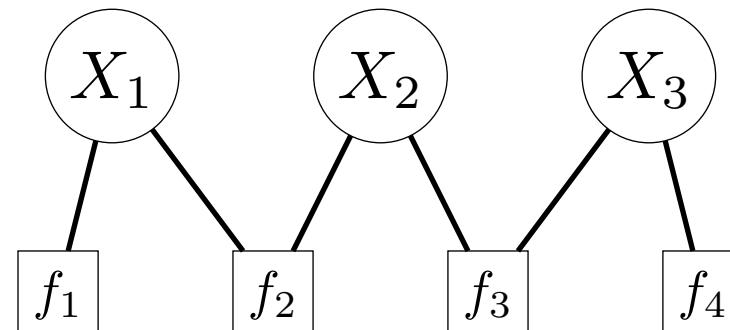
## Definition: consistent assignments

An assignment  $x$  is **consistent** iff  $\text{Weight}(x) = 1$  (i.e., **all** constraints are satisfied).

- Constraint satisfaction problems are just a special case of factor graphs where each of the factors returns either 0 or 1. Such a factor is a **constraint**, where 1 means the constraint is satisfied and 0 means that it is not.
- In a CSP, all assignments have either weight 1 or 0. Assignments with weight 1 are called **consistent** (they satisfy all the constraints), and the assignments with weight 0 are called inconsistent. Our goal is to find any consistent assignment (if one exists).



# Summary so far



**Factor graph** (general)

variables

factors

assignment weight

**CSP** (all or nothing)

variables

constraints

consistent or inconsistent



# Roadmap

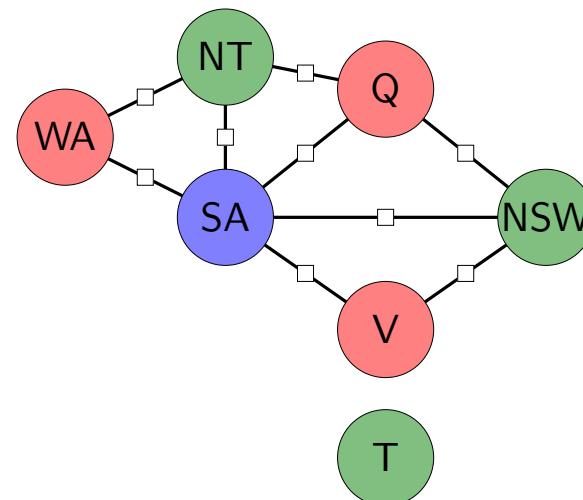
Factor graphs

**Dynamic ordering**

Arc consistency

Modeling

# Extending partial assignments



WA

NT

SA

Q

NSW

V

T

$$[WA \neq NT]$$

$$[WA \neq SA]$$

$$[NT \neq Q]$$

$$[SA \neq NSW]$$

$$[SA \neq V]$$

$$[NT \neq SA]$$

$$[SA \neq Q]$$

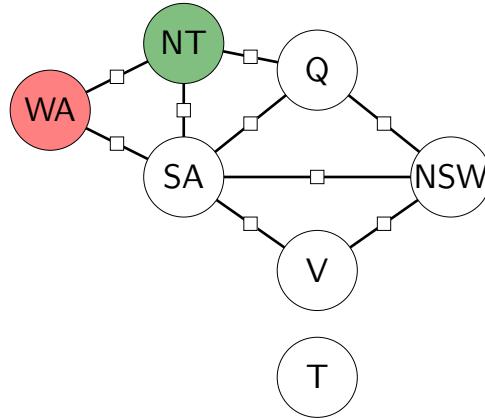
$$[Q \neq NSW]$$

$$[NSW \neq V]$$

- The general idea, as we've already seen in our search-based solution is to work with **partial assignments**. We've defined the weight of a full assignment to be the product of all the factors applied to that assignment.
- We extend this definition to partial assignments: The weight of a partial assignment is defined to be the product of all the factors whose scope includes only assigned variables. For example, if only WA and NT are assigned, the weight is just value of the single factor between them.
- When we assign a new variable a value, the weight of the new extended assignment is defined to be the original weight times all the factors that depend on the new variable and only previously assigned variables.

# Dependent factors

- Partial assignment (e.g.,  $x = \{\text{WA} : \text{R}, \text{NT} : \text{G}\}$ )



## Definition: dependent factors

Let  $D(x, X_i)$  be set of factors depending on  $X_i$  and  $x$  but not on unassigned variables.

$$D(\{\text{WA} : \text{R}, \text{NT} : \text{G}\}, \text{SA}) = \{[\text{WA} \neq \text{SA}], [\text{NT} \neq \text{SA}]\}$$

- Formally, we will use  $D(x, X_i)$  to denote this set of these factors, which we will call **dependent factors**.
- For example, if we assign SA, then  $D(x, \text{SA})$  contains two factors: the one between SA and WA and the one between SA and NT.

# Backtracking search



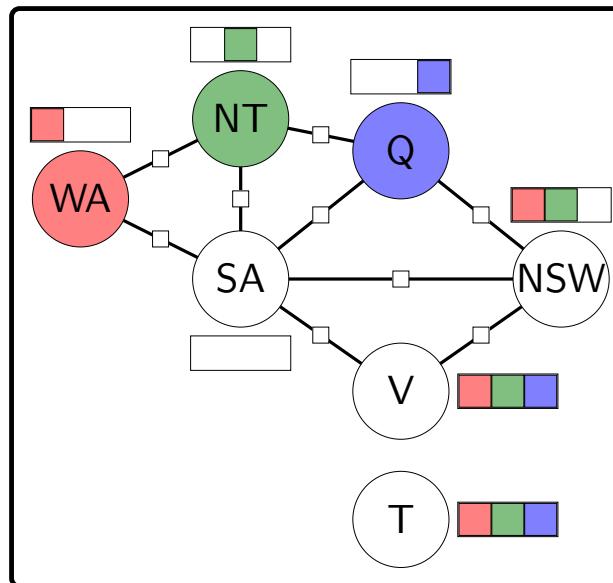
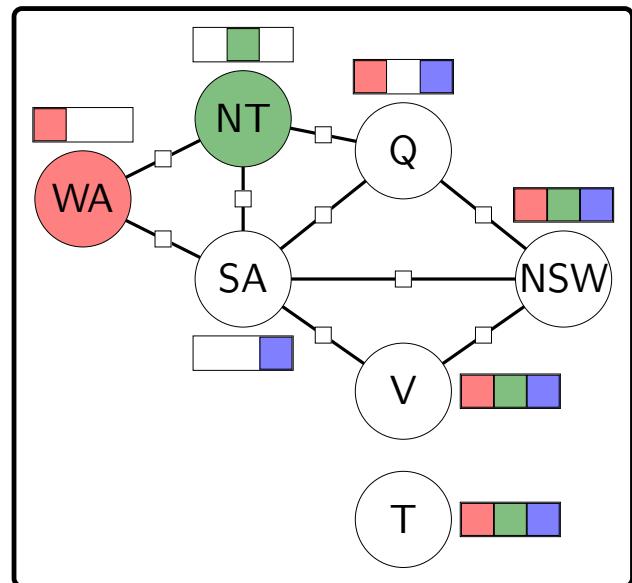
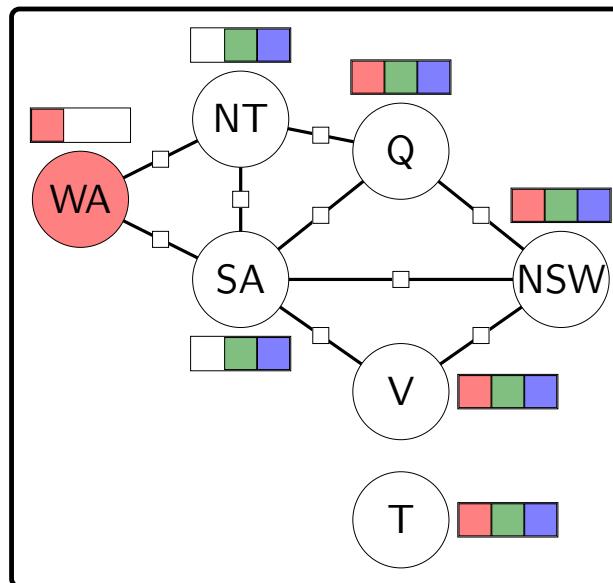
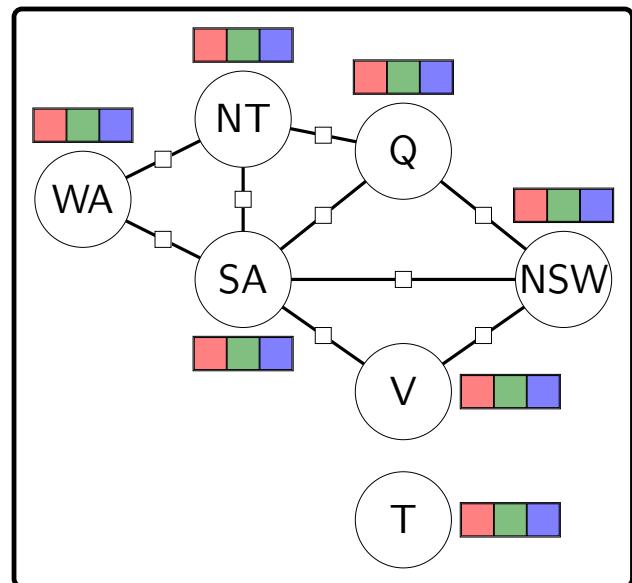
## Algorithm: backtracking search

Backtrack( $x, w, \text{Domains}$ ):

- If  $x$  is complete assignment: update best and return
- Choose unassigned **VARIABLE**  $X_i$
- Order **VALUES**  $\text{Domain}_i$  of chosen  $X_i$
- For each value  $v$  in that order:
  - $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$
  - If  $\delta = 0$ : continue
  - **Domains'  $\leftarrow$  Domains via LOOKAHEAD**
  - Backtrack( $x \cup \{X_i : v\}, w\delta, \text{Domains}'$ )

- Now we are ready to present the full backtracking search, which is a recursive procedure that takes in a partial assignment  $x$ , its weight  $w$ , and the domains of all the variables  $\text{Domains} = (\text{Domain}_1, \dots, \text{Domain}_n)$ .
- If the assignment  $x$  is complete (all variables are assigned), then we update our statistics based on what we're trying to compute: We can increment the total number of assignments seen so far, check to see if  $x$  is better than the current best assignment that we've seen so far (based on  $w$ ), etc. (For CSPs where all the weights are 0 or 1, we can stop as soon as we find one consistent assignment, just as in DFS for search problems.)
- Otherwise, we choose an **unassigned variable**  $X_i$ . Given the choice of  $X_i$ , we choose an **ordering of the values** of that variable  $X_i$ . Next, we iterate through all the values  $v \in \text{Domain}_i$  in that order. For each value  $v$ , we compute  $\delta$ , which is the product of the dependent factors  $D(x, X_i)$ ; recall this is the multiplicative change in weight from assignment  $x$  to the new assignment  $x \cup \{X_i : v\}$ . If  $\delta = 0$ , that means a constraint is violated, and we can ignore this partial assignment completely, because multiplying more factors later on cannot make the weight non-zero.
- We then perform **lookahead**, removing values from the domains  $\text{Domains}$  to produce  $\text{Domains}'$ . This is not required (we can just use  $\text{Domains}' = \text{Domains}$ ), but it can make our algorithm run faster. (We'll see one type of lookahead in the next slide.)
- Finally, we recurse on the new partial assignment  $x \cup \{X_i : v\}$ , the new weight  $w\delta$ , and the new domain  $\text{Domains}'$ .
- If we choose an unassigned variable according to an arbitrary fixed ordering, order the values arbitrarily, and do not perform lookahead, we get the basic tree search algorithm that we would have used if we were thinking in terms of a search problem. We will next start to improve the efficiency by exploiting properties of the CSP.

# Lookahead: forward checking (example)



Inconsistent - prune!

- First, we will look at **forward checking**, which is a way to perform a one-step lookahead. The idea is that as soon as we assign a variable (e.g., WA = **R**), we can pre-emptively remove inconsistent values from the domains of neighboring variables (i.e., those that share a factor).
- If we keep on doing this and get to a point where some variable has an empty domain, then we can stop and backtrack immediately, since there's no possible way to assign a value to that variable which is consistent with the previous partial assignment.
- In this example, after Q is assigned blue, we remove inconsistent values (blue) from SA's domain, emptying it. At this point, we need not even recurse further, since there's no way to extend the current assignment. We would then instead try assigning Q to red.

# Lookahead: forward checking



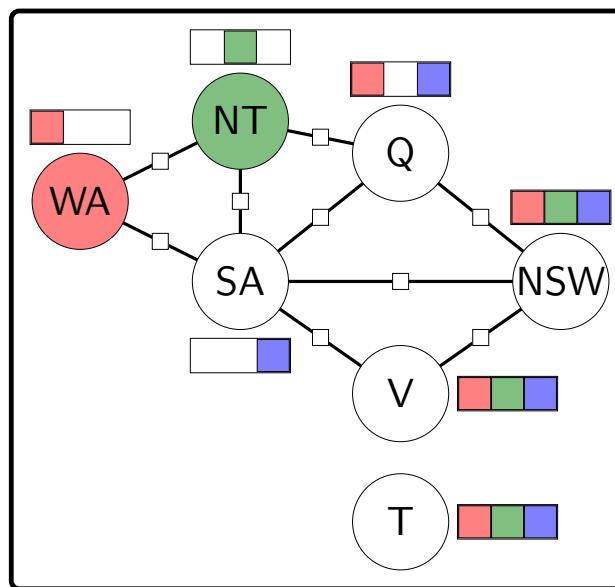
**Key idea: forward checking (one-step lookahead)**

- After assigning a variable  $X_i$ , eliminate inconsistent values from the domains of  $X_i$ 's neighbors.
- If any domain becomes empty, don't recurse.
- When unassign  $X_i$ , restore neighbors' domains.



- When unassigning a variable, remember to restore the domains of its neighboring variables!
- The simplest way to implement this is to make a copy of the domains of the variables before performing forward checking. This is foolproof, but can be quite slow.
- A fancier solution is to keep a counter (initialized to be zero)  $c_{iv}$  for each variable  $X_i$  and value  $v$  in its domain. When we remove a value  $v$  from the domain of  $X_i$ , we increment  $c_{iv}$ . An element is deemed to be "removed" when  $c_{iv} > 0$ . When we want to un-remove a value, we decrement  $c_{iv}$ . This way, the remove operation is reversible, which is important since a value might get removed multiple times due to multiple neighboring variables.
- Later, we will look at arc consistency, which will allow us to lookahead even more.

# Choosing an unassigned variable



Which variable to assign next?



**Key idea: most constrained variable**

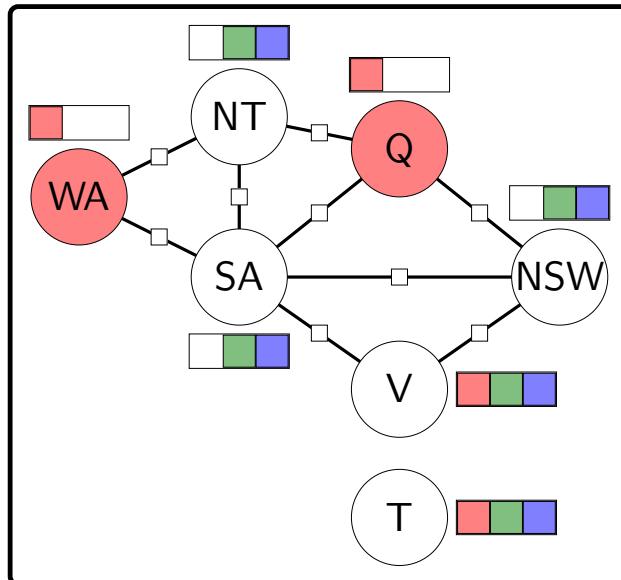
Choose variable that has the fewest consistent values.

This example: SA (has only one value)

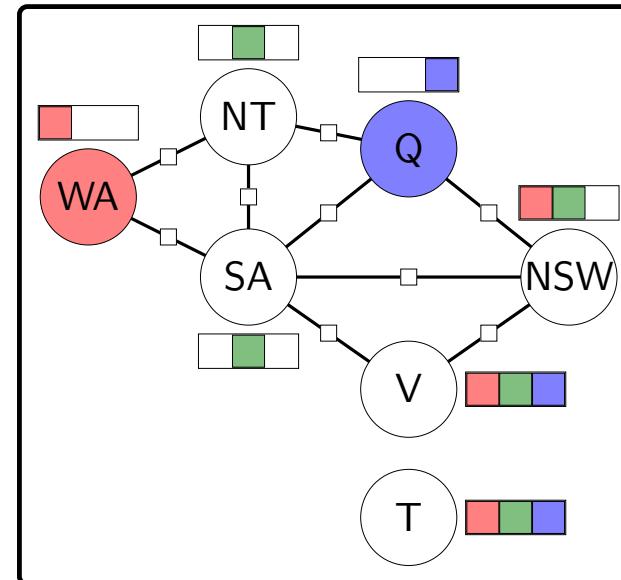
- Now let us look at the problem of choosing an unassigned variable. Intuitively, we want to choose the variable which is most constrained, that is, the variable whose domain has the fewest number of remaining valid values (based on forward checking), because those variables yield smaller branching factors.

# Order values of a selected variable

What values to try for Q?



$$2 + 2 + 2 = 6 \text{ consistent values}$$



$$1 + 1 + 2 = 4 \text{ consistent values}$$



**Key idea: least constrained value**

Order values of selected  $X_i$  by decreasing number of consistent values of neighboring variables.

- Once we've selected an unassigned variable  $X_i$ , we need to figure out which order to try the different values in. Here the principle we will follow is to first try values which are less constrained.
- There are several ways we can think about measuring how constrained a variable is, but for the sake of concreteness, here is the heuristic we'll use: just count the number of values in the domains of all neighboring variables (those that share a factor with  $X_i$ ).
- If we color Q red, then we have 2 valid values for NT, 2 for SA, and 2 for NSW. If we color Q blue, then we have only 1 for NT, 1 for SA, and 2 for NSW. Therefore, red is preferable (6 total valid values versus 4).
- The intuition is that we want values which impose the fewest number of constraints on the neighbors, so that we are more likely to find a consistent assignment.

# When to fail?

Most constrained variable (MCV):

- Must assign **every** variable
- If going to fail, fail early  $\Rightarrow$  more pruning

Least constrained value (LCV):

- Need to choose **some** value
- Choosing value most likely to lead to solution

- The most constrained variable and the least constrained value heuristics might seem conflicting, but there is a good reason for this superficial difference.
- An assignment involves **every** variable whereas for each variable we only need to choose **some** value. Therefore, for variables, we want to try to detect failures early on if possible (because we'll have to confront those variables sooner or later), but for values we want to steer away from possible failures because we might not have to consider those other values.

# When do these heuristics help?

- **Most constrained variable:** useful when **some** factors are constraints (can prune assignments with weight 0)

$$[x_1 = x_2]$$

$$[x_2 \neq x_3] + 2$$

- **Least constrained value:** useful when **all** factors are constraints (all assignment weights are 1 or 0)

$$[x_1 = x_2]$$

$$[x_2 \neq x_3]$$

- **Forward checking:** need to actually prune domains to make heuristics useful!

- Most constrained variable is useful for finding maximum weight assignments in any factor graph as long as there are some factors which are constraints, because we only save work if we can prune away assignments with zero weight, and this only happens with violated constraints (weight 0).
- On the other hand, least constrained value only makes sense if all the factors are constraints (CSPs). In general, ordering the values makes sense if we're going to just find the first consistent assignment. If there are any non-constraint factors, then we need to look at all consistent assignments to see which one has the maximum weight. Analogy: think about when depth-first search is guaranteed to find the minimum cost path.

# Review: backtracking search



## Algorithm: backtracking search

Backtrack( $x, w, \text{Domains}$ ):

- If  $x$  is complete assignment: update best and return
- Choose unassigned **VARIABLE**  $X_i$
- Order **VALUES**  $\text{Domain}_i$  of chosen  $X_i$
- For each value  $v$  in that order:
  - $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$
  - If  $\delta = 0$ : continue
  - **Domains'  $\leftarrow$  Domains via LOOKAHEAD**
  - Backtrack( $x \cup \{X_i : v\}, w\delta, \text{Domains}'$ )



# Roadmap

Factor graphs

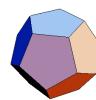
Dynamic ordering

**Arc consistency**

Modeling

# Arc consistency

Idea: eliminate values from domains  $\Rightarrow$  reduce branching



## Example: numbers

Before enforcing arc consistency on  $X_i$ :

$$X_i \in \text{Domain}_i = \{1, 2, 3, 4, 5\}$$

$$X_j \in \text{Domain}_j = \{1, 2\}$$

$$f_1(X) = [X_i + X_j = 4]$$

After enforcing arc consistency on  $X_i$ :

$$X_i \in \text{Domain}_i = \{2, 3\}$$

[whiteboard]

- Now let us return to the issue of using lookahead to eliminate values from domains of unassigned variables. One motivation is that smaller domains lead to smaller branching factors, which makes search faster.
- A second motivation is that since the domain sizes are used in the context of the dynamic ordering heuristics (most constrained variable and least constrained value), we can hope to choose better orderings with domains that more accurately reflect what values are actually possible.
- We've already seen forward checking as a simple way of using lookahead to prune the domains of unassigned variables. Shortly, we will introduce AC-3, which is forward checking without brakes. To build up to that, we need to introduce the idea of arc consistency.
- The idea behind enforcing arc consistency is to look at all the factors that involve just two variables  $X_i$  and  $X_j$  and rule out any values in the domain of  $X_i$  which are obviously bad without even looking at other variables.
- To enforce arc consistency on  $X_i$  with respect to  $X_j$ , we go through each of the values in the domain of  $X_i$  and remove it if there is no value in the domain of  $X_j$  that is consistent with  $X_i$ . For example,  $X_i = 4$  is ruled out because no value  $X_j \in \{1, 2, 3, 4, 5\}$  satisfies  $X_i + X_j = 4$ .

# Arc consistency



## Definition: arc consistency

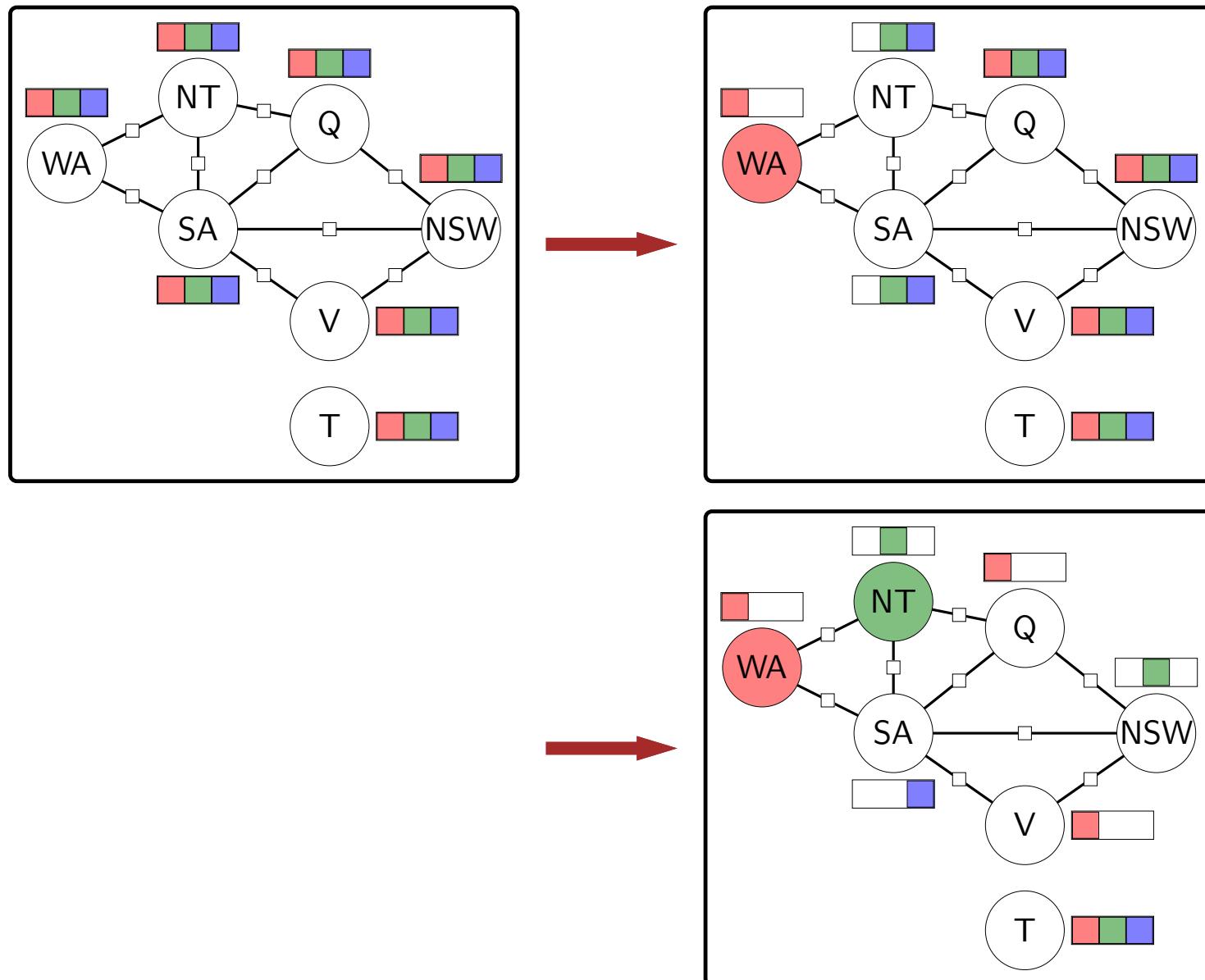
A variable  $X_i$  is **arc consistent** with respect to  $X_j$  if for each  $x_i \in \text{Domain}_i$ , there exists  $x_j \in \text{Domain}_j$  such that  $f(\{X_i : x_i, X_j : x_j\}) \neq 0$  for all factors  $f$  whose scope contains  $X_i$  and  $X_j$ .



## Algorithm: enforce arc consistency

`EnforceArcConsistency( $X_i, X_j$ )`: Remove values from  $\text{Domain}_i$  to make  $X_i$  arc consistent with respect to  $X_j$ .

# AC-3 (example)



# AC-3

**Forward checking:** when assign  $X_j : x_j$ , set  $\text{Domain}_j = \{x_j\}$  and enforce arc consistency on all neighbors  $X_i$  with respect to  $X_j$

**AC-3:** repeatedly enforce arc consistency on all variables



## Algorithm: AC-3

Add  $X_j$  to set.

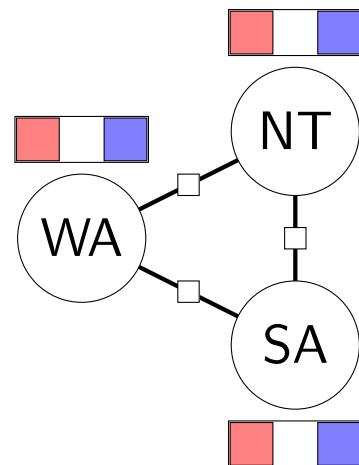
While set is non-empty:

- Remove any  $X_k$  from set.
- For all neighbors  $X_l$  of  $X_k$ :
  - Enforce arc consistency on  $X_l$  w.r.t.  $X_k$ .
  - If  $\text{Domain}_l$  changed, add  $X_l$  to set.

- In fact, we already saw a limited version of arc consistency. In forward checking, when we assign a variable  $X_i$  to a value, we are actually enforcing arc consistency on the neighbors of  $X_i$  with respect to  $X_i$ .
- Why stop there? AC-3 doesn't. In AC-3, we start by enforcing arc consistency on the neighbors of  $X_i$  (forward checking). But then, if the domains of any neighbor  $X_j$  changes, then we enforce arc consistency on the neighbors of  $X_j$ , etc.
- In the example, after we assign WA : **R**, performing AC-3 is the same as forward checking. But after the assignment NT : **G**, AC-3 goes wild and eliminates all but one value from each of the variables on the mainland.
- Note that unlike BFS graph search, a variable could get added to the set multiple times because its domain can get updated more than once. More specifically, we might enforce arc consistency on  $(X_i, X_j)$  up to  $D$  times in the worst case, where  $D = \max_{1 \leq i \leq n} |\text{Domain}_i|$  is the size of the largest domain. There are at most  $m$  different pairs  $(X_i, X_j)$  and each call to enforce arc consistency takes  $O(D^2)$  time. Therefore, the running time of this algorithm is  $O(ED^3)$  in the very worst case where  $E$  is the number of edges (usually, it's much better than this).

# Limitations of AC-3

- Ideally, if no solutions, AC-3 would remove all values from a domain
- AC-3 isn't always effective:



- No consistent assignments, but AC-3 doesn't detect a problem!
- **Intuition:** if we look locally at the graph, nothing blatantly wrong...

- In the best case, if there is no way to consistently assign values all the variables, then running AC-3 will detect that there is no solution by emptying out a domain. However, this is not always the case, as the example above shows. Locally, everything looks fine, even though there's no global solution.
- Advanced: We could generalize arc consistency to fix this problem. Instead of looking at every 2 variables and the factors between them, we could look at every subset of  $k$  variables, and check that there's a way to consistently assign values to all  $k$ , taking into account all the factors involving those  $k$  variables. However, there is a substantial cost to doing this (the running time is exponential in  $k$  in the worst case), so generally arc consistency ( $k = 2$ ) is good enough.



# Summary

- Basic template: backtracking search on partial assignments
- Dynamic ordering: most constrained variable (fail early), least constrained value (try to succeed)
- Lookahead: forward checking (enforces arc consistency on neighbors), AC-3 (enforces arc consistency on neighbors and their neighbors, etc.)



# Roadmap

Factor graphs

Dynamic ordering

Arc consistency

**Modeling**



## Example: LSAT question

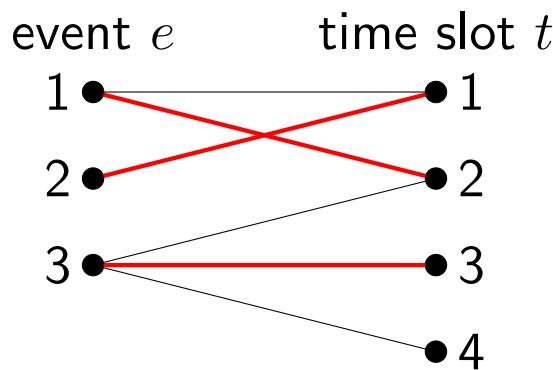
Three sculptures (A, B, C) are to be exhibited in rooms 1, 2 of an art gallery.

The exhibition must satisfy the following conditions:

- Sculptures A and B cannot be in the same room.
- Sculptures B and C must be in the same room.
- Room 2 can only hold one sculpture.

[demo]

# Example: event scheduling (section)

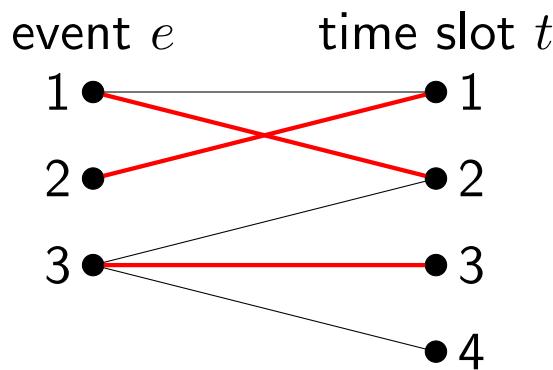


Setup:

- Have  $E$  events and  $T$  time slots
- Each event  $e$  must be put in **exactly one** time slot
- Each time slot  $t$  can have **at most one** event
- Event  $e$  allowed in time slot  $t$  only if  $(e, t) \in A$

- Consider a simple scheduling problem, where we have  $E$  events that we want to schedule into  $T$  time slots. There are three families of requirements: (i) every event must be scheduled into a time slot; (ii) every time slot can have at most one event (zero is possible); and (iii) we are given a fixed set  $A$  of (event, time slot) pairs which are allowed.
- There are in general multiple ways to cast a problem as a CSP, and the purpose of this example is to show two reasonable ways to do it.

# Example: event scheduling (section)

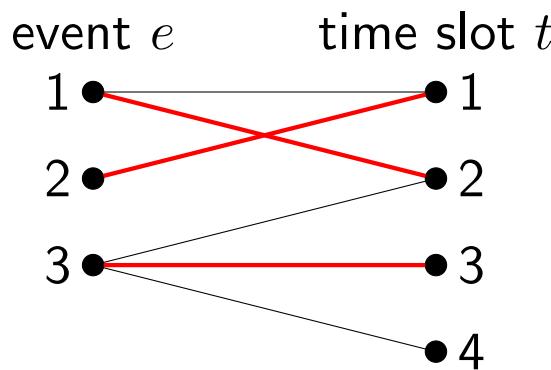


CSP formulation 1:

- Variables: for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only scheduled allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$

- The first formulation is perhaps the more natural one. We make a variable  $X_e$  for each event, whose value will be the time slot that the event is scheduled into. Since each variable can only take on one value, we automatically satisfy the requirement that every event must be put in exactly one time slot.
- However, we need to make sure no two events end up in the same time slot. To do this, we can create a binary constraint between every pair of distinct event variables  $X_e$  and  $X'_e$  that enforces their values to be different ( $X_e \neq X_{e'}$ ).
- Finally, to deal with the requirement that an event is scheduled only in allowed time slots, we just need to add a unary constraint for each variable saying that the time slot  $X_e$  that's chosen for that event is allowed.
- Note that we end up with  $E$  variables with domain size  $T$ , and  $O(E^2)$  binary constraints.

# Example: event scheduling (section)

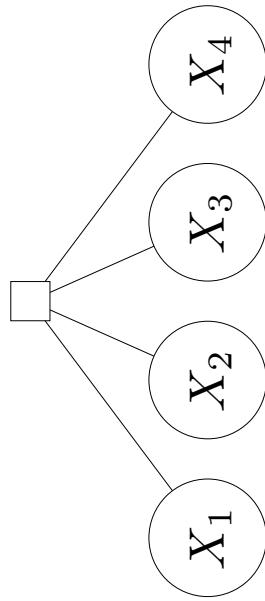


CSP formulation 2:

- Variables: for each time slot  $t$ ,  $Y_t \in \{1, \dots, E\} \cup \{\emptyset\}$
- Constraints (each event is scheduled exactly once): for each event  $e$ , enforce  $[Y_t = e \text{ for exactly one } t]$
- Constraints (only schedule allowed times): for each time slot  $t$ , enforce  $[Y_t = \emptyset \text{ or } (Y_t, t) \in A]$

- Alternatively, we can take the perspective of the time slots and ask which event was scheduled in each time slot. So we introduce a variable  $Y_t$  for each time slot  $t$  which takes on a value equal to one of the events or none ( $\emptyset$ ).
- Unlike the first formulation, we don't get for free the requirement that each event is put in exactly one time slot. To add it, we introduce  $E$  constraints, one for each event. Each constraint needs to depend on all  $T$  variables and check that the number of time slots  $t$  which have event  $e$  assigned to that slot ( $Y_t = e$ ) is exactly 1.
- On the other hand, the requirement that each time slot has at most one event assigned to it we get for free, since each variable takes on exactly one value.
- Finally, we add  $T$  constraints, one for each time slot  $t$  enforcing that if there was an event scheduled there ( $Y_t \neq \emptyset$ ), then it better be allowed according to  $A$ .
- With this formulation, we have  $T$  variables with domain size  $E+1$ , and  $E$   $T$ -ary constraints. We will show shortly that each  $T$ -ary constraints can be converted into  $O(T)$  binary constraints with  $O(T)$  variables. Therefore, the resulting formulation has  $T$  variables with domain size  $E+1$ ,  $O(T^2)$  variables with domain size 2 and  $O(T^2)$  binary constraints.
- Which one is better? Since  $T \gg E$  is required for the existence of a consistent solution, the first formulation is better. If the problem were modified so that not all events had to be scheduled and  $T \ll E$ , then the second formulation would be better.

## N-ary constraints (section)



**Variables:**  $X_1, X_2, X_3, X_4 \in \{0, 1\}$

**Factor:**  $[X_1 \vee X_2 \vee X_3 \vee X_4]$

**Examples:**

$$\text{Weight}(\{X_1 : \mathbf{0}, X_2 : \mathbf{0}, X_3 : \mathbf{0}, X_4 : \mathbf{0}\}) = 0$$

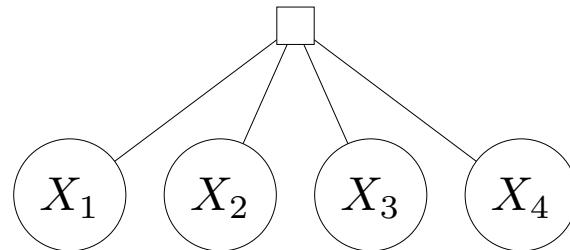
$$\text{Weight}(\{X_1 : \mathbf{0}, X_2 : \mathbf{1}, X_3 : \mathbf{0}, X_4 : \mathbf{0}\}) = 1$$

$$\text{Weight}(\{X_1 : \mathbf{0}, X_2 : \mathbf{1}, X_3 : \mathbf{1}, X_4 : \mathbf{1}\}) = 1$$

**What if inference only take unary/binary factors?**

- Consider the simple problem: given  $n$  variables  $X_1, \dots, X_n$ , where each  $X_i \in \{0, 1\}$ , impose the requirement that at least one  $X_i = 1$ . The case of  $n = 4$  is shown in the slide.

# N-ary constraints: attempt 1 (section)



**Key idea: auxiliary variable**

Auxiliary variables hold intermediate computation.

Factors:

Initialization:  $[A_0 = 0]$

$i \quad 0 \ 1 \ 2 \ 3 \ 4$

Processing:  $[A_i = A_{i-1} \vee X_i]$

$X_i: \quad 0 \ 1 \ 0 \ 1$

Final output:  $[A_4 = 1]$

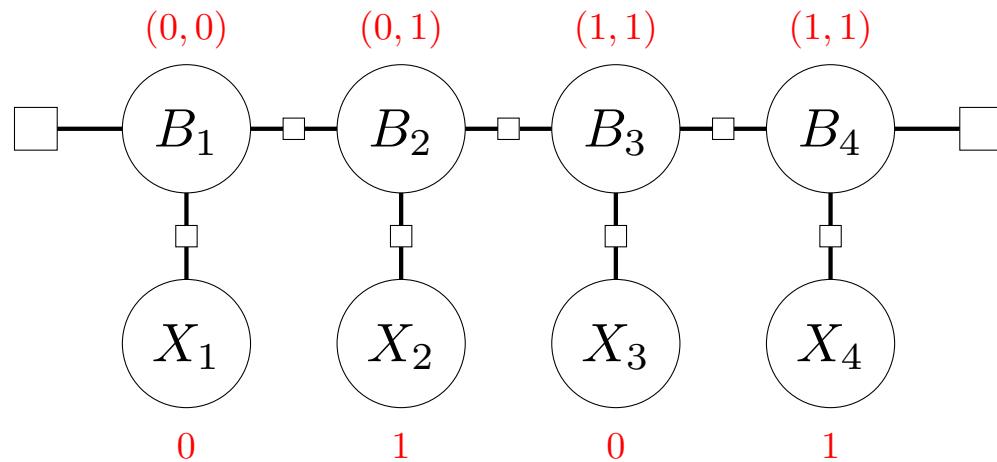
$A_i: \quad 0 \ 0 \ 1 \ 1 \ 1$

Still have factors involving 3 variables...

- The key idea is to break down the computation of the  $n$ -ary constraint into  $n$  simple steps. As a first attempt, let's introduce an auxiliary variable  $A_i$  for  $i = 1, \dots, n$  which represents the OR of variables  $X_1, \dots, X_i$ . Then we can write a simple recurrence that updates  $A_i$  with  $A_{i-1}$ . The constraint  $[A_n = 1]$  enforces that the OR of all the variables is 1.
- It is important to note that while our intuitions are based on procedurally computing  $A_i$ 's, one after the other, these computations are actually represented declaratively as constraints in the CSP.
- We have eliminated the massive  $n$ -ary constraint with ternary constraints (depending on  $A_i, A_{i-1}, X_i$ ). Can we replace the ternary constraint with unary and binary constraints?

# N-ary constraints: attempt 2 (section)

Key idea: pack  $A_{i-1}$  and  $A_i$  into one variable  $B_i$



Variables:  $B_i$  is (pre, post) pair from processing  $X_i$

Factors:

Initialization:  $[B_1[1] = 0]$

Processing:  $[B_i[2] = B_i[1] \vee X_i]$

Final output:  $[B_4[2] = 1]$

Consistency:  $[B_{i-1}[2] = B_i[1]]$

- The key idea to turn the ternary constraint  $[A_i = A_{i-1} \vee X_i]$  into a binary constraint is to merge  $A_{i-1}$  and  $A_i$  into one variable, represented as one variable  $B_i$ . The variable  $B_i$  will represent a pair of booleans, where  $B_i[1]$  represents  $A_{i-1}$  and  $B_i[2]$  represents  $A_i$ .
- Now, the ternary constraint is just a binary constraint:  $[B_i[2] = B_i[1] \vee X_i]!$
- However, note that  $A_{i-1}$  is represented twice, both in  $B_i$  and  $B_{i-1}$ . So we need to add another binary constraint to enforce that the two are equal:  $[B_{i-1}[2] = B_i[1]]$ .
- The initialization and final output factors are the same as before.

# Example: relation extraction

Motivation: build a question-answering system

*Which US presidents played the guitar?*

Prerequisite: learn knowledge by reading the web



Systems:

[NELL (CMU)]

[OpenIE (UW)]

- Now let's look at a different problem. Some background which is unrelated to CSPs: A major area of research in natural language processing is **relation extraction**, the task of building systems that can process the enormous amount of unstructured text on the web, and populate a structured knowledge base, so that we can answer complex questions by querying the knowledge base.

# Example: relation extraction

Input (hundreds of millions of web pages):

*Barack Obama is the 44th and current President of the United States...*

Output (database of relations):

EmployedBy(BarackObama, UnitedStates)

Profession(BarackObama, President)

...

# Example: relation extraction

Typical predictions of classifiers:

BornIn(BarackObama,UnitedStates) 0.9

BornIn(BarackObama,Kenya) 0.6

BornIn(JohnLennon,guitar) 0.7

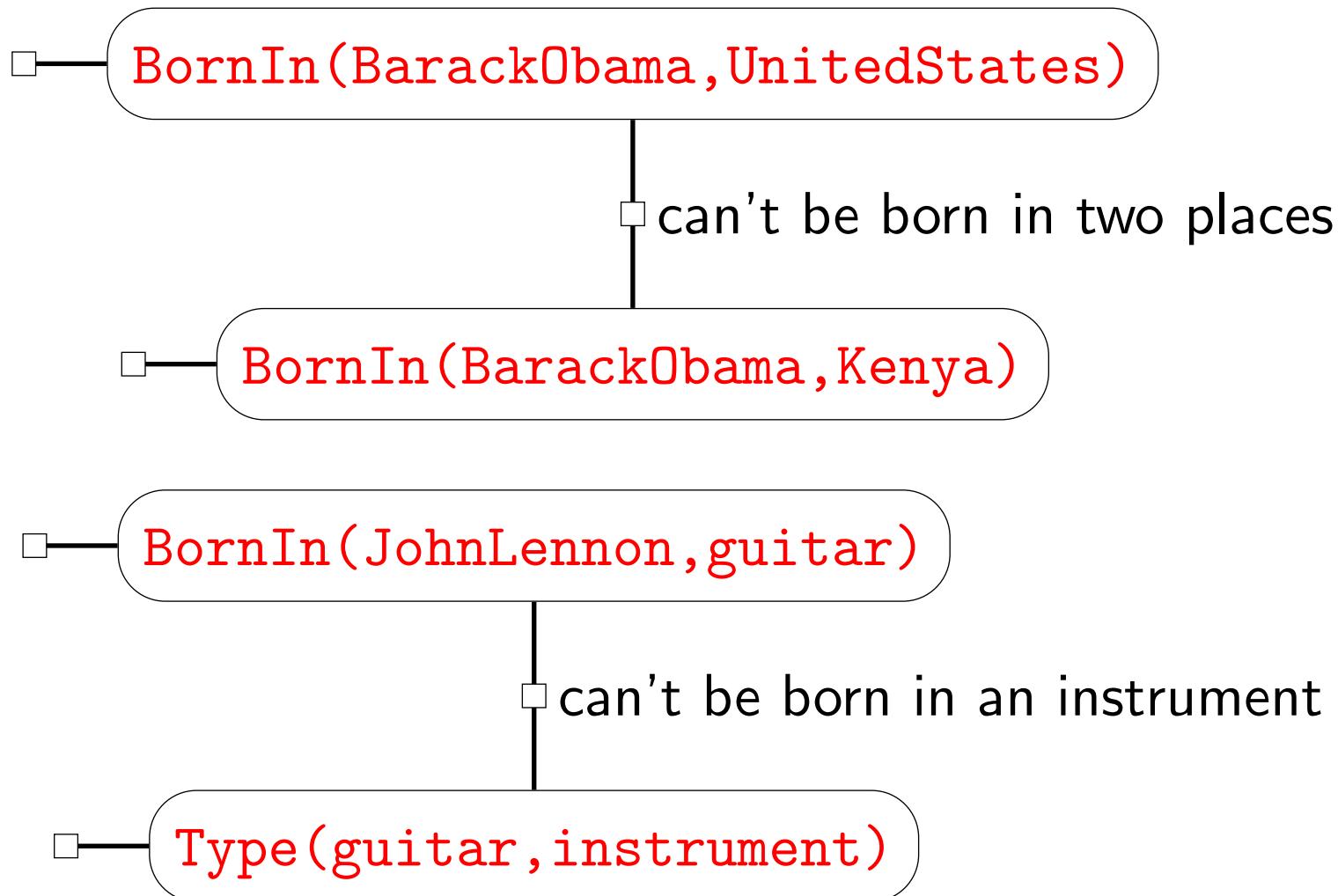
Type(guitar,instrument) 0.9

...

How do reconcile conflicting predictions?

- State-of-the-art methods typically use machine learning, casting it as a classification problem. However, relation extraction is a very difficult problem, and even the best systems today often fail, producing nonsensical facts.
- A key observation is that these classification decisions are not independent, and we have some prior knowledge on how they should be related. For example, you can't be born in two places, and you also can't be born in an instrument (not usually, anyway).

# Example: relation extraction



# General framework

Classification decisions are generally related:

$$Y_1$$

$$Y_2$$

$$Y_3$$

$$Y_4$$

- **Unary factors:** local classifiers (provide evidence)

$$\exp(\mathbf{w} \cdot \phi(x_i) Y_i)$$

- **Binary factors:** enforce that outputs are consistent

$$[Y_i \text{ consistent with } Y_{i'}]$$

- To operationalize this intuition, we can leverage factor graphs. Think about each of the classification decisions as a variable, which can take on 1 or 0 (assume binary classification for now).
- We have a unary factor which specifies the contribution of the classifier. Recall that linear classifiers return a score  $\mathbf{w} \cdot \phi(x_i)$ , which is a real number. Factors must be non-negative, so it's typical to exponentiate the score.
- We can add binary factors between pairs of classification decisions which are related in some way (e.g.,  $[\text{BornIn}(\text{BarackObama}, \text{UnitedStates}) + \text{BornIn}(\text{BarackObama}, \text{Kenya}) \leq 1]$ ). The factors do not have to be hard constraints, but rather general preferences that encode soft preferences (e.g., returning weight 0.01 instead of 0).
- Once we have a CSP, we can ask for the maximum weight assignment, which takes into account all the information available and reasons about it globally.



# Summary

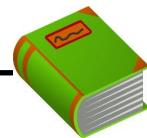
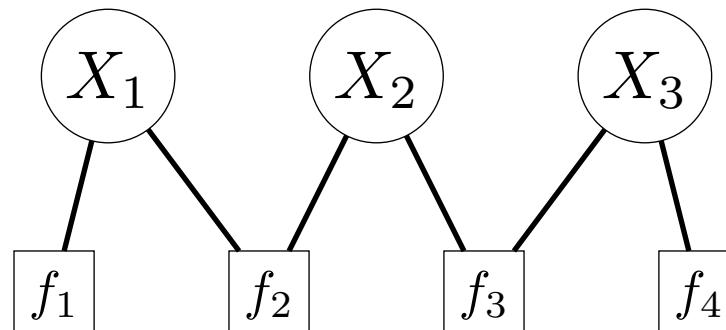
- Factor graphs: modeling framework (variables, factors)
- Key property: ordering decisions pushed to algorithms
- Algorithms: backtracking search + dynamic ordering + lookahead
- Modeling: lots of possibilities!



# Lecture 12: CSPs II

2		5	1	9
5		3		6
6	4			
			1 3 7	
	6		9	
5	9	3		
			4	8
8		5		2
1	7	8		4

# Review: definition



## Definition: factor graph

Variables:

$$X = (X_1, \dots, X_n), \text{ where } X_i \in \text{Domain}_i$$

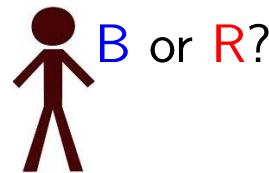
Factors:

$$f_1, \dots, f_m, \text{ with each } f_j(X) \geq 0$$

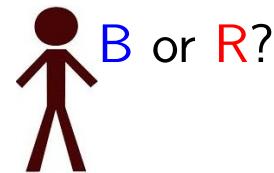
Scope of  $f_j$ : set of dependent variables

- Recall the definition of a factor graph: we have a set of variables  $X_1, \dots, X_n$  and a set of factors  $f_1, \dots, f_m$ .
- Each factor  $f_j$  is a function that takes an assignment to the variables and returns a non-negative number  $f_j(X)$  indicating how much that factor likes that assignment. A zero return value signifies a (hard) constraint that the assignment is to be avoided at all costs.
- Each factor  $f_j$  depends on only variables in its scope, which is usually a much smaller subset of the variables.
- Factor graphs are typically visualized graphically in a way that highlights the dependencies between variables and factors.

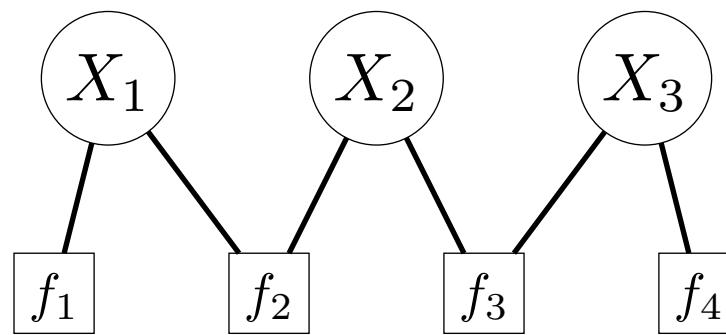
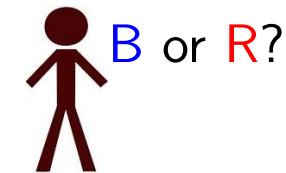
# Factor graph (example)



*must  
agree*



*tend to  
agree*



$x_1$	$f_1(x_1)$
R	0
B	1

$x_1$	$x_2$	$f_2(x_1, x_2)$
R	R	1
R	B	0
B	R	0
B	B	1

$x_2$	$x_3$	$f_3(x_2, x_3)$
R	R	3
R	B	2
B	R	2
B	B	3

$x_3$	$f_4(x_3)$
R	2
B	1

$$f_2(x_1, x_2) = [x_1 = x_2] \quad f_3(x_2, x_3) = [x_2 = x_3] + 2$$

# Review: definition



## Definition: assignment weight

Each **assignment**  $x = (x_1, \dots, x_n)$  has a **weight**:

$$\text{Weight}(x) = \prod_{j=1}^m f_j(x)$$

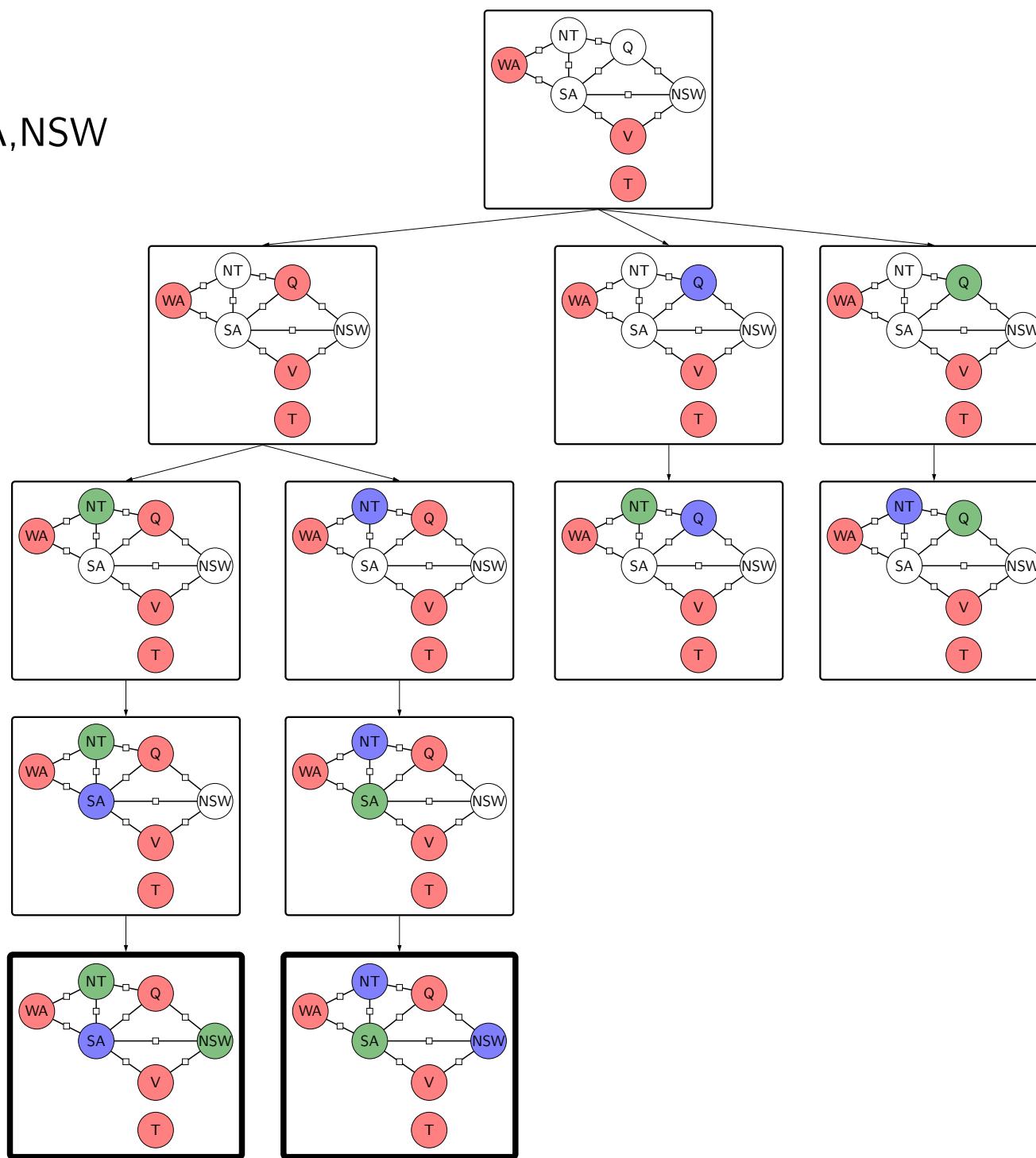
**Objective:** find a maximum weight assignment

$$\arg \max_x \text{Weight}(x)$$

- The weight of an assignment  $x$  is defined as the product of all the factors applied to  $x$ . Since it's a product, all factors have to unanimously like an assignment for the assignment to have high weight.
- Our objective is to find an assignment with the maximum weight (not to be confused with the weights in machine learning).

# Search

WA,V,T,Q,NT,SA,NSW



# Review: backtracking search

Vanilla version:

$$O(|\text{Domain}|^n) \text{ time}$$

Lookahead: forward checking, AC-3

$$O(|\text{Domain}|^n) \text{ time}$$

Dynamic ordering: most constrained variable, least constrained value

$$O(|\text{Domain}|^n) \text{ time}$$

Note: these pruning techniques useful only for constraints

- Last time, we talked about backtracking search as a way to find maximum weight assignments. In the worst case, without any further assumptions on the factor graph, this requires exponential time. We can be more clever and employ lookahead and dynamic ordering, which in some cases can dramatically improve running time, but in the worst case, it's still exponential.
- Also, these heuristics are only helpful when there are hard constraints, which allow us to prune away values in the domains of variables which definitely are not compatible with the current assignment.
- What if all the factors were strictly positive? None of the pruning techniques we encountered would be useful at all. Thus we need new techniques.

# Example: object tracking

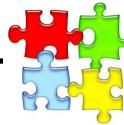
**Setup:** sensors (e.g., camera) provide noisy information about location of an object (e.g., video frames)

**Goal:** infer object's true location



- As motivation, we will consider object (e.g., person) tracking, an important task in computer vision.
- Here, at each discrete time step  $i$ , we are given some noisy information about where the object might be. For example, this noisy information could be the video frame at time step  $i$ . The goal is to answer the question: what trajectory did the object take?

# Modeling object tracking



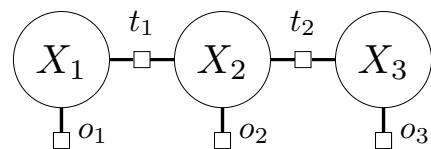
## Problem: object tracking

Noisy sensors report positions: 0, 2, 2.  
Objects don't move very fast.  
What path did the object take?

[whiteboard: trajectories over time]

# Person tracking solution

Factor graph (chain-structured):



- Variables  $X_i$ : location of object at time  $i$
- Observation factors  $o_i(x_i)$ : noisy information compatible with position
- Transition factors  $t_i(x_i, x_{i+1})$ : object positions can't change too much

[demo: create factor graph]

- Let's try to model this problem. Always start by defining the variables: these are the quantities which we don't know. In this case, it's the locations of the object at each time step:  $X_1, X_2, X_3 \in \{0, 1, 2\}$ .
- Now let's think about the factors, which need to capture two things. First, transition factors make sure physics isn't violated (e.g., object positions can't change too much). Second, observation factors make sure the hypothesized locations  $X_i$  are compatible with the noisy information. Note that these numbers are just numbers, not necessarily probabilities; later we'll see how probabilities fit in to factor graphs.
- Having modeled the problem as a factor graph, we can now ask for the maximum weight assignment for that factor graph, which would give us the most likely trajectory for the object.
- Click on the [track] demo to see the definition of this factor graph as well as the maximum weight assignment, which is [1, 2, 2]. Note that we smooth out the trajectory, assuming that the first sensor reading was inaccurate.
- Next we will develop algorithms for finding a maximum weight assignment in a factor graph. These algorithms will be overkill for solving this simple tracking problem, but it will nonetheless be useful for illustrative purposes.



# Roadmap

Beam search

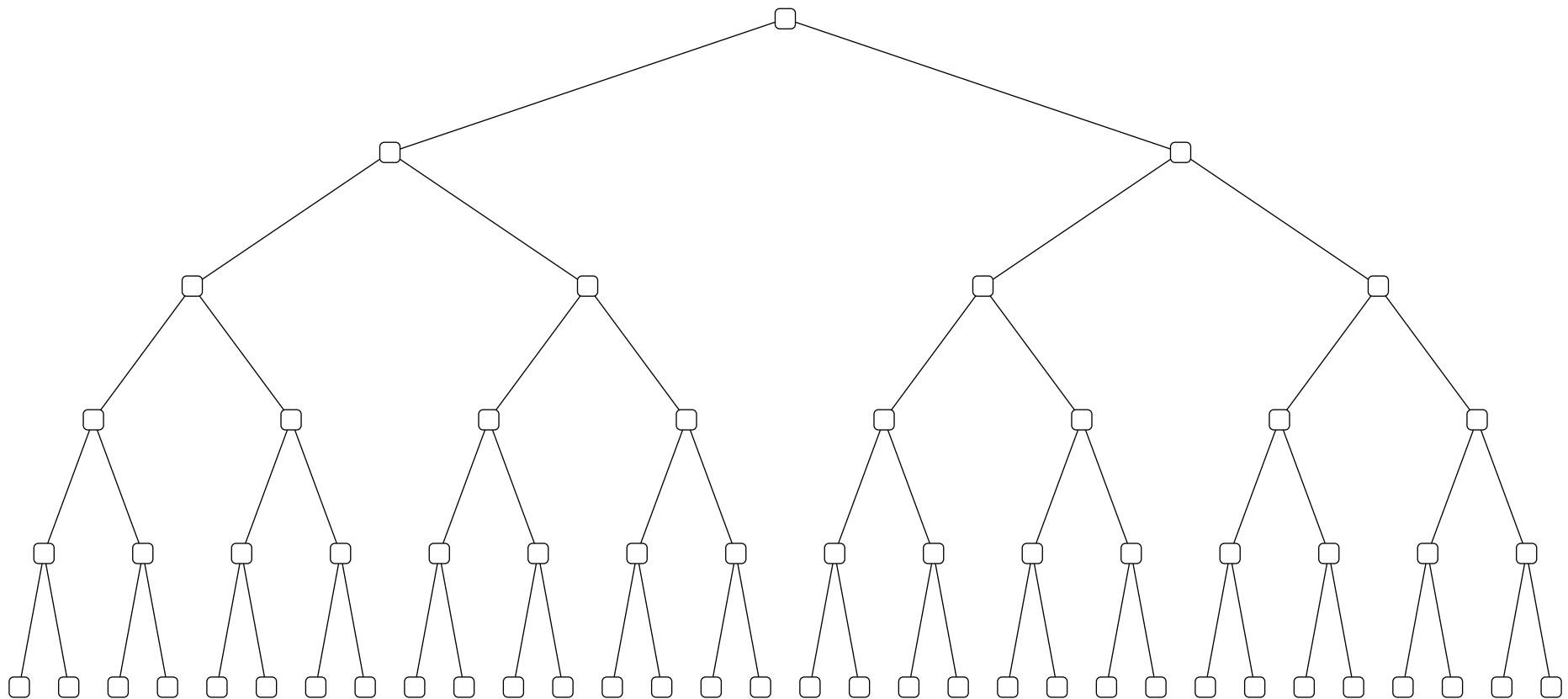
Local search

Conditioning

Elimination

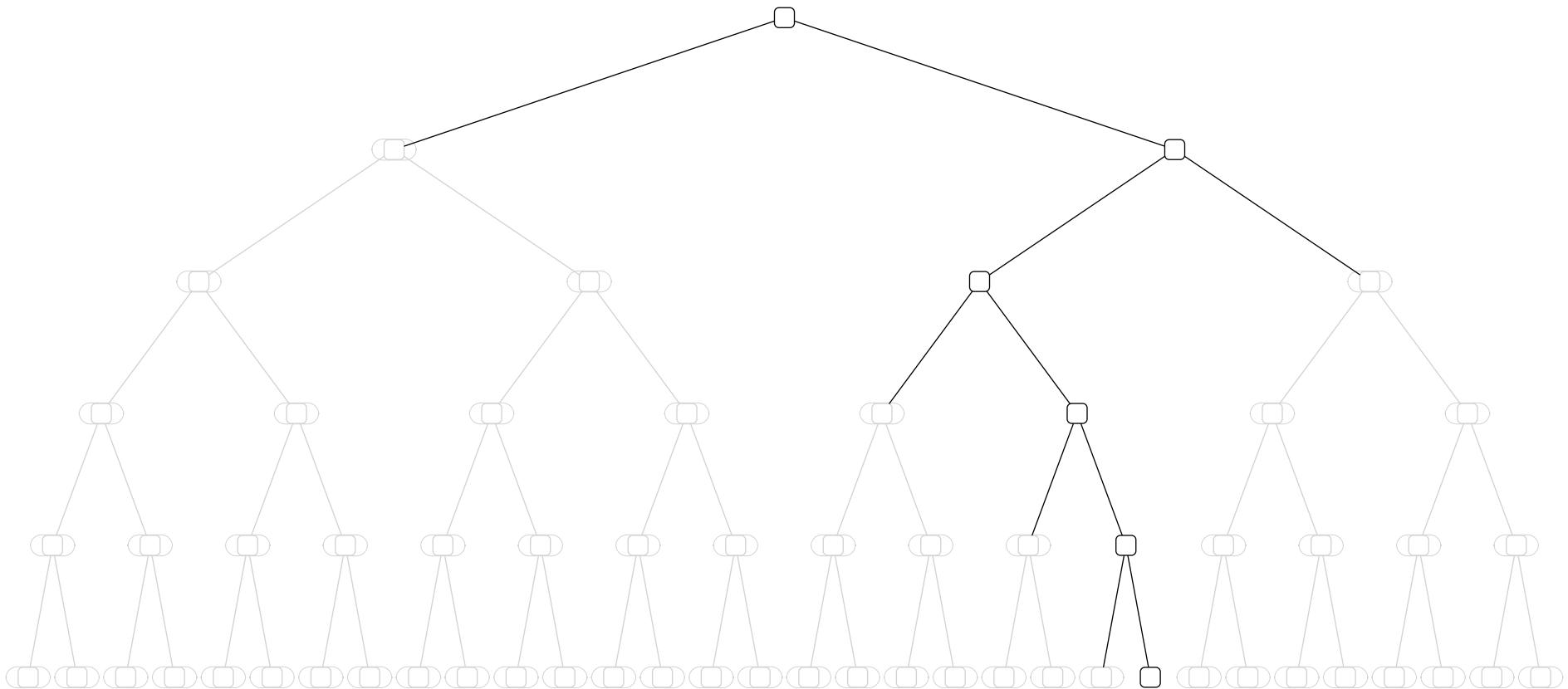
- In this lecture, we will discuss alternative ways to find maximum weight assignments efficiently without incurring the full cost of backtracking search.
- The first two methods (beam search and local search) are approximate algorithms. We give up guarantees of finding the exact maximum weight assignment, but they can still work well in practice.
- Then, we will look at the fact that we have a factor **graph**, and show that for graph structures, we can get exponentially faster algorithms, analogous to the savings we obtained by using dynamic programming in search problems. We will introduce two factor graph operations: conditioning and elimination.

# Backtracking search



- Backtracking search in the worst case performs an exhaustive DFS of the entire search tree, which can take a very very long time. How do we avoid this?

# Greedy search



- One option is to simply not backtrack! In greedy search, we're just going to stick with our guns, marching down one thin slice of the search tree, and never looking back.

# Greedy search

[demo: beamSearch({K:1})]



## Algorithm: greedy search

Partial assignment  $x \leftarrow \{\}$

For each  $i = 1, \dots, n$ :

Extend:

Compute weight of each  $x_v = x \cup \{X_i : v\}$

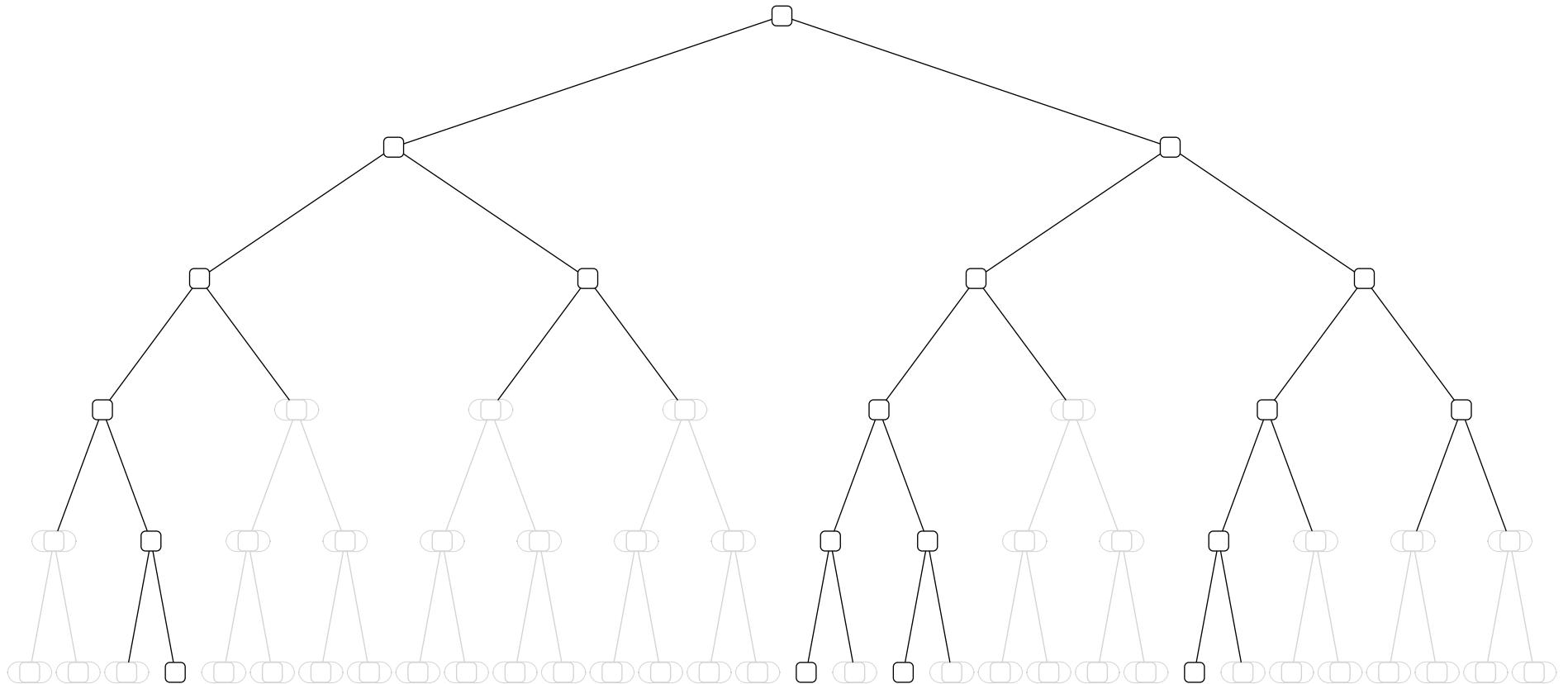
Prune:

$x \leftarrow x_v$  with highest weight

Not guaranteed to find optimal assignment!

- Specifically, we assume we have a fixed ordering of the variables. As in backtracking search, we maintain a partial assignment  $x$  and its weight, which we denote  $w(x)$ . We consider extending  $x$  to include  $X_i : v$  for all possible values  $v \in \text{Domain}_i$ . Then instead of recursing on all possible values of  $v$ , we just commit to the best one according to the weight of the new partial assignment  $x \cup \{X_i : v\}$ .
- It's important to realize that "best" here is only with respect to the weight of the partial assignment  $x \cup \{X_i : v\}$ . The greedy algorithm is by no means guaranteed to find the globally optimal solution. Nonetheless, it is incredibly fast and sometimes good enough.
- In the demo, you'll notice that greedy search produces a suboptimal solution.

# Beam search



Beam size  $K = 4$

# Beam search

[demo: beamSearch({K:3})]

Idea: keep  $\leq K$  **candidate list**  $C$  of partial assignments



## Algorithm: beam search

Initialize  $C \leftarrow [\{\}]$

For each  $i = 1, \dots, n$ :

Extend:

$$C' \leftarrow \{x \cup \{X_i : v\} : x \in C, v \in \text{Domain}_i\}$$

Prune:

$$C \leftarrow K \text{ elements of } C' \text{ with highest weights}$$

Not guaranteed to find optimal assignment!

- The problem with greedy is that it's too myopic. So a natural solution is to keep track of more than just the single best partial assignment at each level of the search tree. This is exactly **beam search**, which keeps track of (at most)  $K$  candidates ( $K$  is called the beam size). It's important to remember that these candidates are not guaranteed to be the  $K$  best at each level (otherwise greedy would be optimal).
- The beam search algorithm maintains a candidate list  $C$  and iterates through all the variables, just as in greedy. It extends each candidate partial assignment  $x \in C$  with every possible  $X_i : v$ . This produces a new candidate list  $C'$ . We sort  $C'$  by decreasing weight, and keep only the top  $K$  elements.
- Beam search also has no guarantees of finding the maximum weight assignment, but it generally works better than greedy at the cost of an increase in running time.
- In the demo, we can see that with a beam size of  $K = 3$ , we are able to find the globally optimal solution.

# Beam search properties

- Running time:  $O(n(Kb) \log(Kb))$  with branching factor  $b = |\text{Domain}|$ , beam size  $K$
- Beam size  $K$  controls tradeoff between efficiency and accuracy
  - $K = 1$  is greedy ( $O(nb)$  time)
  - $K = \infty$  is BFS tree search ( $O(b^n)$  time)
- **Analogy:** backtracking search : DFS :: BFS : beam search (pruned)

- Beam search offers a nice way to tradeoff efficiency and accuracy and is used quite commonly in practice. If you want speed and don't need extremely high accuracy, use greedy ( $K = 1$ ). The running time is  $O(nb)$ , since for each of the  $n$  variables, we need to consider  $b$  possible values in the domain.
- If you want high accuracy, then you need to pay by increasing  $K$ . For each of the  $n$  variables, we keep track of  $K$  candidates, which gets extended to  $Kb$  when forming  $C'$ . Sorting these  $Kb$  candidates by score requires  $Kb \log(Kb)$  time.
- With large enough  $K$  (no pruning), beam search is just doing a BFS traversal rather than a DFS traversal of the search tree, which takes  $O(b^n)$  time. Note that  $K$  doesn't enter in to the expression because the number of candidates is bounded by the total number, which is  $O(b^n)$ . Technically, we could write the running time of beam search as  $O(\min\{b^n, n(Kb) \log(Kb)\})$ , but for small  $K$  and large  $n$ ,  $b^n$  will be much larger, so it can be ignored.
- For moderate values of  $K$ , beam search is a kind of pruned BFS, where we use the factors that we've seen so far to decide which branches of the tree are worth exploring.
- In summary, beam search takes a broader view of the search tree, allowing us to compare partial assignments in very different parts of the tree, something that backtracking search cannot do.



# Roadmap

Beam search

**Local search**

Conditioning

Elimination

# Local search

Backtracking/beam search: extend partial assignments

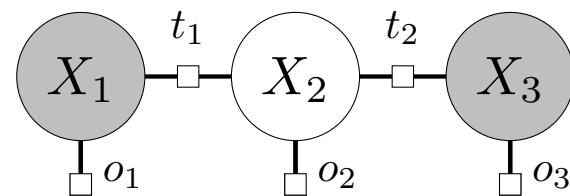


Local search: modify complete assignments



- So far, both backtracking and beam search build up a partial assignment incrementally, and are structured around an ordering of the variables (even if it's dynamically chosen). With backtracking search, we can't just go back and change the value of a variable much higher in the tree due to new information; we have to wait until the backtracking takes us back up, in which case we lose all the information about the more recent variables. With beam search, we can't even go back at all.
- Recall that one of the motivations for moving to variable-based models is that we wanted to downplay the role of ordering. **Local search** (i.e., hill climbing) provides us with additional flexibility. Instead of building up partial assignments, we work with a complete assignment and make repairs by changing one variable at a time.

# Iterated conditional modes (ICM)



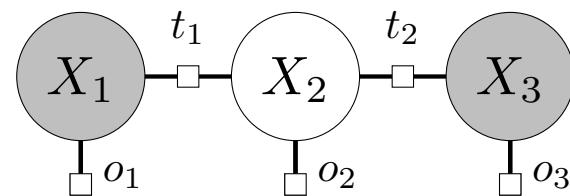
Current assignment:  $(0, 0, 1)$ ; how to improve?

$(x_1, v, x_3)$	weight
$(0, 0, 1)$	$2 \cdot 2 \cdot 0 \cdot 1 \cdot 1 = 0$
$(0, 1, 1)$	$2 \cdot 1 \cdot 1 \cdot 2 \cdot 1 = 4$
$(0, 2, 1)$	$2 \cdot 0 \cdot 2 \cdot 1 \cdot 1 = 0$

New assignment:  $(0, 1, 1)$

- Consider a complete assignment  $(0, 0, 1)$ . Can we make a local change to the assignment to improve the weight? Let's just try setting  $x_2$  to a new value  $v$ . For each possibility, we can compute the weight, and just take the highest scoring option. This results in a new assignment  $(0, 1, 1)$  with a higher weight (4 rather than 0).

# Iterated conditional modes (ICM)



Weight of new assignment  $(x_1, v, x_3)$ :

$$o_1(x_1) \color{red}{t_1(x_1, v)} o_2(v) t_2(v, x_3) o_3(x_3)$$



## Key idea: locality

When evaluating possible re-assignments to  $X_i$ , only need to consider the factors that depend on  $X_i$ .

- If we write down the weights of the various new assignments  $x \cup \{X_2 : v\}$ , we will notice that all the factors return the same value except the ones that depend on  $X_2$ .
- Therefore, we only need to compute the product of these relevant factors and take the maximum value. Because we only need to look at the factors that touch the variable we're modifying, this can be a big saving if the total number of factors is much larger.

# Iterated conditional modes (ICM)



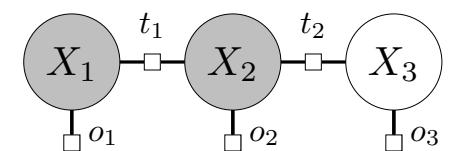
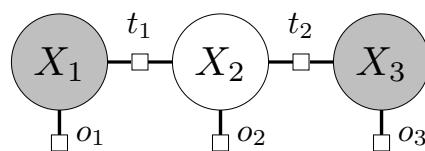
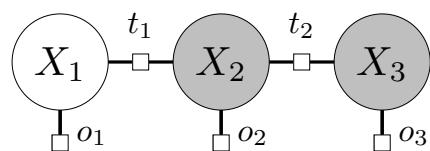
## Algorithm: iterated conditional modes (ICM)

Initialize  $x$  to a random complete assignment

Loop through  $i = 1, \dots, n$  until convergence:

    Compute weight of  $x_v = x \cup \{X_i : v\}$  for each  $v$

$x \leftarrow x_v$  with highest weight

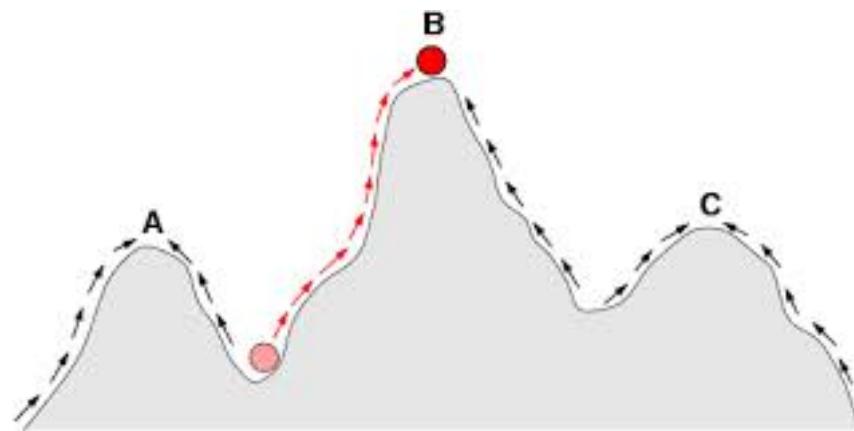


- Now we can state our first algorithm, ICM, which is the local search analogy of the greedy algorithm we described earlier. The idea is simple: we start with a random complete assignment. We repeatedly loop through all the variables  $X_i$ . On variable  $X_i$ , we consider all possible ways of re-assigning it  $X_i : v$  for  $v \in \text{Domain}_i$ , and choose the new assignment that has the highest weight.
- Graphically, we represent each step of the algorithm by having shaded nodes for the variables which are fixed and unshaded for the single variable which is being re-assigned.

# Iterated conditional modes (ICM)

[demo: `iteratedConditionalModes()`]

- $\text{Weight}(x)$  increases or stays the same each iteration
- Converges in a finite number of iterations
- Can get stuck in **local optima**
- Not guaranteed to find optimal assignment!



- Note that ICM will increase the weight of the assignments monotonically and converges, but it will get stuck in local optima, where there is a better assignment elsewhere, but all the one variable changes result in a lower weight assignment.
- Connection: this hill-climbing is called coordinate-wise ascent. We already saw an instance of coordinate-wise ascent in the K-means algorithm which would alternate between fixing the centroids and optimizing the object with respect to the cluster assignments, and fixing the cluster assignments and optimizing the centroids. Recall that K-means also suffered from local optima issues.
- Connection: these local optima are an example of a Nash equilibrium (for collaborative games), where no unilateral change can improve utility.
- Note that in the demo, ICM gets stuck in a local optimum with weight 4 rather than the global optimum's 8.

# Gibbs sampling

Sometimes, need to go downhill to go uphill...



**Key idea: randomness**

Sample an assignment with probability proportional to its weight.



**Example: Gibbs sampling**

Weight( $x \cup \{X_2 : 0\}$ ) = 1 prob. 0.2

Weight( $x \cup \{X_2 : 1\}$ ) = 2 prob. 0.4

Weight( $x \cup \{X_2 : 2\}$ ) = 2 prob. 0.4



- In reinforcement learning, we also had a problem where if we explore by using a greedy policy (always choosing the best action according to our current estimate of the  $Q$  function), then we were doomed to get stuck. There, we used **randomness** via epsilon-greedy to get out of local optima.
- Here, we will do the same, but using a slightly more sophisticated form of randomness. The idea is **Gibbs sampling**, a method originally designed for using Markov chains to sample from a distribution over assignments. We will return to that original use later, but for now, we are going to repurpose it for the problem of finding the maximum weight assignment.

# Gibbs sampling

[demo: gibbsSampling()]



## Algorithm: Gibbs sampling

Initialize  $x$  to a random complete assignment

Loop through  $i = 1, \dots, n$  until convergence:

    Compute weight of  $x_v = x \cup \{X_i : v\}$  for each  $v$

    Choose  $x \leftarrow x_v$  with probability prop. to its weight

Can escape from local optima (not always easy though)!

- The form of the algorithm is identical to ICM. The only difference is that rather than taking the assignment  $x \cup \{X_i : v\}$  with the maximum weight, we choose the assignment with probability proportional to its weight.
- In this way, even if an assignment has lower weight, we won't completely rule it out, but just choose it with lower probability. Of course if an assignment has zero weight, we will choose it with probability zero (which is to say, never).
- Randomness is not a panacea and often Gibbs sampling can get ensnarled in local optima just as much as ICM. In theory, under the assumption that we could move from the initial assignment and the maximum weight assignment with non-zero probability, Gibbs sampling will move there eventually (but it could take exponential time in the worst case).
- Advanced: just as beam search is greedy search with  $K$  candidates instead of 1, we could extend ICM and Gibbs sampling to work with more candidates. This leads us to the area of particle swarm optimization, which includes genetic algorithms, which is beyond the scope of this course.



# Question

Which of the following algorithms are guaranteed to find the maximum weight assignment (select all that apply)?

backtracking search

greedy search

beam search

Iterated Conditional Modes

Gibbs sampling



# Summary so far

Algorithms for max-weight assignments in factor graphs:

(1) Extend partial assignments:

- Backtracking search: exact, exponential time
- Beam search: approximate, linear time

(2) Modify complete assignments:

- Iterated conditional modes: approximate, deterministic
- Gibbs sampling: approximate, randomized



# Roadmap

Beam search

Local search

**Conditioning**

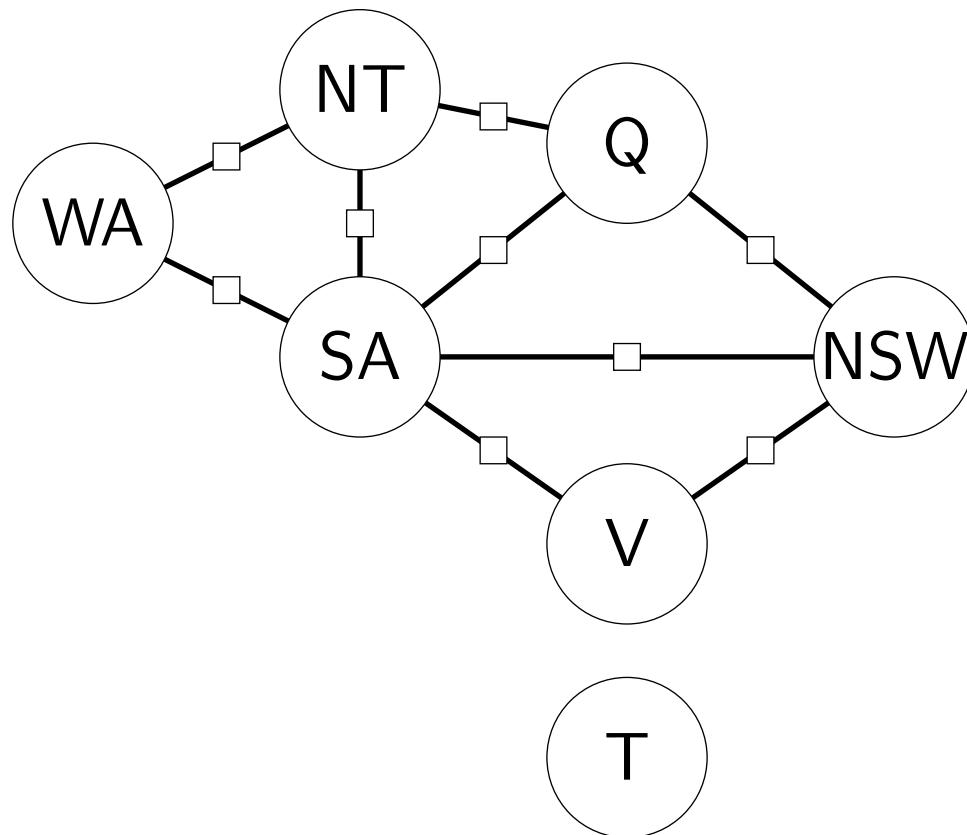
Elimination

# Motivation



**Key idea: graph**

Leverage graph properties to derive efficient algorithms which are exact.

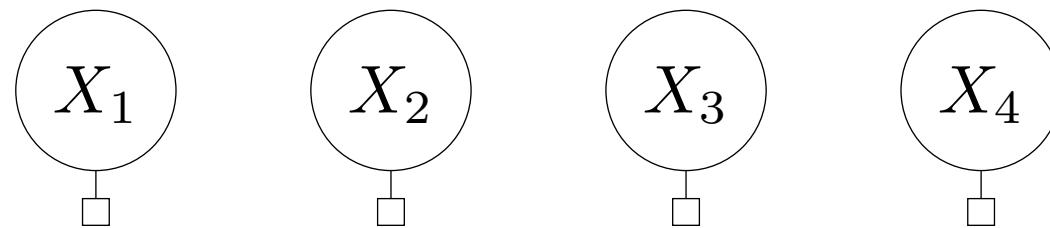


- The goal in the second part of the lecture is to take advantage of the fact that we have a factor **graph**. We will see how exploiting the graph properties can lead us to more efficient algorithms as well as a deeper understanding of the structure of our problem.

# Motivation

Backtracking search:

exponential time in number of variables  $n$



Efficient algorithm:

maximize each variable separately

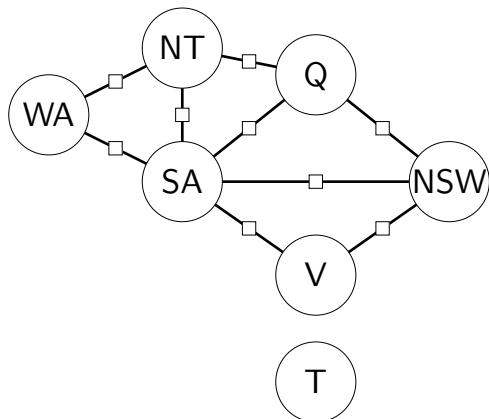
- Recall that backtracking search takes time exponential in the number of variables  $n$ . While various heuristics can have dramatic speedups in practice, it is not clear how to characterize those improvements rigorously.
- As a motivating example, consider a fully disconnected factor graph. (Imagine  $n$  people trying to vote red or blue, but they don't talk to each other.) It's clear that to get the maximum weight assignment, we can just choose the value of each variable that maximizes its own unary factor without worrying about other variables.

# Independence



## Definition: independence

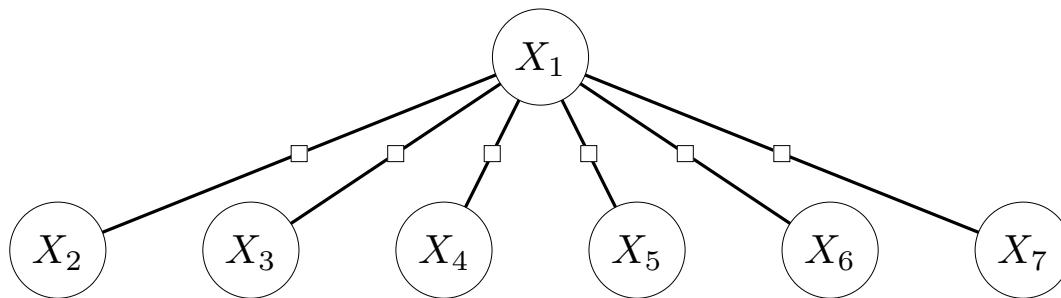
- Let  $A$  and  $B$  be a partitioning of variables  $X$ .
- We say  $A$  and  $B$  are **independent** if there are no edges between  $A$  and  $B$ .
- In symbols:  $A \perp\!\!\!\perp B$ .



$\{WA, NT, SA, Q, NSW, V\}$  and  $\{T\}$  are independent.

- Let us formalize this intuition with the notion of **independence**. It turns out that this notion of independence is deeply related to the notion of independence in probability, as we will see in due time.
- Note that we are defining independence purely in terms of the graph structure, which will be important later once we start operating on the graph using two transformations: conditioning and elimination.

# Non-independence

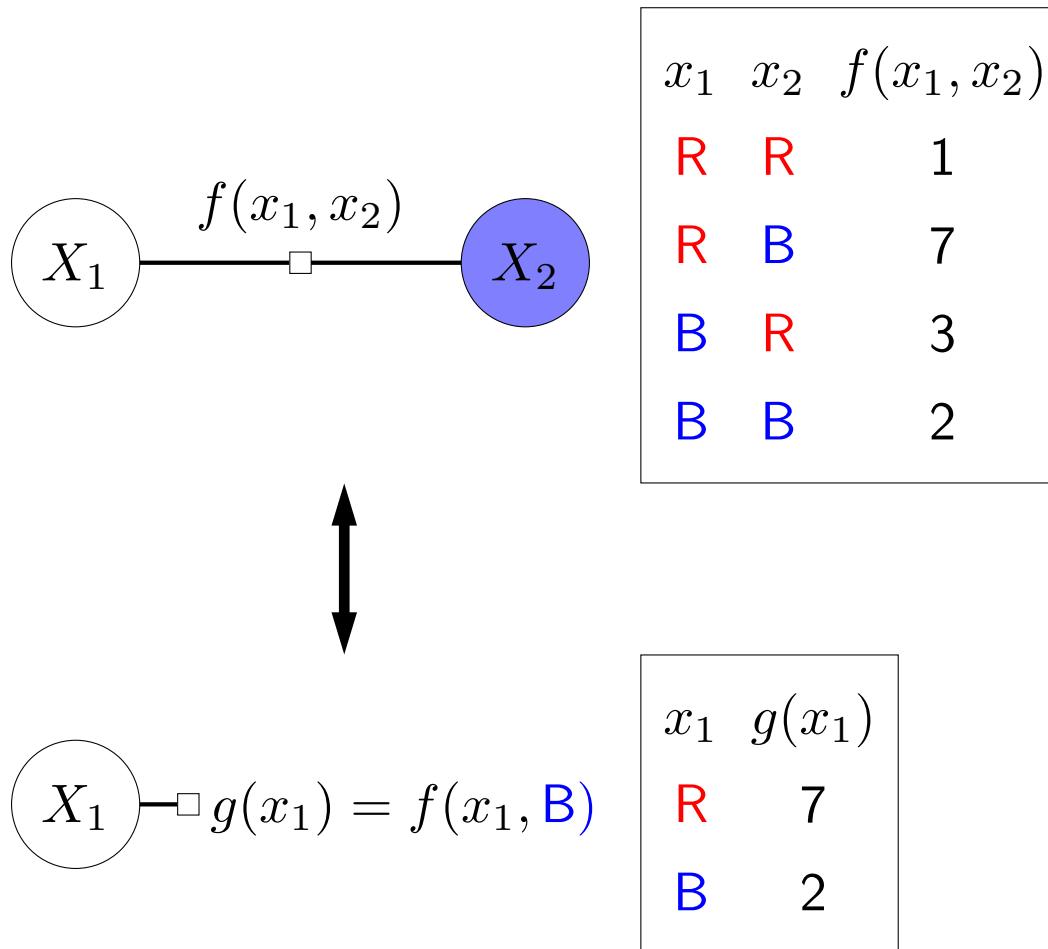


No variables are independent of each other, but feels close...

- When all the variables are independent, finding the maximum weight assignment is easily solvable in time linear in  $n$ , the number of variables. However, this is not a very interesting factor graph, because the whole point of a factor graph is to model dependencies (preferences and constraints) between variables.
- Consider the tree-structured factor graph, which corresponds to  $n - 1$  people talking only through a leader. Nothing is independent here, but intuitively, this graph should be pretty close to independent.

# Conditioning

Goal: try to disconnect the graph



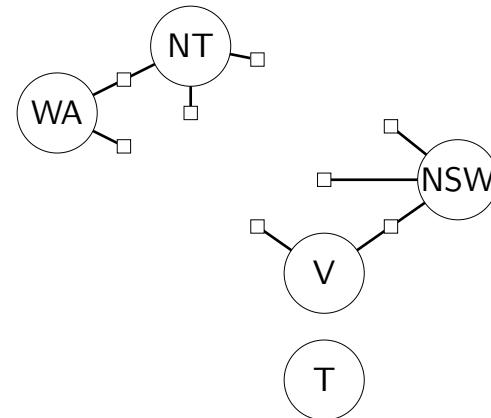
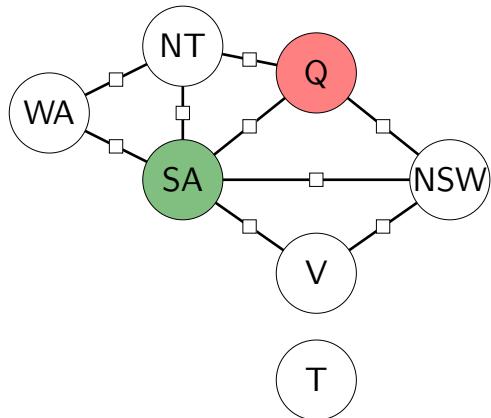
Condition on  $X_2 = \text{B}$ : remove  $X_2, f$  and add  $g$

# Conditioning: example



## Example: map coloring

Condition on  $Q = \text{R}$  and  $\text{SA} = \text{G}$ .



New factors:

$$[\text{NT} \neq \text{R}]$$

$$[\text{NSW} \neq \text{R}]$$

$$[\text{WA} \neq \text{G}]$$

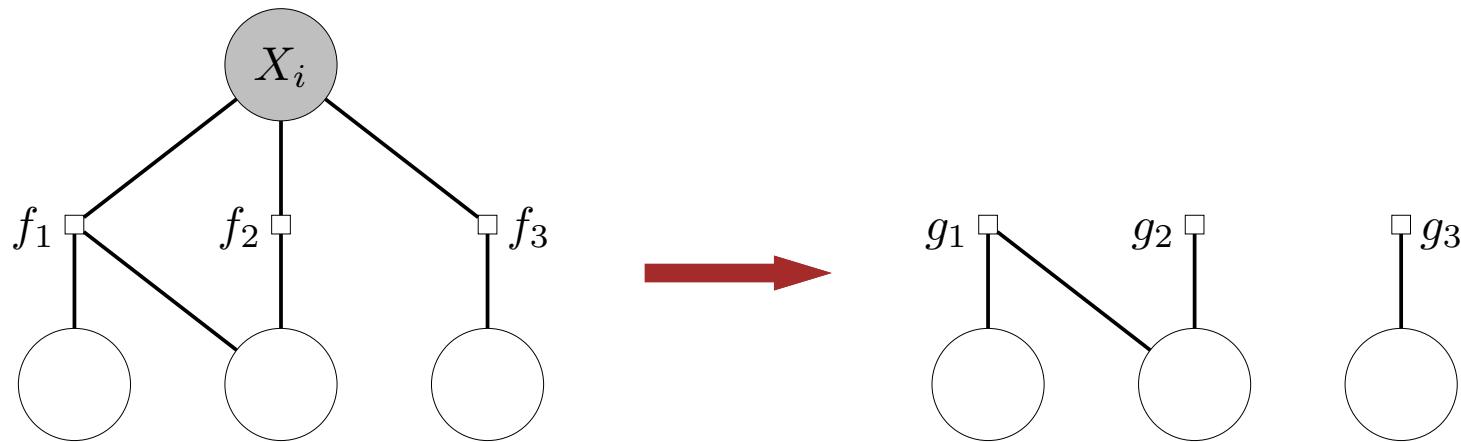
$$[\text{NT} \neq \text{G}]$$

$$[\text{NSW} \neq \text{G}]$$

$$[\text{V} \neq \text{G}]$$

# Conditioning: general

Graphically: remove edges from  $X_i$  to dependent factors



## Definition: conditioning

- To **condition** on a variable  $X_i = v$ , consider all factors  $f_1, \dots, f_k$  that depend on  $X_i$ .
- Remove  $X_i$  and  $f_1, \dots, f_k$ .
- Add  $g_j(x) = f_j(x \cup \{X_i : v\})$  for  $j = 1, \dots, k$ .

- In general, factor graphs are not going to have many partitions which are independent (we got lucky with Tasmania, Australia). But perhaps we can transform the graph to make variables independent. This is the idea of **conditioning**: when we condition on a variable  $X_i = v$ , this is simply saying that we're just going to clamp the value of  $X_i$  to  $v$ .
- We can understand conditioning in terms of a graph transformation. For each factor  $f_j$  that depends on  $X_i$ , we create a new factor  $g_j$ . The new factor depends on the scope of  $f_j$  excluding  $X_i$ ; when called on  $x$ , it just invokes  $f_j$  with  $x \cup \{X_i : v\}$ . Think of  $g_j$  as a partial evaluation of  $f_j$  in functional programming. The transformed factor graph will have each  $g_j$  in place of the  $f_j$  and also not have  $X_i$ .

# Conditional independence



## Definition: conditional independence

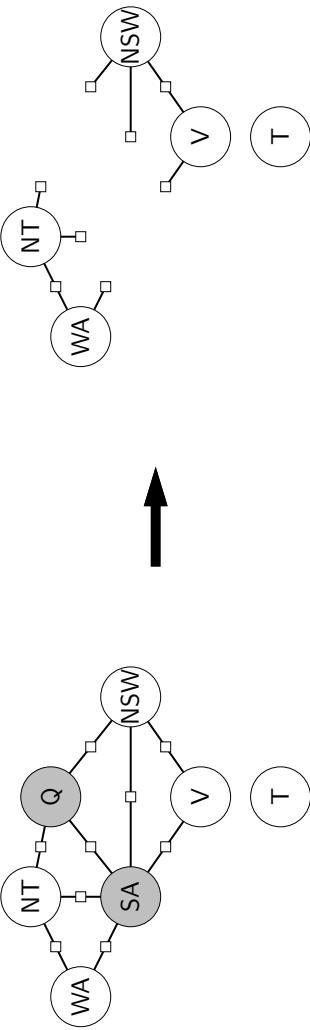
- Let  $A, B, C$  be a partitioning of the variables.
- We say  $A$  and  $B$  are **conditionally independent** given  $C$  if conditioning on  $C$  produces a graph in which  $A$  and  $B$  are independent.
- In symbols:  $A \perp\!\!\!\perp B \mid C$ .

Equivalently: every path from  $A$  to  $B$  goes through  $C$ .

## Conditional independence



Example: map coloring



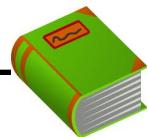
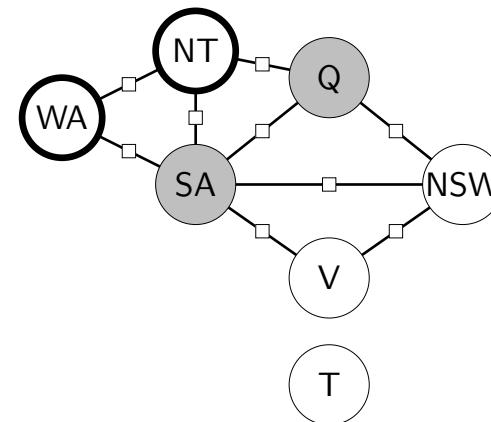
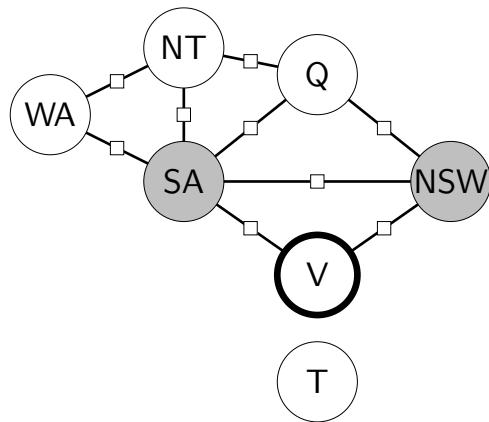
Conditional independence assertion:

$$\{WA, NT\} \perp\!\!\!\perp \{V, NSW, T\} \mid \{SA, Q\}$$

- With conditioning in hand, we can define **conditional independence**, perhaps the most important property in factor graphs.
- Graphically, if we can find a subset of the variables  $C \subset X$  that disconnects the rest of the variables into  $A$  and  $B$ , then we say that  $A$  and  $B$  are conditionally independent given  $C$ .
- Later, we'll see how this definition relates to the definition of conditional independence in probability.

# Markov blanket

How can we separate an arbitrary set of nodes from everything else?

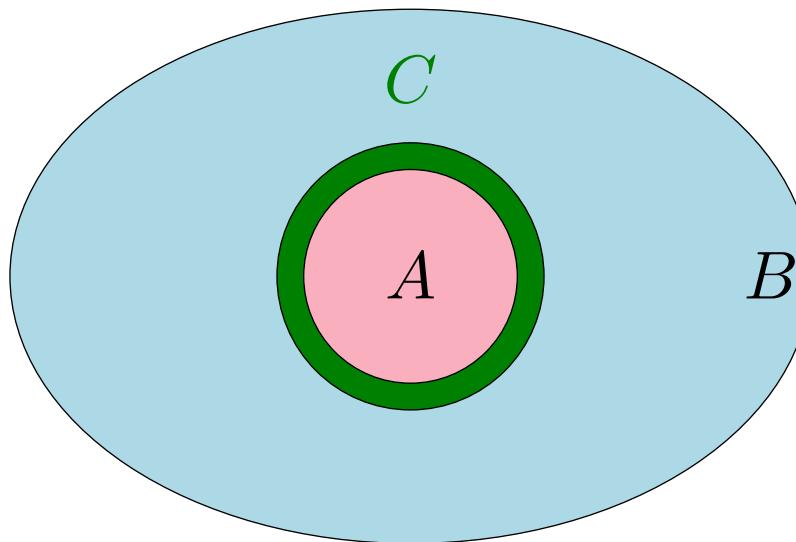


## Definition: Markov blanket

Let  $A \subseteq X$  be a subset of variables.

Define  $\text{MarkovBlanket}(A)$  be the neighbors of  $A$  that are not in  $A$ .

# Markov blanket



## Proposition: conditional independence

Let  $C = \text{MarkovBlanket}(A)$ .

Let  $B$  be  $X \setminus (A \cup C)$ .

Then  $A \perp\!\!\!\perp B \mid C$ .

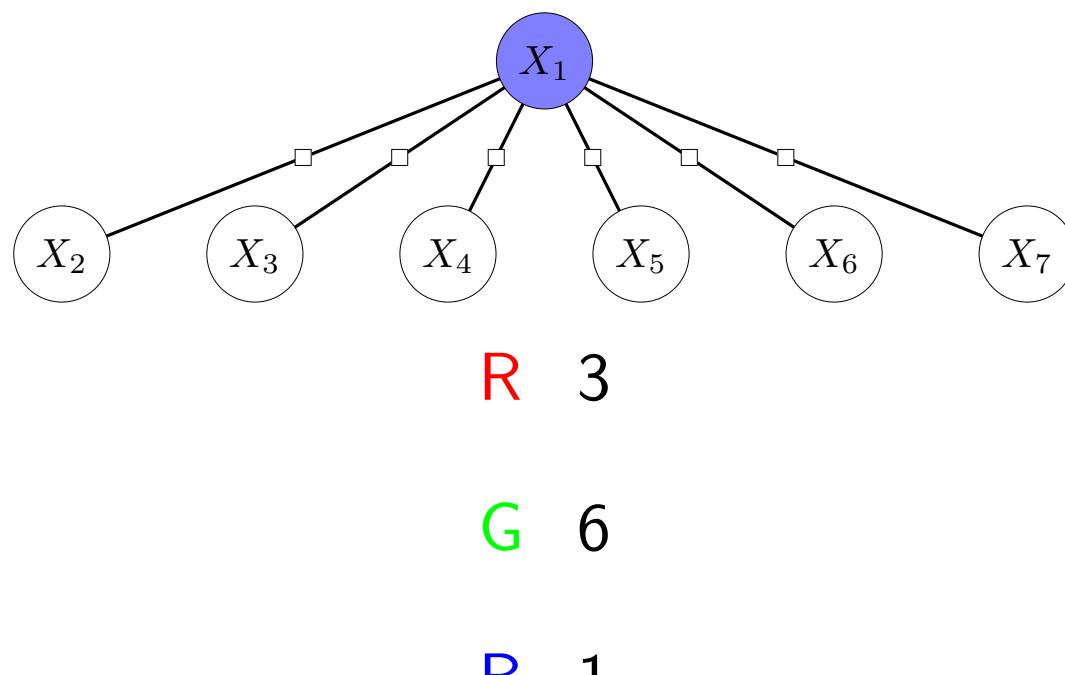
- Suppose we wanted to disconnect a subset of variables  $A \subset X$  from the rest of the graph. What is the smallest set of variables  $C$  that we need to condition on to make  $A$  and the rest of the graph ( $B = X \setminus (A \cup C)$ ) conditionally independent.
- It's intuitive that the answer is simply all the neighbors of  $A$  (those that share a common factor) which are not in  $A$ . This concept is useful enough that it has a special name: **Markov blanket**.
- Intuitively, the smaller the Markov blanket, the easier the factor graph is to deal with.

# Using conditional independence

For each value  $v = \text{R}, \text{G}, \text{B}$ :

Condition on  $X_1 = v$ .

Find the maximum weight assignment (easy).



maximum weight is 6

- Now that we understand conditional independence, how is it useful?
- First, this formalizes the fact that if someone tells you the value of a variable, you can condition on that variable, thus potentially breaking down the problem into simpler pieces.
- If we are not told the value of a variable, we can simply try to condition on all possible values of that variable, and solve the remaining problem using any method. If conditioning breaks up the factor graph into small pieces, then solving the problem becomes easier.
- In this example, conditioning on  $X_1 = v$  results in a fully disconnected graph, the maximum weight assignment for which can be computed in time linear in the number of variables.



# Summary so far

**Independence:** when sets of variables  $A$  and  $B$  are disconnected; can solve separately.

**Conditioning:** assign variable to value, replaces binary factors with unary factors

**Conditional independence:** when  $C$  blocks paths between  $A$  and  $B$

**Markov blanket:** what to condition on to make  $A$  conditionally independent of the rest.

- **Independence** is the key property that allows us to solve subproblems in parallel. It is worth noting that the savings is huge — exponential, not linear. Suppose the factor graph has two disconnected variables, each taking on  $m$  values. Then backtracking search would take  $m^2$  time, whereas solving each subproblem separately would take  $2m$  time.
- However, the factor graph isn't always disconnected (which would be uninteresting). In these cases, we can **condition** on particular values of a variable. Doing so potentially disconnects the factor graph into pieces, which can be again solved in parallel.
- Factor graphs are interesting because every variable can still influence every other variable, but finding the maximum weight assignment is efficient if there are small bottlenecks that we can condition on.



# Roadmap

Beam search

Local search

Conditioning

**Elimination**

# Conditioning versus elimination

## Conditioning:

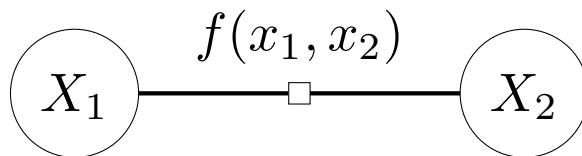
- Removes  $X_i$  from the graph
- Add factors that use fixed value of  $X_i$

## Elimination (max):

- Removes  $X_i$  from the graph
- Add factors that maximize over all values of  $X_i$

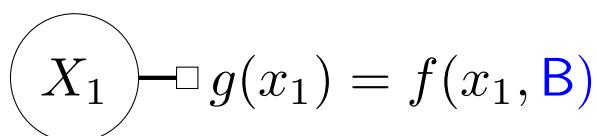
- Now we discuss the second important factor graph transformation: **elimination**. Conditioning was great at breaking the factor graph apart but required a fixed value on which to condition. If we don't know what the value should be, we just have to try all of them.
- Elimination (more precisely, max elimination) also removes variables from the graph, but actually chooses the best value for the eliminated variable  $X_i$ . But how do we talk about the best value? The answer is that we compute the best one for all possible assignments to the Markov blanket of  $X_i$ . The result of this computation can be stored as a new factor.

# Conditioning versus elimination



$x_1$	$x_2$	$f(x_1, x_2)$
R	R	1
R	B	7
B	R	3
B	B	2

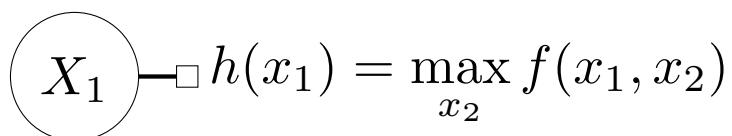
Conditioning:



consider **one** value ( $X_2 = \text{B}$ )

$x_1$	$g(x_1)$
R	7
B	2

Elimination:

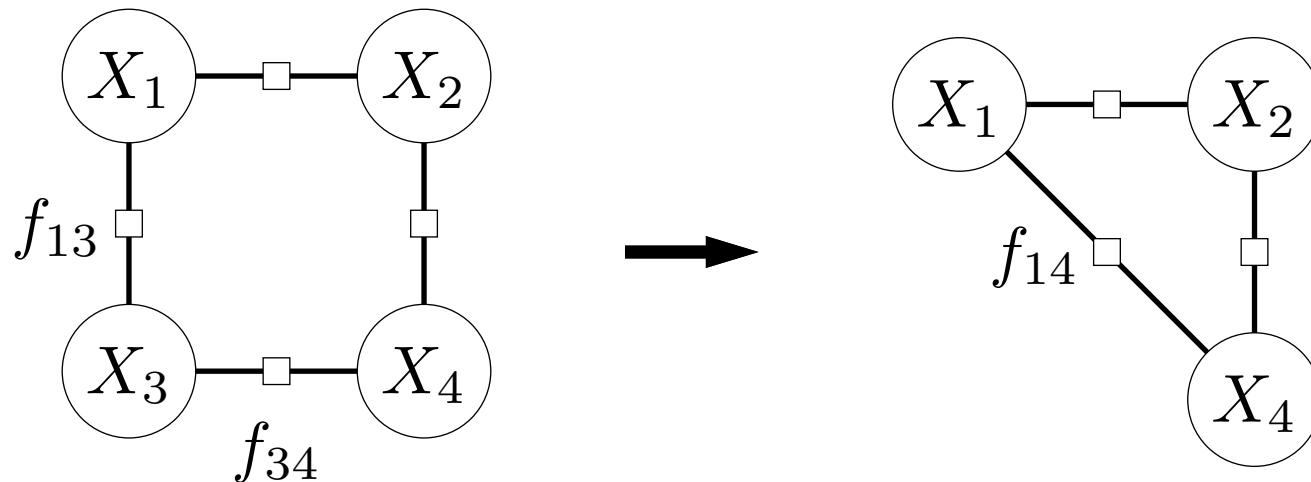


consider **all** values

$x_1$	$h(x_1)$
R	7
B	3

- If we eliminate  $X_2$  in this simple example, we produce a factor graph with the same structure as what we got for conditioning (but in general, this is not the case), but a different factor.
- In conditioning, the new factor produced  $g(x_1)$  was just  $f$  evaluated with  $x_2 = \text{B}$ . In elimination, the new factor produced  $h(x_1)$  is the maximum value of  $f$  over all possible values of  $x_2$ .

# Elimination: example



$$f_{14}(x_1, x_4) = \max_{x_3} [f_{13}(x_1, x_3) f_{34}(x_3, x_4)]$$

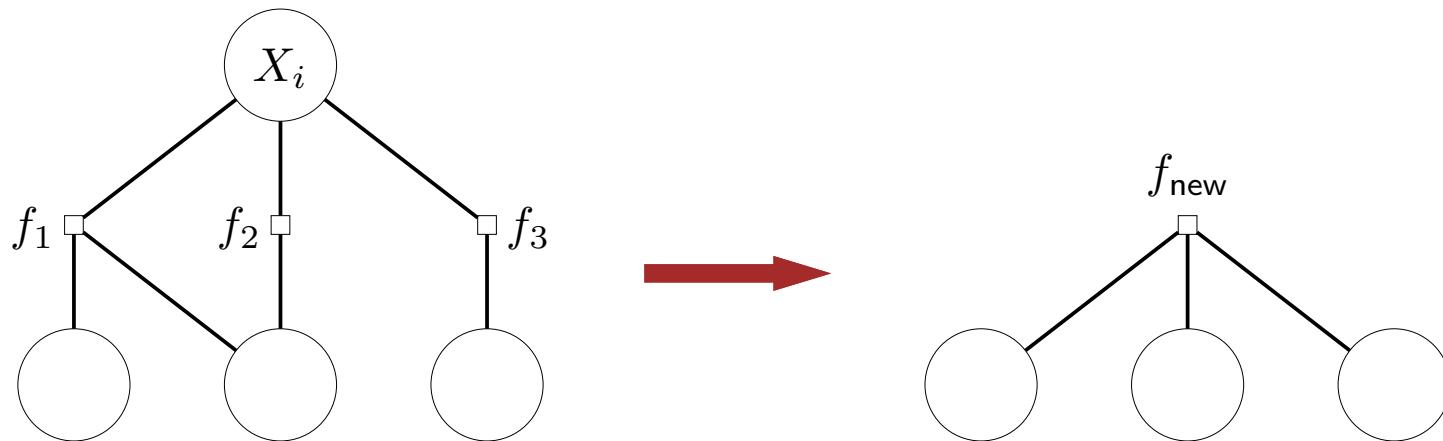
(maximum weight of assignment to  $X_3$  given  $X_1, X_4$ )

$x_1$	$x_3$	$f_{13}(x_1, x_3)$	$x_3$	$x_4$	$f_{34}(x_3, x_4)$	$x_1$	$x_4$	$f_{14}(x_1, x_4)$
R	R	4	R	R	1	R	R	$\max(4 \cdot 1, 1 \cdot 2) = 4$
R	B	1	R	B	2	R	B	$\max(4 \cdot 2, 1 \cdot 1) = 8$
B	R	1	B	R	2	B	R	$\max(1 \cdot 1, 4 \cdot 2) = 8$
B	B	4	B	B	1	B	B	$\max(1 \cdot 2, 4 \cdot 1) = 4$

$\max_{x_3}$

- Now let us look at a more complex example. Suppose we want to eliminate  $X_3$ . Now we have two factors  $f_{13}$  and  $f_{34}$  that depend on  $X_3$ .
- Again, recall that we should think of elimination as solving the maximum weight assignment problem over  $X_3$  conditioned on the Markov blanket  $\{X_1, X_4\}$ .
- The result of this computation is stored in the new factor  $f_{14}(x_1, x_4)$ , which depends on the Markov blanket.

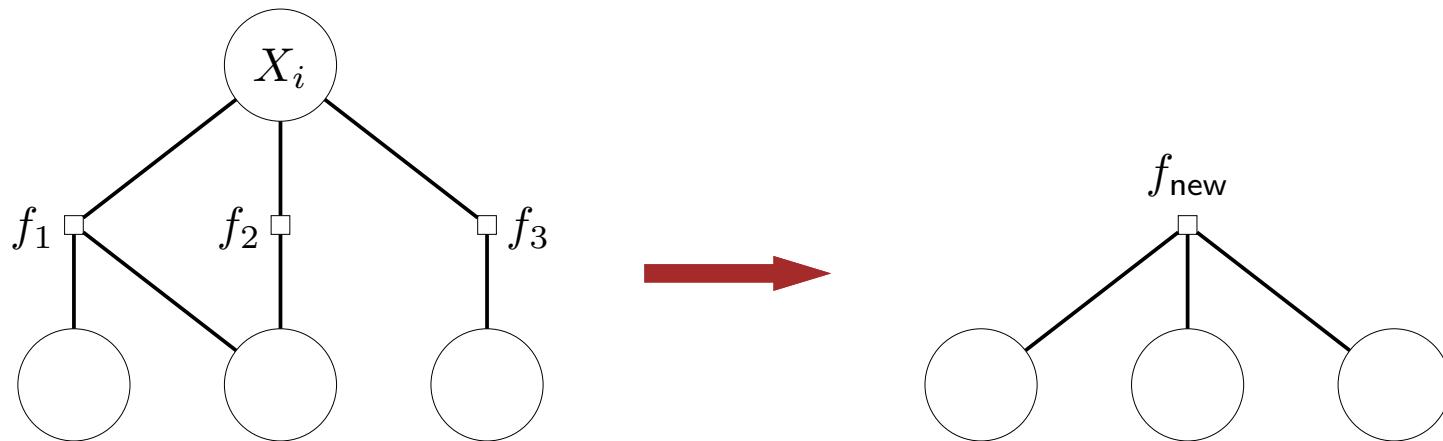
# Elimination: general



## Definition: elimination

- To **eliminate** a variable  $X_i$ , consider all factors  $f_1, \dots, f_k$  that depend on  $X_i$ .
- Remove  $X_i$  and  $f_1, \dots, f_k$ .
- Add  $f_{\text{new}}(x) = \max_{x_i} \prod_{j=1}^k f_j(x)$

# Elimination: general



$$f_{\text{new}}(x) = \max_{x_i} \prod_{j=1}^k f_j(x)$$

- Solves a mini-problem over  $X_i$  conditioned on its Markov blanket!
- Scope of  $f_{\text{new}}$  is  $\text{MarkovBlanket}(X_i)$

- In general, to eliminate a variable  $X_i$ , we look at all factors which depend on it, just like in conditioning. We then remove those factors  $f_1, \dots, f_k$  and  $X_i$ , just as in conditioning. Where elimination differs is that it produces a single factor which depends on the Markov blanket rather than a new factor for each  $f_j$ .
- Note that eliminating a variable  $X_i$  is much more costly than conditioning, and will produce a new factor which can have quite high arity (if  $X_i$  depends on many other variables).
- But the good news is that once a variable  $X_i$  is eliminated, we don't have to revisit it again. If we have an assignment to the Markov blanket of  $X_i$ , then the new factor gives us the weight of the best assignment to  $X_i$ , which is stored in the new factor.
- If for every new factor  $f_{\text{new}}$ , we store for each input, not only the value of the max, but also the argmax, then we can quickly recover the best assignment to  $X_i$ .



# Question

Suppose we have a star-shaped factor graph. Which of the following is true (select all that apply)?

Conditioning on the hub produces unary factors.

Eliminating the hub produces unary factors.

# Variable elimination algorithm



## Algorithm: variable elimination

For  $i = 1, \dots, n$ :

    Eliminate  $X_i$  (produces new factor  $f_{\text{new},i}$ ).

For  $i = n, \dots, 1$ :

    Set  $X_i$  to the maximizing value in  $f_{\text{new},i}$ .

[demo: query(''); maxVariableElimination()]

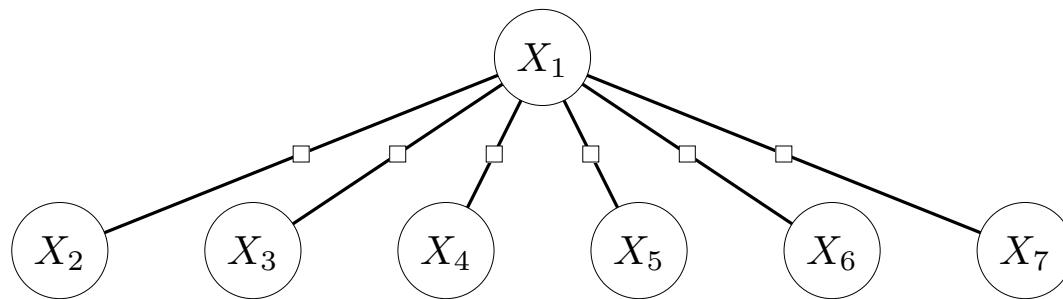
Let max-arity be the maximum arity of any  $f_{\text{new},i}$ .

Running time:  $O(n \cdot |\text{Domain}|^{\text{max-arity}+1})$

- We can turn elimination directly into an actual algorithm for computing the maximum weight assignment by just repeating it until we are left with one variable. This is called the **variable elimination** algorithm.
- The running time of this algorithm is exponential in the maximum arity of the factor produced along the way in variable elimination. The arity in the worst case is  $n - 1$ , but in the best case it could be a lot better, as we will see.

# Variable ordering

What's the maximum arity?



If eliminate leaves first, all factors have arity 1 (**good**)

If eliminate root first, get giant factor have arity 6 (**bad**)

Degree heuristic: eliminate variables with the fewest neighbors

- The arity of the factors produced during variable elimination depends on the ordering of the variables. In this extreme example, the difference is between 1 and 6.
- A useful heuristic is to eliminate variables with the smallest Markov blanket. In this example, the heuristic would eliminate the leaves and we'd only end up with factors with arity 1.



# Treewidth



## Definition: treewidth

The **treewidth** of a factor graph is the maximum arity of any factor created by variable elimination with the **best** variable ordering.

[whiteboard]

- Treewidth of a chain is 1.
- Treewidth of a tree is 1.
- Treewidth of simple cycle is 2.
- Treewidth of  $m \times n$  grid is  $\min(m, n)$ .

- If we use the best ordering, the arity of the largest factor produced is known as the **treewidth**, a very important property in graph theory. Computing the treewidth in general is NP-complete, and verifying that treewidth is  $k$  is exponential in  $k$  (but linear in the number of nodes).
- However, in practice, it's useful to remember a few examples.
- The treewidth of a chain is 1, by just eliminating all the variables left to right.
- The treewidth of a tree is also 1 by eliminating the variables from the leaves first.
- The treewidth of a simple cycle is 2. By symmetry, we can pick any variable on the cycle; eliminating it creates a factor that connects its two neighbors.
- The treewidth of an  $m \times n$  grid is more complex. Without loss of generality, assume that  $m \leq n$ . One can eliminate the variables by going along the columns left to right and processing the variables from the top row to the bottom row. Verify that when eliminating variable  $X_{ij}$  (in the  $i$ -th row and the  $j$ -th column), its Markov blanket is all the variables in column  $j + 1$  and row  $\leq i$  as well as all the variables in column  $j$  but in row  $> i$ .
- Note that even if we don't know the exact treewidth, having an upper bound gives us a handle on the running time of variable elimination.



# Summary

- Beam search: follows the most promising branches of search tree based on myopic information (think pruned BFS search)
- Local search: can freely re-assign variables; use randomness to get out of local optima
- Conditioning: break up a factor graph into smaller pieces (divide and conquer); can use in backtracking
- Elimination: solve a small subproblem conditioned on its Markov blanket

- Last lecture, we focused on algorithms that worked in the backtracking search framework. This lecture explores two classes of methods for efficiently finding the maximum weight assignment in a factor graph.
- The first class of methods are approximate methods. **Beam search**, like backtracking search, builds up partial assignments, but extends multiple partial assignments over the same subset of variables at once, and heuristically keeping the ones that seem most promising so far. It is quite possible that the actual maximum weight assignment will "fall off the beam". **Local search**, in contrast, works with complete assignments, modifying the value of one variable at time. In both beam and local search, one considers one variable at a time, and we only need to look at the factors touching that one variable.
- The second class of methods are exact methods that rely on (conditional) **independence** structure of the graph, in particular, that the graph is weakly connected, for example, a chain or a tree. We approached this methods by thinking about two graph operations, conditioning and elimination. **Conditioning** sets the value of a variable, and breaks up any factors that touch that variable. **Elimination** maximizes over a variable, but since the maximum value depends on the Markov blanket, this maximization is encoded in a new factor that depends on all the variables in the Markov blanket. The variable elimination computes the maximum weight assignment by applying elimination to each variable sequentially.

# Constraint Satisfaction Problems (CSPs)

CS 221 Section – 10/31/19

Chuma Kabaghe

Will Deaderick

# Agenda

- CSP Problem Modeling
- N-ary Constraints
- Exam Problem Solving

# Factor Graph and CSP Applications

- Scheduling problems: event scheduling, resource and assembly scheduling
- Inferring relations from data
- Puzzles: sudoku, crosswords
- Satisfiability problems
- Map and graph coloring
- Object tracking
- Decoding noisy signals (images, messages etc.)

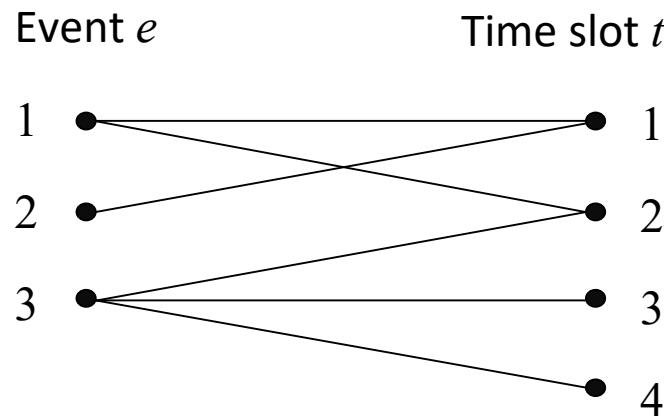
# Agenda

- CSP Problem Modeling
- N-ary Constraints
- Exam Problem Solving

# Event Scheduling

## Setup:

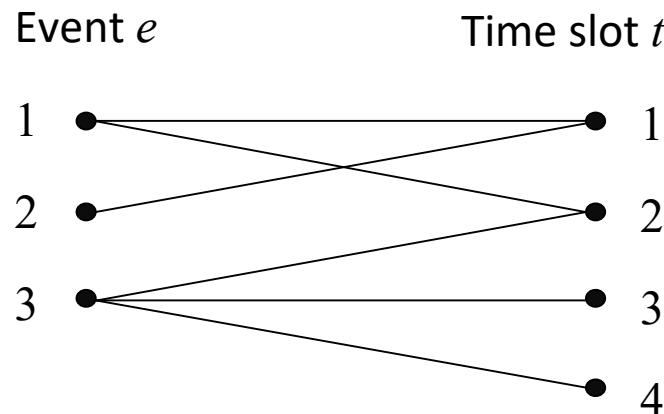
- Have  $E$  events and  $T$  time slots
- Each event  $e$  must be put in **exactly one** time slot
- Each time slot  $t$  can have **at most one** event
- Event  $e$  only allowed at time slot  $t$  if  $(e, t)$  in  $A$



# Event Scheduling

## Setup:

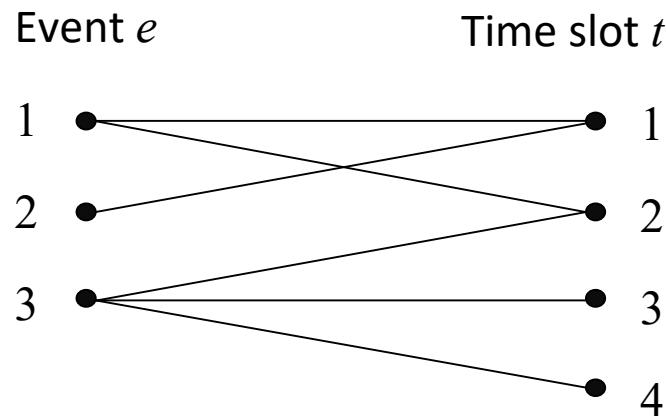
- Have  $E$  events and  $T$  time slots
- Each event  $e$  must be put in **exactly one** time slot
- Each time slot  $t$  can have **at most one** event
- Event  $e$  only allowed at time slot  $t$  if  $(e, t)$  in  $A$



# Event Scheduling

Formulation 1a:

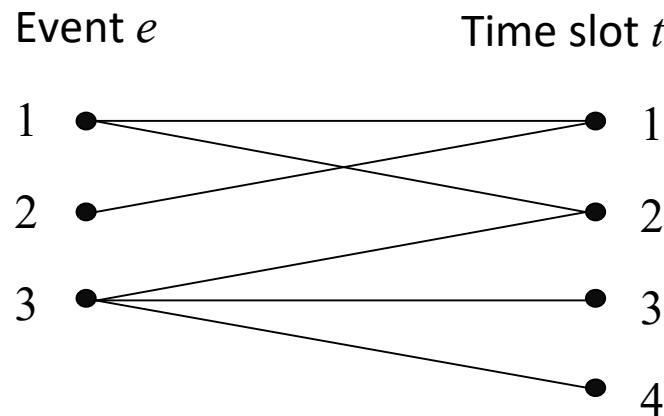
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$



# Event Scheduling

Formulation 1a:

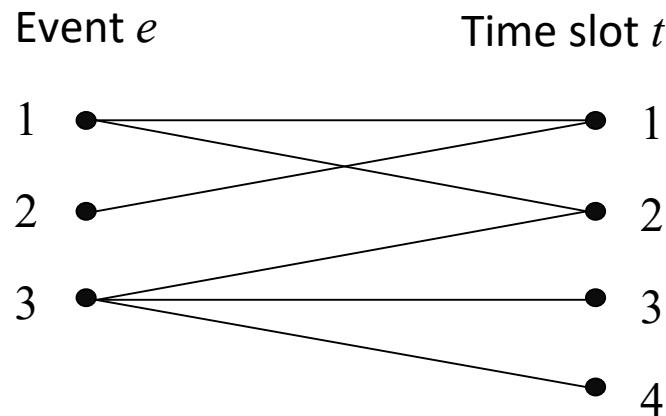
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$



# Event Scheduling

Formulation 1a:

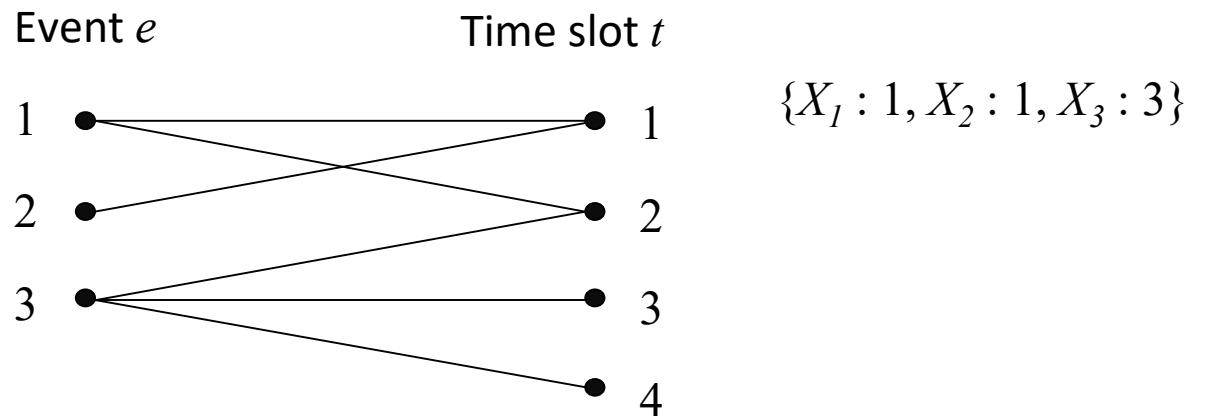
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$



# Event Scheduling

Formulation 1a:

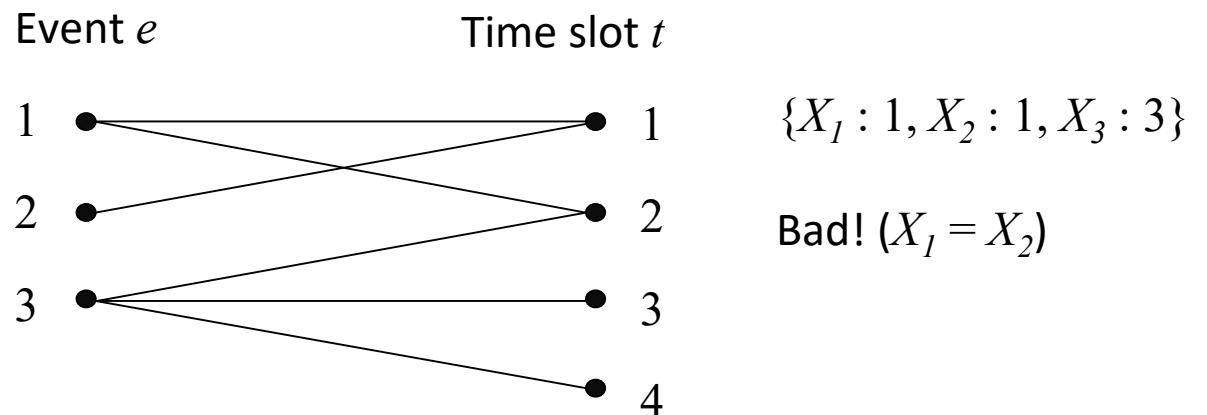
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$



# Event Scheduling

Formulation 1a:

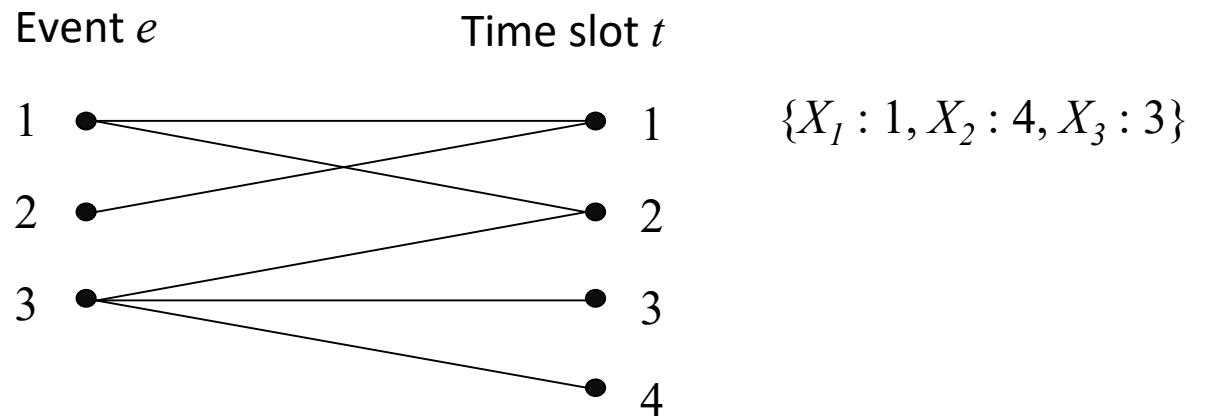
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$



# Event Scheduling

Formulation 1a:

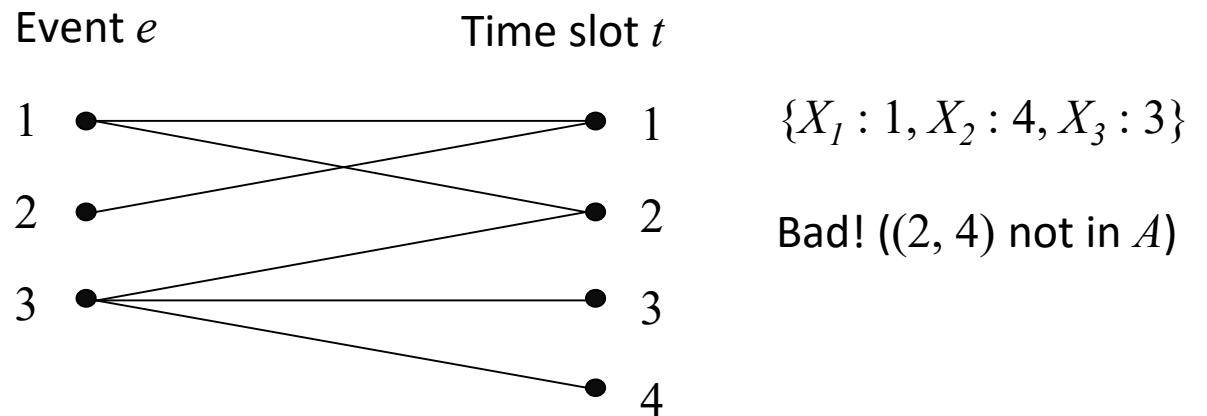
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$



# Event Scheduling

Formulation 1a:

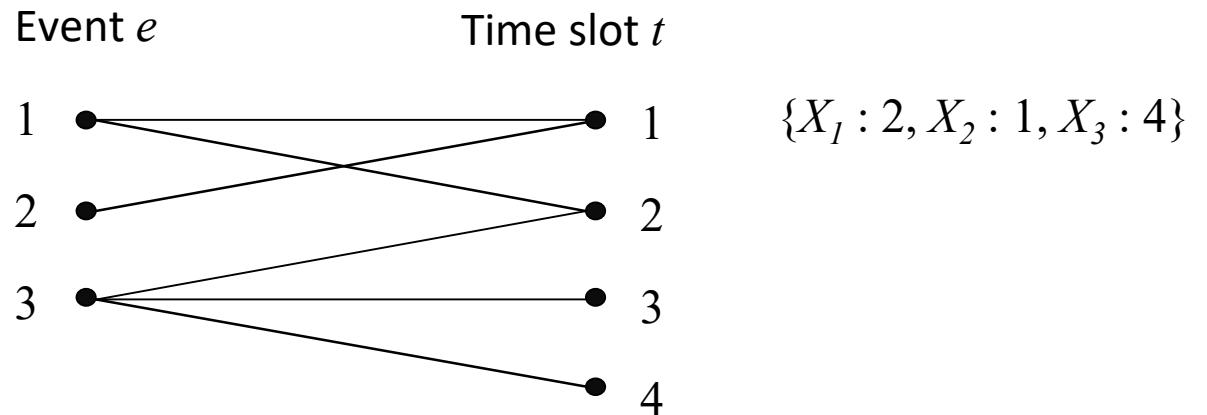
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$



# Event Scheduling

Formulation 1a:

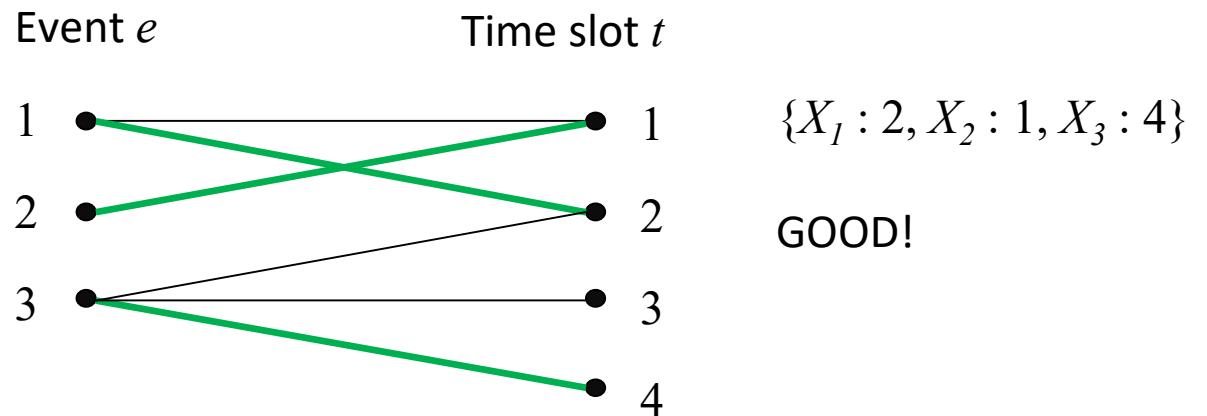
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$



# Event Scheduling

Formulation 1a:

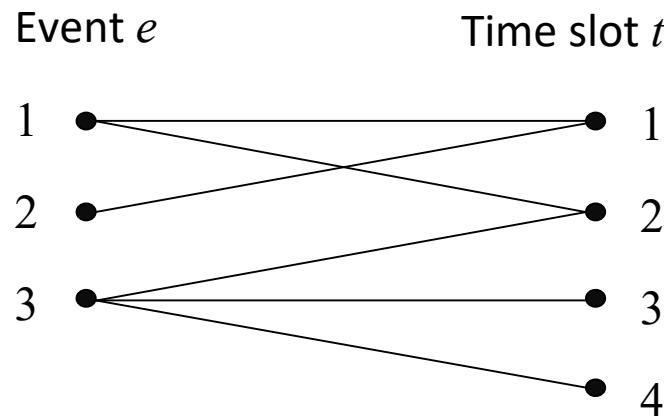
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$



# Event Scheduling

## Formulation 1a:

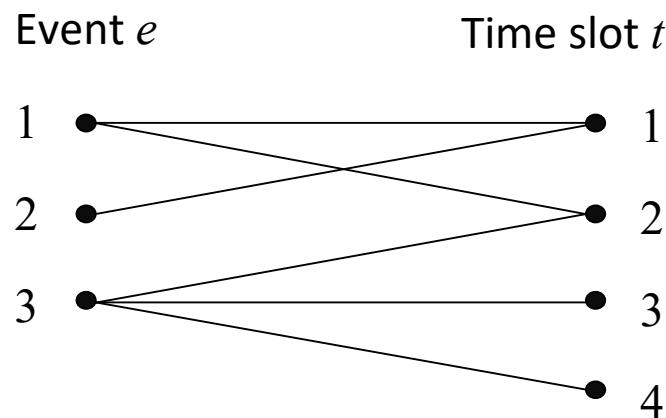
- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$



# Event Scheduling

Formulation 1b:

- Variables for each event  $e$ ,  $X_1, \dots, X_E$

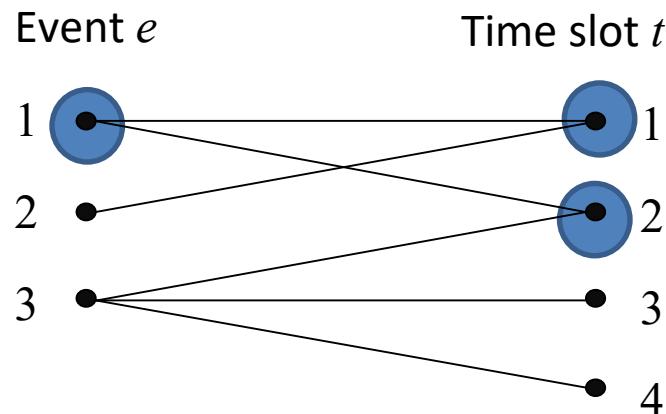


# Event Scheduling

Formulation 1b:

- Variables for each event  $e$ ,  $X_1, \dots, X_E$

$$Domain_i = \{t : (i, t) \in A\}$$



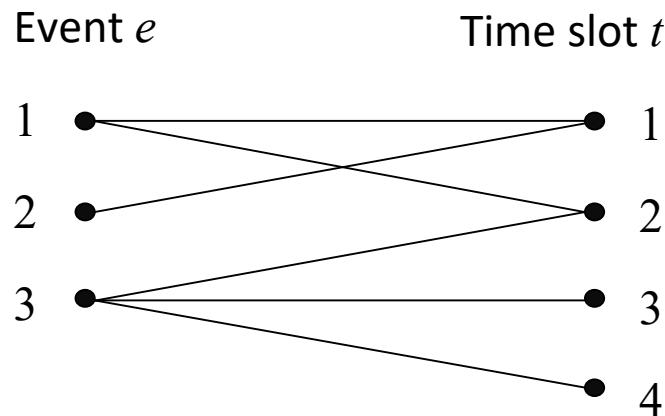
# Event Scheduling

Formulation 1b:

- Variables for each event  $e$ ,  $X_1, \dots, X_E$

$$Domain_i = \{t : (i, t) \in A\}$$

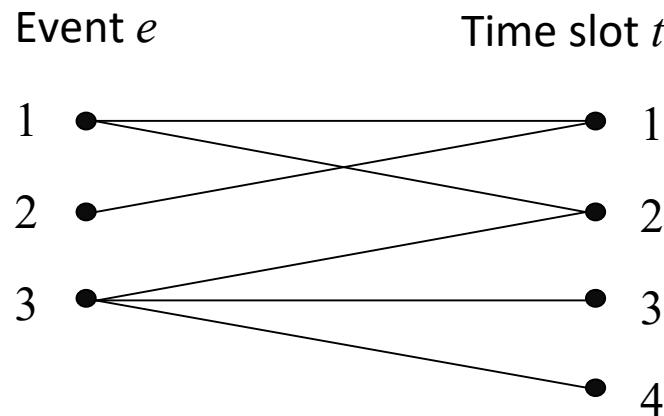
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$



# Event Scheduling

Formulation 2a:

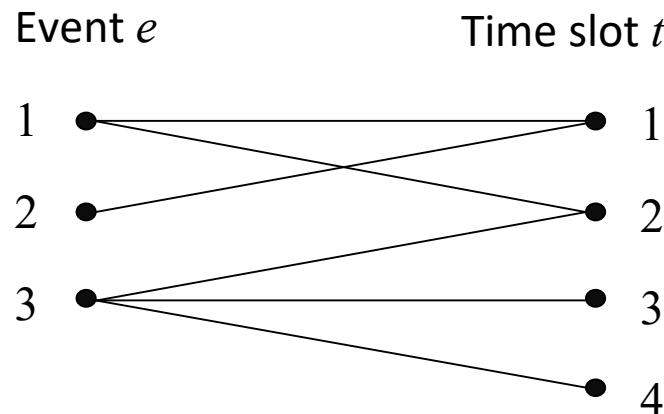
- Variables for each time slot  $t$ :  $Y_t \in \{1, \dots, E\} \cup \{\emptyset\}$



# Event Scheduling

Formulation 2a:

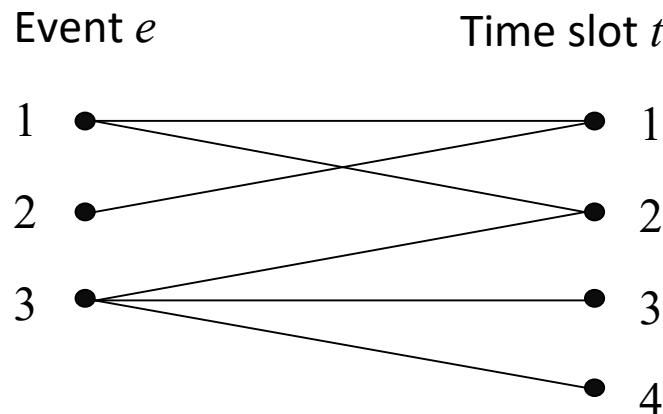
- Variables for each time slot  $t$ :  $Y_t \in \{1, \dots, E\} \cup \{\emptyset\}$
- Constraints (each event is scheduled exactly once): for each event  $e$ , enforce [ $Y_t = e$  for exactly one  $t$ ]



# Event Scheduling

Formulation 2a:

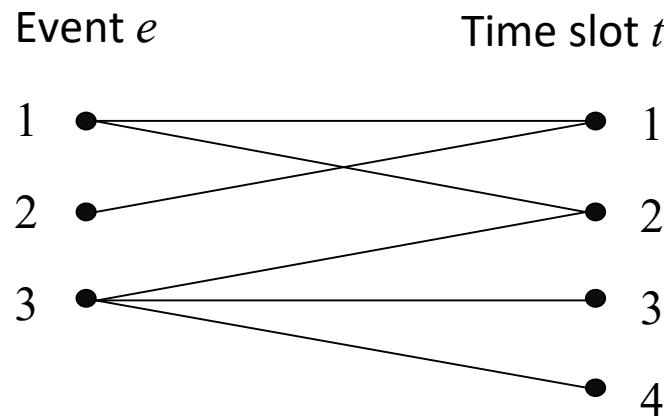
- Variables for each time slot  $t$ :  $Y_t \in \{1, \dots, E\} \cup \{\emptyset\}$
- Constraints (each event is scheduled exactly once): for each event  $e$ , enforce  $[Y_t = e \text{ for exactly one } t]$
- Constraints (only schedule allowed times): for each time slot  $t$ , enforce  $[Y_t = \emptyset \text{ or } (Y_t, t) \in A]$



# Event Scheduling

## Formulation 2a:

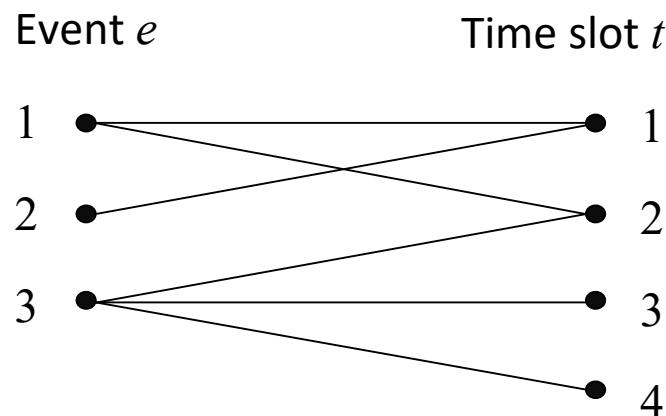
- Variables for each time slot  $t$ :  $Y_t \in \{1, \dots, E\} \cup \{\emptyset\}$
- Constraints (each event is scheduled exactly once): for each event  $e$ , enforce  $[Y_t = e \text{ for exactly one } t]$
- Constraints (only schedule allowed times): for each time slot  $t$ , enforce  $[Y_t = \emptyset \text{ or } (Y_t, t) \in A]$



# Event Scheduling

Formulation 2b:

- Variables for each time slot  $t$ :  $Y_1, \dots, Y_T$

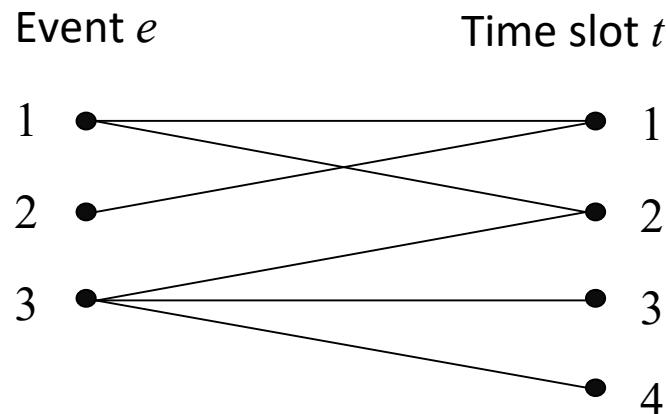


# Event Scheduling

Formulation 2b:

- Variables for each time slot  $t$ :  $Y_1, \dots, Y_T$

$$Domain_i = \{e : (e, i) \in A\} \cup \{\emptyset\}$$



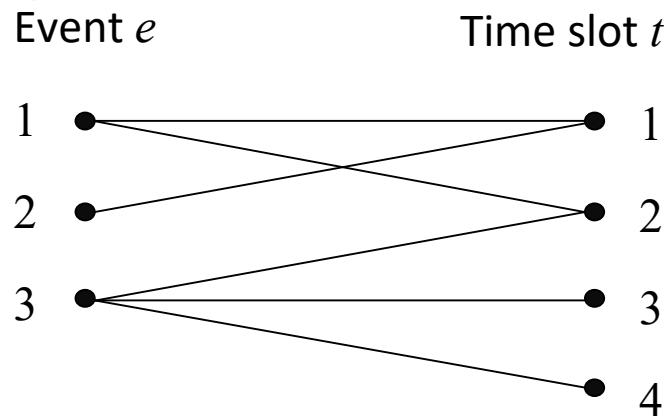
# Event Scheduling

Formulation 2b:

- Variables for each time slot  $t$ :  $Y_1, \dots, Y_T$

$$Domain_i = \{e : (e, i) \in A\} \cup \{\emptyset\}$$

- Constraints (each event is scheduled exactly once): for each event  $e$ , enforce [ $Y_t = e$  for exactly one  $t$ ]



# Event Scheduling

## Formulation 1a:

- Variables for each event  $e$ ,  $X_e \in \{1, \dots, T\}$  E variables with domain size T, and  $O(E^2)$  binary constraints.
- Constraints (only one event per time slot): for each pair of events  $e \neq e'$ , enforce  $[X_e \neq X_{e'}]$
- Constraints (only schedule allowed times): for each event  $e$ , enforce  $[(e, X_e) \in A]$   $T$  variables with domain size  $E+1$   
 $O(T^2)$  variables with domain size 2 and  $O(T^2)$  binary constraints.

## Formulation 2a:

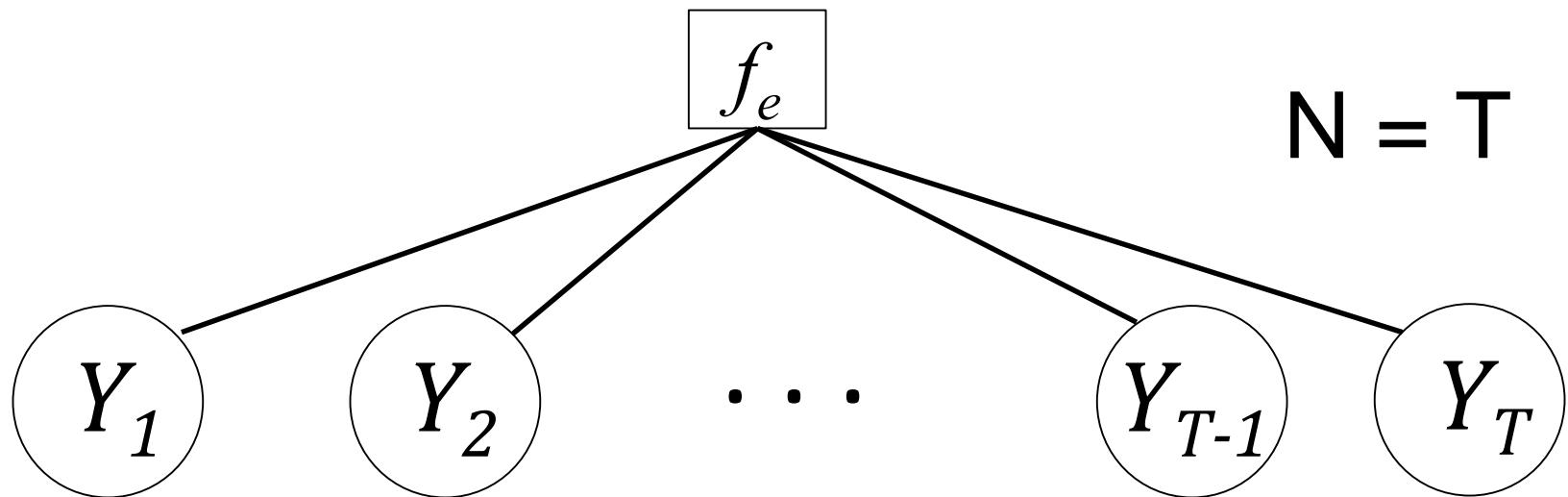
- Variables for each time slot  $t$ :  $Y_t \in \{1, \dots, E\} \cup \{\emptyset\}$
- Constraints (each event is scheduled exactly once): for each event  $e$ , enforce  $[Y_t = e \text{ for exactly one } t]$
- Constraints (only schedule allowed times): for each time slot  $t$ , enforce  $[Y_t = \emptyset \text{ or } (Y_t, t) \in A]$

# Agenda

- CSP Problem Modeling
- N-ary Constraints
- Exam Problem Solving

# N-ary Constraints

- From event scheduling:
  - Constraints (each event is scheduled exactly once): for each event  $e$ , enforce  
 $[Y_t = e \text{ for exactly one } t]$



# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

$$e = 1$$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0				

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

Processing:  $[A_i = A_{i-1} + 1[Y_i = e]]$

$$e = 1$$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0				

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

Processing:  $[A_i = \min(A_{i-1} + 1[Y_i = e], 2)]$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0				

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

Processing:  $[A_i = \min(A_{i-1} + 1[Y_i = e], 2)]$

$$e = 1$$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0	0			

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

Processing:  $[A_i = \min(A_{i-1} + 1[Y_i = e], 2)]$

$$e = 1$$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0	0	1		

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

Processing:  $[A_i = \min(A_{i-1} + 1[Y_i = e], 2)]$

$$e = 1$$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0	0	1	1	

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

Processing:  $[A_i = \min(A_{i-1} + 1[Y_i = e], 2)]$

$$e = 1$$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0	0	1	1	2

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

Processing:  $[A_i = \min(A_{i-1} + 1[Y_i = e], 2)]$

Final Output:  $1[A_T = 1]$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0	0	1	1	2

# N-ary Constraints

## Key Idea: Auxiliary Variables

Auxiliary Variables hold intermediate computation.

Represent “for exactly one” as counting the number of values equal to  $e$  and constraining that count to be equal to one.

Factors:

Initialization:  $[A_0 = 0]$

Processing:  $[A_i = \min(A_{i-1} + 1[Y_i = e], 2)]$

Final Output:  $1[A_T = 1]$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0	0	1	1	2

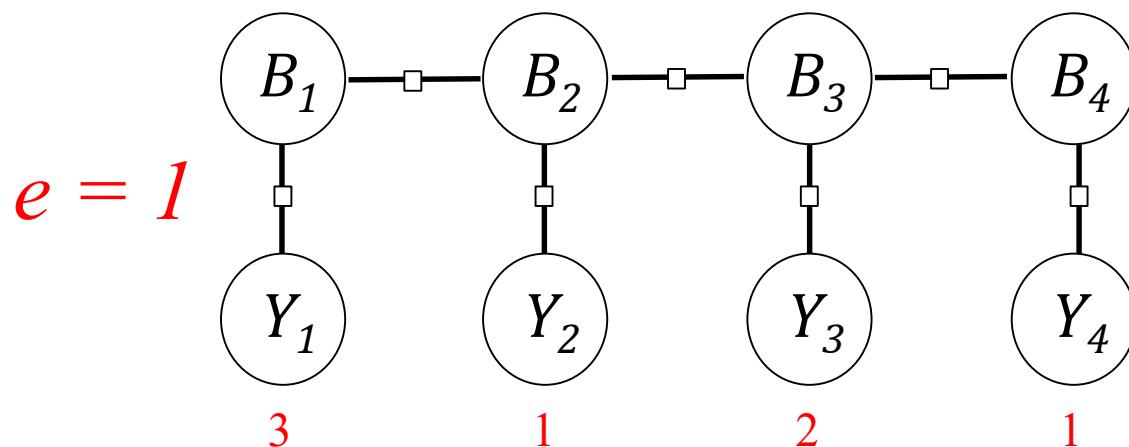
Still have factors with three variables...

# N-ary Constraints

Key idea: Combine  $A_{i-1}$  and  $A_i$  into one variable  $B_i$

# N-ary Constraints

Key idea: Combine  $A_{i-1}$  and  $A_i$  into one variable  $B_i$

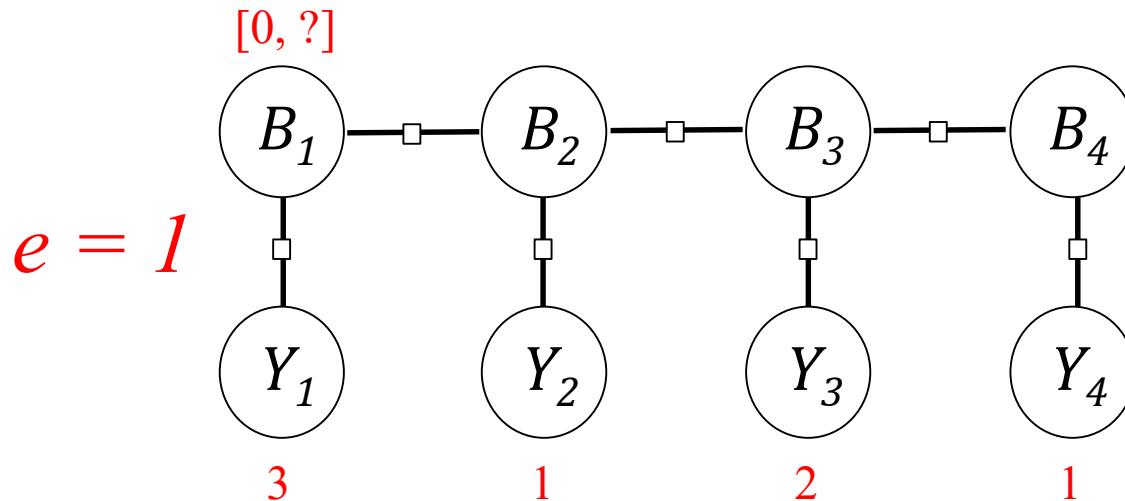


$e = 1$

$i$	0	1	2	3	4
$Y_i$		3	1	2	1
$A_i$	0	0	1	1	2

# N-ary Constraints

Key idea: Combine  $A_{i-1}$  and  $A_i$  into one variable  $B_i$



Factors:

Initialization:  $[B_1[0] = 0]$

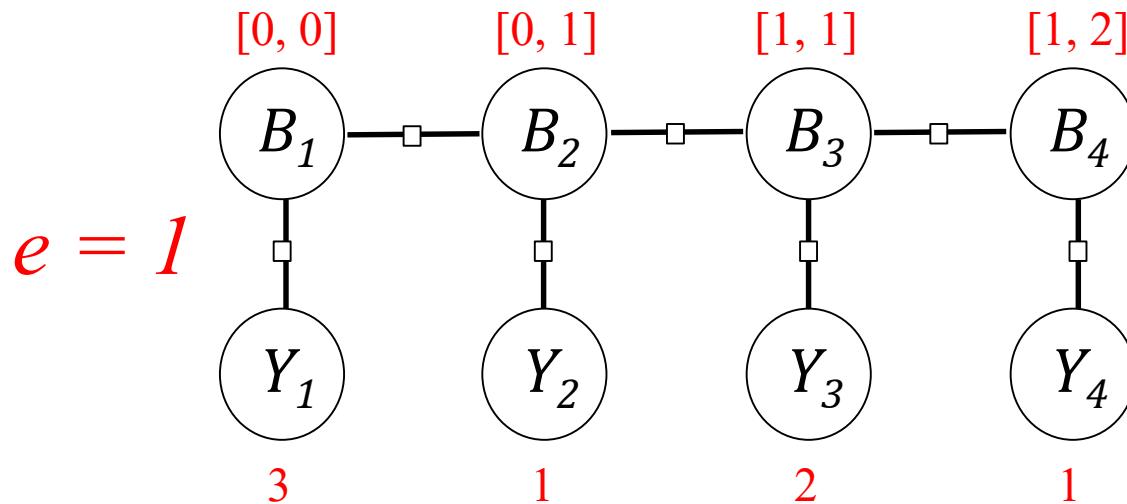
Processing:  $[B_i[1] = \min(B_i[0] + 1[Y_i = e], 2)]$

Final Output:  $1[B_T[1] = 1]$

Consistency:  $[B_{i-1}[1] = B_i[0]]$

# N-ary Constraints

Key idea: Combine  $A_{i-1}$  and  $A_i$  into one variable  $B_i$



Factors:

Initialization:  $[B_1[0] = 0]$

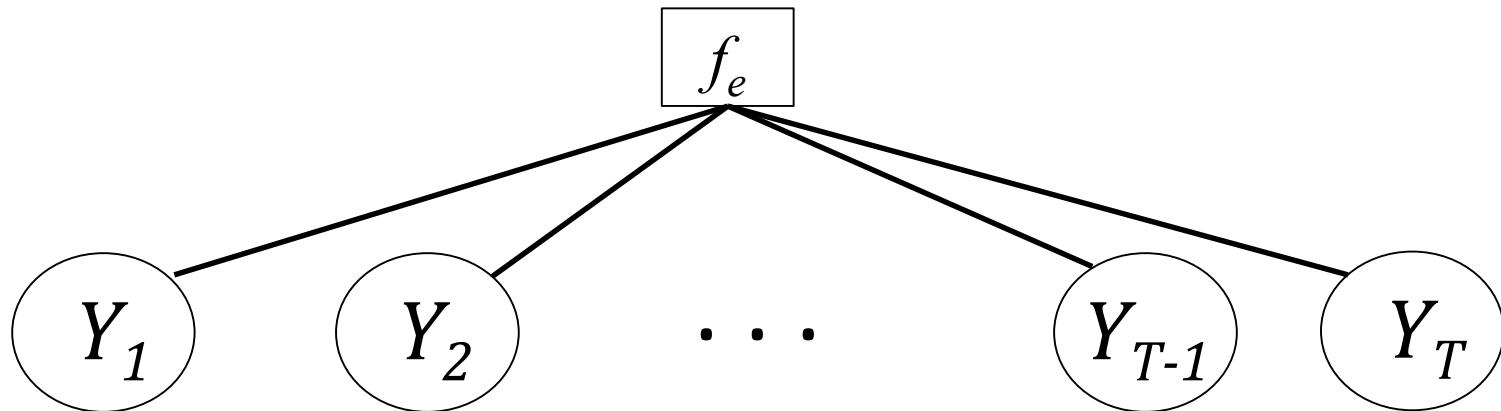
Processing:  $[B_i[1] = \min(B_i[0] + 1[Y_i = e], 2)]$

Final Output:  $1[B_T[1] = 1]$

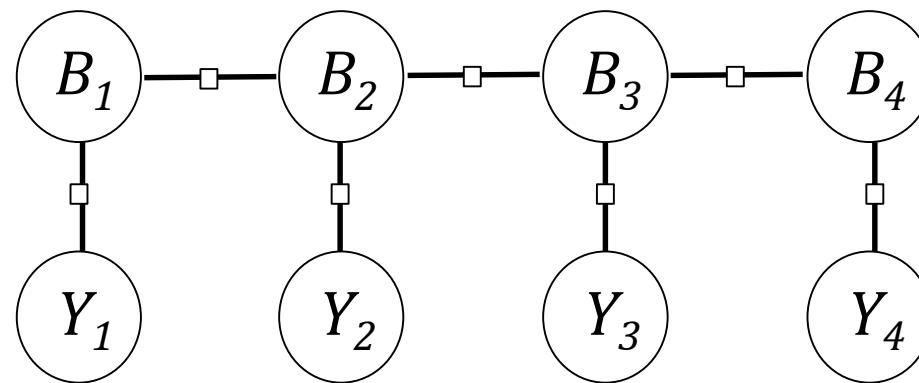
Consistency:  $[B_{i-1}[1] = B_i[0]]$

# N-ary Constraints

[ $Y_t = e$  for exactly one  $t$ ]



44



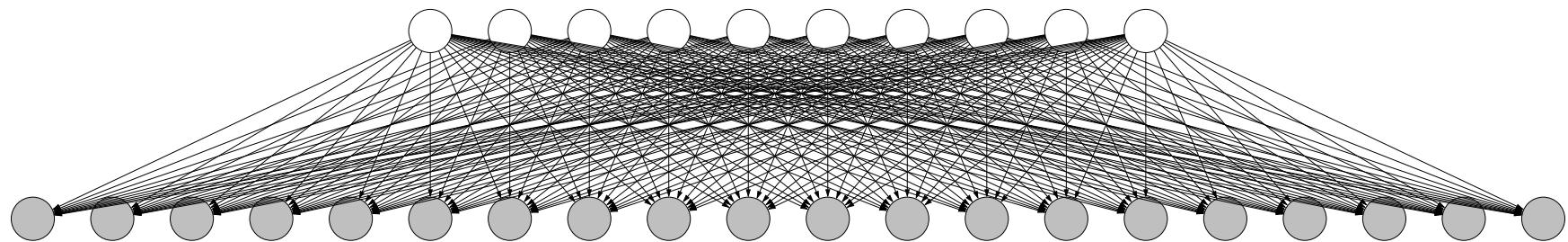
44

# Agenda

- CSP Problem Modeling
- N-ary Constraints
- Exam Problem Solving



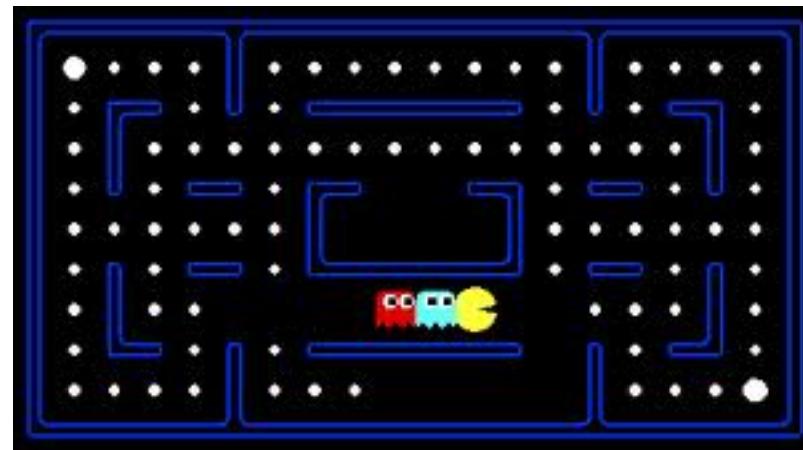
# Lecture 13: Bayesian networks I



# Announcements

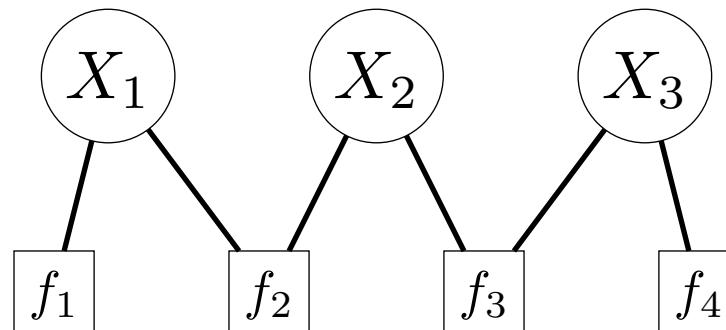
- **scheduling** is due tomorrow
- **car** is due next Tuesday
- **p-progress** is due next Thursday
- **exam** is in two weeks

# Pac-Man competition



1. (1765) Adam Keppler
2. (1764) Allan Li
3. (1763.5) Jiahao Zhang

# Review: definition



## Definition: factor graph

Variables:

$X = (X_1, \dots, X_n)$ , where  $X_i \in \text{Domain}_i$

Factors:

$f_1, \dots, f_m$ , with each  $f_j(X) \geq 0$

$$\text{Weight}(x) = \prod_{j=1}^m f_j(x)$$

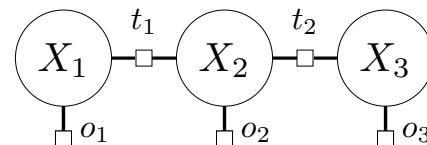
- Last week, we talked about factor graphs, which uses the product of factors to specify a weight  $\text{Weight}(x)$  for each assignment  $x$  in a compact way. The stated objective was to find the maximum weight assignment.
- Given any factor graph, we saw a number of algorithms (backtracking search, beam search, Gibbs sampling, variable elimination) for (approximately) optimizing this objective.

# Review: object tracking



## Problem: object tracking

Sensors report positions: 0, 2, 2. Objects don't move very fast and sensors are a bit noisy. What path did the object take?



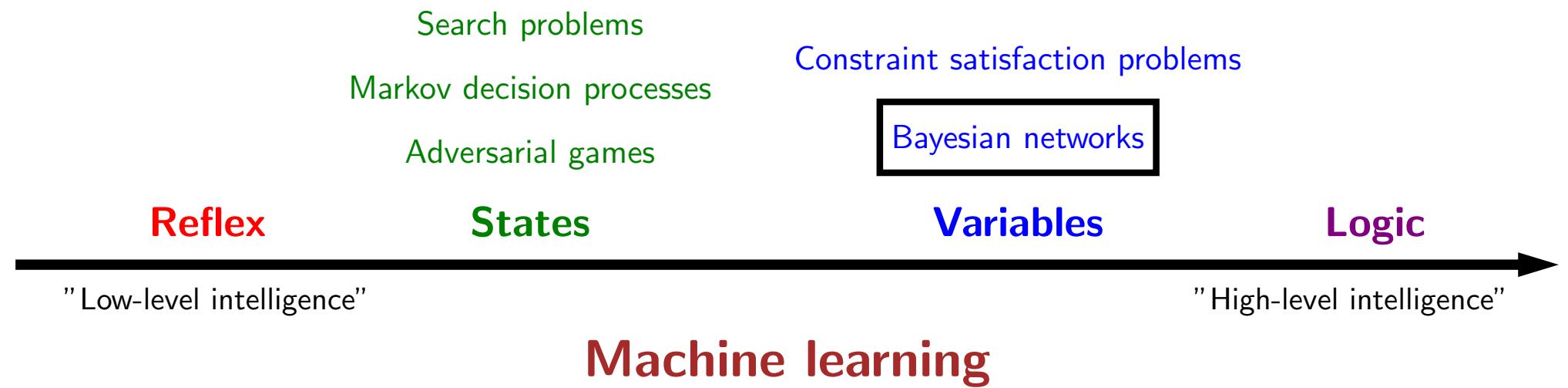
- Variables  $X_i$ : location of object at time  $i$
- Transition factors  $t_i(x_i, x_{i+1})$ : incorporate physics
- Observation factors  $o_i(x_i)$ : incorporate sensors

[demo: `maxVariableElimination()`]

How do we **interpret** the factors?

- As an example, recall the object tracking example. We defined observation factors to capture the fact that the true object position is close to the sensor reading, and the transition factors to capture the fact that the true object positions across time are close to each other.
- We just set them rather arbitrarily. Is there a more principled way to think about these factors beyond being non-negative functions?

# Course plan



- Much of this class has been on developing modeling frameworks. We started with state-based models, where we cast real-world problems as finding paths or policies through a state graph.
- Then, we saw that for a large class of problems (such as scheduling), it was much more convenient to use the language of factor graphs.
- While factor graphs could be reduced to state-based models by fixing the variable ordering, we saw that they also led to notions of treewidth and variable elimination, which allowed us to understand our models much better.
- In this lecture, we will introduce another modeling framework, Bayesian networks, which are factor graphs imbued with the language of probability. This will give probabilistic life to the factors of factor graphs.



# Roadmap

Basics

Probabilistic programs

Inference

- Bayesian networks were popularized in AI by Judea Pearl in the 1980s, who showed that having a coherent probabilistic framework is important for **reasoning under uncertainty**.
- There is a lot to say about the Bayesian networks (CS228 is an entire course about them and their cousins, Markov networks). So we will devote most of this lecture focusing on modeling.



# Review: probability

**Random variables:** sunshine  $S \in \{0, 1\}$ , rain  $R \in \{0, 1\}$

**Joint distribution:**

$s$	$r$	$\mathbb{P}(S = s, R = r)$
0	0	0.20
0	1	0.08
1	0	0.70
1	1	0.02

**Marginal distribution:**

$s$	$\mathbb{P}(S = s)$
0	0.28
1	0.72

(aggregate rows)

**Conditional distribution:**

$s$	$\mathbb{P}(S = s   R = 1)$
0	0.8
1	0.2

(select rows, normalize)

- Before introducing Bayesian networks, let's review probability (at least the relevant parts). We start with an example about the weather. Suppose we have two boolean random variables,  $S$  and  $R$  representing whether it is sunny and rainy. Think of an assignment to  $(S, R)$  as representing a possible state of the world.
- The **joint distribution** specifies a probability for each assignment to  $(S, R)$  (state of the the world). We use lowercase letters (e.g.,  $s$  and  $r$ ) to denote values and uppercase letters (e.g.,  $S$  and  $R$ ) to denote random variables. Note that  $\mathbb{P}(S = s, R = r)$  is a probability (a number) while  $\mathbb{P}(S, R)$  is a distribution (a table of probabilities). We don't know what state of the world we're in, but we know what the probabilities are (there are no unknown unknowns). The joint distribution contains all the information and acts as the central source of truth, like a database.
- From it, we can derive a **marginal distribution** over a subset of the variables. We get this by aggregating the rows that share the same value of  $S$ . The interpretation is that we are interested in  $S$ . We don't explicitly care about  $R$ , but we want to take into account  $R$ 's effect on  $S$ . We say that  $R$  is **marginalized out**. This is a special form of elimination. In the last lecture, we leveraged max-elimination, where we took the max over the eliminated variables; here, we are taking a sum.
- The **conditional distribution** selects rows of the table matching the condition (right of the bar), and then normalizes the probabilities so that they sum to 1. The interpretation is that we observe the condition ( $R = 1$ ) and are interested in  $S$ . This is the conditioning that we saw for factor graphs, but where we normalize the selected rows to get probabilities.

# Probabilistic inference

Joint distribution (probabilistic database):

$$\mathbb{P}(S, R, T, A)$$

Probabilistic inference:

- **Condition** on evidence (traffic, autumn):  $T = 1, A = 1$
- Interested in **query** (rain?):  $R$

$$\mathbb{P}(\underbrace{R}_{\text{query}} \mid \underbrace{T = 1, A = 1}_{\text{condition}})$$

( $S$  is **marginalized out**)

- We should think about each assignment  $x$  as a possible state of the world (it's raining, it's not sunny, there is traffic, it is autumn, etc.). Think of the joint distribution as one giant database that contains full information about how the world works.
- In practice, we'd like to ask questions by querying this probabilistic database. First, we observe some evidence, which effectively fixes some of the variables. Second, we are interested in the distribution of some set of variables which we didn't observe. This forms a query, and the process of answering this query (computing the desired distribution) is called **probabilistic inference**.

# Challenges

**Modeling:** How to specify a joint distribution  $\mathbb{P}(X_1, \dots, X_n)$  **compactly?**

Bayesian networks (factor graphs to specify joint distributions)

**Inference:** How to compute queries  $\mathbb{P}(R \mid T = 1, A = 1)$  **efficiently?**

Variable elimination, Gibbs sampling, particle filtering (analogue of algorithms for finding maximum weight assignment)

- In general, a joint distribution over  $n$  variables has size exponential in  $n$ . From a modeling perspective, how do we even specify an object that large? Here, we will see that Bayesian networks, based on factor graphs, offer an elegant solution.
- From an algorithms perspective, there is still the question of how we perform probabilistic inference efficiently. In the next lecture, we will see how we can adapt all of the algorithms that we saw before for computing maximum weight assignments in factor graphs, essentially by replacing a max with a sum.
- The two desiderata are rather synergistic, and it is the same property — conditional independence — that makes both possible.



## Question

Earthquakes and burglaries are independent events that will cause an alarm to go off. Suppose you hear an alarm. How does hearing on the radio that there's an earthquake change your beliefs about burglary?

it increases the probability of burglary

it decreases the probability of burglary

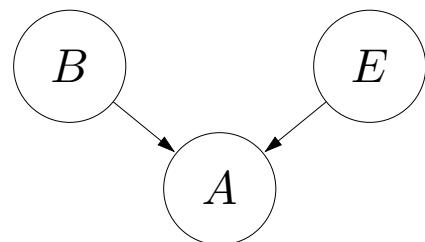
it does not change the probability of burglary

- Situations like these arise all the time in practice: we have a lot of unknowns which are all dependent on one another. If we obtain evidence on some of these unknowns, how does that affect our belief about the other unknowns? This is called **reasoning under uncertainty**.
- In this lecture, we'll see how we can perform this type of reasoning under uncertainty in a principled way using Bayesian networks.



# Bayesian network (alarm)

$$\mathbb{P}(B = b, E = e, A = a) \stackrel{\text{def}}{=} p(b)p(e)p(a | b, e)$$



b	$p(b)$
1	$\epsilon$
0	$1 - \epsilon$

e	$p(e)$
1	$\epsilon$
0	$1 - \epsilon$

b	e	a	$p(a   b, e)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

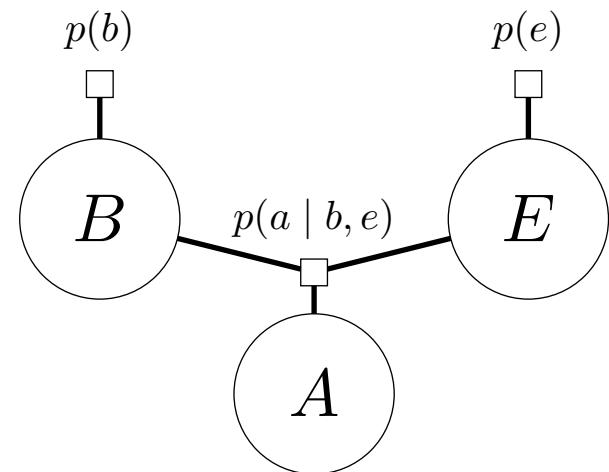
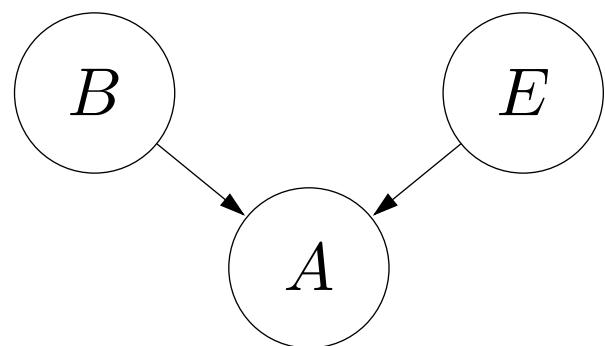
$$p(b) = \epsilon \cdot [b = 1] + (1 - \epsilon) \cdot [b = 0]$$

$$p(e) = \epsilon \cdot [e = 1] + (1 - \epsilon) \cdot [e = 0]$$

$$p(a | b, e) = [a = (b \vee e)]$$

- Let us try to model the situation. First, we establish that there are three variables,  $B$  (burglary),  $E$  (earthquake), and  $A$  (alarm).
- Second, we connect up the variables to model the dependencies. Unlike in factor graphs, these dependencies are represented as **directed** edges. You can intuitively think about the directionality as suggesting causality, though what this actually means is a deeper question and beyond the scope of this class.
- Third, for each variable, we specify a **local conditional distribution** (a factor) of that variable given its parent variables. In this example,  $B$  and  $E$  have no parents while  $A$  has two parents,  $B$  and  $E$ . This local conditional distribution is what governs how a variable is generated.
- Fourth, we define the joint distribution over all the random variables as the product of all the local conditional distributions.
- Note that we write the local conditional distributions using  $p$ , while  $\mathbb{P}$  is reserved for the joint distribution over all random variables, which is defined as the product.

# Bayesian network (alarm)



$$\mathbb{P}(B = b, E = e, A = a) = p(b)p(e)p(a \mid b, e)$$

Bayesian networks are a special case of factor graphs!

Note: single factor that connects **all** parents!

- Note that the local conditional distributions (e.g.,  $p(a | b, e)$ ) are non-negative so they can be thought of simply as factors of a factor graph. The joint probability of an assignment is then the weight of that assignment.
- In this light, Bayesian networks are just a type of factor graph, but with additional structure and interpretation.

# Probabilistic inference (alarm)

Joint distribution:

$b$	$e$	$a$	$\mathbb{P}(B = b, E = e, A = a)$
0	0	0	$(1 - \epsilon)^2$
0	0	1	0
0	1	0	0
0	1	1	$(1 - \epsilon)\epsilon$
1	0	0	0
1	0	1	$\epsilon(1 - \epsilon)$
1	1	0	0
1	1	1	$\epsilon^2$

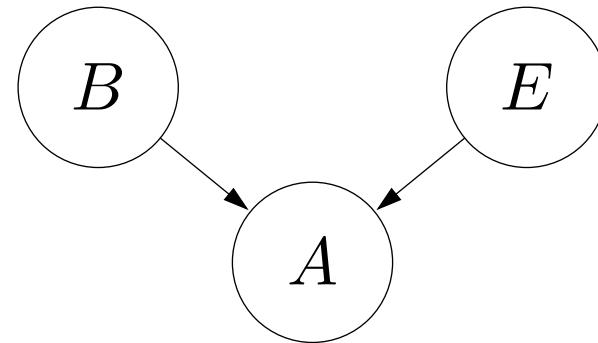
Queries:  $\mathbb{P}(B)? \mathbb{P}(B | A = 1)? \mathbb{P}(B | A = 1, E = 1)?$

[demo:  $\epsilon = 0.05$ ]

- Bayesian networks can be used to capture common reasoning patterns under uncertainty (which was one of their first applications).
- Consider the following model: Suppose the probability of an earthquake is  $\epsilon$  and the probability of a burglary is  $\epsilon$  and both are independent. Suppose that the alarm always goes off if either an earthquake or a burglary occurs.
- In the prior, we can eliminate  $A$  and  $E$  and get that the probability of the burglary is  $\epsilon$ .
- Now suppose we hear the alarm  $A = 1$ . The probability of burglary is now  $\mathbb{P}(B = 1 | A = 1) = \frac{1}{2-\epsilon}$ .
- Now suppose that you hear on the radio that there was an earthquake ( $E = 1$ ). Then the probability of burglary goes down to  $\mathbb{P}(B = 1 | A = 1, E = 1) = \epsilon$  again.



# Explaining away

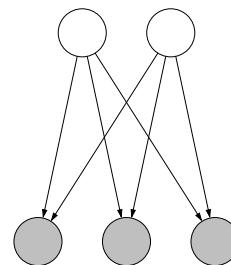


## Key idea: explaining away

Suppose two causes positively influence an effect. Conditioned on the effect, conditioning on one cause reduces the probability of the other cause.

- This last phenomenon has a special name: **explaining away**. Suppose we have two **cause** variables  $B$  and  $E$ , which are parents of an **effect** variable  $A$ . Assume the causes influence the effect positively (e.g., through the OR function).
- Conditioned on the effect  $A = 1$ , there is some posterior probability of  $B$ . Conditioned on the effect  $A = 1$  and the other cause  $E = 1$ , the new posterior probability is reduced. We then say that the other cause  $E$  has explained away  $B$ .

# Definition



## Definition: Bayesian network

Let  $X = (X_1, \dots, X_n)$  be random variables.

A **Bayesian network** is a directed acyclic graph (DAG) that specifies a **joint distribution** over  $X$  as a product of **local conditional distributions**, one for each node:

$$\mathbb{P}(X_1 = x_1, \dots, X_n = x_n) \stackrel{\text{def}}{=} \prod_{i=1}^n p(x_i \mid x_{\text{Parents}(i)})$$

- Without further ado, let's define a Bayesian network formally. A Bayesian network defines a large joint distribution in a modular way, one variable at a time.
- First, the graph structure captures what other variables a given variable depends on.
- Second, we specify a local conditional distribution for variable  $X_i$ , which is a function that specifies a distribution over  $X_i$  given an assignment  $x_{\text{Parents}(i)}$  to its parents in the graph (possibly no parents). The joint distribution is simply **defined** to be the product of all of the local conditional distributions together.
- Notationally, we use lowercase  $p$  (in  $p(x_i \mid x_{\text{Parents}(i)})$ ) to denote a local conditional distribution, and uppercase  $\mathbb{P}$  to denote the induced joint distribution over all variables. While we will see that the two coincide, it is important to keep these things separate in your head!

# Special properties

Key difference from general factor graphs:



**Key idea: locally normalized**

All factors (local conditional distributions) satisfy:

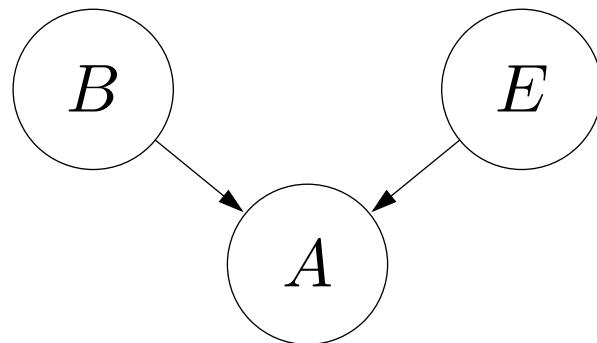
$$\sum_{x_i} p(x_i \mid x_{\text{Parents}(i)}) = 1 \text{ for each } x_{\text{Parents}(i)}$$

Implications:

- Consistency of sub-Bayesian networks
- Consistency of conditional distributions

- But Bayesian networks are more than that. The key property is that all the local conditional distributions, being distributions, sum to 1 over the first argument.
- This simple property results in two important properties of Bayesian networks that are not present in general factor graphs.

# Consistency of sub-Bayesian networks



A short calculation:

$$\begin{aligned}\mathbb{P}(B = b, E = e) &\stackrel{\text{def}}{=} \sum_a \mathbb{P}(B = b, E = e, A = a) \\ &\stackrel{\text{def}}{=} p(b)p(e)p(a \mid b, e) \\ &= p(b)p(e) \sum_a p(a \mid b, e) \\ &= p(b)p(e)\end{aligned}$$

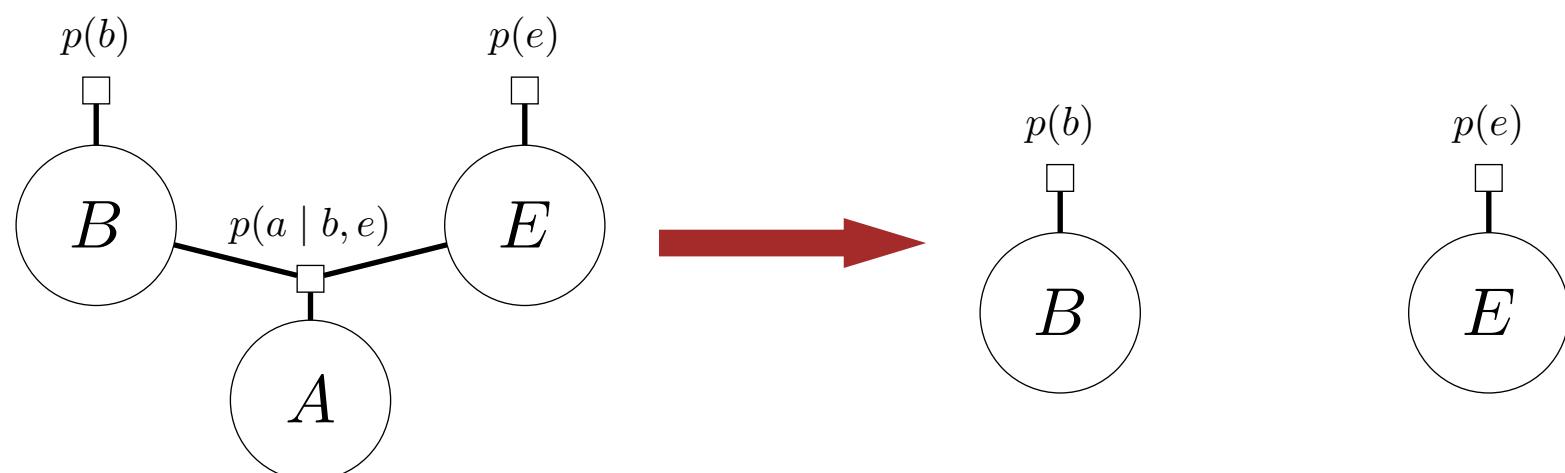
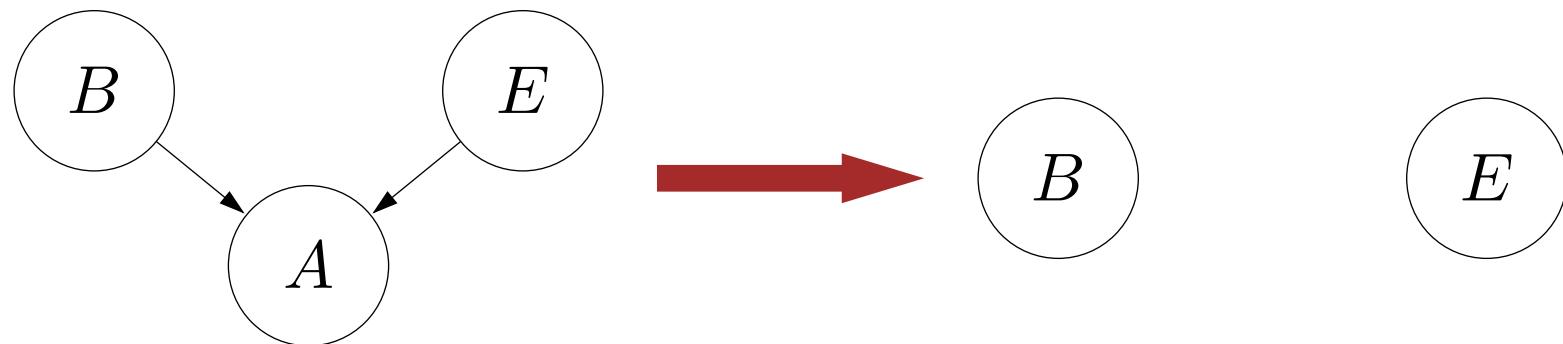
- First, let's see what happens when we marginalize  $A$  (by performing algebra on the joint probability). We see that we end up with  $p(b)p(e)$ , which actually defines a sub-Bayesian network with one fewer variable, and the same local conditional probabilities.
- If one marginalizes out all the variables, then one gets 1, which verifies that a Bayesian network actually defines a probability distribution.
- The philosophical ramifications of this property is that there could be many other variables that depend on the variables you've modeled (earthquakes also impact traffic) but as long as you don't observe them, they can be ignored mathematically (ignorance is bliss). Note that this doesn't mean that knowing about the other things isn't useful.

# Consistency of sub-Bayesian networks



## Key idea: marginalization

Marginalization of a leaf node yields a Bayesian network without the node.



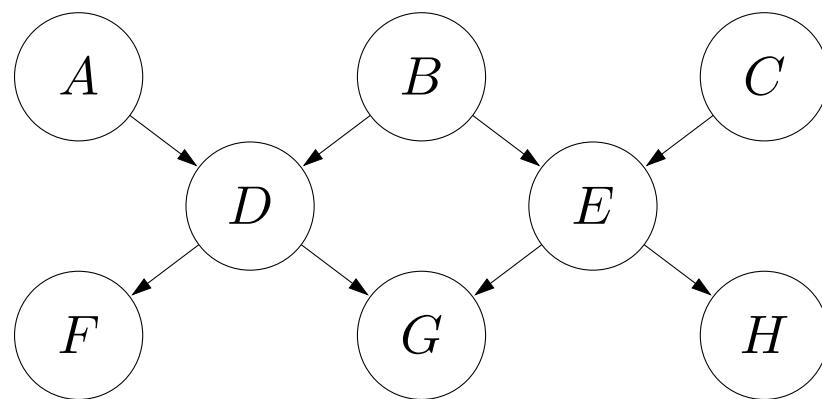
- This property is very attractive, because it means that whenever we have a large Bayesian network, where we don't care about some of the variables, we can just remove them (graph operations), and this encodes the same distribution as we would have gotten from marginalizing out variables (algebraic operations). The former, being visual, can be more intuitive.
- Note that if we marginalized only based on the factor graph representation, we would have kept the factor between  $B$  and  $E$ , which is too conservative. This is because the factor graph representation doesn't "know" about the probabilistic structure of its factors, and the factor has to be kept in general.

# Consistency of local conditionals



**Key idea: local conditional distributions**

Local conditional distributions (factors) are the true conditional distributions.



$$\underbrace{\mathbb{P}(D = d \mid A = a, B = b)}_{\text{from probabilistic inference}} = \underbrace{p(d \mid a, b)}_{\text{by definition}}$$

- Note that the local conditional distributions  $p(d | a, b)$  are simply defined by the user. On the other hand, the quantity  $\mathbb{P}(D = d | A = a, B = b)$  is not defined, but follows from probabilistic inference on the joint distribution defined by the Bayesian network.
- It's not clear a priori that the two have anything to do with each other. The second special property that we get from using Bayesian networks is that the two are actually the same.
- To show this, we can remove all non-ancestors of  $D$  by the consistency of sub-Bayesian networks, leaving us with the Bayesian network  $\mathbb{P}(A = a, B = b, D = d) = p(a)p(b)p(d | a, b)$ . By the chain rule,  $\mathbb{P}(A = a, B = b, D = d) = \mathbb{P}(A = a, B = b)\mathbb{P}(D = d | A = a, B = b)$ . If we marginalize out  $D$ , then we are left with the Bayesian network  $\mathbb{P}(A = a, B = b) = p(a)p(b)$ . From this, we can conclude that  $\mathbb{P}(D = d | A = a, B = b) = p(d | a, b)$ .
- This argument generalizes to any Bayesian network and local conditional distribution.



# Medical diagnosis



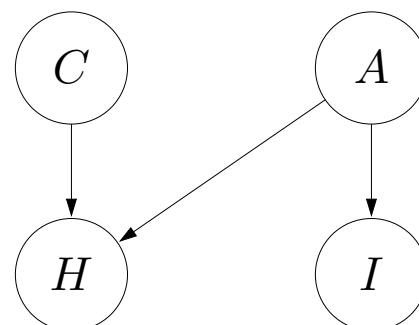
## Problem: cold or allergies?

You are coughing and have itchy eyes. Do you have a cold or allergies?

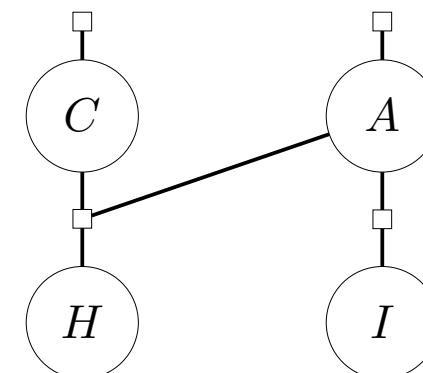
[whiteboard] [demo]

Variables: Cold, Allergies, Cough, Itchy eyes

Bayesian network:



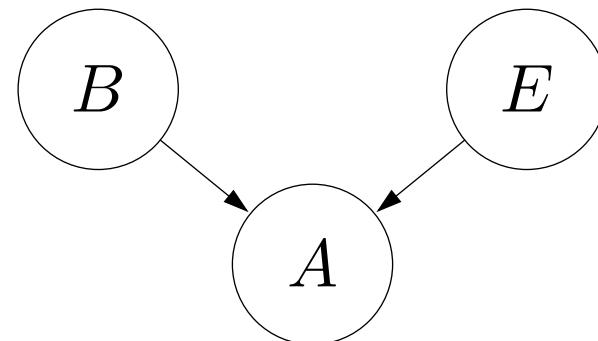
Factor graph:



- Here is another example (a cartoon version of Bayesian networks for medical diagnosis). Allergies and cold are the two hidden variables that we'd like to infer (we have some prior over these two). Cough and itchy eyes are symptoms that we observe as evidence, and we have some likelihood model of these symptoms given the hidden causes.
- Formally, we are interested in  $\mathbb{P}(C, A \mid H = 1, I = 1)$ .
- We can use the demo to infer the hidden state given the evidence.



# Summary so far



- Random variables capture state of world
- Edges between variables represent dependencies
- Local conditional distributions  $\Rightarrow$  joint distribution
- Probabilistic inference: ask questions about world
- Captures reasoning patterns (e.g., explaining away)
- Factor graph interpretation (for inference later)



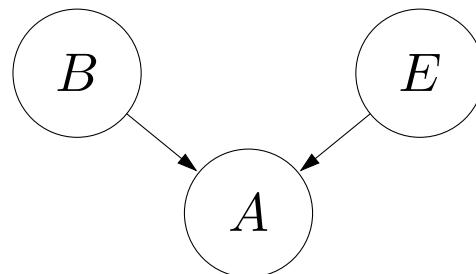
# Roadmap

Basics

**Probabilistic programs**

Inference

# Probabilistic programs



Probabilistic program: alarm

$$B \sim \text{Bernoulli}(\epsilon)$$

$$E \sim \text{Bernoulli}(\epsilon)$$

$$A = B \vee E$$



Key idea: probabilistic program

A randomized program that sets the random variables.

```
def Bernoulli(epsilon):  
    return random.random() < epsilon
```

- There is another way of writing down Bayesian networks other than graphically or mathematically, and that is as a probabilistic program. A **probabilistic program** is a randomized program that invokes a random number generator to make random choices. Executing this program will assign values to a collection of random variables  $X_1, \dots, X_n$ ; that is, generating an assignment.
- The probability (e.g., fraction of times) that the program generates that assignment is exactly the probability under the joint distribution specified by that program.
- We should think of this program as outputting the state of the world (or at least the part of the world that we care about for our task).
- Note that the probabilistic program is only used to define joint distributions. We usually wouldn't actually run this program directly.
- For example, we show the probabilistic program for alarm.  $B \sim \text{Bernoulli}(\epsilon)$  simply means that  $\mathbb{P}(B = 1) = \epsilon$ . Here, we can think about  $\text{Bernoulli}(\epsilon)$  as a randomized function (`random() < epsilon`) that returns 1 with probability  $\epsilon$  and 0 with probability  $1 - \epsilon$ .

# Probabilistic program: example



## Probabilistic program: object tracking

$$X_0 = (0, 0)$$

For each time step  $i = 1, \dots, n$ :

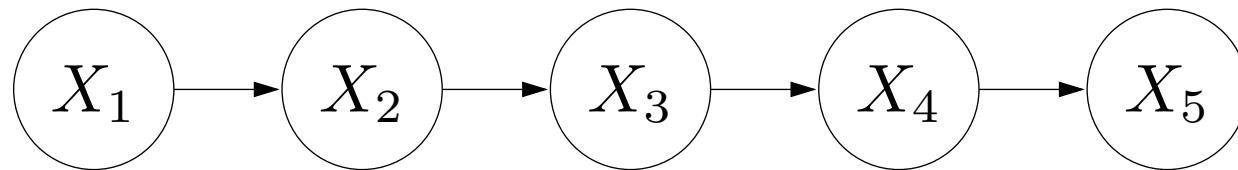
With probability  $\alpha$ :

$$X_i = X_{i-1} + (1, 0) \text{ [go right]}$$

With probability  $1 - \alpha$ :

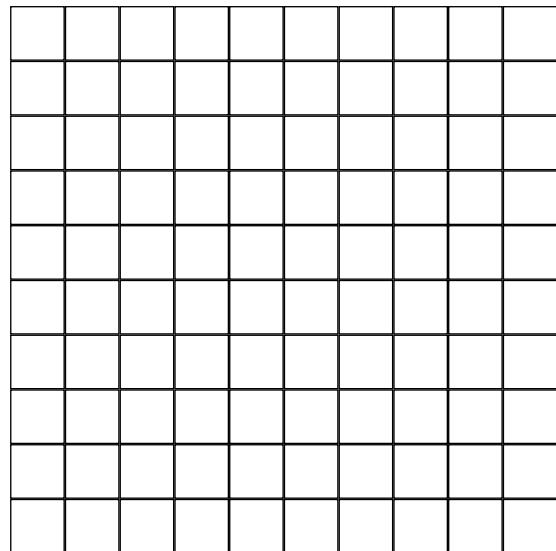
$$X_i = X_{i-1} + (0, 1) \text{ [go down]}$$

Bayesian network structure:



- This is a more interesting generative model since it has a for loop, which allows us to determine the distribution over a templated set of  $n$  variables rather than just 3 or 4.
- In these cases, variables are generally indexed by something like time or location.
- We can also draw the Bayesian network. Each  $X_i$  only depends on  $X_{i-1}$ . This is a chain-structured Bayesian network, called a **Markov model**.

# Probabilistic program: example



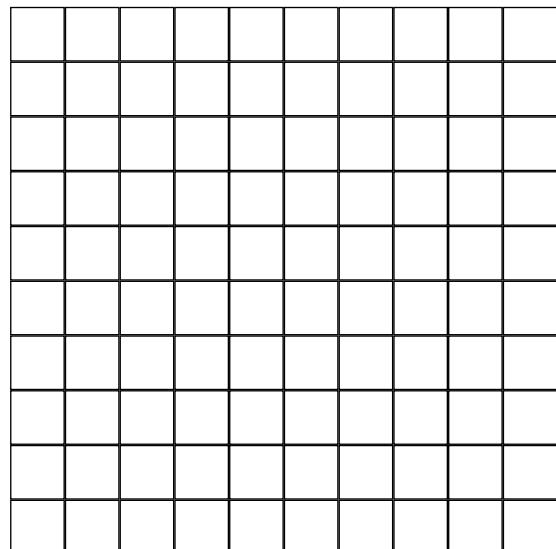
(press ctrl-enter to save)

Run

- Try clicking [Run]. Each time a new assignment of  $(X_1, \dots, X_n)$  is chosen.

# Probabilistic inference: example

**Query:** what are possible trajectories given **evidence**  $X_{10} = (8, 2)$ ?



(press ctrl-enter to save)

Run

- This program only serves for defining the distribution. Now we can query that distribution and ask the question: suppose the program set  $X_{10} = (8, 2)$ ; what is the distribution over the other variables?
- In the demo, note that all trajectories are constrained to go through  $(8, 2)$  at time step 10.

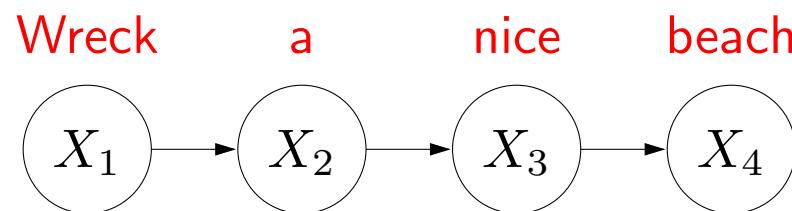
# Application: language modeling



## Probabilistic program: Markov model

For each position  $i = 1, 2, \dots, n$ :

Generate word  $X_i \sim p(X_i \mid X_{i-1})$



- In the context of natural language, a Markov model is known as a bigram model. A higher-order generalization of bigram models are  $n$ -gram models (more generally known as higher-order Markov models).
- Language models are often used to measure the "goodness" of a sentence, mostly within the context of a larger system such as speech recognition or machine translation.

# Application: object tracking

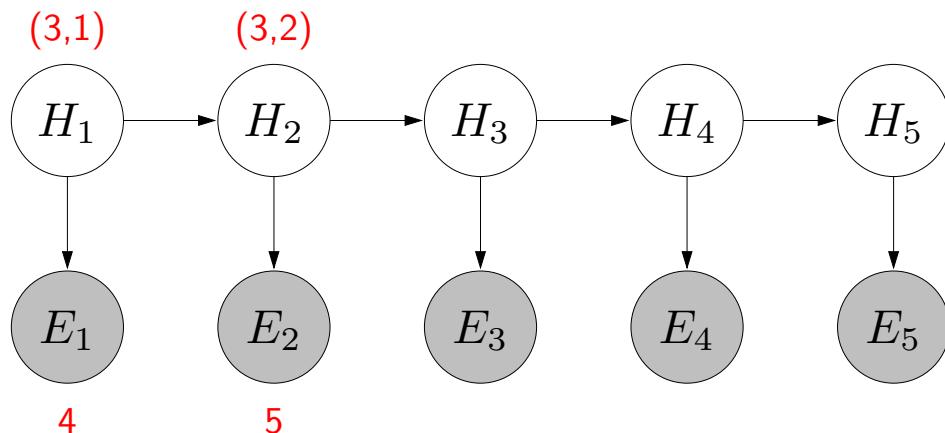


## Probabilistic program: hidden Markov model (HMM)

For each time step  $t = 1, \dots, T$ :

Generate object location  $H_t \sim p(H_t | H_{t-1})$

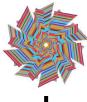
Generate sensor reading  $E_t \sim p(E_t | H_t)$



Inference: given sensor readings, where is the object?

- Markov models are limiting because they do not have a way of talking about noisy evidence (sensor readings). They can be extended quite easily to hidden Markov models, which introduce a parallel sequence of observation variables.
- For example, in object tracking,  $H_t$  denotes the true object location, and  $E_t$  denotes the noisy sensor reading, which might be (i) the location  $H_t$  plus noise, or (ii) the distance from  $H_t$  plus noise, depending on the type of sensor.
- In speech recognition,  $H_t$  would be the phonemes or words and  $E_t$  would be the raw acoustic signal.

## Application: multiple object tracking



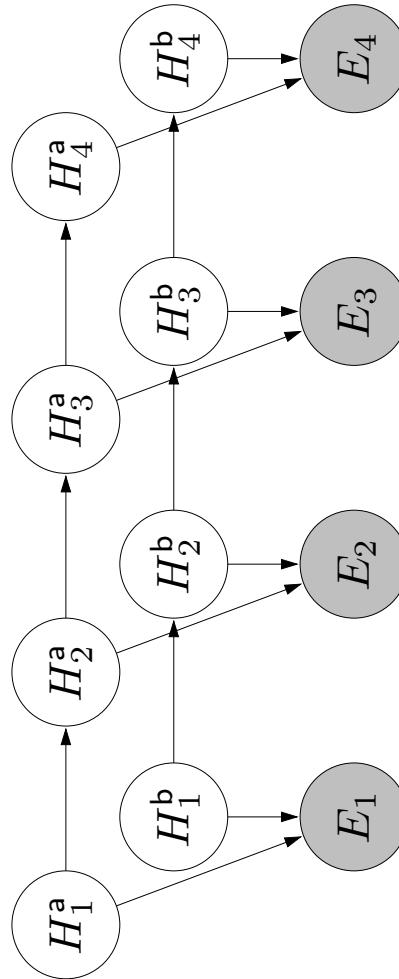
### Probabilistic program: factorial HMM

For each time step  $t = 1, \dots, T$ :

For each object  $o \in \{\text{a, b}\}$ :

Generate location  $H_t^o \sim p(H_t^o \mid H_{t-1}^o)$

Generate sensor reading  $E_t \sim p(E_t \mid H_t^a, H_t^b)$



- An extension of an HMM, called a **factorial HMM**, can be used to track multiple objects. We assume that each object moves independently according to a Markov model, but that we get one sensor reading which is some noisy aggregated function of the true positions.
- For example,  $E_t$  could be the set  $\{H_t^a, H_t^b\}$ , which reveals where the objects are, but doesn't say which object is responsible for which element in the set.

# Application: document classification

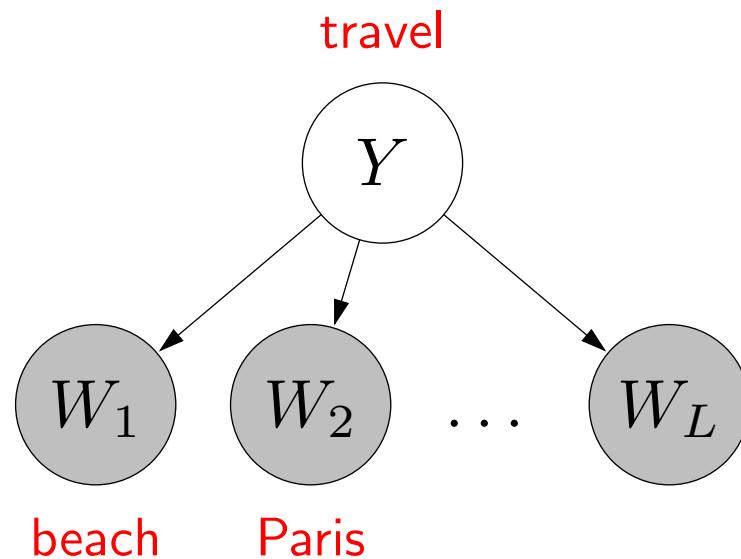


## Probabilistic program: naive Bayes

Generate label  $Y \sim p(Y)$

For each position  $i = 1, \dots, L$ :

Generate word  $W_i \sim p(W_i \mid Y)$



Inference: given a text document, what is it about?

- Naive Bayes is a very simple model which can be used for classification. For document classification, we generate a label and all the words in the document given that label.
- Note that the words are all generated independently, which is not a very realistic model of language, but naive Bayes models are surprisingly effective for tasks such as document classification.
- These types of models are traditionally called generative models as opposed to discriminative models for classification. Rather than thinking about how you take the input and produce the output label (e.g., using a neural network), you go the other way around: think about how the input is generated from the output (which is usually the purer, more structured form of the input).
- One advantage of using Naive Bayes for classification is that "training" is extremely easy and fast and just requires counting (as opposed to performing gradient descent).

# Application: topic modeling



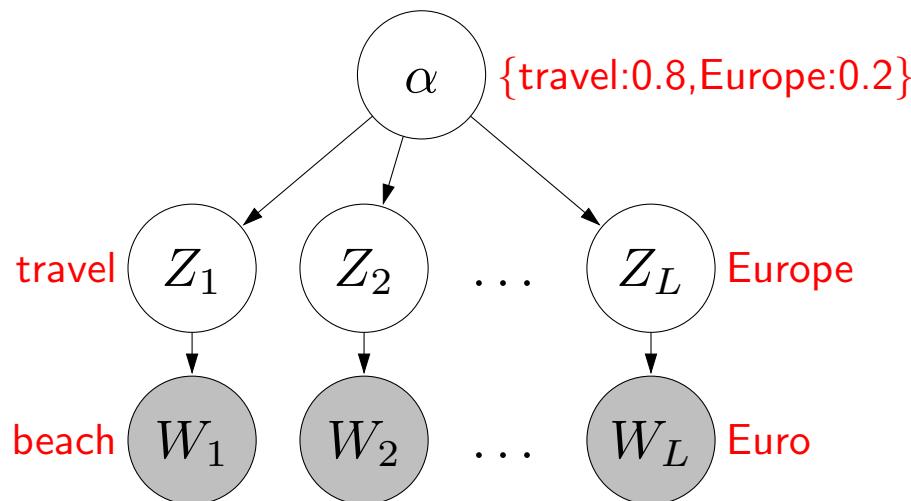
## Probabilistic program: latent Dirichlet allocation

Generate a distribution over topics  $\alpha \in \mathbb{R}^K$

For each position  $i = 1, \dots, L$ :

Generate a topic  $Z_i \sim p(Z_i | \alpha)$

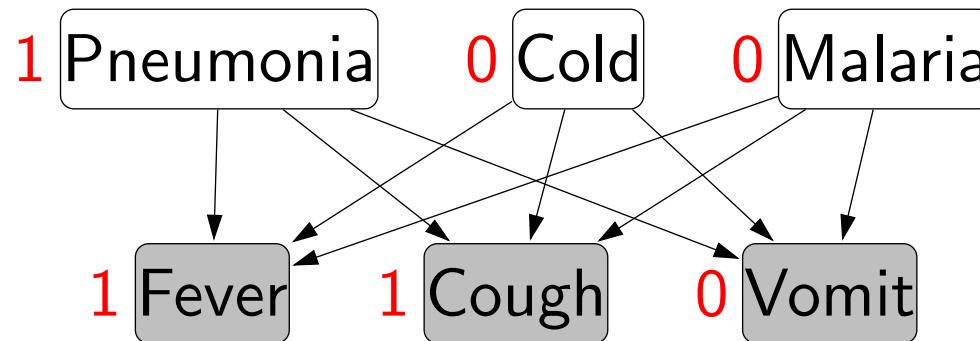
Generate a word  $W_i \sim p(W_i | Z_i)$



Inference: given a text document, what topics is it about?

- A more sophisticated model of text is latent Dirichlet Allocation (LDA), which allows a document to not just be about one topic (which was true in naive Bayes), but about multiple topics.
- Here, the distribution over topics  $\alpha$  is chosen per document from a Dirichlet distribution. Note that  $\alpha$  is a continuous-valued random variable. For each position, we choose a topic according to that per-document distribution and generate a word given that topic.
- Latent Dirichlet Allocation (LDA) has been very influential for modeling not only text but images, videos, music, etc.; any sort of data with hidden structure. It is very related to matrix factorization.

# Application: medical diagnostics



## Probabilistic program: diseases and symptoms

For each disease  $i = 1, \dots, m$ :

Generate activity of disease  $D_i \sim p(D_i)$

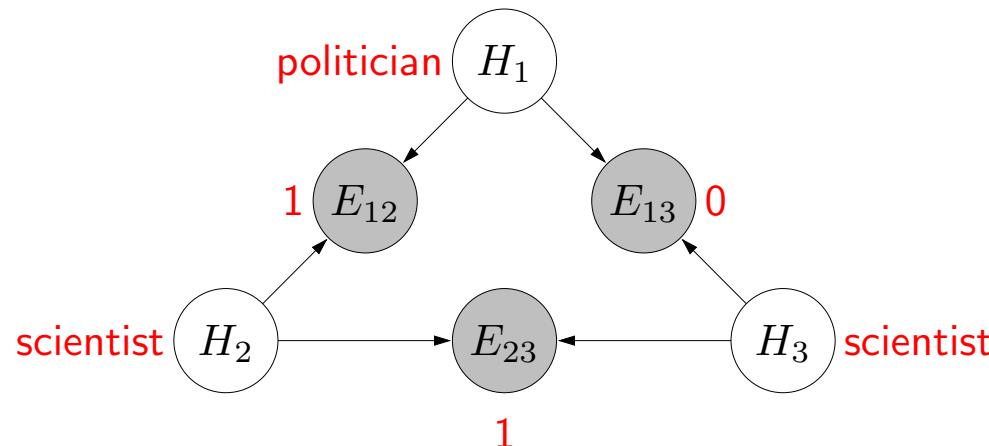
For each symptom  $j = 1, \dots, n$ :

Generate activity of symptom  $S_j \sim p(S_j | D_{1:m})$

Inference: If a patient has has a cough and fever, what disease(s) does he/she have?

- We already saw a special case of this model. In general, we would like to diagnose many diseases and might have measured many symptoms and vitals.

# Application: social network analysis



## Probabilistic program: stochastic block model

For each person  $i = 1, \dots, n$ :

Generate person type  $H_i \sim p(H_i)$

For each pair of people  $i \neq j$ :

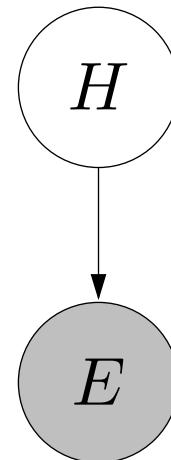
Generate connectedness  $E_{ij} \sim p(E_{ij} \mid H_i, H_j)$

**Inference:** Given a social network (graph over  $n$  people), what types of people are there?

- One can also model graphs such as social networks. A very naive-Bayes-like model is that each node (person) has a "type". Whether two people interact with each other is determined solely by their types and random chance.
- Note: there are extensions called mixed membership models which, like LDA, allow each person to have multiple types.
- In summary, it is quite easy to come up with probabilistic programs that tell a story of how the world works for the domain of interest. These probabilistic programs define joint distributions over assignments to a collection of variables. Usually, these programs describe how some collection of hidden variables  $H$  that you're interested in behave, and then describe the generation of the evidence  $E$  that you see conditioned on  $H$ . After defining the model, one can do probabilistic inference to compute  $\mathbb{P}(H \mid E = e)$ .



# Summary so far



- Many many different types of models
- Mindset: come up with stories of how the data (input) was generated through quantities of interest (output)
- Opposite of how we normally do classification!



# Roadmap

Basics

Probabilistic programs

Inference

# Review: probabilistic inference

## Input

Bayesian network:  $\mathbb{P}(X_1, \dots, X_n)$

Evidence:  $E = e$  where  $E \subseteq X$  is subset of variables

Query:  $Q \subseteq X$  is subset of variables



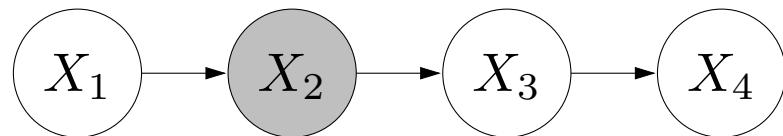
## Output

$\mathbb{P}(Q = q \mid E = e)$  for all values  $q$

Example: if coughing and itchy eyes, have a cold?

$$\mathbb{P}(C \mid H = 1, I = 1)$$

# Example: Markov model



Query:  $\mathbb{P}(X_3 = x_3 \mid X_2 = 5)$  for all  $x_3$

Tedious way:

$$\propto \sum_{x_1, x_4} p(x_1) p(x_2 = 5 \mid x_1) p(x_3 \mid x_2 = 5) p(x_4 \mid x_3)$$

$$\propto \left( \sum_{x_1} p(x_1) p(x_2 = 5 \mid x_1) \right) p(x_3 \mid x_2 = 5)$$

$$\propto p(x_3 \mid x_2 = 5)$$

Fast way:

[whiteboard]

- Let's first compute the query the old-fashioned way by grinding through the algebra. Then we'll see a faster, more graphical way, of doing this.
- We start by transforming the query into an expression that references the joint distribution, which allows us to rewrite as the product of the local conditional probabilities. To do this, we invoke the definition of marginal and conditional probability.
- One convenient shortcut we will take is to make use of the proportional-to ( $\propto$ ) relation. Note that in the end, we need to construct a distribution over  $X_3$ . This means that any quantity (such as  $\mathbb{P}(X_2 = 5)$ ) which doesn't depend on  $X_3$  can be folded into the proportionality constant. If you don't believe this, keep it around to convince yourself that it doesn't matter. Using  $\propto$  can save you a lot of work.
- Next, we do some algebra to push the summations inside. We notice that  $\sum_{x_4} p(x_4 | x_3) = 1$  because it's a local conditional distribution. The factor  $\sum_{x_1} p(x_1)p(x_2 = 5 | x_1)$  can also be folded into the proportionality constant.
- The final result is  $p(x_3 | x_2 = 5)$ , which matches the query as we expected by the consistency of local conditional distributions.

# General strategy

Query:

$$\mathbb{P}(Q \mid E = e)$$

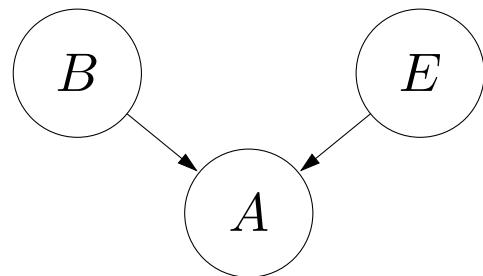


## Algorithm: general probabilistic inference strategy

- Remove (marginalize) variables that are not ancestors of  $Q$  or  $E$ .
- Convert Bayesian network to factor graph.
- Condition on  $E = e$  (shade nodes + disconnect).
- Remove (marginalize) nodes disconnected from  $Q$ .
- Run probabilistic inference algorithm (manual, variable elimination, Gibbs sampling, particle filtering).

- Our goal is to compute the conditional distribution over the query variables  $Q \subseteq H$  given evidence  $E = e$ . We can do this with our bare hands by chugging through all the algebra starting with the definition of marginal and conditional probability, but there is an easier way to do this that exploits the structure of the Bayesian network.
- Step 1: remove variables which are not ancestors of  $Q$  or  $E$ . Intuitively, these don't have an influence on  $Q$  and  $E$ , so they can be removed. Mathematically, this is due to the consistency of sub-Bayesian networks.
- Step 2: turn this Bayesian network into a factor graph by simply introducing one factor per node which is connected to that node and its parents. It's important to include all the parents and the child into one factor, not separate factors. From here out, all we need to think about is factor graphs.
- Step 3: condition on the evidence variables. Recall that conditioning on nodes in a factor graph shades them in, and as a graph operation, rips out those variables from the graph.
- Step 4: remove nodes which are not connected to  $Q$ . These are independent of  $Q$ , so they have no impact on the results.
- Step 5: Finally, run a standard probabilistic inference algorithm on the reduced factor graph. We'll do this manually for now using variable elimination. Later we'll see automatic methods for doing this.

# Example: alarm



$b$	$p(b)$
1	$\epsilon$
0	$1 - \epsilon$

$e$	$p(e)$
1	$\epsilon$
0	$1 - \epsilon$

$b$	$e$	$a$	$p(a   b, e)$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

[whiteboard]

Query:  $\mathbb{P}(B)$

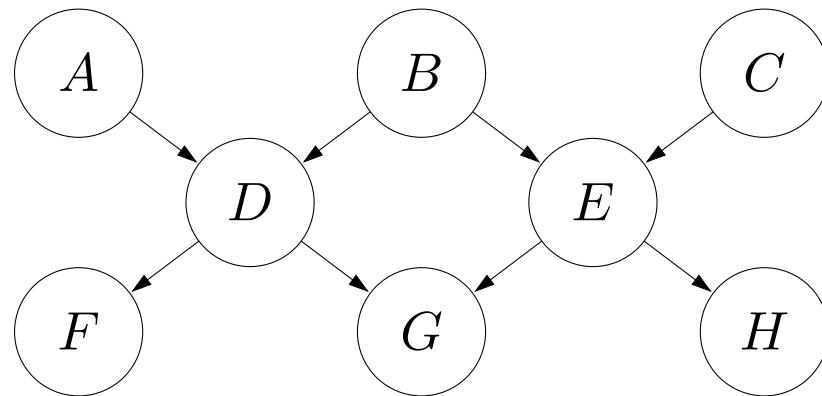
- Marginalize out  $A, E$

Query:  $\mathbb{P}(B | A = 1)$

- Condition on  $A = 1$

- Here is another example: the simple v-structured alarm network from last time.
- $\mathbb{P}(B) = p(b)$  trivially after marginalizing out  $A$  and  $E$  (step 1).
- For  $\mathbb{P}(B | A = 1)$ , step 1 doesn't do anything. Conditioning (step 3) creates a factor graph with factors  $p(b)$ ,  $p(e)$ , and  $p(a = 1 | b, e)$ . In step 5, we eliminate  $E$  by replacing it and its incident factors with a new factor  $f(b) = \sum_e p(e)p(a = 1 | b, e)$ . Then, we multiply all the factors (which should only be unary factors on the query variable  $B$ ) and normalize:  $\mathbb{P}(B = b | A = 1) \propto p(b)f(b)$ .
- To flesh this out, for  $b = 1$ , we have  $\epsilon(\epsilon + (1 - \epsilon)) = \epsilon$ . For  $b = 0$ , we have  $(1 - \epsilon)(\epsilon + 0) = \epsilon(1 - \epsilon)$ . The normalized result is thus  $\mathbb{P}(B = 1 | A = 1) = \frac{\epsilon}{\epsilon + \epsilon(1 - \epsilon)} = \frac{1}{2 - \epsilon}$ .
- For a probabilistic interpretation, note that all we've done is calculate  $\mathbb{P}(B = b | A = 1) = \frac{\mathbb{P}(B=b)\mathbb{P}(A=1|B=b)}{\mathbb{P}(A=1)} = \frac{p(b)f(b)}{\sum_{b_i \in \text{Domain}(B)} p(b_i)f(b_i)}$ , where the first equality follows from Bayes' rule and the second follows from the fact that the local conditional distributions are the true conditional distributions. The Bayesian network has simply given us a methodical, algorithmic way to calculate this probability.

# Example: A-H (section)



[whiteboard]

Query:  $\mathbb{P}(C \mid B = b)$

- Marginalize out everything else, note  $C \perp\!\!\!\perp B$

Query:  $\mathbb{P}(C, H \mid E = e)$

- Marginalize out  $A, D, F, G$ , note  $C \perp\!\!\!\perp H \mid E$

- In the first example, once we marginalize out all variables we can, we are left with  $C$  and  $B$ , which are disconnected. We condition on  $B$ , which just removes that node, and so we're just left with  $\mathbb{P}(C) = p(c)$ , as expected.
- In the second example, note that the two query variables are independent, so we can compute them separately. The result is  $\mathbb{P}(C = c, H = h \mid E = e) \propto p(c)p(h \mid e) \sum_b p(b)p(e \mid b, c)$ .
- If we had the actual values of these probabilities, we can compute these quantities.



# Summary

Bayesian networks: modular definition of large joint distribution over variables



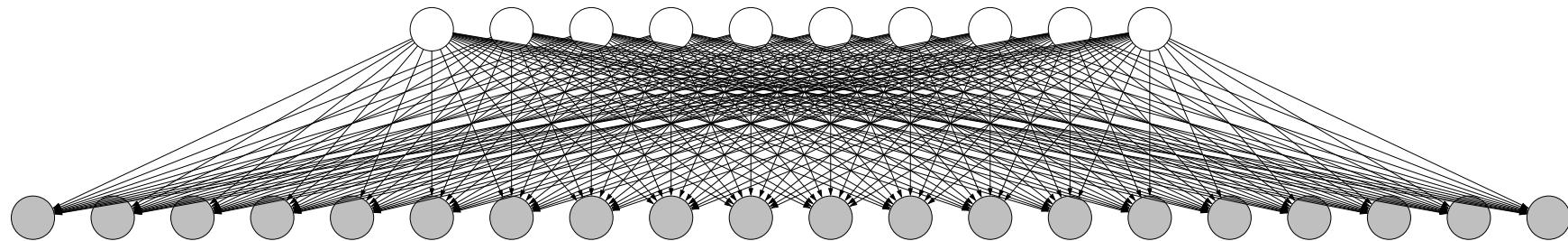
Probabilistic inference: condition on evidence, query variables of interest



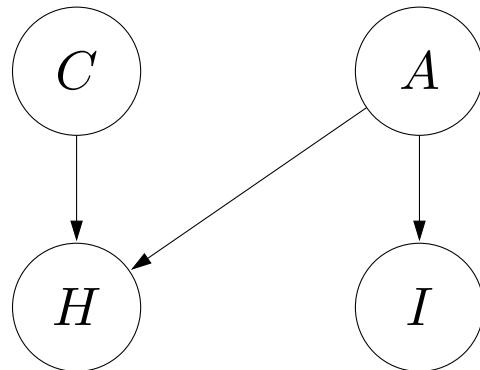
Next time: algorithms for probabilistic **inference**



# Lecture 14: Bayesian networks II



# Review: Bayesian network



$$\begin{aligned}\mathbb{P}(C = c, A = a, H = h, I = i) \\ \stackrel{\text{def}}{=} p(c)p(a)p(h | c, a)p(i | a)\end{aligned}$$



## Definition: Bayesian network

Let  $X = (X_1, \dots, X_n)$  be random variables.

A **Bayesian network** is a directed acyclic graph (DAG) that specifies a **joint distribution** over  $X$  as a product of **local conditional distributions**, one for each node:

$$\mathbb{P}(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^n p(x_i | x_{\text{Parents}(i)})$$

- Last time, we talked about Bayesian networks, which was a fun and convenient modeling framework. We posit a collection of variables that describe the state of the world, and then create a story on how the variables are generated (recall the probabilistic program interpretation).
- Think factor graphs + probability.
- A Bayesian network specifies two parts: (i) a graph structure which governs the qualitative relationship between the variables, and (ii) local conditional distributions, which specify the quantitative relationship.
- Formally, a Bayesian network defines a **joint** probability distribution over many variables (e.g.,  $\mathbb{P}(C, A, H, I)$ ) via the **local** conditional distributions (e.g.,  $p(i | a)$ ). This joint distribution specifies all the information we know about how the world works.

# Review: probabilistic inference

## Input

Bayesian network:  $\mathbb{P}(X_1 = x_1, \dots, X_n = x_n)$

Evidence:  $E = e$  where  $E \subseteq X$  is subset of variables

Query:  $Q \subseteq X$  is subset of variables



## Output

$\mathbb{P}(Q = q \mid E = e)$  for all values  $q$

Example: if coughing but no itchy eyes, have a cold?

$$\mathbb{P}(C \mid H = 1, I = 0)$$

- Think of the joint probability distribution defined by the Bayesian network as a guru. Probabilistic inference allows you to ask the guru anything: what is the probability of having a cold? What if I'm coughing? What if I don't have itchy eyes?
- In the last lecture, we performed probabilistic inference manually. In this lecture, we're going to build a guru that can answer these queries efficiently and automatically.



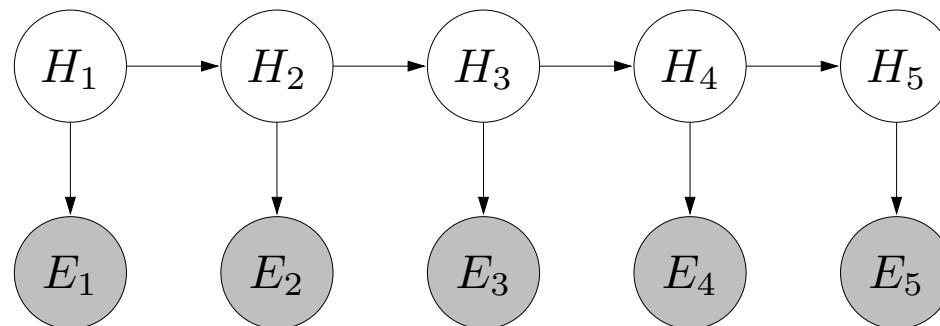
# Roadmap

Forward-backward

Particle filtering

Gibbs sampling

# Hidden Markov model



## Problem: object tracking

$H_i \in \{1, \dots, K\}$ : location of object at time step  $i$

$E_i \in \{1, \dots, K\}$ : sensor reading at time step  $i$

Start  $p(h_1)$ : e.g., uniform over all locations

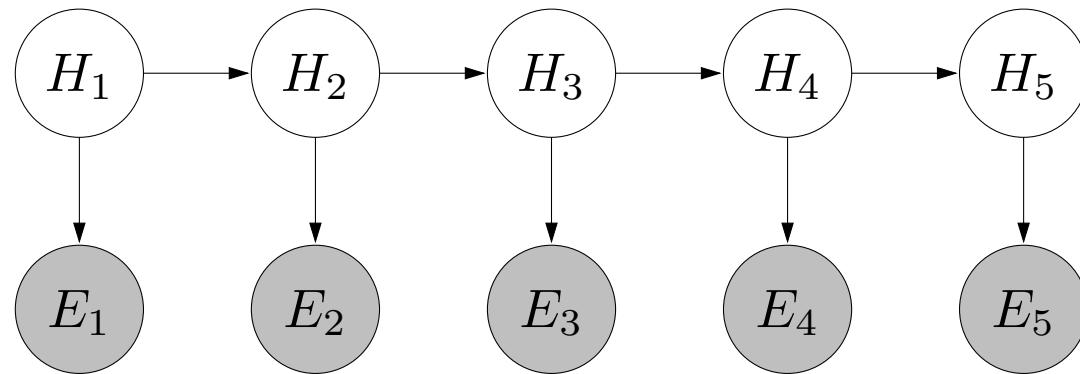
Transition  $p(h_i | h_{i-1})$ : e.g., uniform over adjacent loc.

Emission  $p(e_i | h_i)$ : e.g., uniform over adjacent loc.

$$\mathbb{P}(H = h, E = e) = \underbrace{p(h_1)}_{\text{start}} \prod_{i=2}^n \underbrace{p(h_i | h_{i-1})}_{\text{transition}} \prod_{i=1}^n \underbrace{p(e_i | h_i)}_{\text{emission}}$$

- So far, we have computed various ad-hoc probabilistic queries on ad-hoc Bayesian networks by hand. We will now switch gears and focus on the popular hidden Markov model (HMM), and show how the forward-backward algorithm can be used to compute typical queries of interest.
- As motivation, consider the problem of tracking an object. The probabilistic story is as follows: An object starts at  $H_1$  uniformly drawn over all possible locations. Then at each time step thereafter, it **transitions** to an adjacent location (e.g., with equal probability). For example, if  $H_2 = 3$ , then  $H_3 \in \{2, 4\}$  with equal probability. At each time step, we obtain a sensor reading  $E_i$  given  $H_i$  (e.g., also uniform over locations adjacent to  $H_i$ ).

# Hidden Markov model inference



Question (**filtering**):

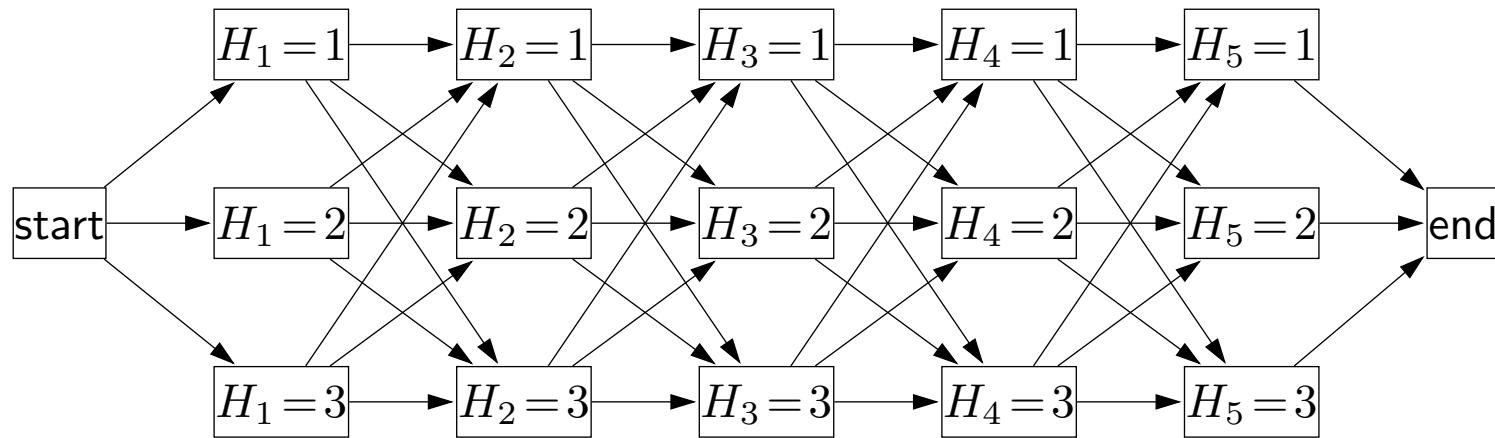
$$\mathbb{P}(H_3 \mid E_1 = e_1, E_2 = e_2, E_3 = e_3)$$

Question (**smoothing**):

$$\mathbb{P}(H_3 \mid E_1 = e_1, E_2 = e_2, E_3 = e_3, E_4 = e_4, E_5 = e_5)$$

- In principle, you could ask any type of questions on an HMM, but there are two common ones: filtering and smoothing.
- Filtering asks for the distribution of some hidden variable  $H_i$  conditioned on only the evidence up until that point. This is useful when you're doing real-time object tracking, and you can't see the future.
- Smoothing asks for the distribution of some hidden variable  $H_i$  conditioned on all the evidence, including the future. This is useful when you have collected all the data and want to retroactively go and figure out what  $H_i$  was.

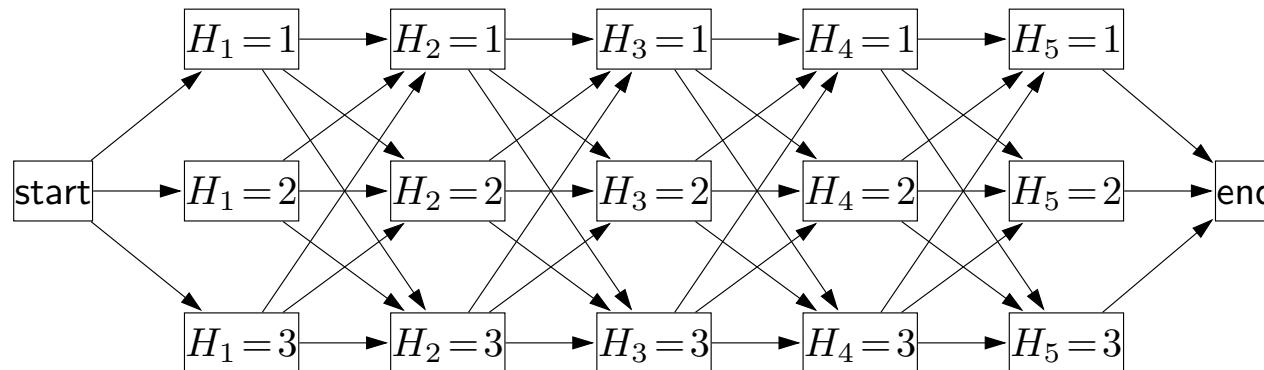
# Lattice representation



- Edge  $\boxed{\text{start}} \Rightarrow \boxed{H_1 = h_1}$  has weight  $p(h_1)p(e_1 | h_1)$
- Edge  $\boxed{H_{i-1} = h_{i-1}} \Rightarrow \boxed{H_i = h_i}$  has weight  $p(h_i | h_{i-1})p(e_i | h_i)$
- Each path from  $\boxed{\text{start}}$  to  $\boxed{\text{end}}$  is an assignment with weight equal to the product of node/edge weights

- Now let's actually compute these queries. We will do smoothing first. Filtering is a special case: if we're asking for  $H_i$  given  $E_1, \dots, E_i$ , then we can marginalize out the future, reducing the problem to a smaller HMM.
- A useful way to think about inference is returning to state-based models. Consider a graph with a start node, an end node, and a node for each assignment of a value to a variable  $H_i = v$ . The nodes are arranged in a lattice, where each column corresponds to one variable  $H_i$  and each row corresponds to a particular value  $v$ . Each path from the start to the end corresponds exactly to a complete assignment to the nodes.
- Note that in the reduction from a variable-based model to a state-based model, we have committed to an ordering of the variables.
- Each edge has a weight (a single number) determined by the local conditional probabilities (more generally, the factors in a factor graph). For each edge into  $H_i = h_i$ , we multiply by the transition probability into  $h_i$  and emission probability  $p(e_i | h_i)$ . This defines a weight for each path (assignment) in the graph equal to the joint probability  $P(H = h, E = e)$ .
- Note that the lattice contains  $O(nK)$  nodes and  $O(nK^2)$  edges, where  $n$  is the number of variables and  $K$  is the number of values in the domain of each variable.

# Lattice representation



Forward:  $F_i(h_i) = \sum_{h_{i-1}} F_{i-1}(h_{i-1})w(h_{i-1}, h_i)$

sum of weights of paths from  $\boxed{\text{start}}$  to  $\boxed{H_i = h_i}$

Backward:  $B_i(h_i) = \sum_{h_{i+1}} B_{i+1}(h_{i+1})w(h_i, h_{i+1})$

sum of weights of paths from  $\boxed{H_i = h_i}$  to  $\boxed{\text{end}}$

Define  $S_i(h_i) = F_i(h_i)B_i(h_i)$ :

sum of weights of paths from  $\boxed{\text{start}}$  to  $\boxed{\text{end}}$  through  $\boxed{H_i = h_i}$

- The point of bringing back the search-based view is that we can cast the probability queries we care about in terms of sums over paths, and effectively use dynamic programming.
- First, define the forward message  $F_i(v)$  to be the sum of the weights over all paths from the start node to  $H_i = v$ . This can be defined recursively: any path that goes  $H_i = h_i$  will have to go through some  $H_{i-1} = h_{i-1}$ , so we can sum over all possible values of  $h_{i-1}$ .
- Analogously, let the backward message  $B_i(v)$  be the sum of the weights over all paths from  $H_i = v$  to the end node.
- Finally, define  $S_i(v)$  to be the sum of the weights over all paths from the start node to the end node that pass through the intermediate node  $X_i = v$ . This quantity is just the product of the weights of paths going into  $H_i = h_i$  ( $F_i(h_i)$ ) and those leaving it ( $B_i(h_i)$ ).

# Lattice representation

Smoothing queries (marginals):

$$\mathbb{P}(H_i = h_i \mid E = e) \propto S_i(h_i)$$



## Algorithm: forward-backward algorithm

Compute  $F_1, F_2, \dots, F_n$

Compute  $B_n, B_{n-1}, \dots, B_1$

Compute  $S_i$  for each  $i$  and normalize

Running time:  $O(nK^2)$

- Let us go back to the smoothing queries:  $\mathbb{P}(H_i = h_i \mid E = e)$ . This is just gotten by normalizing  $S_i$ .
- The algorithm is thus as follows: for each node  $H_i = h_i$ , we compute three numbers:  $F_i(h_i), B_i(h_i), S_i(h_i)$ . First, we sweep forward to compute all the  $F_i$ 's recursively. At the same time, we sweep backward to compute all the  $B_i$ 's recursively. Then we compute  $S_i$  by pointwise multiplication.
- Implementation note: we technically can normalize  $S_i$  to get  $\mathbb{P}(H_i \mid E = e)$  at the very end but it's useful to normalize  $F_i$  and  $B_i$  at each step to avoid underflow. In addition, normalization of the forward messages yields  $\mathbb{P}(H_i = v \mid E_1 = e_1, \dots, E_i = e_i) \propto F_i(v)$ .



# Summary

- Lattice representation: paths are assignments (think state-based models)
- Dynamic programming: compute sums efficiently
- Forward-backward algorithm: share intermediate computations across different queries



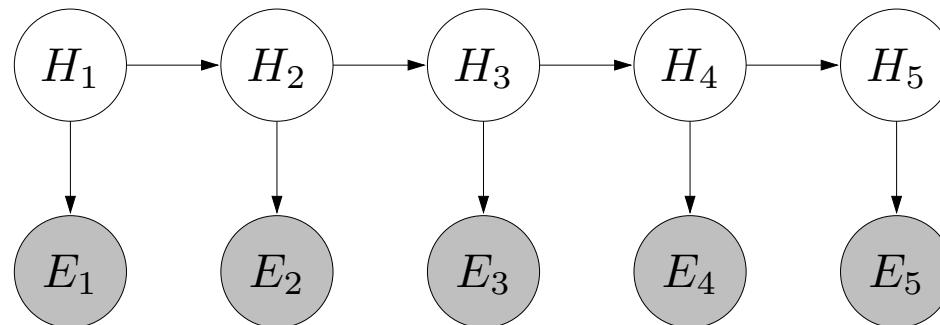
# Roadmap

Forward-backward

Particle filtering

Gibbs sampling

# Hidden Markov models



Query (**filtering**):

$$\mathbb{P}(H_1 \mid E_1 = e_1)$$

$$\mathbb{P}(H_2 \mid E_1 = e_1, E_2 = e_2)$$

$$\mathbb{P}(H_3 \mid E_1 = e_1, E_2 = e_2, E_3 = e_3)$$

**Motivation:** if  $H_i$  can take on many values, forward-backward is too slow ( $O(nK^2)$ )...

- The forward-backward algorithm runs in  $O(nK^2)$ , where  $K$  is the number of possible values (e.g., locations) that  $H_i$  can take on. This could be a very large number, which makes the forward-backward algorithm very slow (though not exponentially so).
- The motivation of particle filtering is to perform **approximate probabilistic inference**, and leveraging the fact that most of the  $K^2$  pairs are very improbable.
- Although particle filtering applies to general factor graphs, we will present them for hidden Markov models for concreteness.
- As the name suggests, we will use particle filtering for answering filtering queries.

# Review: beam search

Idea: keep  $\leq K$  **candidate list**  $C$  of partial assignments



## Algorithm: beam search

Initialize  $C \leftarrow [\{\}]$

For each  $i = 1, \dots, n$ :

Extend:

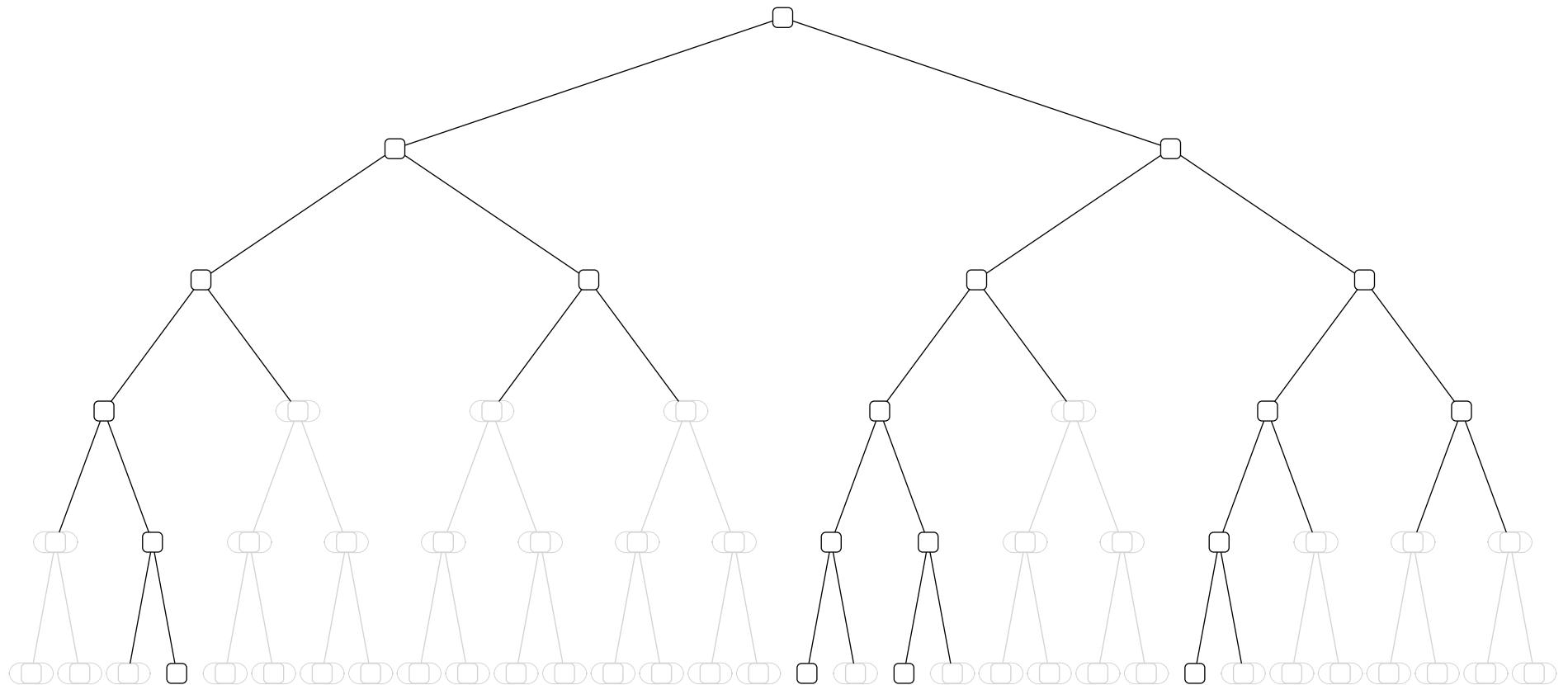
$$C' \leftarrow \{h \cup \{H_i : v\} : h \in C, v \in \text{Domain}_i\}$$

Prune:

$C \leftarrow K$  elements of  $C'$  with highest weights

[demo: beamSearch({K:3})]

# Review: beam search



Beam size  $K = 4$

- Recall that beam search effectively does a pruned BFS of the search tree of partial assignments, where at each level, we keep track of the  $K$  partial assignments with the highest weight.
- There are two phases. In the first phase, we **extend** all the existing candidates  $C$  to all possible assignments to  $H_i$ ; this results in  $K = |\text{Domain}_i|$  candidates  $C'$ . These  $C'$  are sorted by weight and **pruned** by taking the top  $K$ .

# Beam search

End result:

- Candidate list  $C$  is set of particles
- Use  $C$  to compute marginals

Problems:

- Extend: slow because requires considering every possible value for  $H_i$
- Prune: greedily taking best  $K$  doesn't provide diversity

Solution (3 steps): **propose, weight, resample**

- Beam search does generate a set of particles, but there are two problems.
- First, it can be slow if  $\text{Domain}_i$  is large because we have to try every single value. Perhaps we can be smarter about which values to try.
- Second, we are greedily taking the top  $K$  candidates, which can be too myopic. Can we somehow encourage more diversity?
- Particle filtering addresses both of these problems. There are three steps: propose, which extends the current parital assignment, and weight/resample, which redistributes resources on the particles based on evidence.

# Step 1: propose

Old particles:  $\approx \mathbb{P}(H_1, H_2 \mid E_1 = 0, E_2 = 1)$

[0, 1]

[1, 0]



**Key idea: proposal distribution**

For each old particle  $(h_1, h_2)$ , sample  $H_3 \sim p(h_3 \mid h_2)$ .

New particles:  $\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 1)$

[0, 1, 1]

[1, 0, 0]

- Suppose we have a set of particles that approximates the filtering distribution over  $H_1, H_2$ . The first step is to extend each current partial assignment (particle) from  $h_{1:i-1} = (h_1, \dots, h_{i-1})$  to  $h_{1:i} = (h_1, \dots, h_i)$ .
- To do this, we simply go through each particle and extend it stochastically, using the transition probability  $p(h_i | h_{i-1})$  to sample a new value of  $H_i$ .
- (For concreteness, think of what will happen if  $p(h_i | h_{i-1}) = 0.8$  if  $h_i = h_{i-1}$  and 0.2 otherwise.)
- We can think of advancing each particle according to the dynamics of the HMM. These particles approximate the probability of  $H_1, H_2, H_3$ , but still conditioned on the same evidence.

## Step 2: weight

Old particles:  $\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 1)$

[0, 1, 1]

[1, 0, 0]



**Key idea: weighting**

For each old particle  $(h_1, h_2, h_3)$ , weight it by  $w(h_1, h_2, h_3) = p(e_3 \mid h_3)$ .

New particles:

$\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 1, E_3 = 1)$

[0, 1, 1] (0.8)

[1, 0, 0] (0.4)

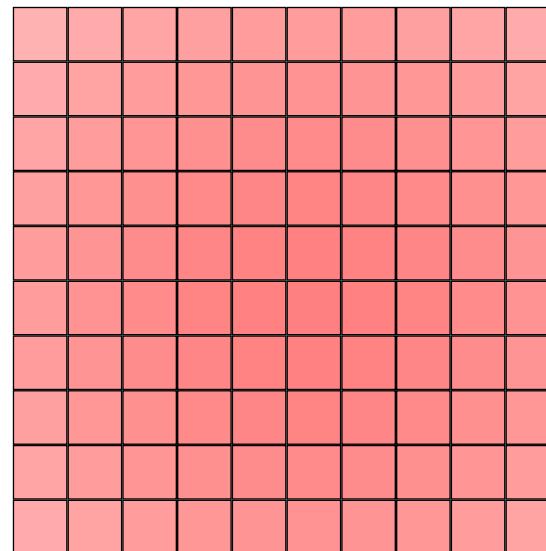
- Having generated a set of  $K$  candidates, we need to now take into account the new evidence  $E_i = e_i$ . This is a deterministic step that simply weights each particle by the probability of generating  $E_i = e_i$ , which is the emission probability  $p(e_i | h_i)$ .
- Intuitively, the proposal was just a guess about where the object will be  $H_3$ . To get a more realistic picture, we condition on  $E_3 = 1$ . Supposing that  $p(e_i = 1 | h_i = 1) = 0.8$  and  $p(e_i = 1 | h_i = 0) = 0.4$ , we then get the weights 0.8 and 0.4. Note that these weights do not sum to 1.

# Step 3: resample

Question: given weighted particles, which to choose?

Tricky situation:

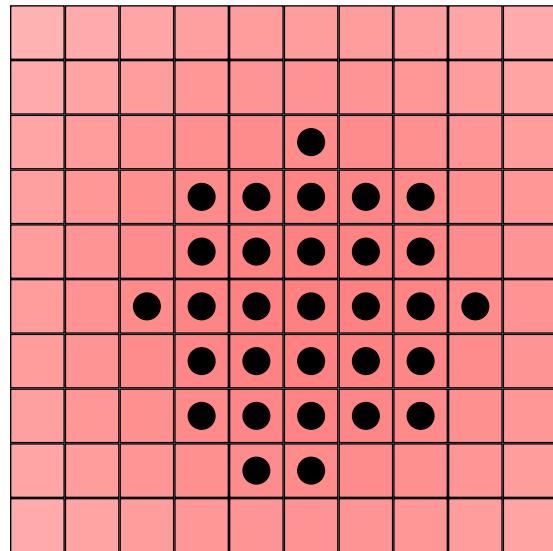
- Target distribution close to uniform
- Fewer particles than locations



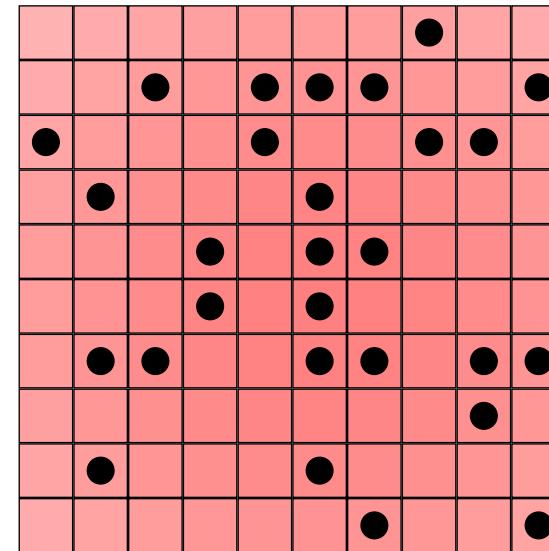
- Having proposed extensions to the particles and computed a weight for each particle, we now come to the question of which particles to keep.
- Intuitively, if a particle has very small weight, then we might want to prune it away. On the other hand, if a particle has high weight, maybe we should dedicate more resources to it.
- As a motivating example, consider an almost uniform distribution over a set of locations, and trying to represent this distribution with fewer particles than locations. This is a tough situation to be in.

# Step 3: resample

$K$  with highest weight



$K$  sampled from distribution



Intuition: top  $K$  assignments not representative.

Maybe random samples will be more representative...

- Beam search, which would choose the  $K$  locations with the highest weight, would clump all the particles near the mode. This is risky, because we have no support out farther from the center, where there is actually substantial probability.
- However, if we sample from the distribution which is proportional to the weights, then we can hedge our bets and get a more representative set of particles which cover the space more evenly.

# Step 3: resample



## Key idea: resampling

Given a distribution  $\mathbb{P}(A = a)$  with  $n$  possible values, draw a sample  $K$  times.

Intuition: redistribute particles to more promising areas



## Example: resampling

$a$	$\mathbb{P}(A = a)$
a1	0.70
a2	0.20
a3	0.05
a4	0.05



sample 1	a1
sample 2	a2
sample 3	a1
sample 4	a1

- After proposing and weighting, we end up with a set of samples  $h_{1:i}$ , each with some weight  $w(h_{1:i})$ . Intuitively, if  $w(h_{1:i})$  is really small, then it might not be worth keeping that particle around.
- Resampling allows us to put (possibly multiple) particles on high weight particles. In the example above, we don't sample a3 and a4 because they have low probability of being sampled.

## Step 3: resample

Old particles:

$$\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 1, E_3 = 1)$$

$$[0, 1, 1] (0.8) \Rightarrow 2/3$$

$$[1, 0, 0] (0.4) \Rightarrow 1/3$$

New particles:

$$\approx \mathbb{P}(H_1, H_2, H_3 \mid E_1 = 0, E_2 = 1, E_3 = 1)$$

$$[0, 1, 1]$$

$$[0, 1, 1]$$

- In our example, we normalize the particle weights to form a distribution over particles. Then we draw  $K = 2$  independent samples from that distribution. Higher weight particles will get more samples.

# Particle filtering



## Algorithm: particle filtering

Initialize  $C \leftarrow [\{\}]$

For each  $i = 1, \dots, n$ :

    Propose (extend):

$$C' \leftarrow \{h \cup \{H_i : h_i\} : h \in C, h_i \sim p(h_i \mid h_{i-1})\}$$

    Reweight:

        Compute weights  $w(h) = p(e_i \mid h_i)$  for  $h \in C'$

    Resample (prune):

$C \leftarrow K$  elements drawn independently from  $\propto w(h)$

[demo: `particleFiltering({K:100})`]

- The final algorithm here is very similar to beam search. We go through all the variables  $H_1, \dots, H_n$ .
- For each candidate  $h_{i-1} \in C$ , we propose  $h_i$  according to the transition distribution  $p(h_i | h_{i-1})$ .
- We then weight this particle using  $w(h) = p(e_i | h_i)$ .
- Finally, we select  $K$  particles from  $\propto w(h)$  by sampling  $K$  times independently.

# Particle filtering: implementation

- If only care about last  $H_i$ , collapse all particles with same  $H_i$

001  $\Rightarrow$  1

101  $\Rightarrow$  1

010  $\Rightarrow$  0

010  $\Rightarrow$  0

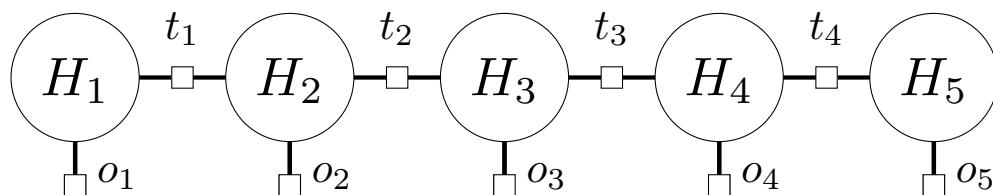
110  $\Rightarrow$  0

- If many particles are the same, can just store counts

1  
1  
0  $\Rightarrow$  1 : 2  
0 : 3  
0  
0

- In particle filtering as it is currently defined, each particle is an entire trajectory in the context of object tracking (assignment to all the variables).
- Often in tracking applications, we only care about the last location  $H_i$ , and the HMM is such that the future  $(H_{i+1}, \dots, H_n)$  is conditionally independent of  $H_1, \dots, H_{i-1}$  given  $H_i$ . Therefore, we often just store the value of  $H_i$  rather than its entire ancestry.
- When we only keep track of the  $H_i$ , we might have many particles that have the same value, so it can be useful to store just the counts of each value rather than having duplicates.

# Application: object tracking



## Example: object tracking

- $H_i$ : position of object at  $i$
- Transitions:  $t_i(h_i, h_{i-1}) = [h_i \text{ near } h_{i-1}]$
- Observations:  $o_i(h_i)$  = sensor reading...

$$\mathbb{P}(H = h) \propto \prod_{i=1}^n o_i(h_i) t_i(h_i, h_{i-1})$$

# Particle filtering demo

[see web version]

- Consider a tracking application where an object is moving around in a grid and we are trying to figure out its location  $H_i \in \{1, \dots, \text{grid-width}\} \times \{1, \dots, \text{grid-height}\}$ .
- The transition factors say that from one time step to the next, the object is equally likely to have moved north, south, east, west, or stayed put.
- Each observation is a location on the grid (a yellow dot). The observation factor is a user-defined function which depends on the vertical and horizontal distance.
- Play around with the demo to get a sense of how particle filtering works, especially the different observation factors.



# Roadmap

Forward-backward

Particle filtering

**Gibbs sampling**

# Gibbs sampling

Setup:

$\text{Weight}(x)$



## Algorithm: Gibbs sampling

Initialize  $x$  to a random complete assignment

Loop through  $i = 1, \dots, n$  until convergence:

    Compute weight of  $x \cup \{X_i : v\}$  for each  $v$

    Choose  $x \cup \{X_i : v\}$  with probability prop. to weight

[demo]

- Recall that Gibbs sampling proceeds by going through each variable  $X_i$ , considering all the possible assignments of  $X_i$  with some  $v \in \text{Domain}_i$ , computing the weight of the resulting assignment  $x \cup \{X_i : v\}$ , and choosing an assignment with probability proportional to the weight.
- We first introduced Gibbs sampling in the context of local search to get out of local optima.

# Gibbs sampling

Setup:

$$\mathbb{P}(X = x) \propto \text{Weight}(x)$$

## Gibbs sampling (probabilistic interpretation)

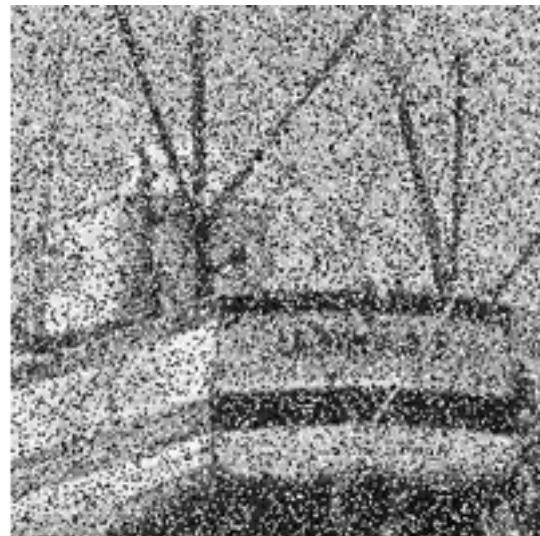
Initialize  $x$  to a random complete assignment

Loop through  $i = 1, \dots, n$  until convergence:

Set  $X_i = v$  with prob.  $\mathbb{P}(X_i = v \mid X_{-i} = x_{-i})$   
 $(X_{-i}$  denotes all variables except  $X_i$ )

- Now, we will use it for its original purpose, which is to draw samples from a probability distribution. (In particular, our goal is to draw from the joint distribution over  $X_1, X_2, \dots, X_n$ .) To do this, we need to define a probability distribution given an arbitrary factor graph. Recall that a general factor graph defines a  $\text{Weight}(x)$ , which is the product of all its factors. We can simply normalize the weight to get a distribution:  $\mathbb{P}(X = x) \propto \text{Weight}(x)$ . Then, Gibbs sampling is exactly sampling according to the conditional distribution  $\mathbb{P}(X_i = v | X_{-i} = x_{-i})$ .
- Note that the convergence criteria is on the *distribution* of the sample values, not the values themselves.
- Advanced: Gibbs sampling is an instance of a Markov Chain Monte Carlo (MCMC) algorithm which generates a sequence of particles  $X^{(1)}, X^{(2)}, X^{(3)}, \dots$ . A Markov chain is irreducible if there is positive probability of getting from any assignment to any other assignment (now the probabilities are over the random choices of the sampler). When the Gibbs sampler is irreducible, then in the limit as  $t \rightarrow \infty$ , the distribution of  $X^{(t)}$  converges to the true distribution  $\mathbb{P}(X)$ . MCMC is a very rich topic which we will not talk about very much here.

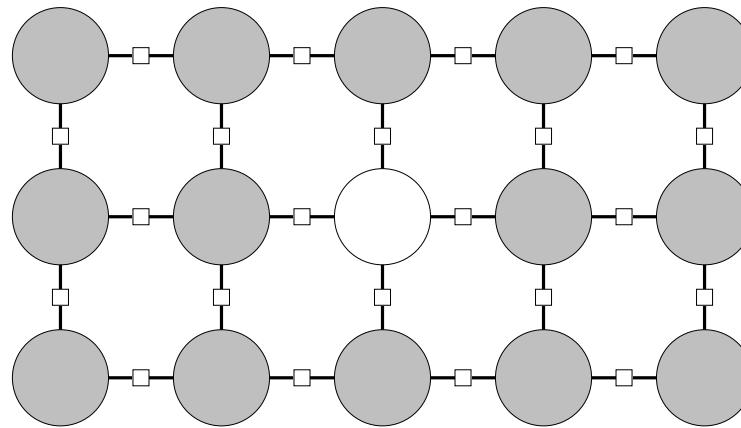
# Application: image denoising



# Application: image denoising



## Example: image denoising



- $X_i \in \{0, 1\}$  is pixel value in location  $i$
- Subset of pixels are observed

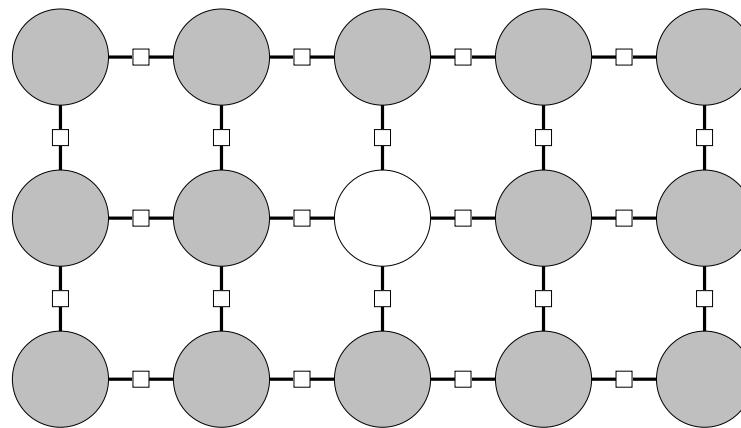
$$o_i(x_i) = [x_i = \text{observed value at } i]$$

- Neighboring pixels more likely to be same than different

$$t_{ij}(x_i, x_j) = [x_i = x_j] + 1$$

- Factor graphs (sometimes referred to as Markov random fields) play an important role in computer vision applications. Here we take a look at a very simple image denoising application and construct a factor graph.
- We assume that we have observed some fraction of the pixels in an image, and we wish to recover the pixels which have been removed. Our simple factor graph has two types of factors.
- First,  $o_i(x_i) = 0$  if the pixel at  $i$  is observed and  $x_i$  does not match it. This is analogous to the emission probabilities in an HMM, but here, the factor is deterministic (0 or 1).
- Second,  $t_{ij}(x_i, x_j)$  exists for every pair of neighboring pixels, and encourages them to agree (both be 0 or both be 1). Weight 2 is given to those pairs which are the same and 1 if the pair is different. This is analogous to the transition probabilities in an HMM.

# Application: image denoising



[whiteboard]



## Example: image denoising

If neighbors are 1, 1, 1, 0 and  $X_i$  not observed:

$$\mathbb{P}(X_i = 1 \mid X_{-i} = x_{-i}) = \frac{2 \cdot 2 \cdot 2 \cdot 1}{2 \cdot 2 \cdot 2 \cdot 1 + 1 \cdot 1 \cdot 1 \cdot 2} = 0.8$$

If neighbors are 0, 1, 0, 1 and  $X_i$  not observed:

$$\mathbb{P}(X_i = 1 \mid X_{-i} = x_{-i}) = \frac{1 \cdot 2 \cdot 1 \cdot 2}{1 \cdot 2 \cdot 1 \cdot 2 + 2 \cdot 1 \cdot 2 \cdot 1} = 0.5$$

- Let us compute the Gibbs sampling update. We go through each pixel  $X_i$  and try to update its value. Specifically, we condition on  $X_{-i} = x_{-i}$  being the current value.
- To compute the conditional  $\mathbb{P}(X_i = x_i \mid X_{-i} = x_{-i})$ , we only need to multiply in the factors that are touching  $X_i$ , since conditioning disconnects the graph completely. Many of the factors simply can be ignored.
- For example, suppose we are updating variable  $X_i$ , which has neighbors  $X_j, X_k, X_l, X_m$ . Suppose we condition on  $X_j = 1, X_k = 1, X_l = 0, X_m = 1$ . Then  $\mathbb{P}(X_i = x_i \mid X_{-i} = x_{-i}) \propto t_{ij}(x_i, 1)t_{ik}(x_i, 1)t_{il}(x_i, 0)t_{im}(x_m, 1)o_i(x_i)$ . We can compute the RHS for both  $x_i = 1$  and  $x_i = 0$ , and then normalize to get the conditional probability 0.8.
- Intuitively, the neighbors are all trying to pull  $X_i$  towards their values, and 0.8 reflects the fact that the pull towards 1 is stronger.

# Gibbs sampling: demo

[see web version]

- Try playing with the demo by modifying the settings to get a feeling for what Gibbs sampling is doing. Each iteration corresponds to resampling each pixel (variable).
- When you hit ctrl-enter for the first time, red and black correspond to 1 and 0, and white corresponds to unobserved.
- `showMarginals` allows you to either view the particles produced or the marginals estimated from the particles (this gives you a smoother probability estimate of what the pixel values are).
- If you increase `missingFrac`, the problem becomes harder.
- If you set `coherenceFactor` to 1, this is equivalent to turning off the edge factors.
- If you set `icm` to true, we will use local search rather than Gibbs sampling, which produces very bad solutions.
- In summary, Gibbs sampling is an algorithm that can be applied to any factor graph. It performs a guided random walk in the space of all possible assignments. Under some mild conditions, Gibbs sampling is guaranteed to converge to a sample from the true distribution  $\mathbb{P}(X = x) \propto \text{Weight}(x)$  but this might take until the heat death of the universe for complex models. Nonetheless, Gibbs sampling is widely used due to its simplicity and can work reasonably well.



# Probabilistic inference

Model (Bayesian network or factor graph):

$$\mathbb{P}(X = x) = \prod_{i=1}^n p(x_i \mid x_{\text{Parents}(i)})$$

Probabilistic inference:

$$\mathbb{P}(Q \mid E = e)$$

Algorithms:

- Forward-backward: HMMs, exact
- Particle filtering: HMMs, approximate
- Gibbs sampling: general, approximate

Next time: learning

# BAYESIAN NETWORKS

**CS221 Fall 2019**

Dhruv Kedia

Jon Kotker



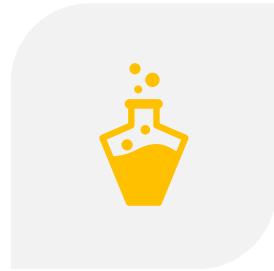
# What are we covering today?



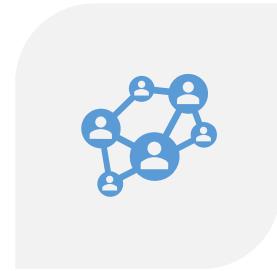
INTRODUCTION TO  
BAYESIAN NETWORKS



PROBABILISTIC  
INFERENCE COOKBOOK

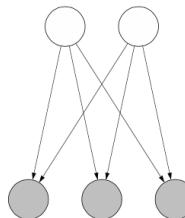


SAMPLE PROBLEMS



CONDITIONAL  
INDEPENDENCE

# Bayesian Networks



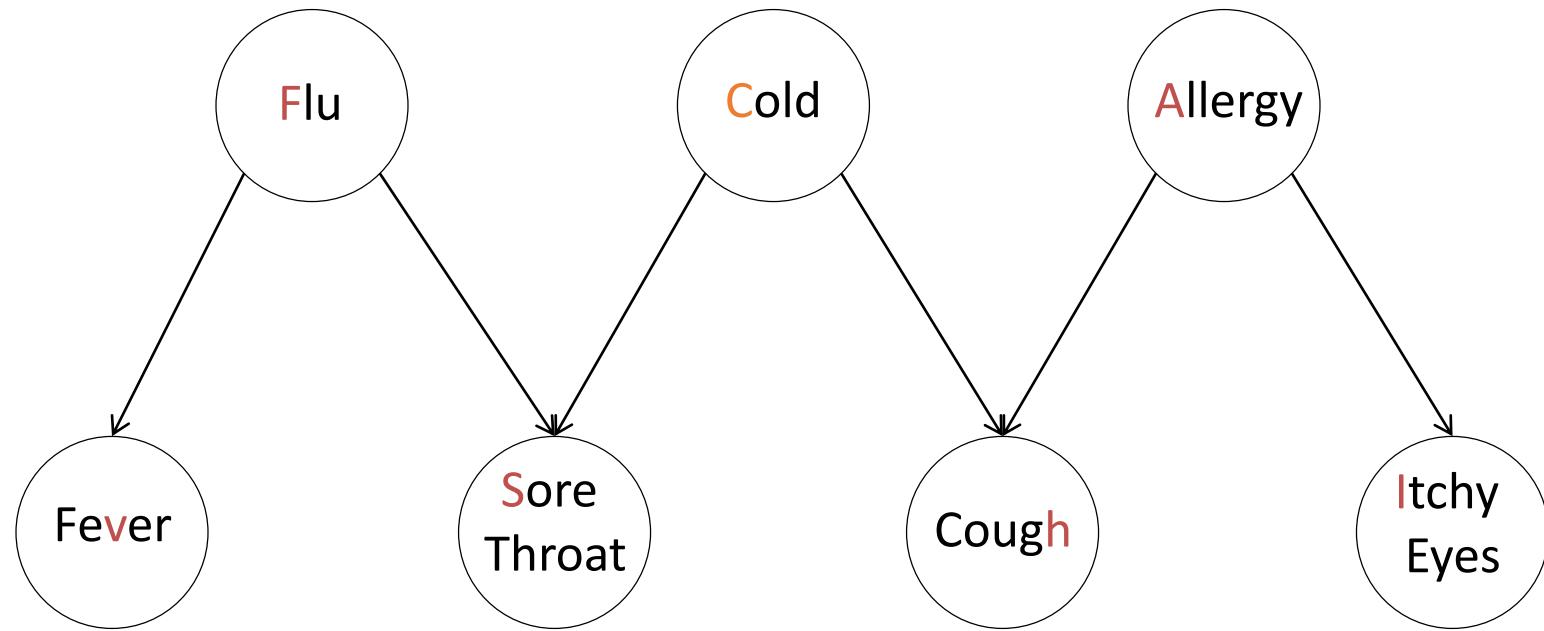
## Definition: Bayesian network

Let  $X = (X_1, \dots, X_n)$  be random variables.

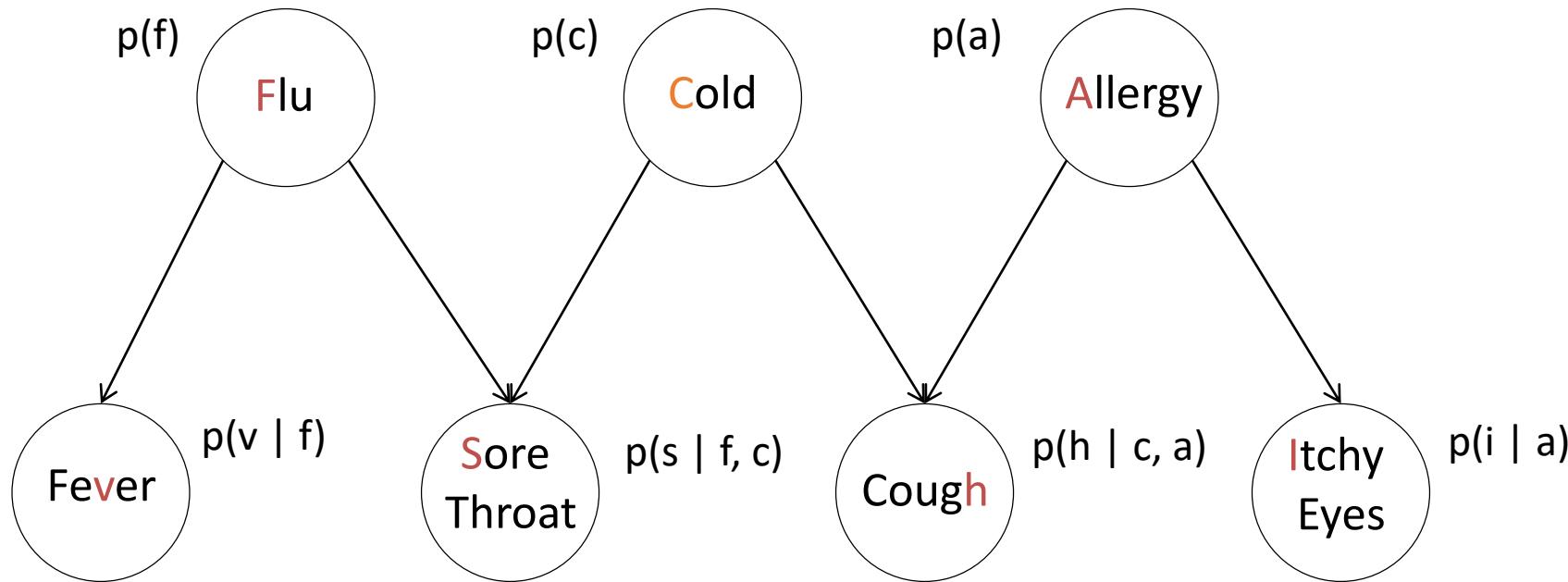
A **Bayesian network** is a directed acyclic graph (DAG) that specifies a **joint distribution** over  $X$  as a product of **local conditional distributions**, one for each node:

$$\mathbb{P}(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^n p(x_i | x_{\text{Parents}(i)})$$

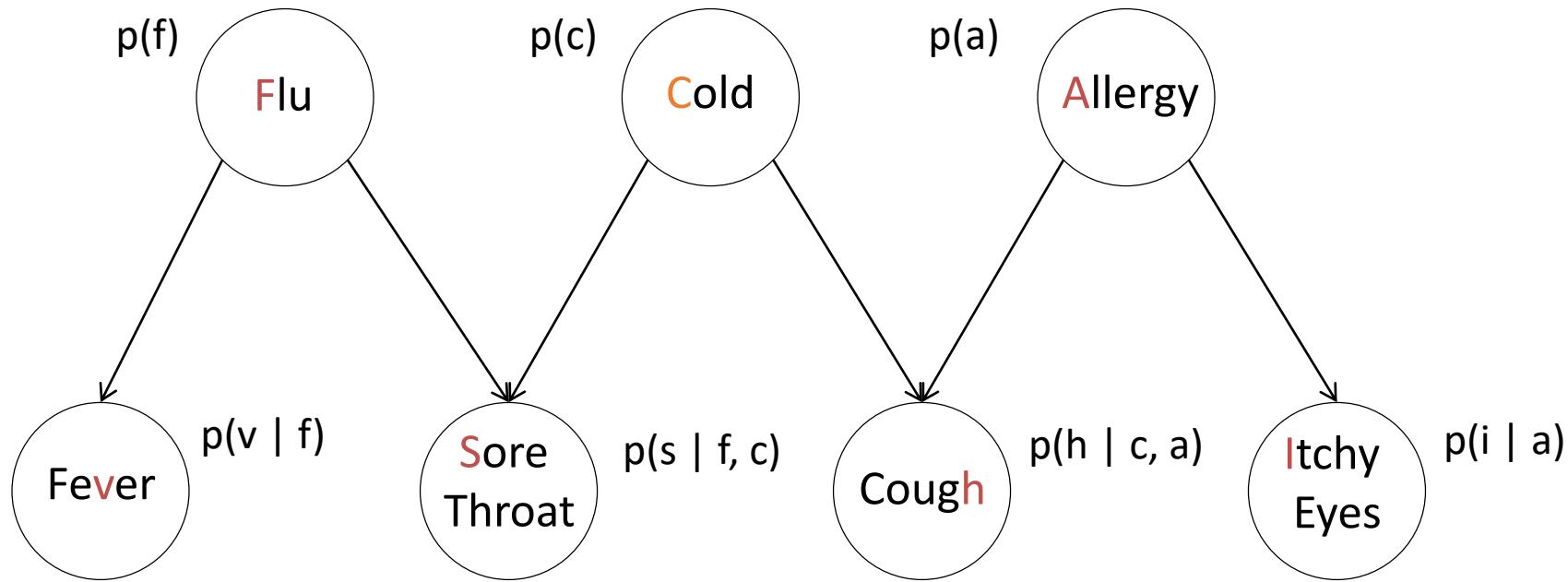
# Bayesian Networks



A Bayesian network represents a joint probability distribution.



# A Bayesian network represents a joint probability distribution.



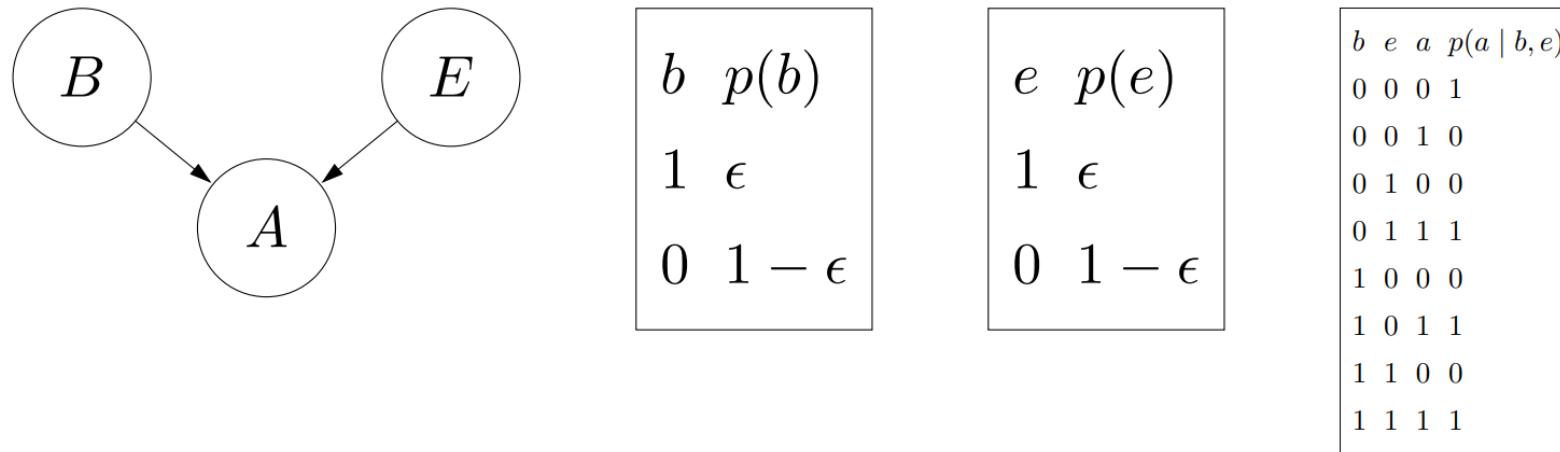
$$P(F=f, C=c, A=a, V=v, S=s, H=h, I=i) = p(f) p(c) p(a) p(v | f) p(s | f, c) p(h | c, a) p(i | a)$$

# Probabilistic Inference Cookbook

Given a query  $P(Q | E = e)$ :

1. Remove (marginalize) variables not ancestors of Q or E.
2. Convert Bayesian network to factor graph.
3. Condition (shade nodes / disconnect) on  $E = e$ .
4. Remove (marginalize) nodes disconnected from Q.
5. Run probabilistic inference algorithm (manual, variable elimination, Gibbs sampling, particle filtering).

# Example: alarm



[whiteboard]

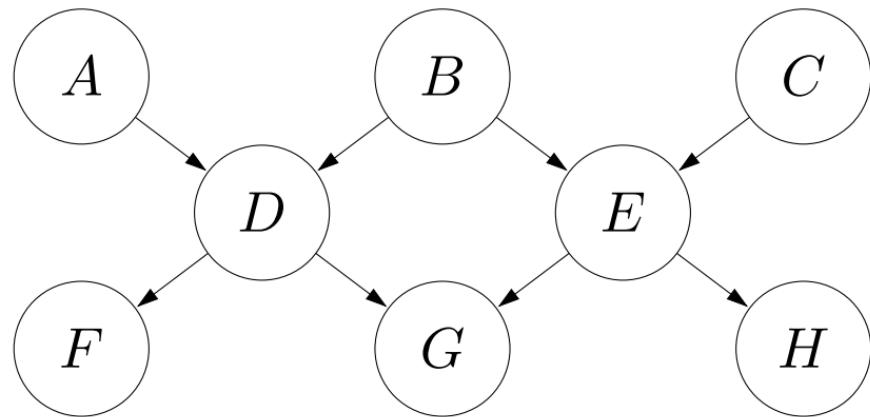
Query:  $\mathbb{P}(B)$

- Marginalize out  $A, E$

Query:  $\mathbb{P}(B | A = 1)$

- Condition on  $A = 1$

# Example: A-H (section)



[whiteboard]

Query:  $\mathbb{P}(C \mid B = b)$

- Marginalize out everything else, note  $C \perp\!\!\!\perp B$

Query:  $\mathbb{P}(C, H \mid E = e)$

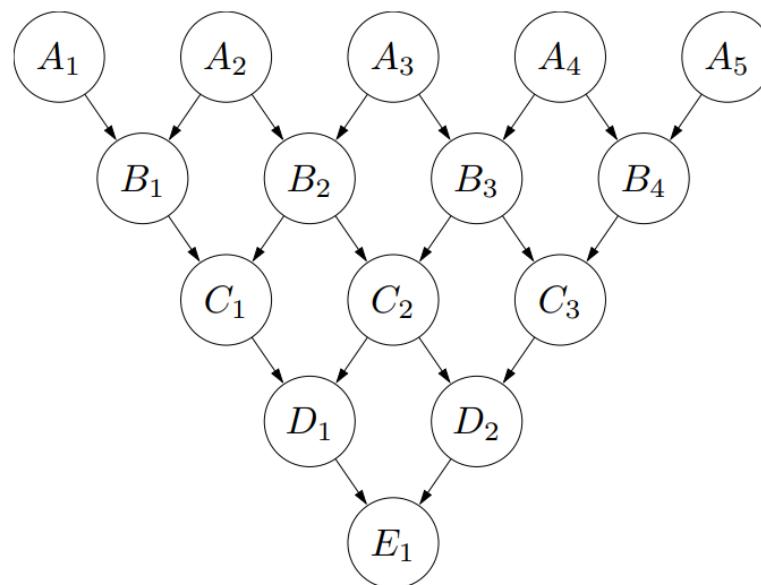
- Marginalize out  $A, D, F, G$ , note  $C \perp\!\!\!\perp H \mid E$

# Bayesian Lights

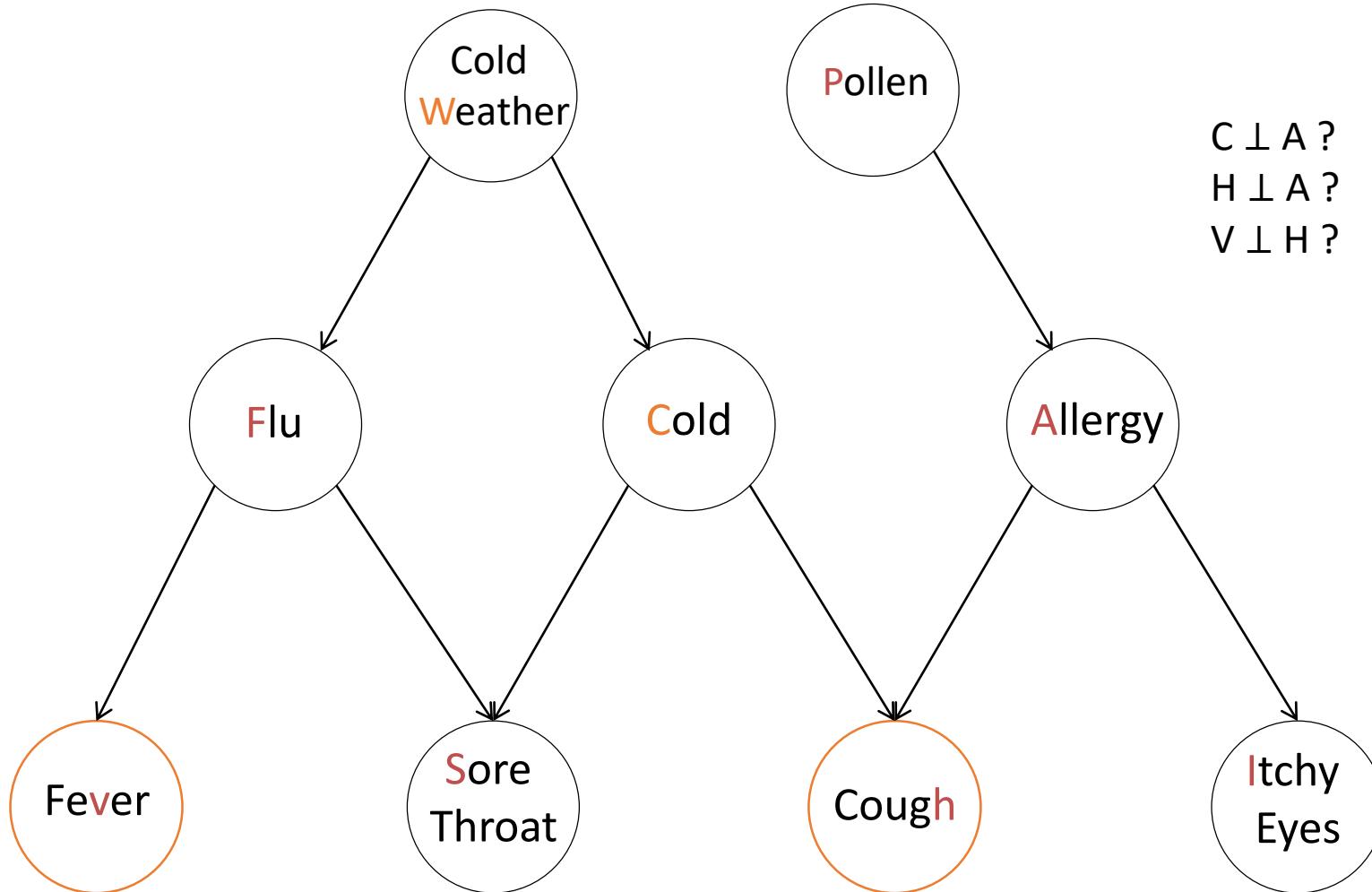
(Fall 2016 Midterm)

This holiday season, you decide to put your knowledge of Bayesian networks to good use. You decide to create Bayesian Lights™, an arrangement of lights that turn on and off randomly according to the joint distribution of a Bayesian network.

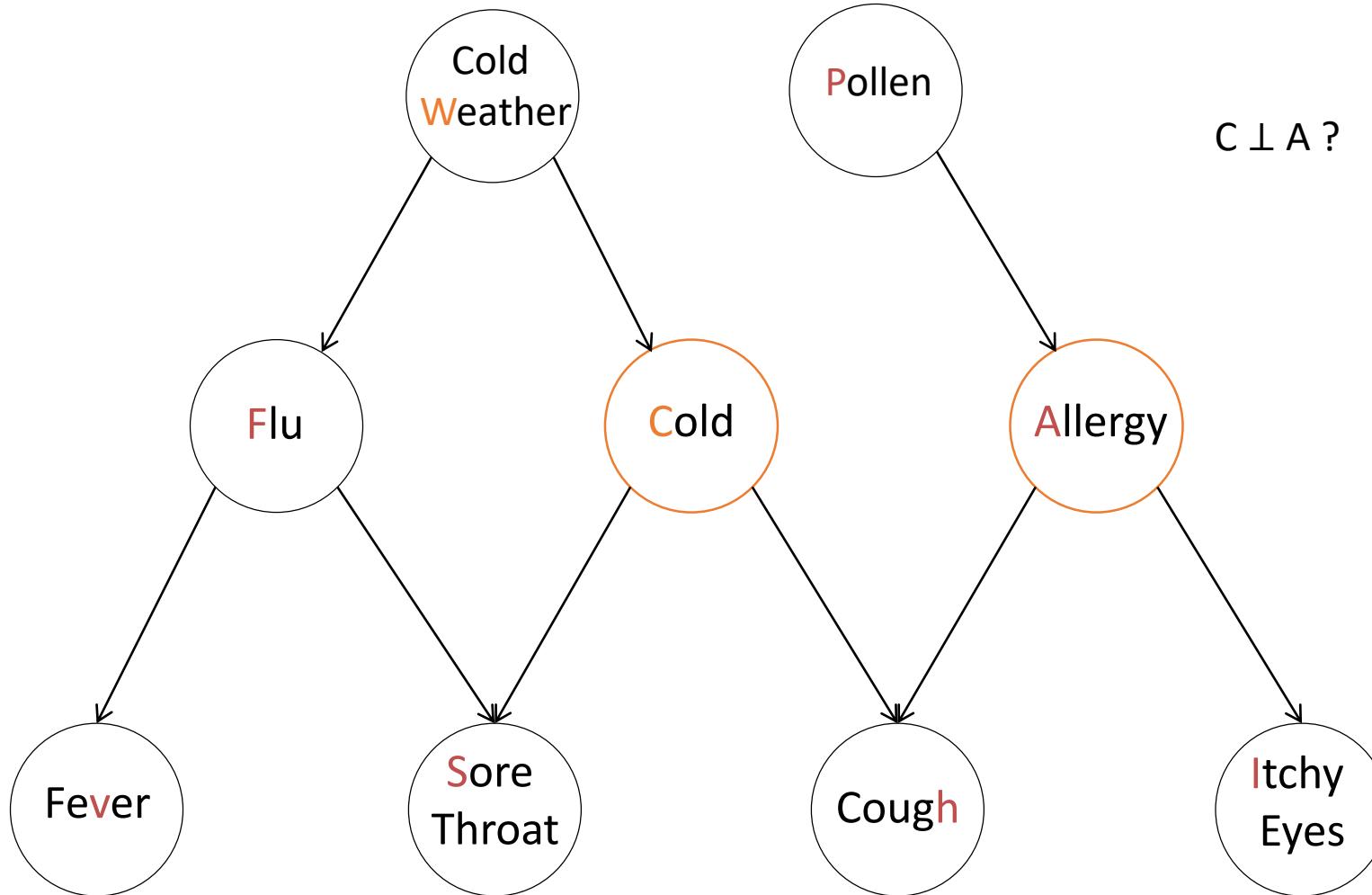
Figure 1 shows the Bayesian network corresponding to the lights. Each light is associated with a variable which takes on values in  $\{0, 1\}$  (off: 0, on: 1). For example,  $A_1$  is the light in the upper-left corner and  $E_1$  is the light at the very bottom. A light in the top row is on with probability  $\alpha$ . The status of a light in subsequent rows is governed by the two parent lights directly above it, and it is on with probability  $\beta$  when the two parent lights above it have different statuses (on-off or off-on), and off with probability  $\beta$  when the two parent lights above it have the same status. In other words, each light is the result of applying a noisy XOR function.



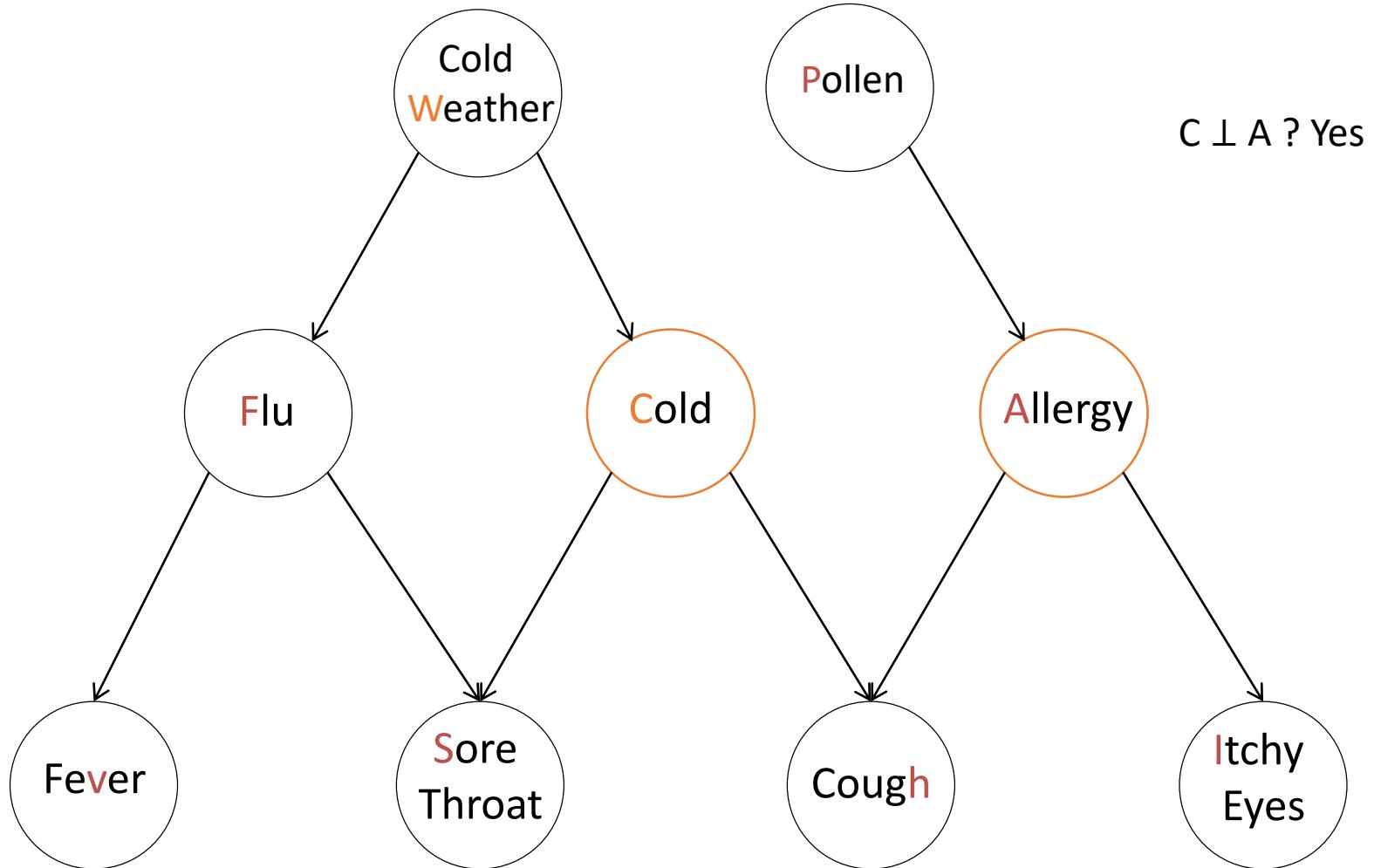
# Conditional Independence



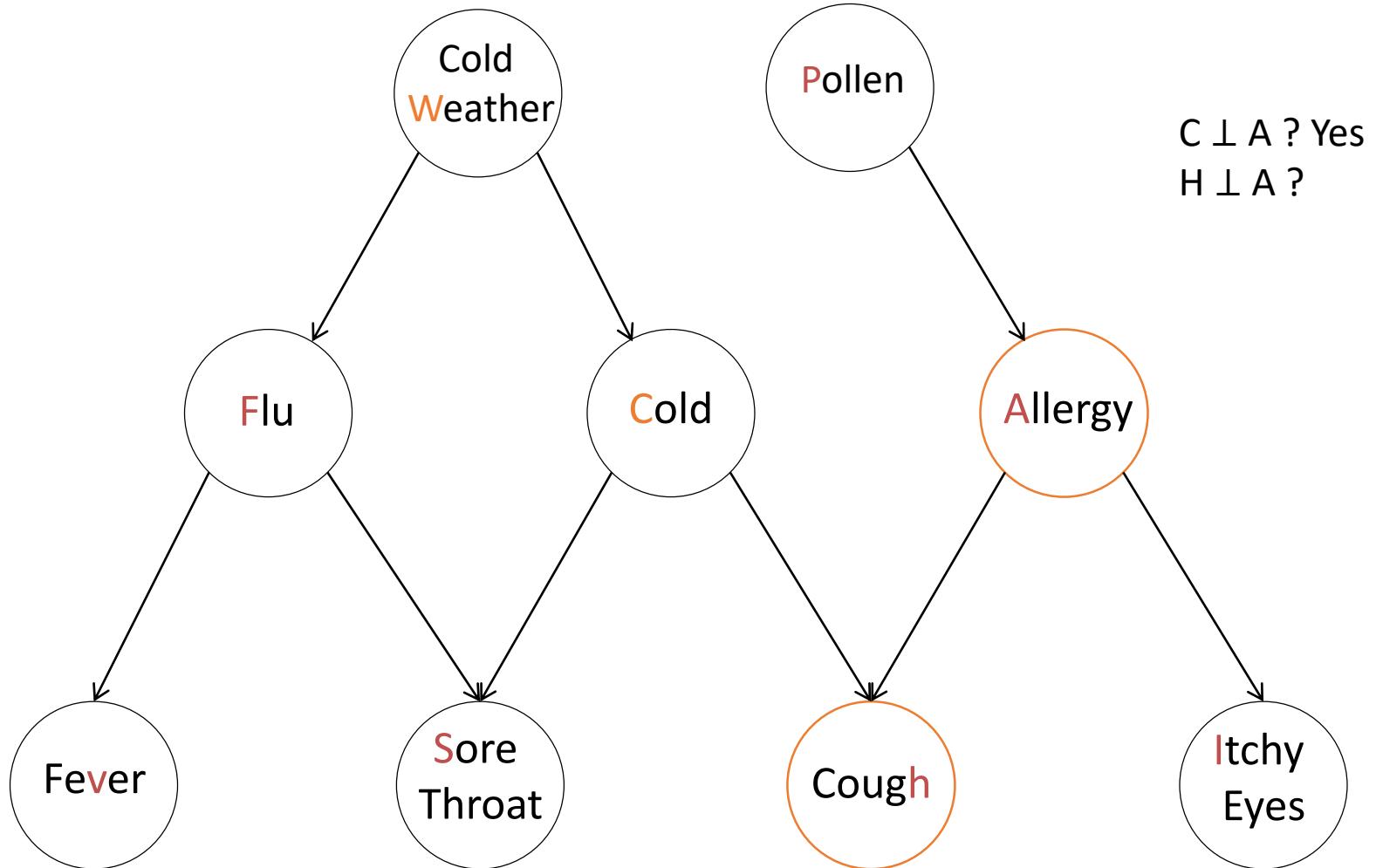
# Conditional Independence



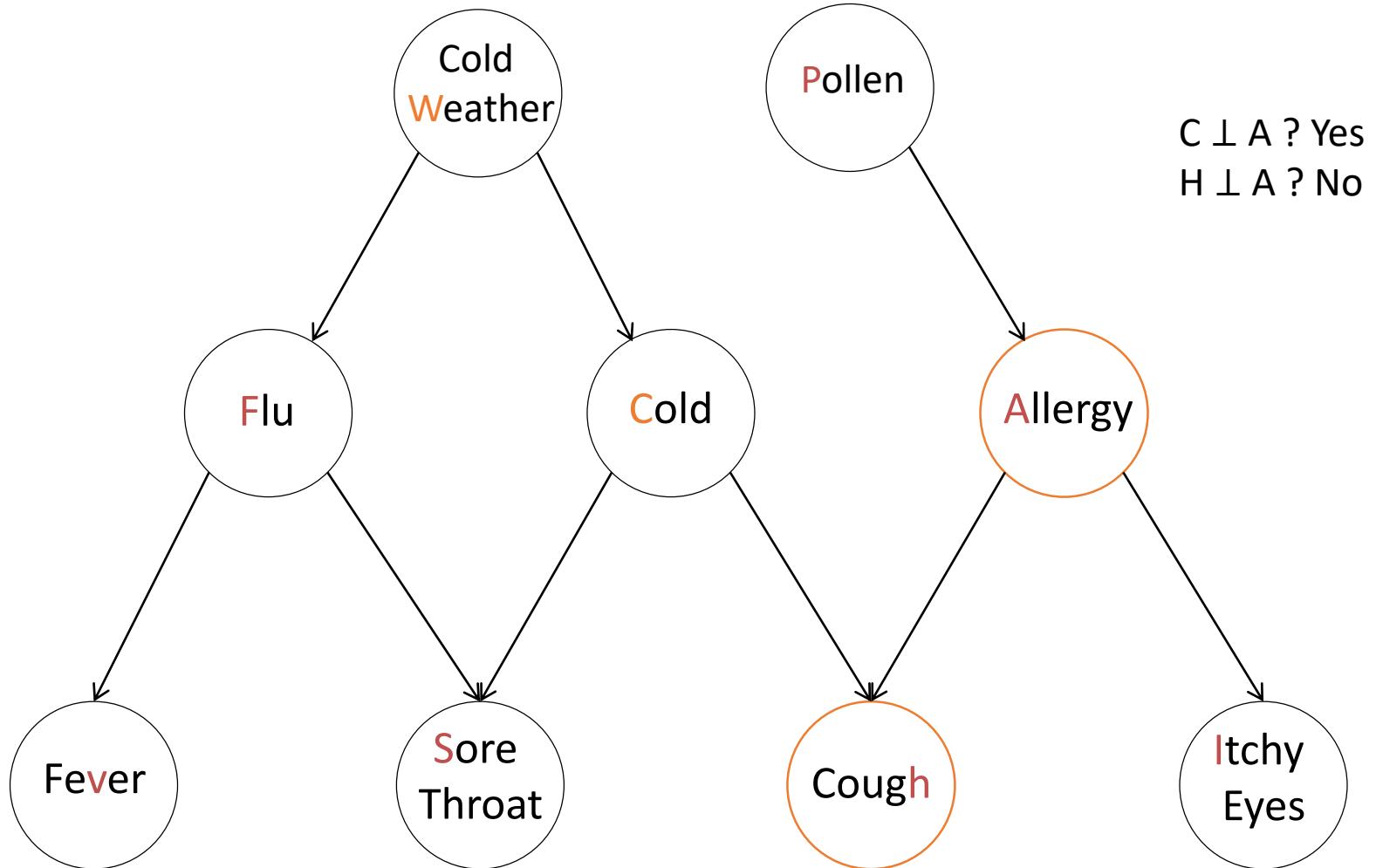
# Conditional Independence



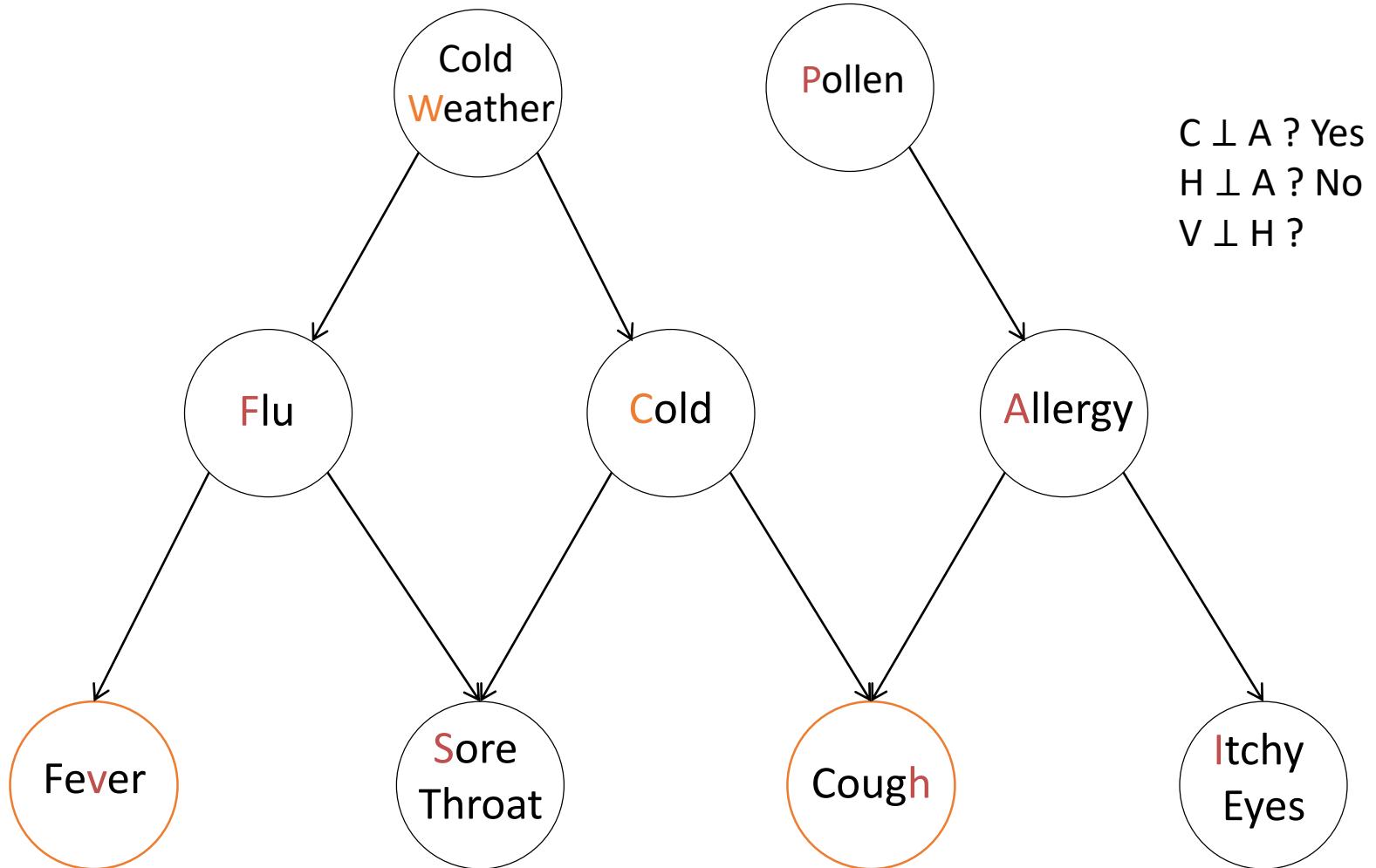
# Conditional Independence



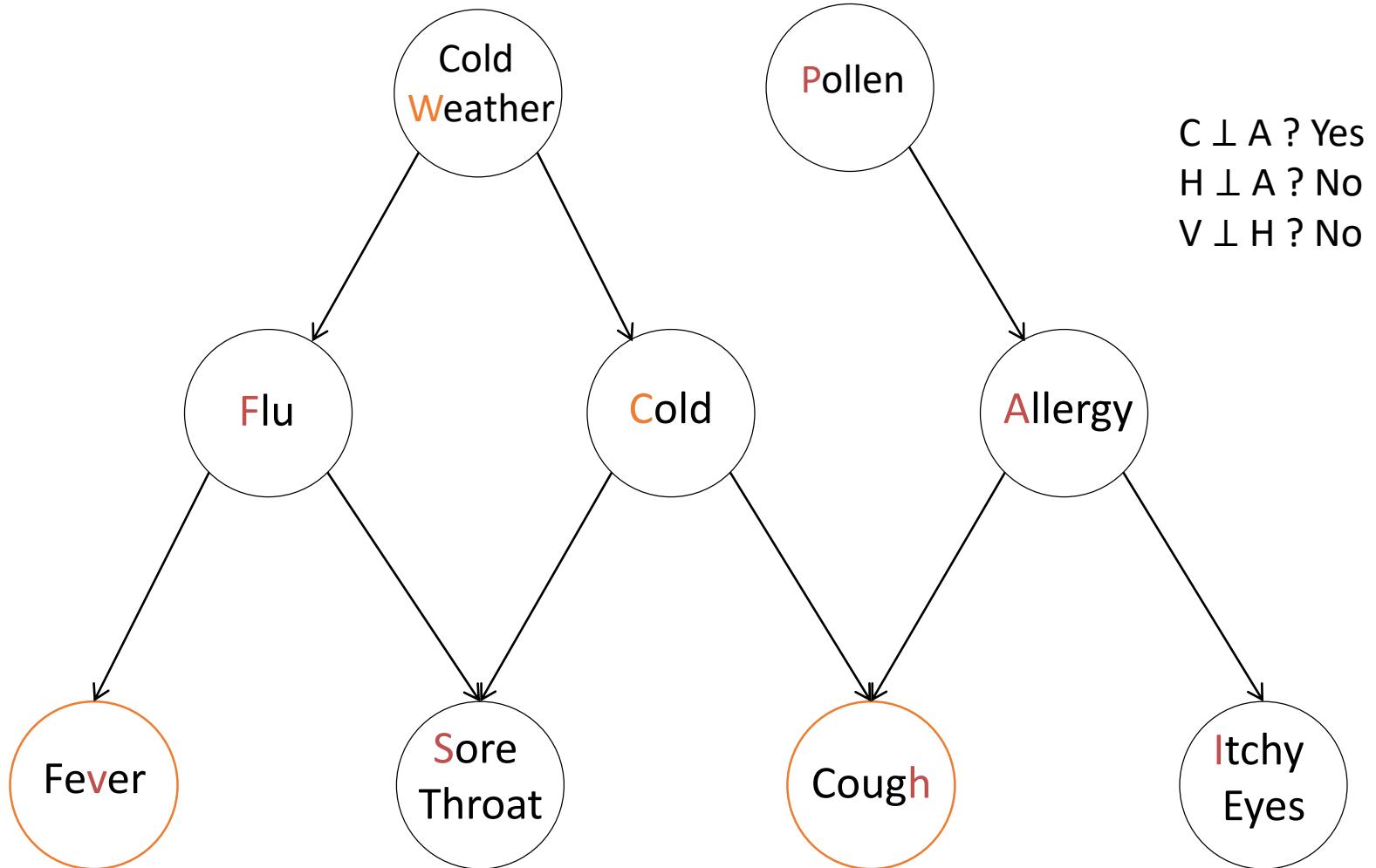
# Conditional Independence



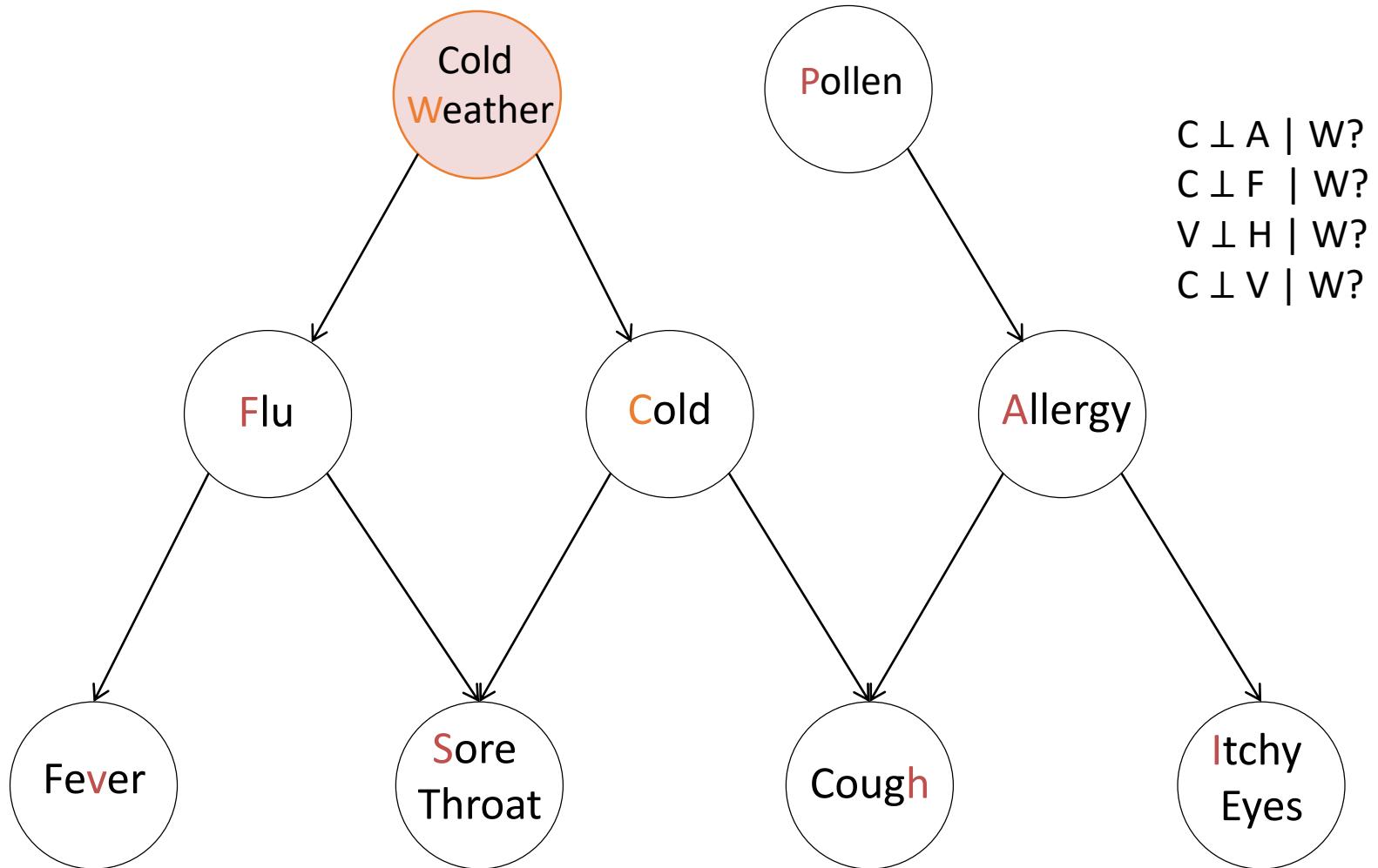
# Conditional Independence



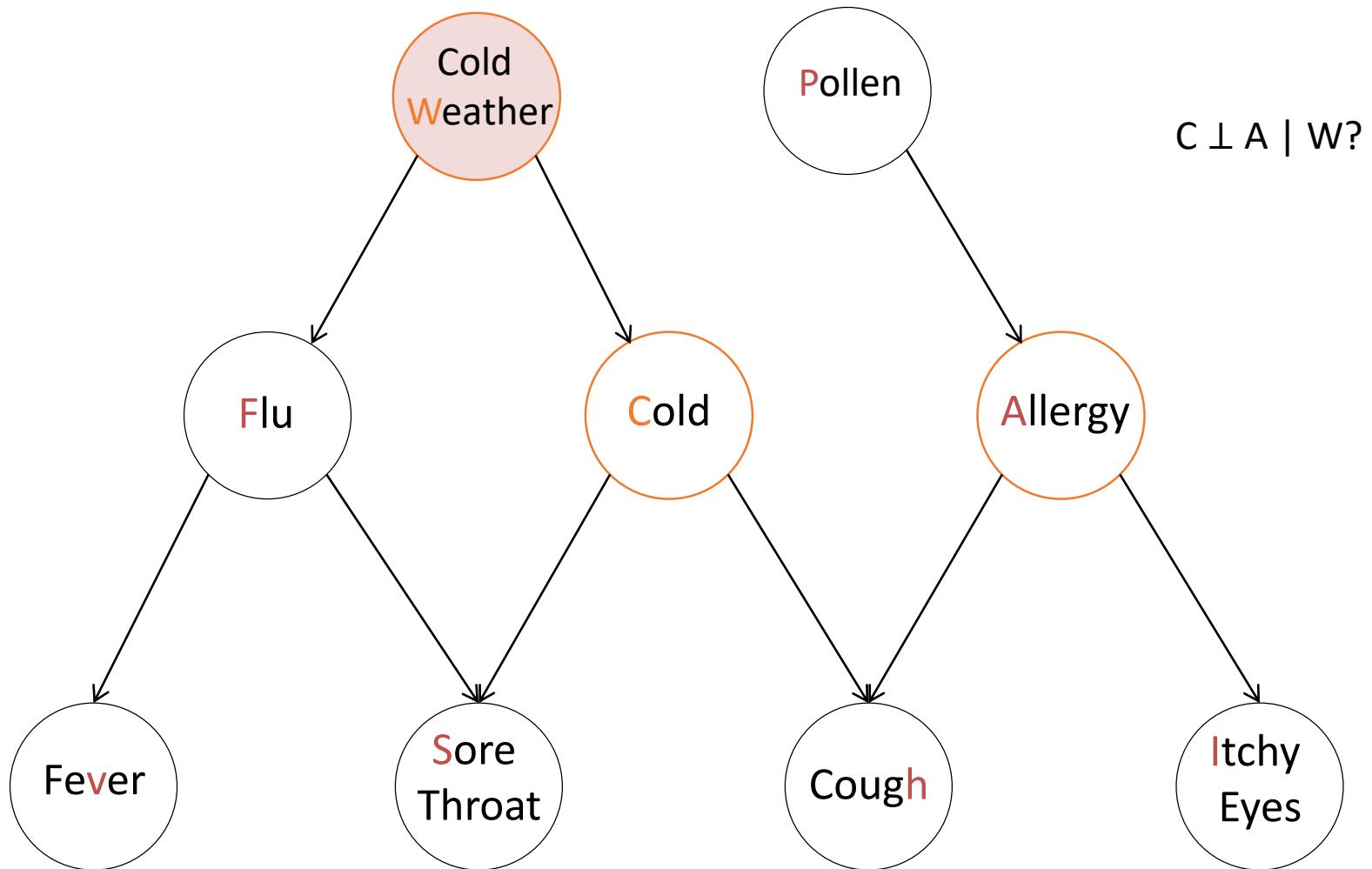
# Conditional Independence



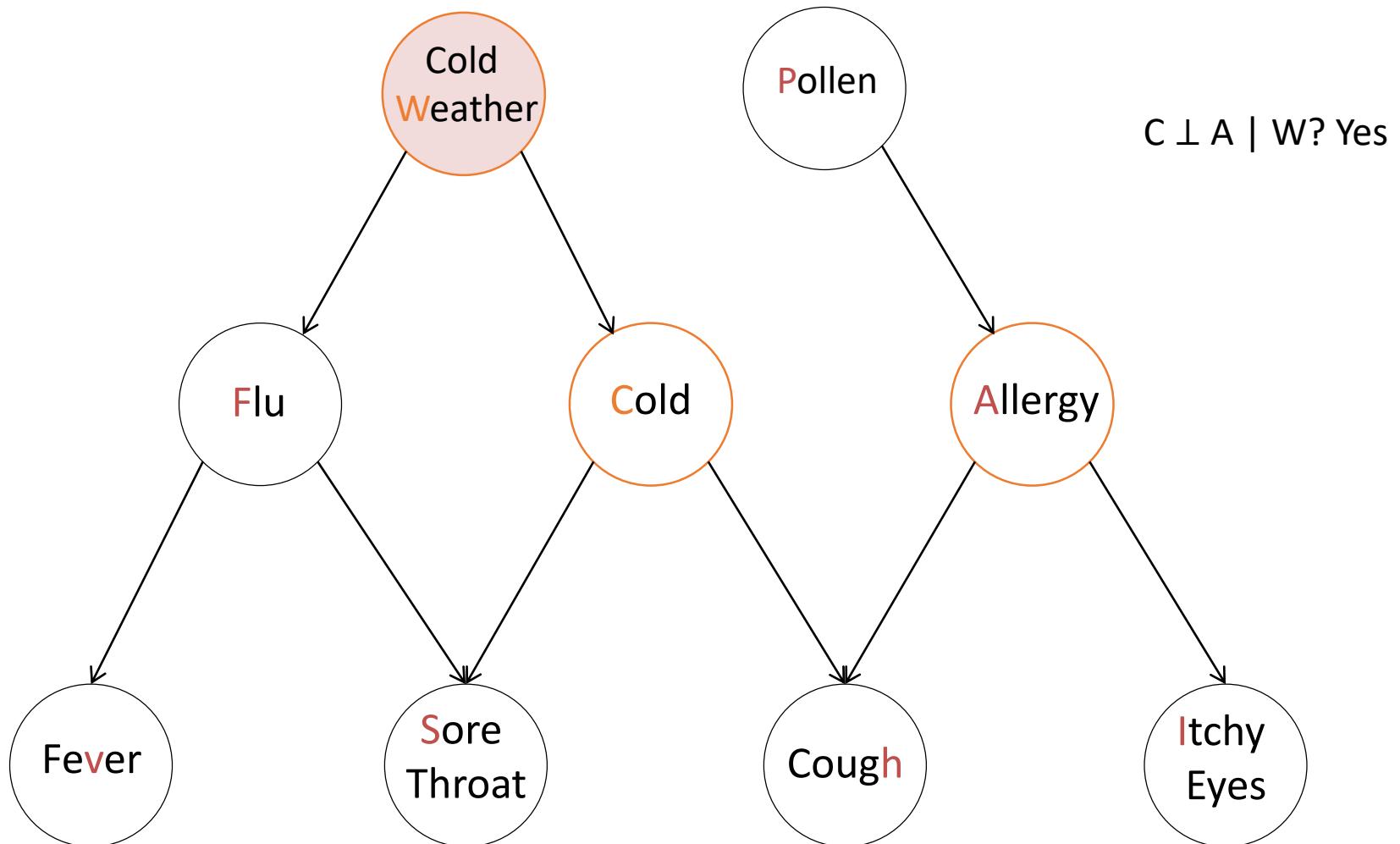
# Conditional Independence



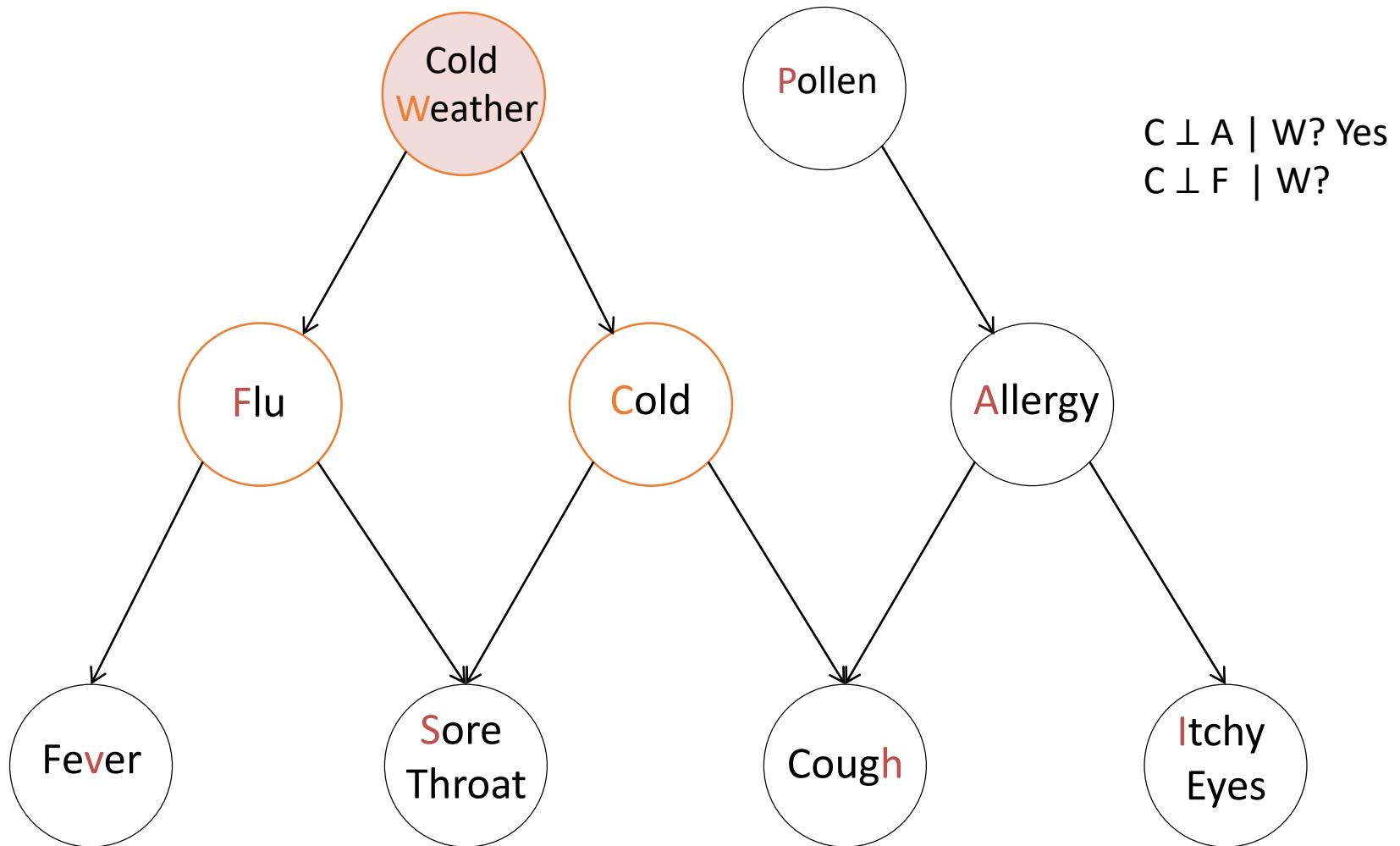
# Conditional Independence



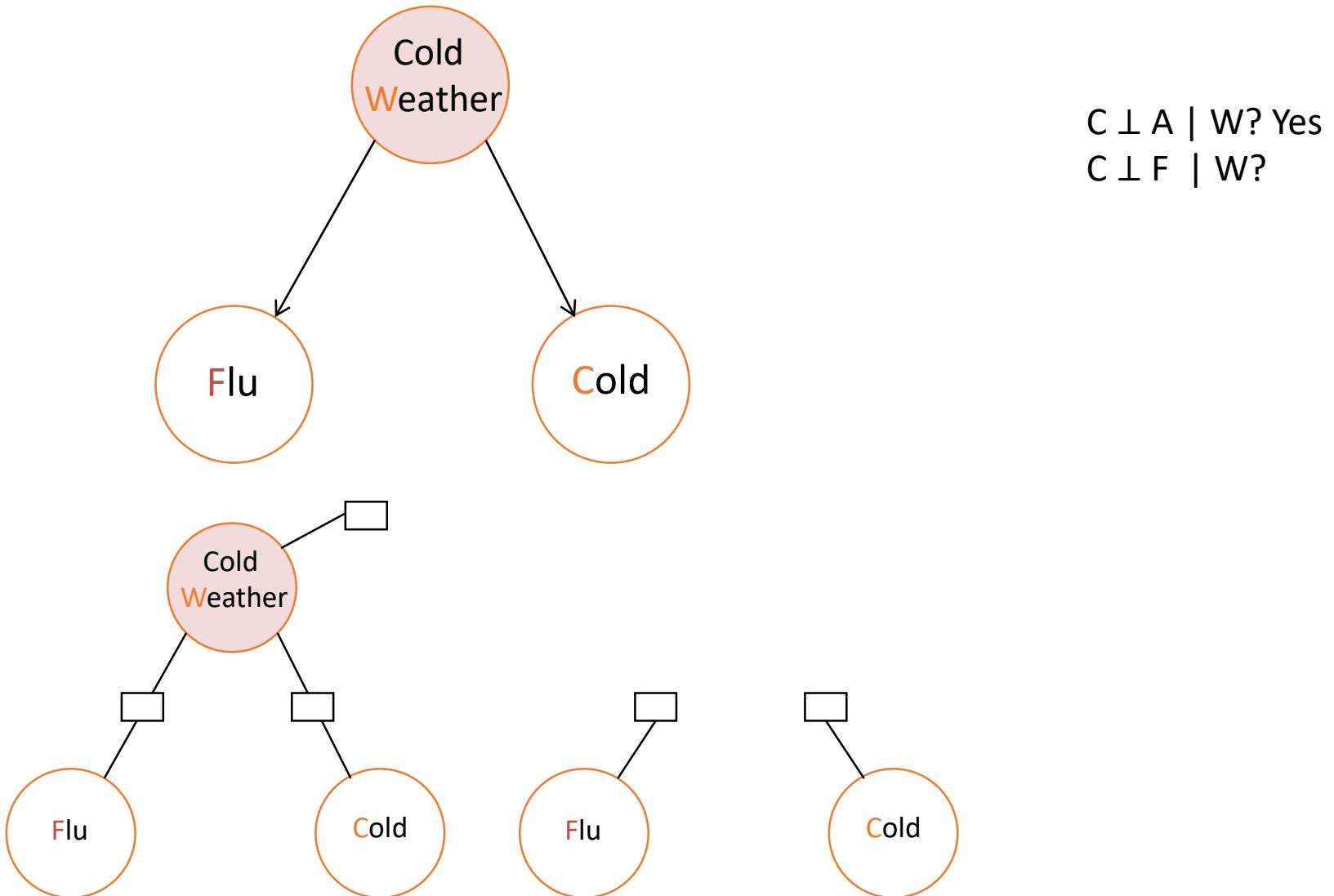
# Conditional Independence



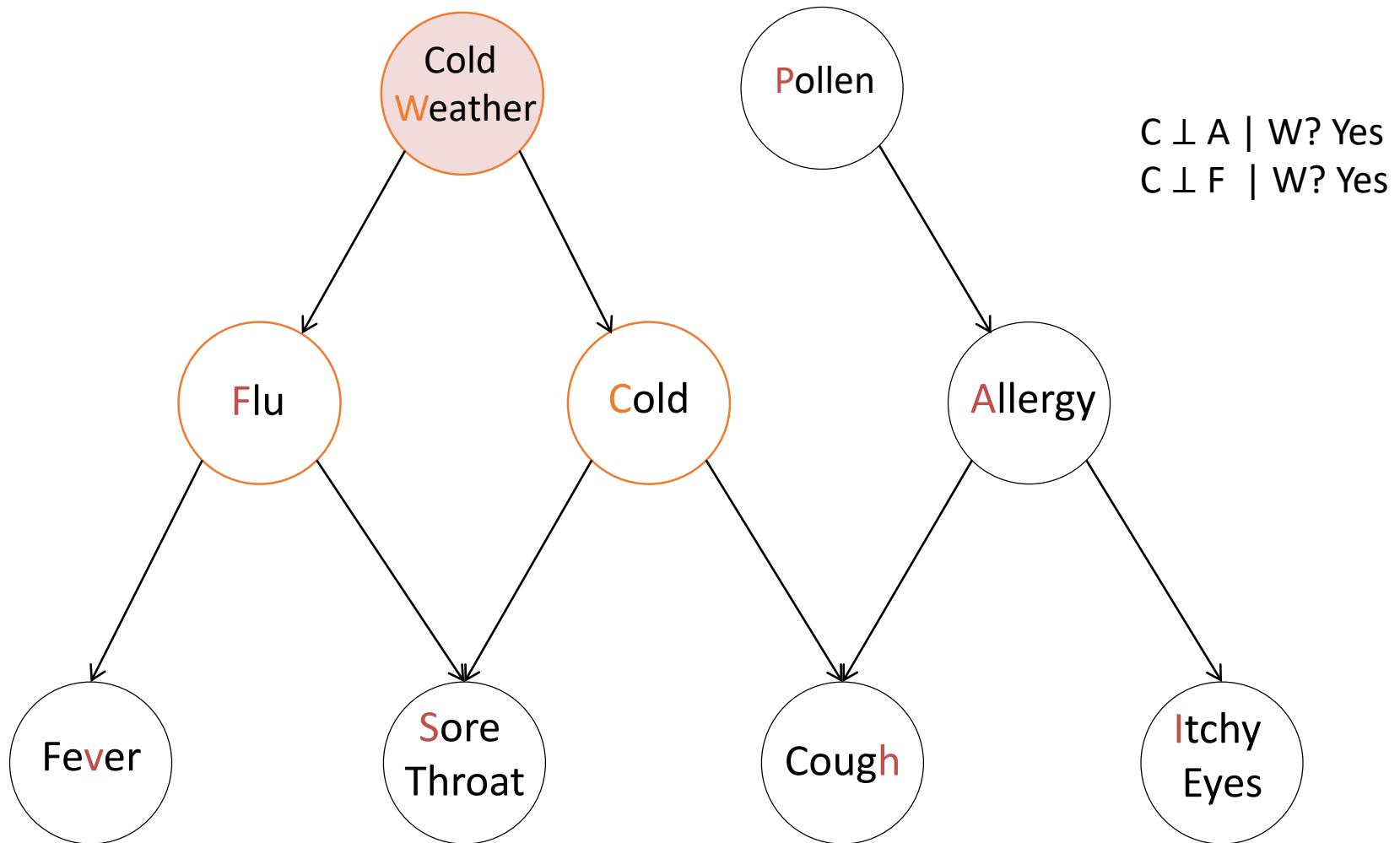
# Conditional Independence



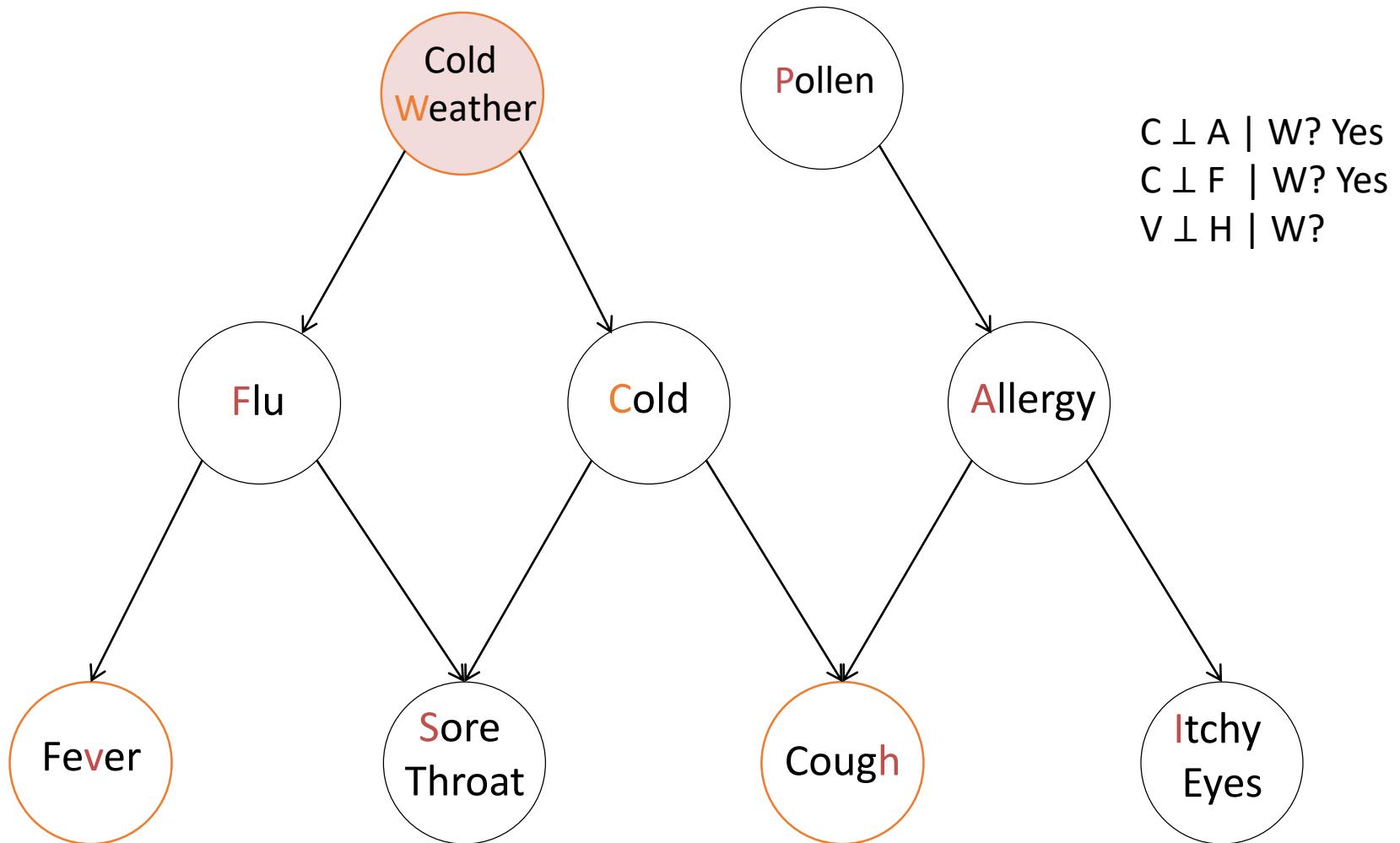
# Conditional Independence



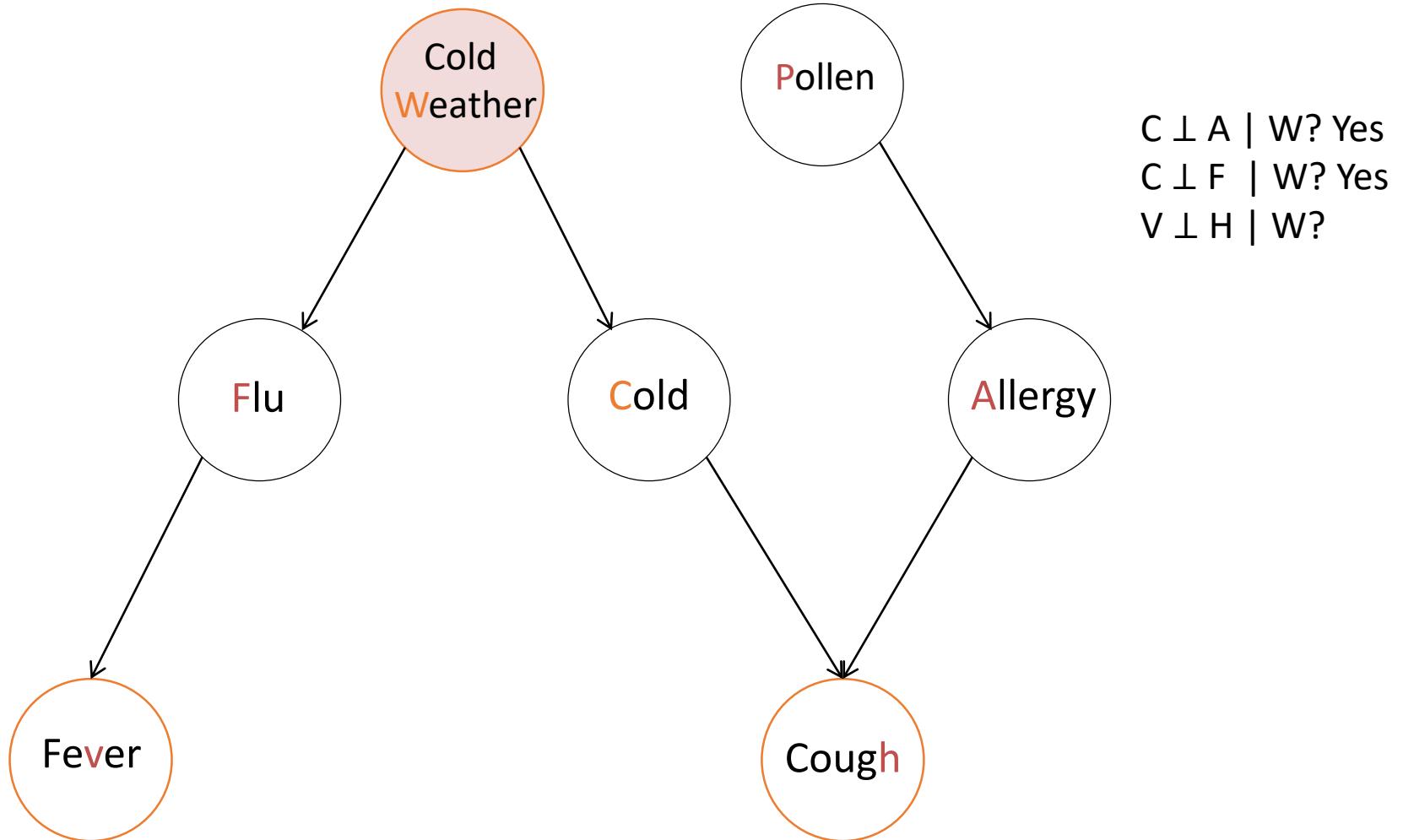
# Conditional Independence



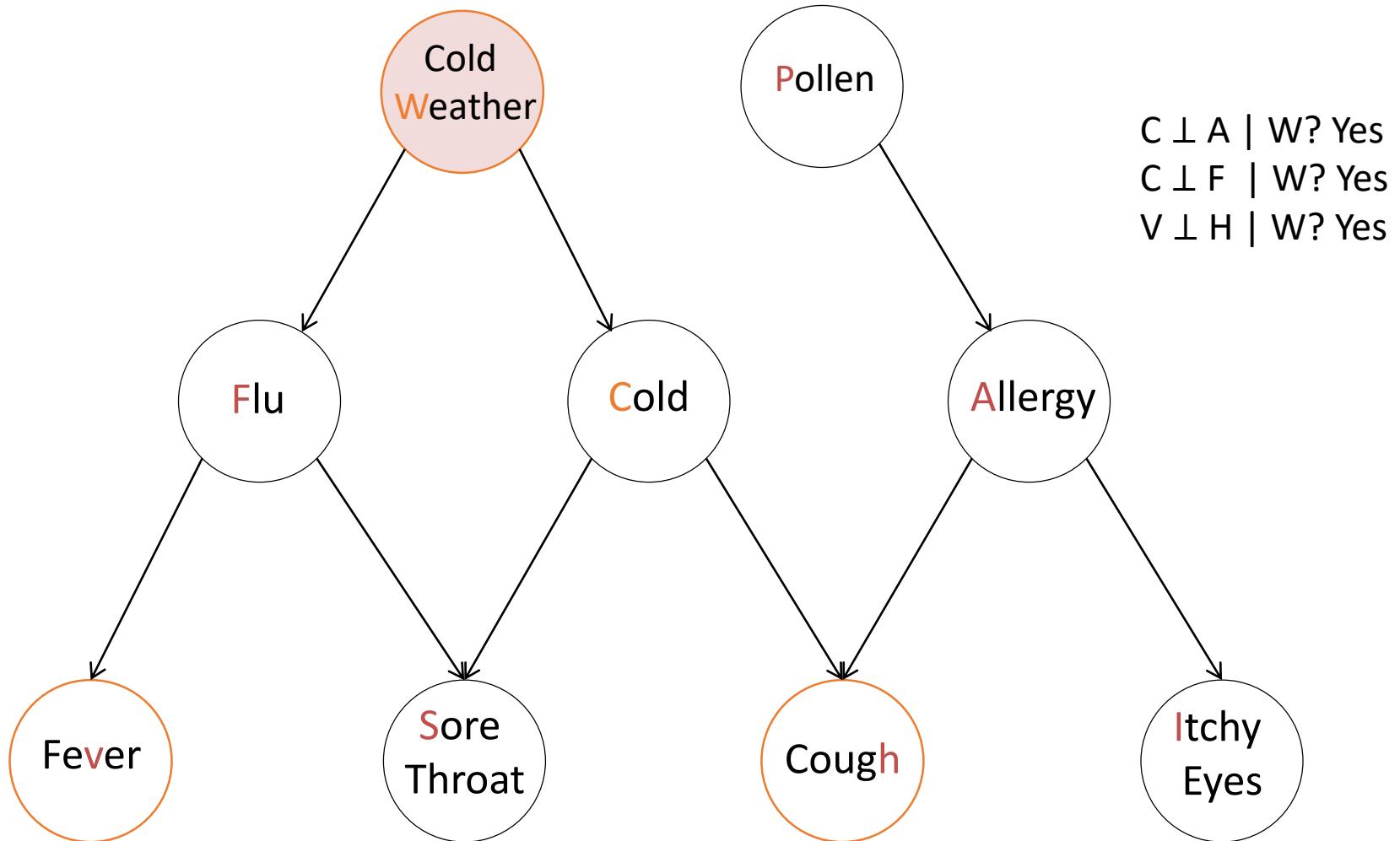
# Conditional Independence



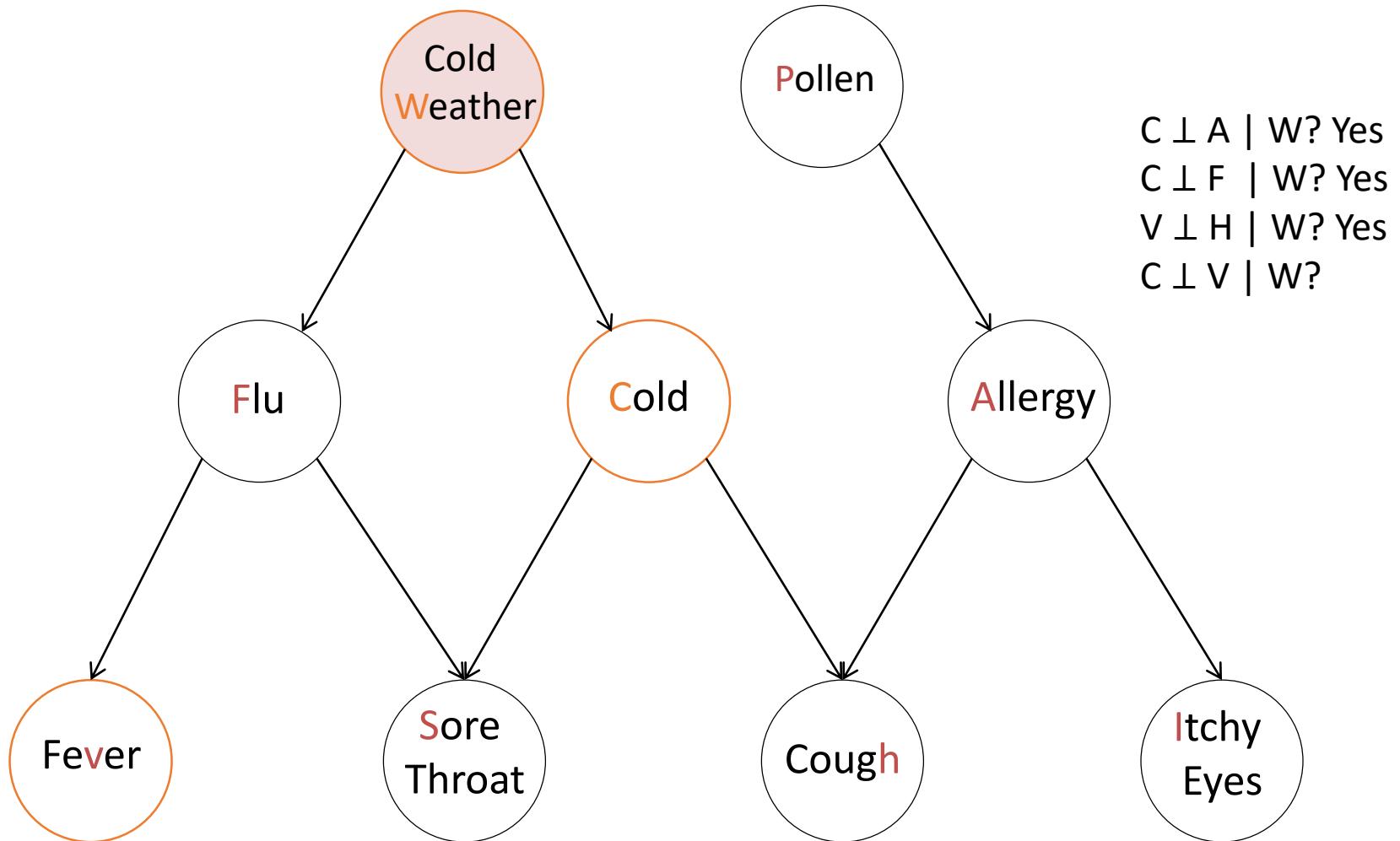
# Conditional Independence



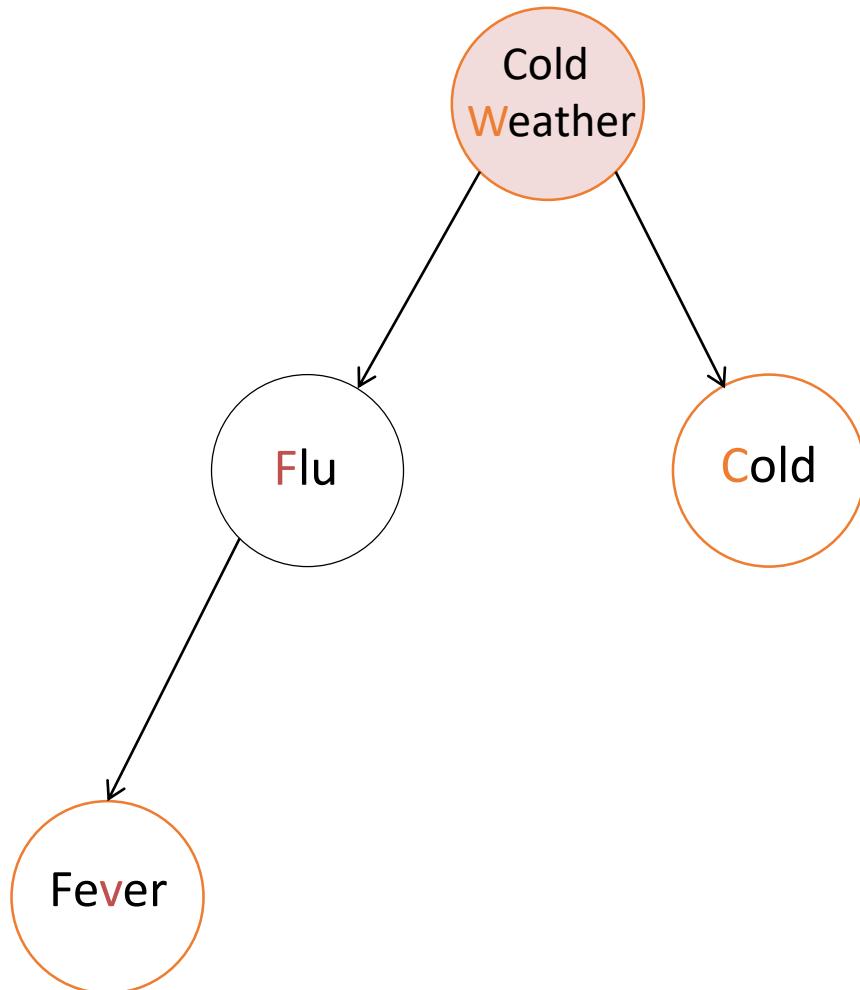
# Conditional Independence



# Conditional Independence

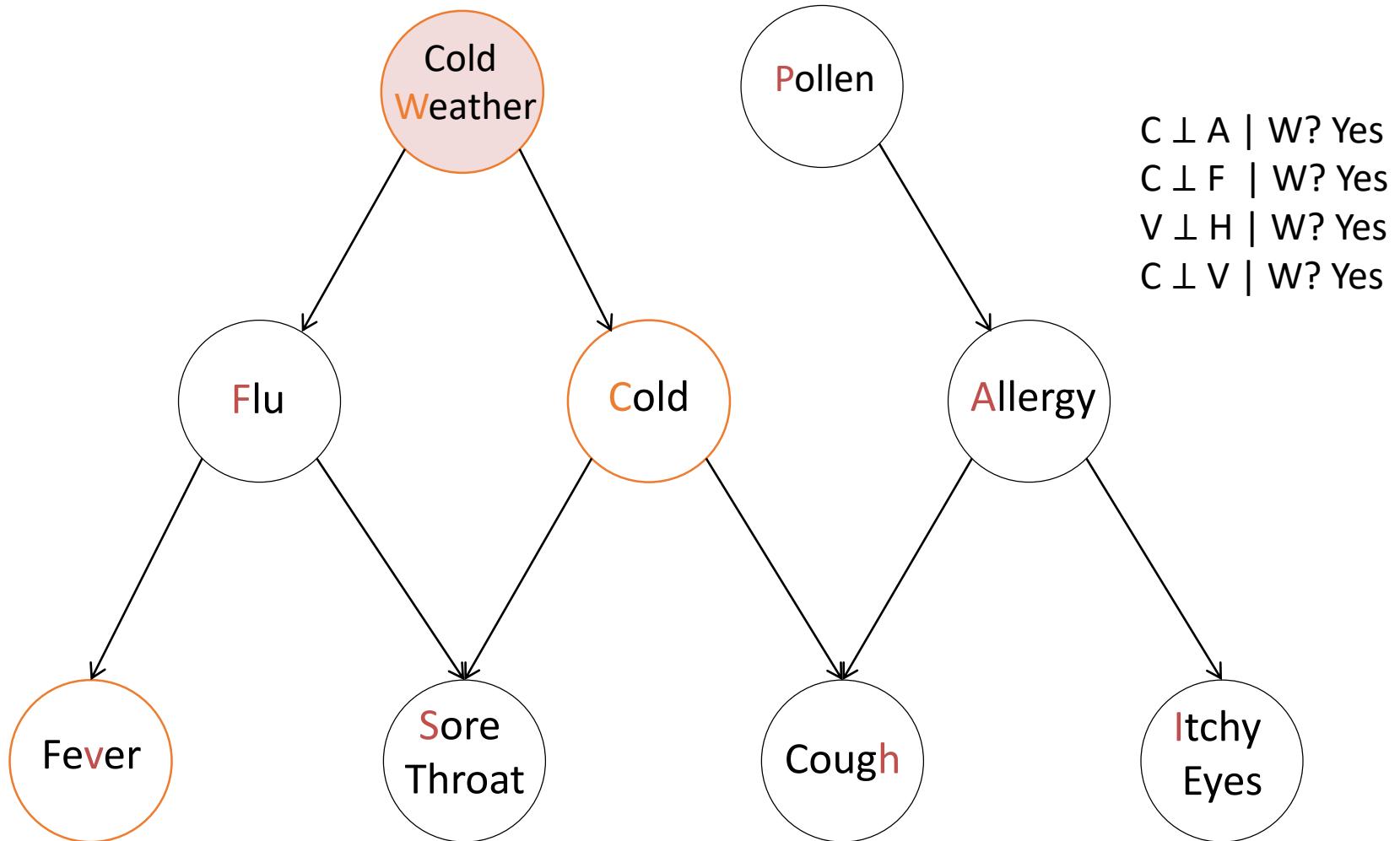


# Conditional Independence

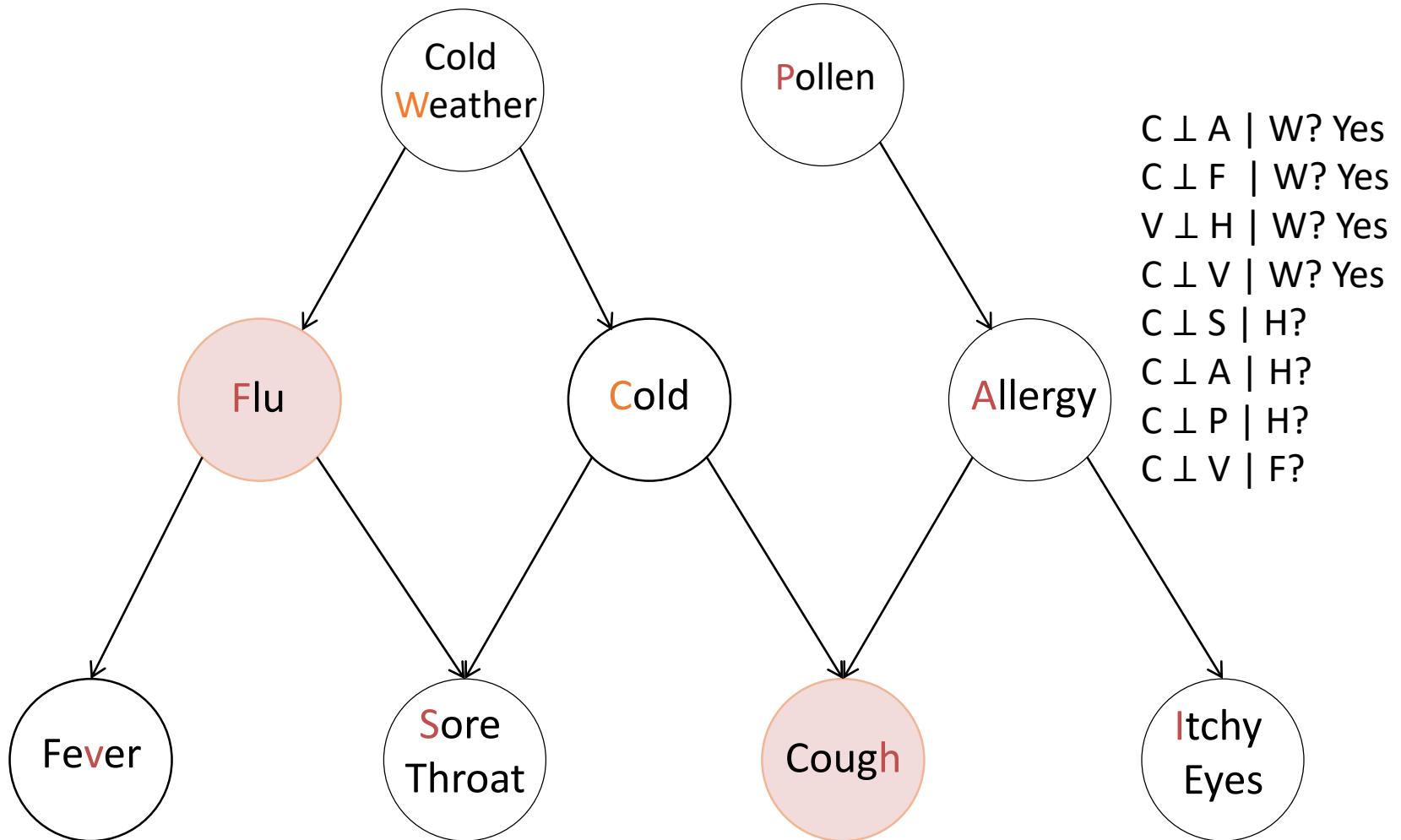


$C \perp A \mid W? \text{ Yes}$   
 $C \perp F \mid W? \text{ Yes}$   
 $V \perp H \mid W? \text{ Yes}$   
 $C \perp V \mid W?$

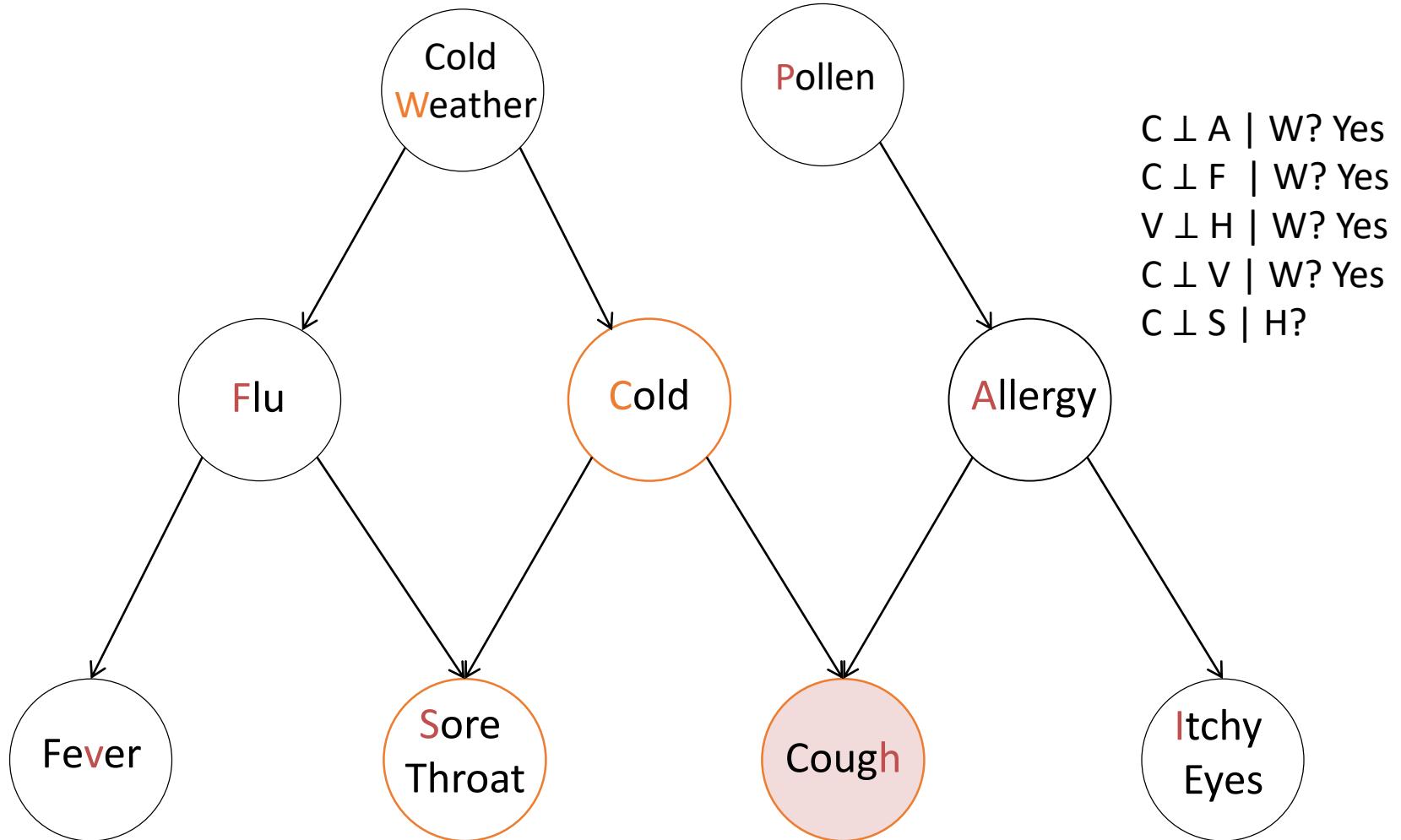
# Conditional Independence



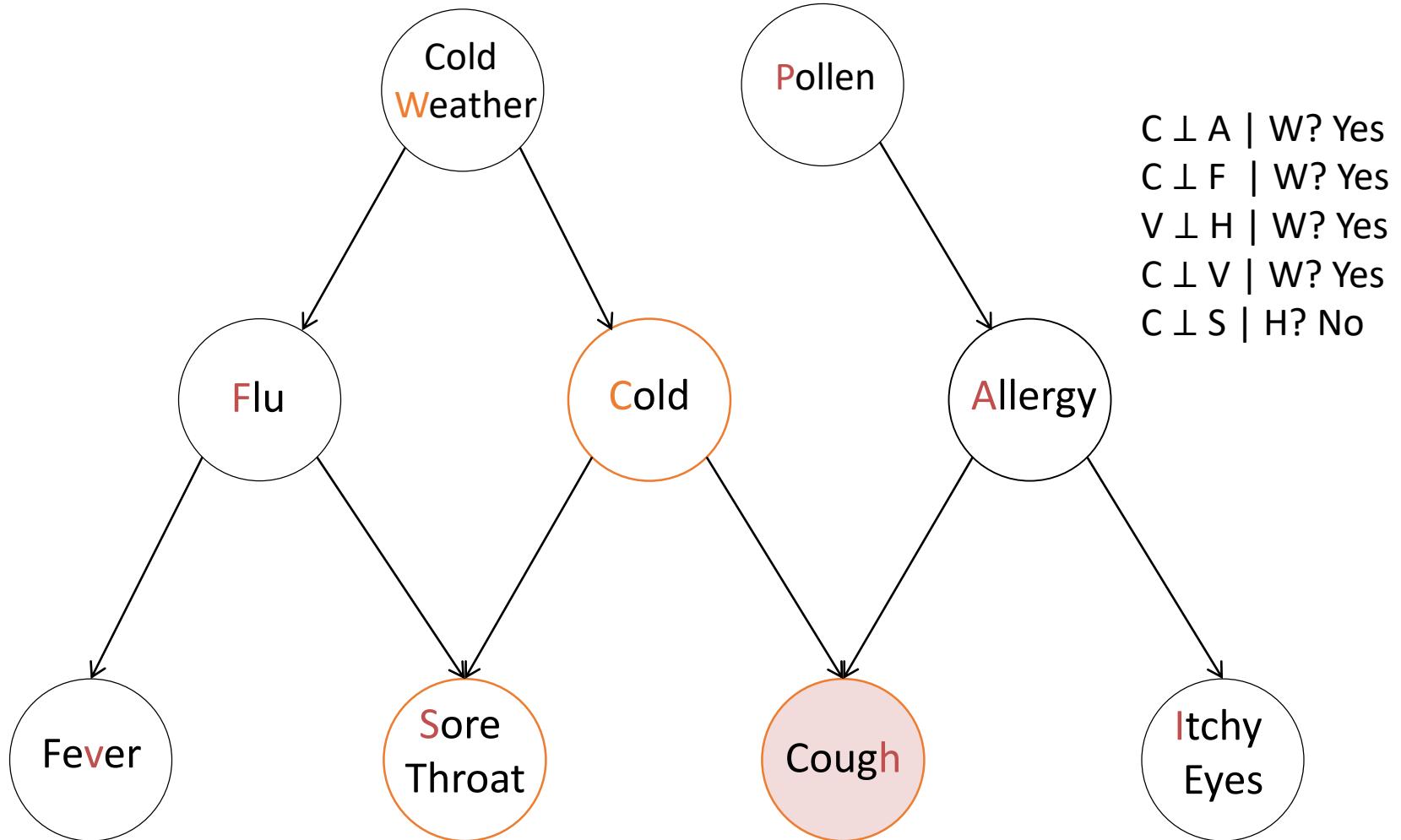
# Conditional Independence



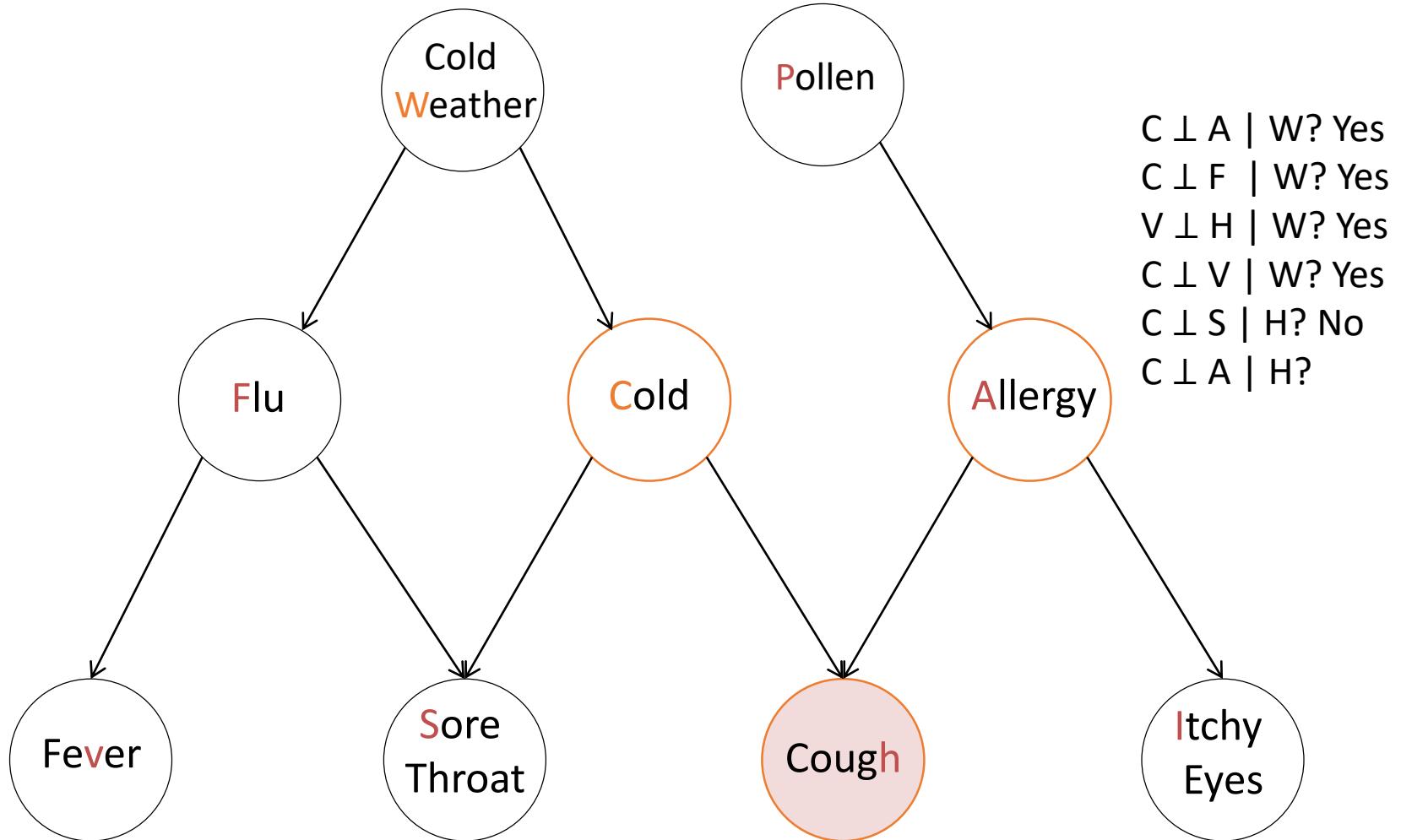
# Conditional Independence



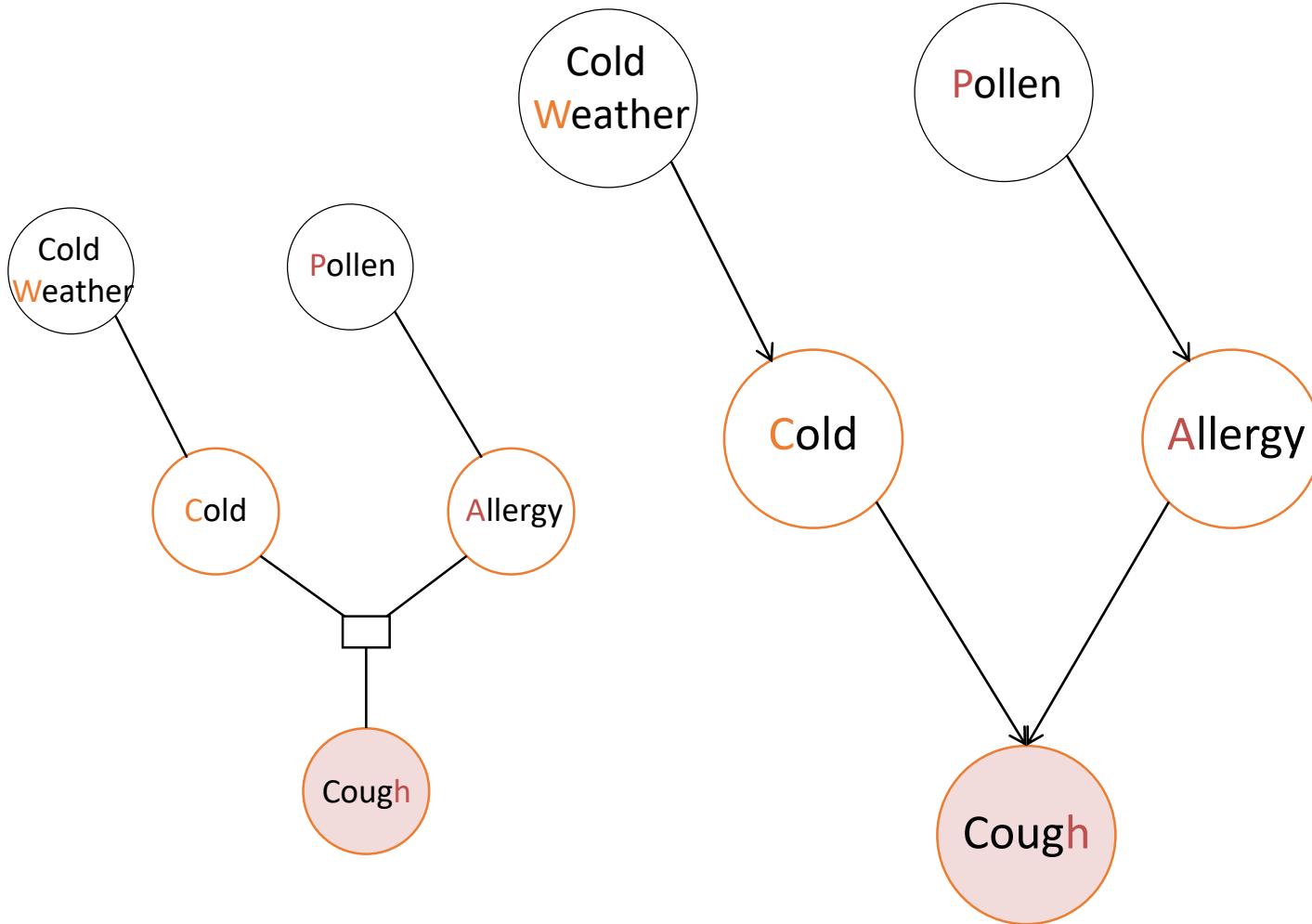
# Conditional Independence



# Conditional Independence



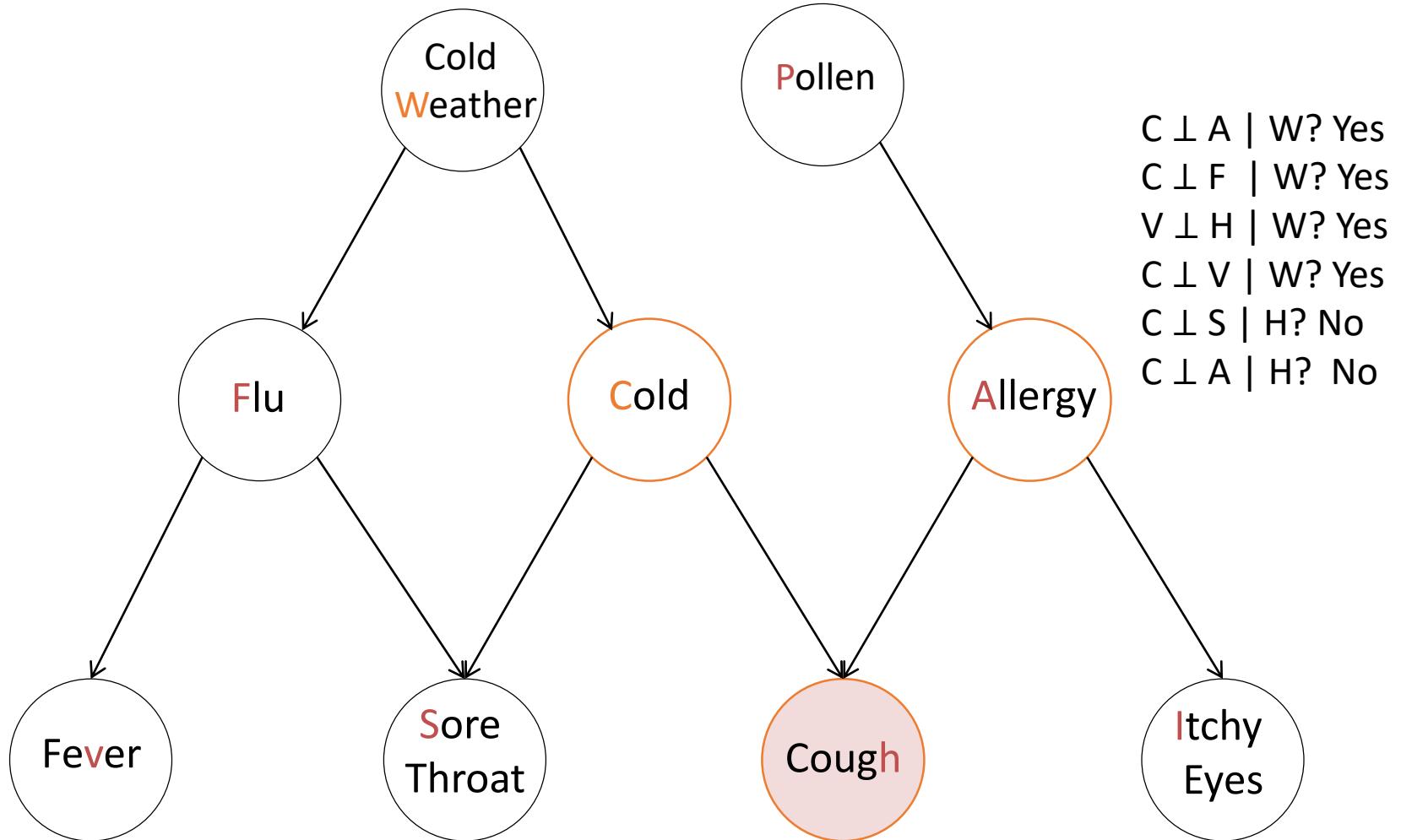
# Conditional Independence



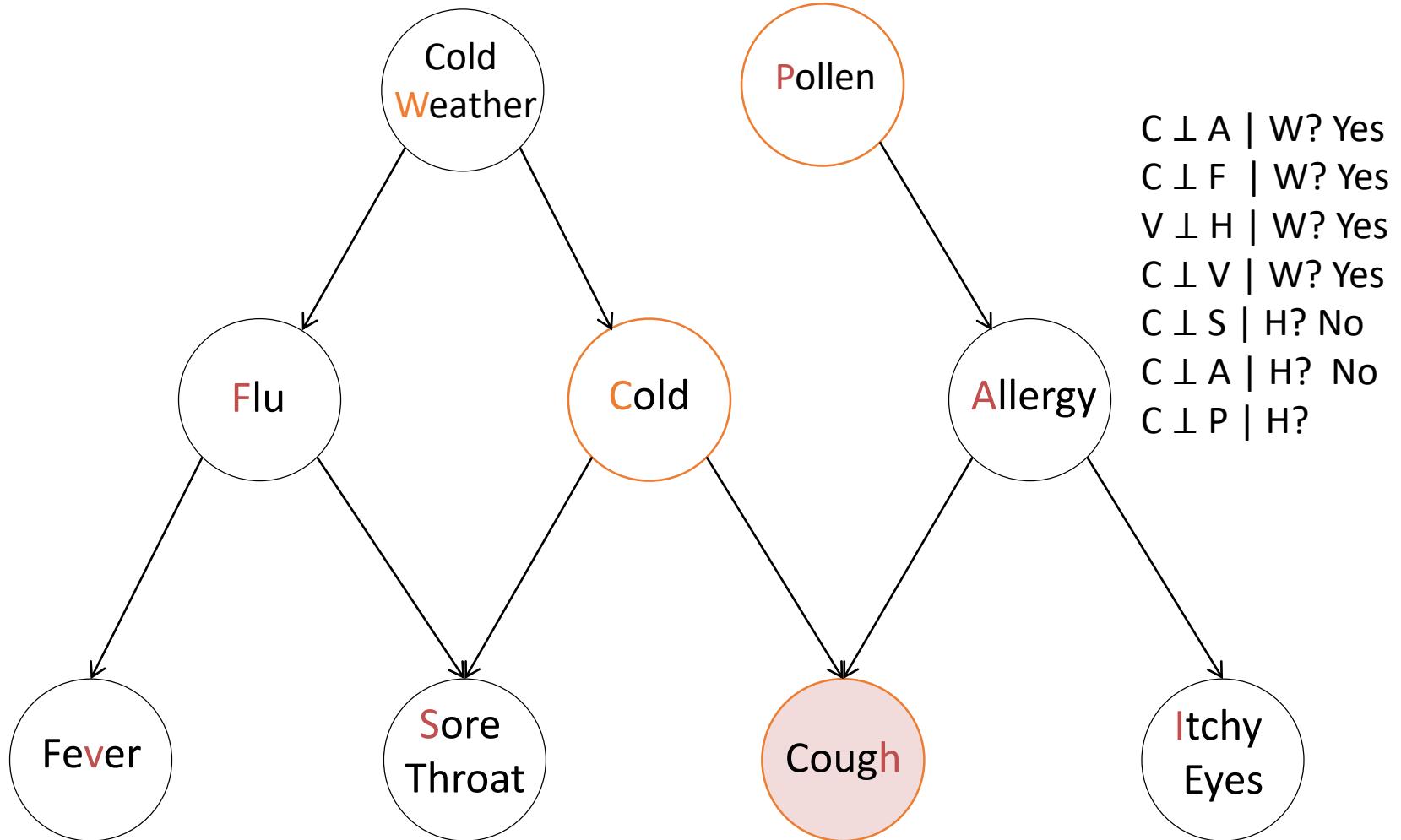
- $C \perp A \mid W? \text{ Yes}$
- $C \perp F \mid W? \text{ Yes}$
- $V \perp H \mid W? \text{ Yes}$
- $C \perp V \mid W? \text{ Yes}$
- $C \perp S \mid H? \text{ No}$
- $C \perp A \mid H?$

Explaining Away!

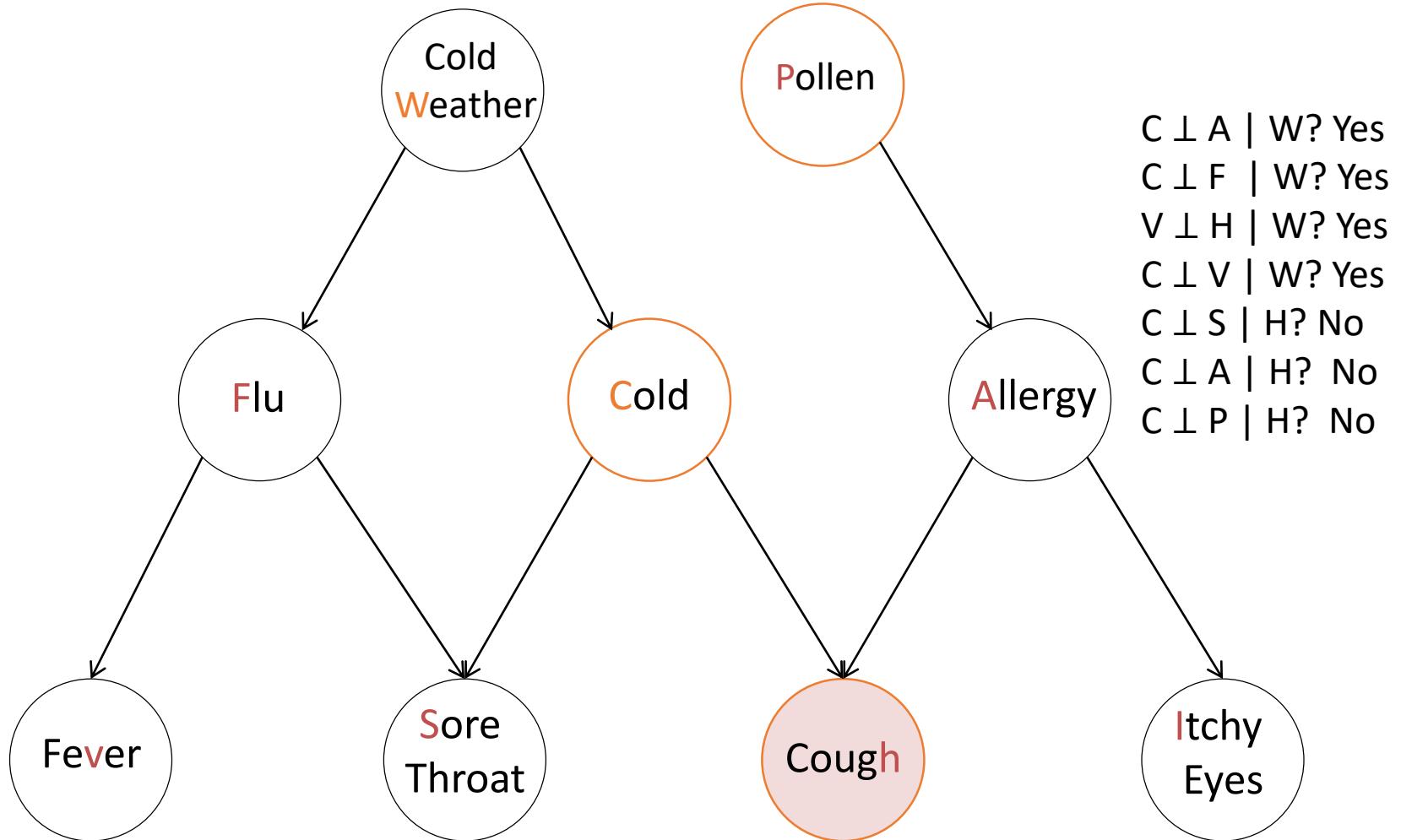
# Conditional Independence



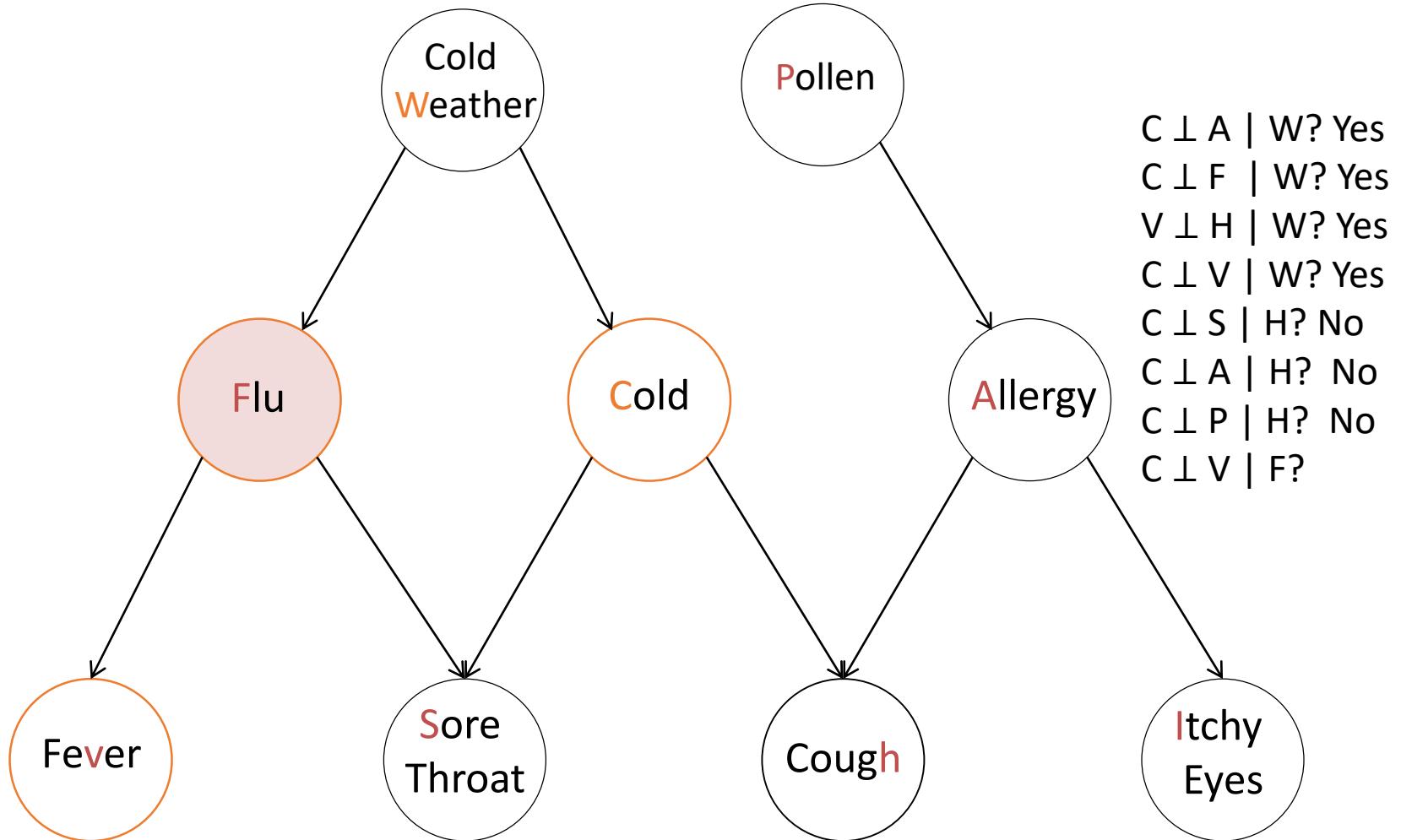
# Conditional Independence



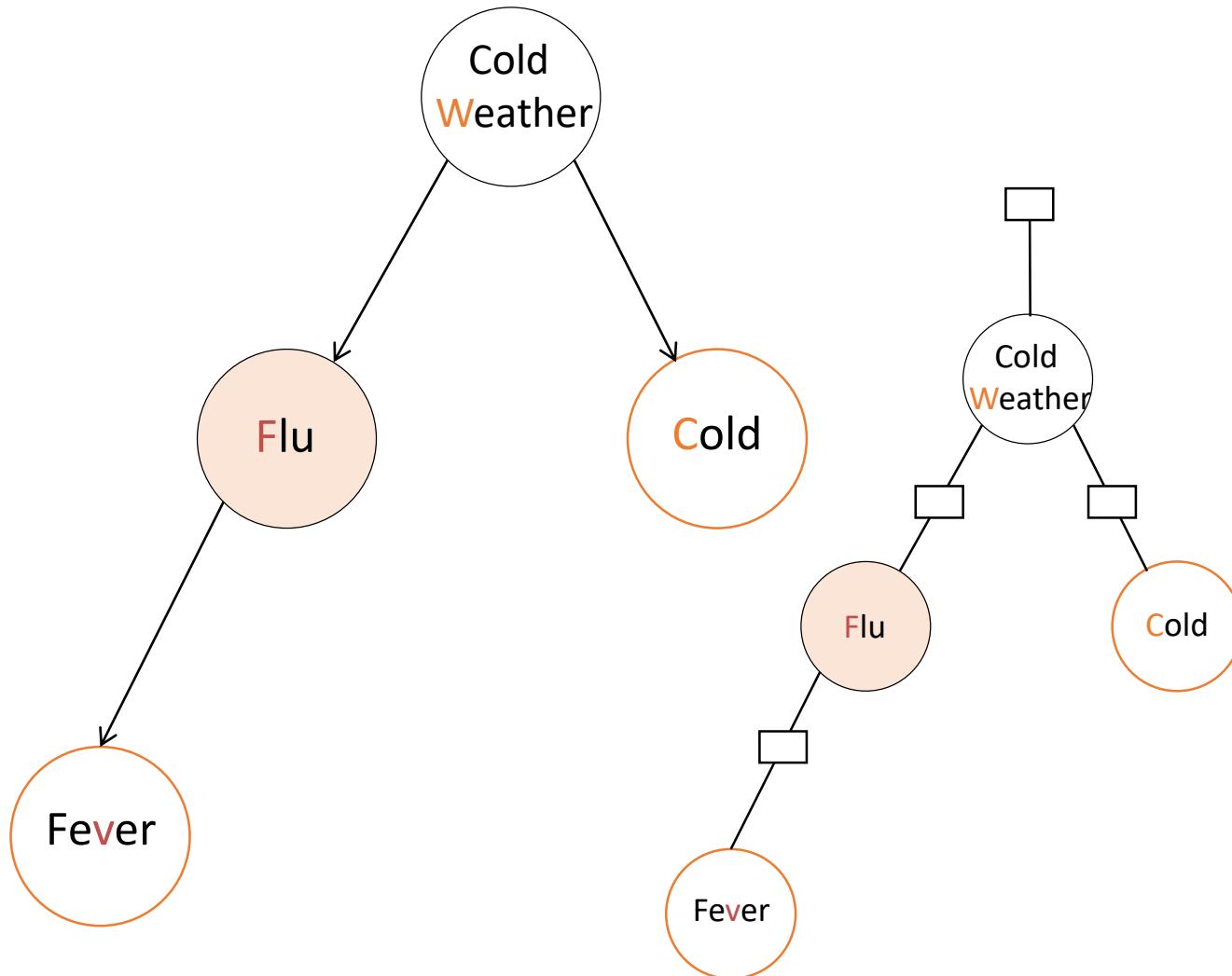
# Conditional Independence



# Conditional Independence

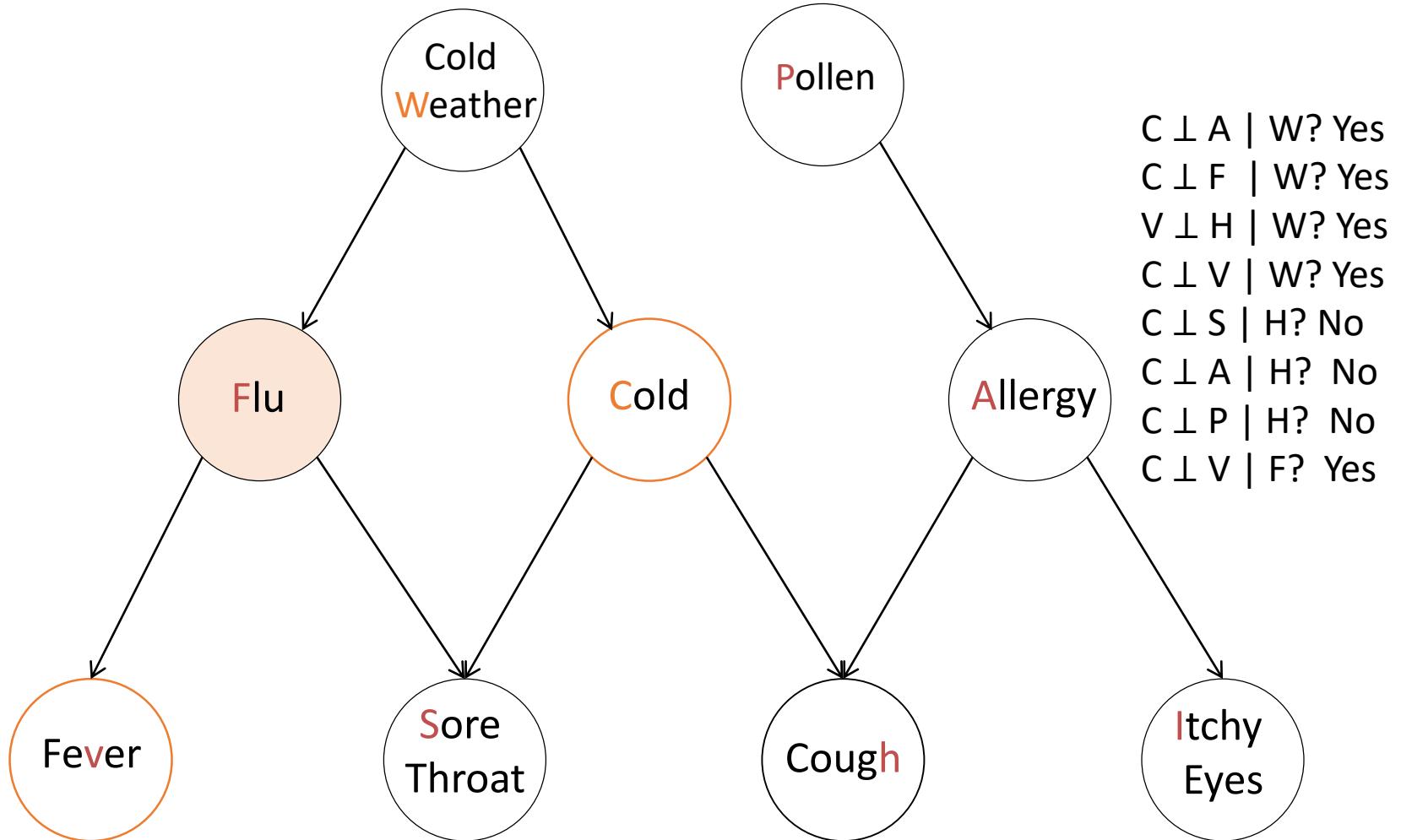


# Conditional Independence

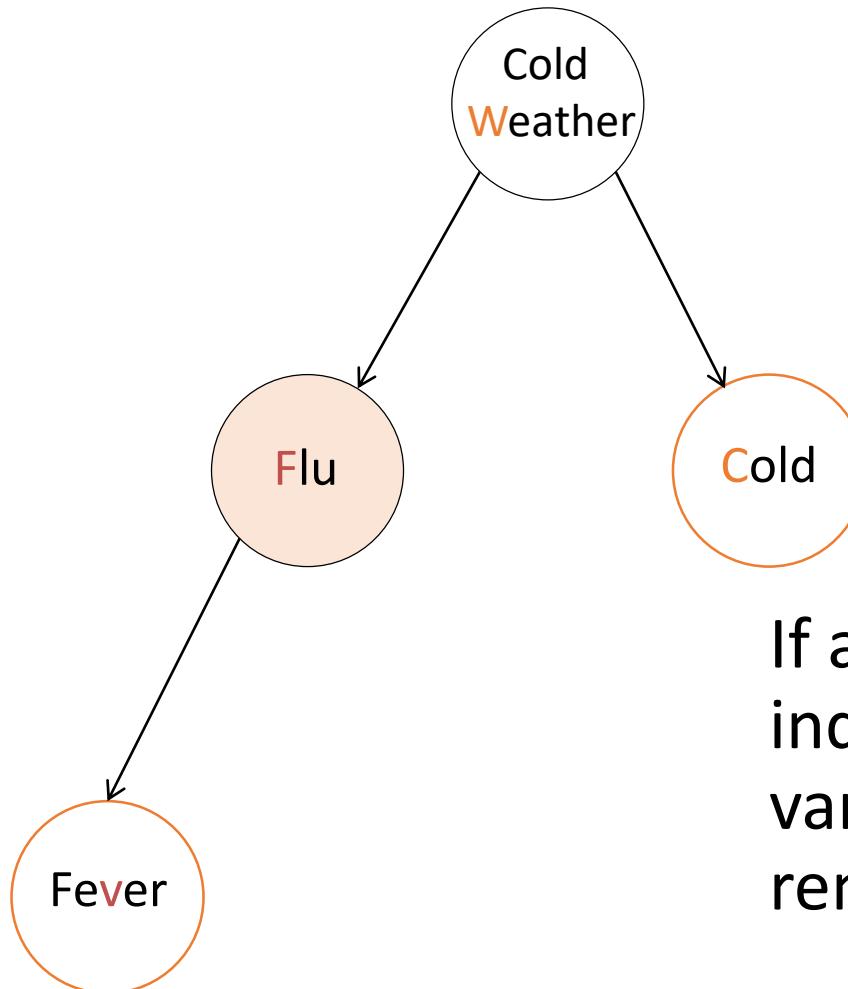


- $C \perp A \mid W? \text{ Yes}$
- $C \perp F \mid W? \text{ Yes}$
- $V \perp H \mid W? \text{ Yes}$
- $C \perp V \mid W? \text{ Yes}$
- $C \perp S \mid H? \text{ No}$
- $C \perp A \mid H? \text{ No}$
- $C \perp P \mid H? \text{ No}$
- $C \perp V \mid F?$

# Conditional Independence



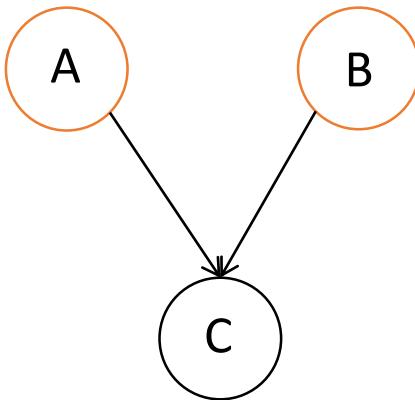
# Conditional Independence



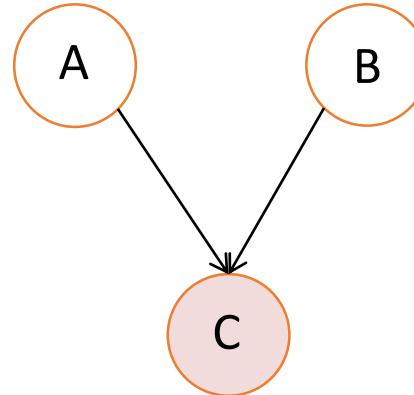
$$P(C=c | F=f) = ?$$

If a variable (Fever) is independent of the Query variable Q (Cold), we can remove (marginalize) it.

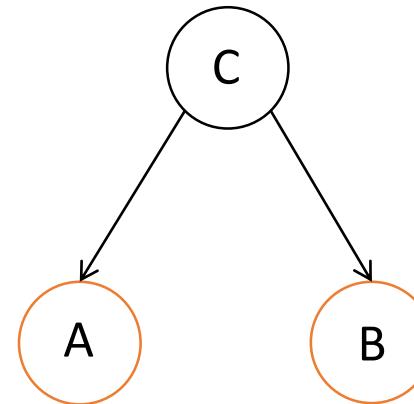
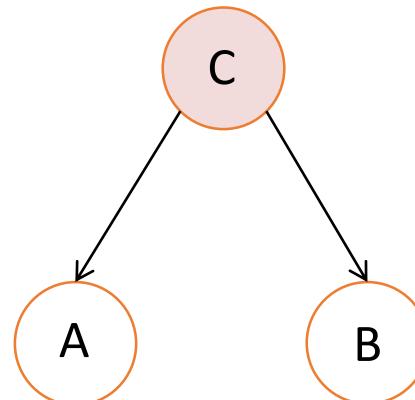
# Patterns



Independent

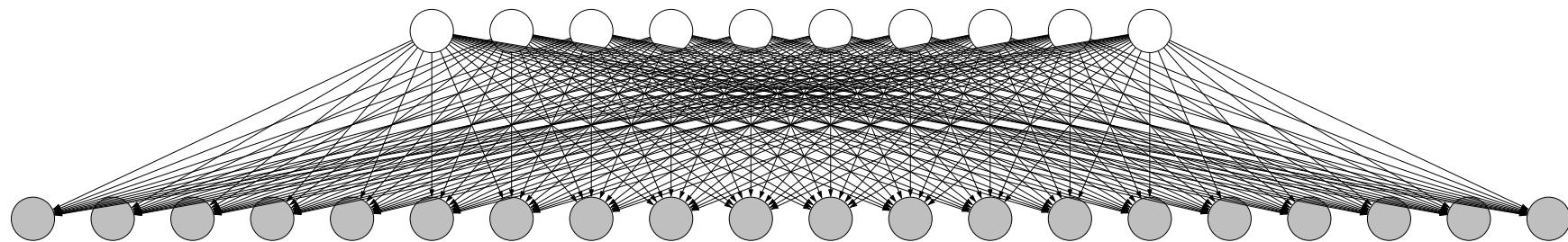


Dependent





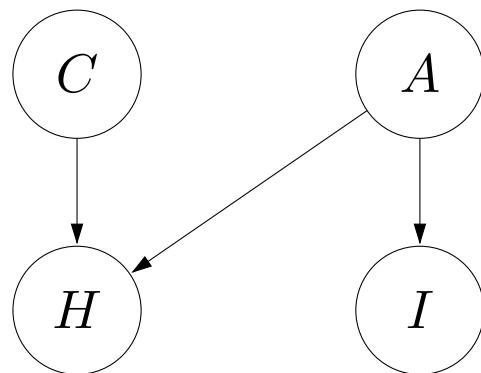
# Lecture 15: Bayesian networks III



# Announcements

- **car** is due tomorrow
- **p-progress** is due Thursday
- **exam** is next Tuesday
  - covers material up to and including today's lecture
  - reviews in Section this Thursday AND Friday
  - all alternative exams have been scheduled
  - problem solving, practice with old exam problems

# Review: Bayesian network



$$\begin{aligned}\mathbb{P}(C = c, A = a, H = h, I = i) \\ = p(c)p(a)p(h \mid c, a)p(i \mid a)\end{aligned}$$



## Definition: Bayesian network

Let  $X = (X_1, \dots, X_n)$  be random variables.

A **Bayesian network** is a directed acyclic graph (DAG) that specifies a **joint distribution** over  $X$  as a product of **local conditional distributions**, one for each node:

$$\mathbb{P}(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^n p(x_i \mid x_{\text{Parents}(i)})$$

- A Bayesian network allows us to define a **joint** probability distribution over many variables (e.g.,  $\mathbb{P}(C, A, H, I)$ ) by specifying **local** conditional distributions (e.g.,  $p(i | a)$ ). Two lectures ago, we talked about modeling: how can we use Bayesian networks to represent real-world problems.

# Review: probabilistic inference

Bayesian network:

$$\mathbb{P}(X = x) = \prod_{i=1}^n p(x_i \mid x_{\text{Parents}(i)})$$

Probabilistic inference:

$$\mathbb{P}(Q \mid E = e)$$

Algorithms:

- Forward-backward: HMMs, exact
- Particle filtering: HMMs, approximate
- Gibbs sampling: general, approximate

- Last lecture, we focused on algorithms for probabilistic inference: how do we efficiently compute queries of interest? We can do many things in closed form by leveraging the conditional independence structure of Bayesian networks.
- For HMMs, we could use the forward-backward algorithm to perform smoothing and filtering efficiently and exactly.
- For extremely large domains, particle filtering allows you to perform filtering efficiently but only approximately.
- For general Bayesian networks and factor graphs, we can apply Gibbs sampling, which is also approximate.

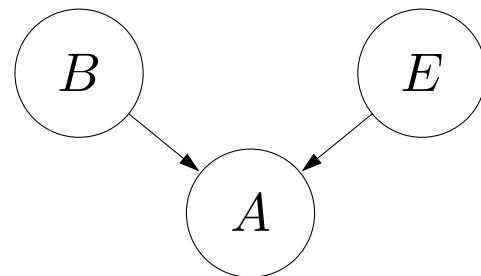
# Paradigm

Modeling

Inference

Learning

# Where do parameters come from?



$b$	$p(b)$
1	?
0	?

$e$	$p(e)$
1	?
0	?

$b$	$e$	$a$	$p(a \mid b, e)$
0	0	0	?
0	0	1	?
0	1	0	?
0	1	1	?
1	0	0	?
1	0	1	?
1	1	0	?
1	1	1	?

- Today's lecture focuses on the following question: where do all the local conditional distributions come from? These local conditional distributions are the parameters of the Bayesian network.



# Roadmap

**Supervised learning**

Laplace smoothing

Unsupervised learning with EM

- The plan for today is to start with the supervised setting, where our training data consists of a set of **complete assignments**. The algorithms here are just counting and normalizing.
- Then we'll talk about smoothing, which is a mechanism for guarding against overfitting. This involves just a small tweak to existing algorithms.
- Finally, we'll consider the unsupervised or missing data setting, where our training data consists of a set of **partial assignments**. Here, the workhorse algorithm is the EM algorithm which breaks up the problem into probabilistic inference + supervised algorithms.

# Learning task

**Training data**

$\mathcal{D}_{\text{train}}$  (an example is an assignment to  $X$ )



**Parameters**

$\theta$  (local conditional probabilities)

- As with any learning algorithm, we start with the data. We will first consider the supervised setting, where each data point (example) is a complete assignment to all the variables in the Bayesian network.
- We will first develop the learning algorithm intuitively on some simple examples. Later, we will provide the algorithm for the general case and a formal justification based on maximum likelihood.



# Question

Which is computationally more expensive for Bayesian networks?

probabilistic inference given the parameters

learning the parameters given fully labeled data

- Probabilistic inference assumes you know the parameters, whereas learning does not, so one might think that inference should be easier.
- However, for Bayesian networks, somewhat surprisingly, it turns out that learning (at least in the fully-supervised setting) is easier.

# Example: one variable

Setup:

- One variable  $R$  representing the rating of a movie  $\{1, 2, 3, 4, 5\}$

$$R \qquad \mathbb{P}(R = r) = p(r)$$

Parameters:

$$\theta = (p(1), p(2), p(3), p(4), p(5))$$

Training data:

$$\mathcal{D}_{\text{train}} = \{1, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5\}$$

- Suppose you want to study how people rate movies. We will develop several Bayesian networks of increasing complexity, and show how to learn the parameters of these models. (Along the way, we'll also practice doing a bit of modeling.)
- Let's start with the world's simplest Bayesian network, which has just one variable representing the movie rating. Here, there are 5 parameters, each one representing the probability of a given rating.
- (Technically, there are only 4 parameters since the 5 numbers sum to 1 so knowing 4 of the 5 is enough. But we will call it 5 for simplicity.)

# Example: one variable

Learning:

$$\mathcal{D}_{\text{train}} \Rightarrow \theta$$

Intuition:  $p(r) \propto$  number of occurrences of  $r$  in  $\mathcal{D}_{\text{train}}$

Example:

$$\mathcal{D}_{\text{train}} = \{1, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5\}$$



$\theta$ :

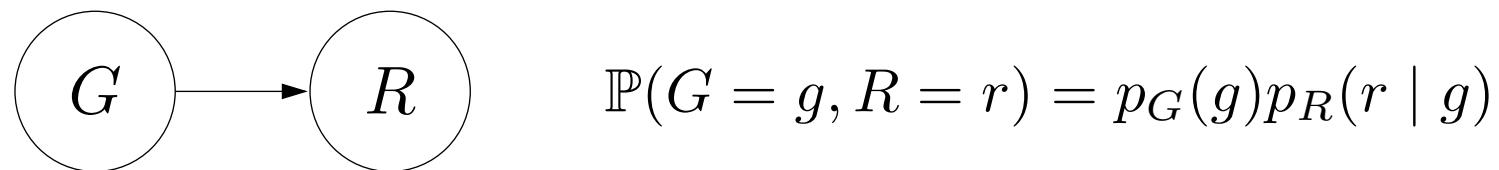
$r$	$p(r)$
1	0.1
2	0.0
3	0.1
4	0.5
5	0.3

- Given the data, which consists of a set of ratings (the order doesn't matter here), the natural thing to do is to set each parameter  $p(r)$  to be the empirical fraction of times that  $r$  occurs in  $\mathcal{D}_{\text{train}}$ .

# Example: two variables

Variables:

- Genre  $G \in \{\text{drama, comedy}\}$
- Rating  $R \in \{1, 2, 3, 4, 5\}$

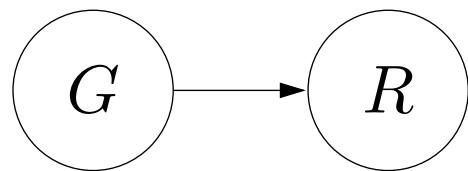


$$\mathcal{D}_{\text{train}} = \{(d, 4), (d, 4), (d, 5), (c, 1), (c, 5)\}$$

Parameters:  $\theta = (p_G, p_R)$

- Let's enrich the Bayesian network, since people don't rate movies completely randomly; the rating will depend on a number of factors, including the genre of the movie. This yields a two-variable Bayesian network.
- We now have two local conditional distributions,  $p_G(g)$  and  $p_R(r | g)$ , each consisting of a set of probabilities, one for each setting of the values.
- Note that we are explicitly using the subscript  $G$  and  $R$  to uniquely identify the local conditional distribution inside the parameters. In this case, we could just infer it from context, but when we talk about parameter sharing later, specifying the precise local conditional distribution will be important.
- There should be  $2 + 2 \cdot 5 = 12$  total parameters in this model. (Again technically there are  $1 + 2 \cdot 4 = 9$ .).

# Example: two variables



$$\mathbb{P}(G = g, R = r) = p_G(g)p_R(r | g)$$

$$\mathcal{D}_{\text{train}} = \{(d, 4), (d, 4), (d, 5), (c, 1), (c, 5)\}$$

**Intuitive strategy:** Estimate each local conditional distribution ( $p_G$  and  $p_R$ ) separately

$\theta$ :

$g$	$p_G(g)$
d	3/5
c	2/5

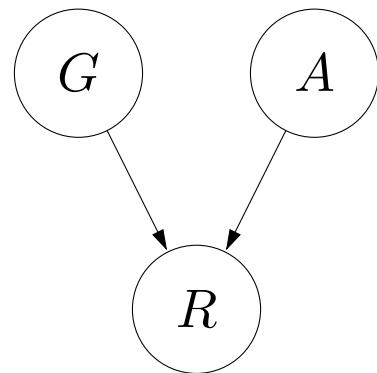
$g$	$r$	$p_R(r   g)$
d	4	2/3
d	5	1/3
c	1	1/2
c	5	1/2

- To learn the parameters of this model, we can handle each local conditional distribution separately (this will be justified later). This leverages the modular structure of Bayesian networks.
- To estimate  $p_G(g)$ , we look at the data but just ignore the value of  $r$ . To estimate  $p_R(r | g)$ , we go through each value of  $g$  and estimate the probability for each  $r$ .
- Operationally, it's convenient to first keep the integer count for each local assignment, and then just normalize by the total count for each assignment.

# Example: v-structure

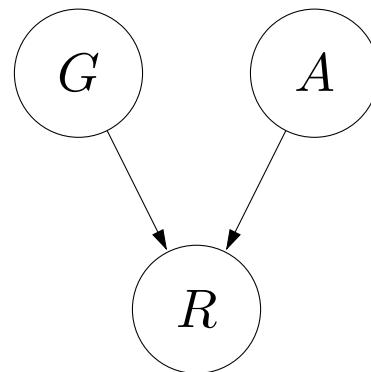
Variables:

- Genre  $G \in \{\text{drama, comedy}\}$
- Won award  $A \in \{0, 1\}$
- Rating  $R \in \{1, 2, 3, 4, 5\}$



$$\mathbb{P}(G = g, A = a, R = r) = p_G(g)p_A(a)p_R(r \mid g, a)$$

# Example: v-structure



$$\mathcal{D}_{\text{train}} = \{(d, 0, 3), (d, 1, 5), (d, 0, 1), (c, 0, 5), (c, 1, 4)\}$$

Parameters:  $\theta = (p_G, p_A, p_R)$

$\theta:$

$g$	$p_G(g)$
d	3/5
c	2/5

$a$	$p_A(a)$
0	3/5
1	2/5

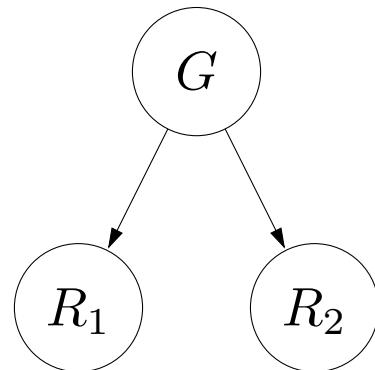
$g$	$a$	$r$	$p_R(r   g, a)$
d	0	1	1/2
d	0	3	1/2
d	1	5	1
c	0	5	1
c	1	4	1

- While probabilistic inference with v-structures was quite subtle, learning parameters for V-structures is really the same as any other Bayesian network structure. We just need to remember that the parameters include the conditional probabilities for each joint assignment to both parents.
- In this case, there are roughly  $2 + 2 + (2 \cdot 2 \cdot 5) = 24$  parameters to set (or 18 if you're more clever). Given the five data points though, most of these parameters will be zero (without smoothing, which we'll talk about later).

# Example: inverted-v structure

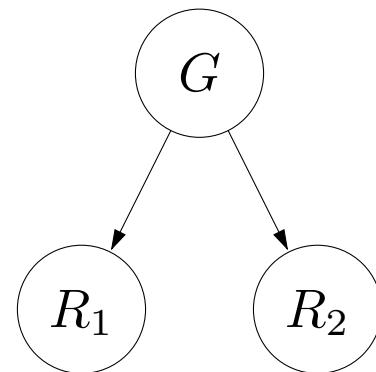
Variables:

- Genre  $G \in \{\text{drama, comedy}\}$
- Jim's rating  $R_1 \in \{1, 2, 3, 4, 5\}$
- Martha's rating  $R_2 \in \{1, 2, 3, 4, 5\}$



$$\mathbb{P}(G = g, R_1 = r_1, R_2 = r_2) = p_G(g)p_{R_1}(r_1 \mid g)p_{R_2}(r_2 \mid g)$$

# Example: inverted-v structure



$$\mathcal{D}_{\text{train}} = \{(d, 4, 5), (d, 4, 4), (d, 5, 3), (c, 1, 2), (c, 5, 4)\}$$

Parameters:  $\theta = (p_G, p_{R_1}, p_{R_2})$

$\theta:$

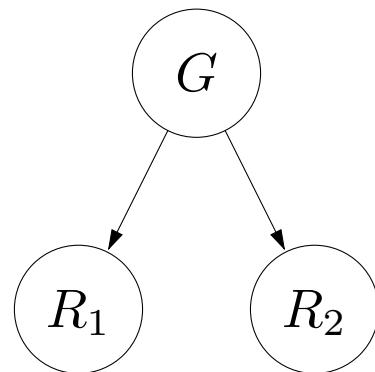
$g$	$p_G(g)$
d	3/5
c	2/5

$g$	$r_1$	$p_{R_1}(r   g)$
d	4	2/3
d	5	1/3
c	1	1/2
c	5	1/2

$g$	$r_2$	$p_{R_2}(r   g)$
d	3	1/3
d	4	1/3
d	5	1/3
c	2	1/2
c	4	1/2

- Let's suppose now that you're trying to model two people's ratings, those of Jim and Martha. We can define a three-node Bayesian network.
- Learning the parameters in this way works the same as before.

# Example: inverted-v structure



$$\mathcal{D}_{\text{train}} = \{(d, 4, 5), (d, 4, 4), (d, 5, 3), (c, 1, 2), (c, 5, 4)\}$$

Parameters:  $\theta = (p_G, p_R)$

$\theta:$

$g$	$p_G(g)$
d	3/5
c	2/5

$g$	$r$	$p_R(r   g)$
d	3	1/6
d	4	3/6
d	5	2/6
c	1	1/4
c	2	1/4
c	4	1/4
c	5	1/4

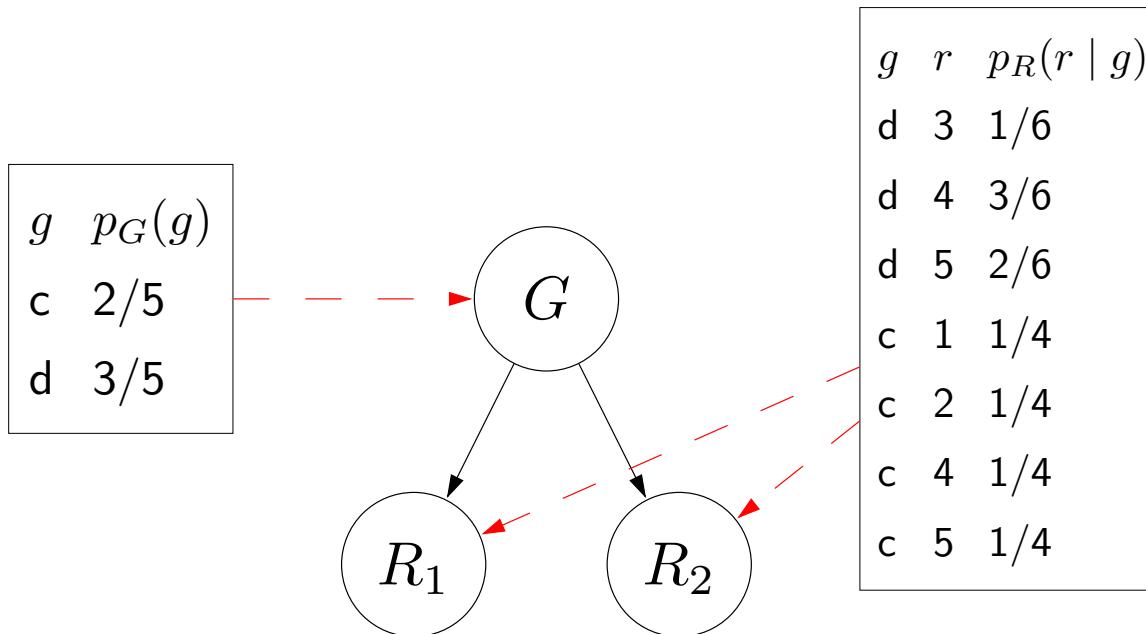
- Recall that currently, every local conditional distribution is estimated separately. But this is non-ideal if some variables behave similarly (e.g., if Jim and Martha have similar movie tastes).
- In this case, it would make more sense to have one local conditional distribution. To perform estimation in this setup, we simply go through each example (e.g.,  $(d, 4, 5)$ ) and each variable, and increment the counts on the appropriate local conditional distribution (e.g., 1 for  $p_G(d)$ , 1 for  $p_R(4 | d)$ , and 1 for  $p_R(5 | d)$ ). Finally, we normalize the counts to get local conditional distributions.

# Parameter sharing



**Key idea: parameter sharing**

The local conditional distributions of different variables use the same parameters.



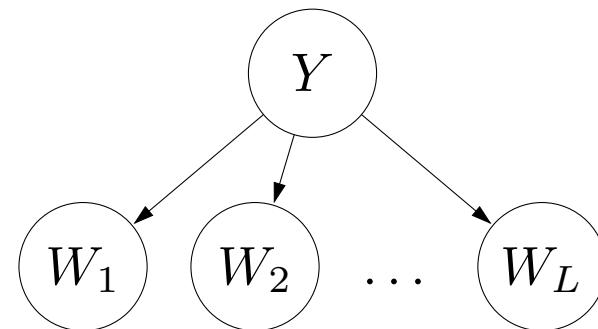
**Impact:** more reliable estimates, less expressive model

- This is the idea of **parameter sharing**. Think of each variable as being powered by a local conditional distribution (a table). Importantly, each table can drive multiple variables.
- Note that when we were talking about probabilistic inference, we didn't really care about where the conditional distributions came from, because we were just reading from them; it didn't matter whether  $p(r_1 | g)$  and  $p(r_2 | g)$  came from the same source. In learning, we have to write to those distributions, and where we write to matters. As an analogy, think of when passing by value and passing by reference yield the same answer.

# Example: Naive Bayes

Variables:

- Genre  $Y \in \{\text{comedy}, \text{drama}\}$
- Movie review (sequence of words):  $W_1, \dots, W_L$



$$\mathbb{P}(Y = y, W_1 = w_1, \dots, W_L = w_L) = p_{\text{genre}}(y) \prod_{j=1}^L p_{\text{word}}(w_j \mid y)$$

Parameters:  $\theta = (p_{\text{genre}}, p_{\text{word}})$

- As an extension of the previous example, consider the popular Naive Bayes model, which can be used to model the contents of documents (say, movie reviews about comedies versus dramas). The model is said to be "naive" because all the words are assumed to be conditionally independent given class variable  $Y$ .
- In this model, there is a lot of parameter sharing: each word  $W_j$  is generated from the same distribution  $p_{\text{word}}$ .



# Question

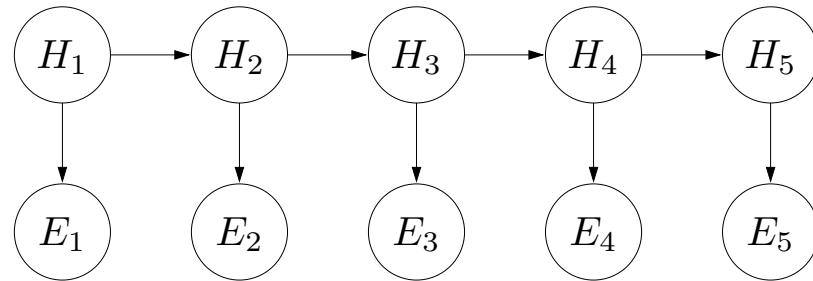
If  $Y$  can take on 2 values and each  $W_j$  can take on  $D$  values, how many parameters are there?

- There are  $L + 1$  variables, but all but  $Y$  are powered by the same local conditional distribution. We have 2 parameters for  $p_{\text{genre}}$  and  $2D$  for  $p_{\text{word}}$ , for a total of  $2 + 2D = O(D)$ . Importantly, due to parameter sharing, there is no dependence on  $L$ .

# Example: HMMs

Variables:

- $H_1, \dots, H_n$  (e.g., actual positions)
- $E_1, \dots, E_n$  (e.g., sensor readings)



$$\mathbb{P}(H = h, E = e) = p_{\text{start}}(h_1) \prod_{i=2}^n p_{\text{trans}}(h_i \mid h_{i-1}) \prod_{i=1}^n p_{\text{emit}}(e_i \mid h_i)$$

Parameters:  $\theta = (p_{\text{start}}, p_{\text{trans}}, p_{\text{emit}})$

$\mathcal{D}_{\text{train}}$  is a set of full assignments to  $(H, E)$

- The HMM is another model, which we saw was useful for object tracking.
- With  $K$  possible hidden states (values that  $H_t$  can take on) and  $D$  possible observations, the HMM has  $K^2$  transition parameters and  $KD$  emission parameters.

# General case

Bayesian network: variables  $X_1, \dots, X_n$

Parameters: collection of distributions  $\theta = \{p_d : d \in D\}$  (e.g.,  $D = \{\text{start}, \text{trans}, \text{emit}\}$ )

Each variable  $X_i$  is generated from distribution  $p_{d_i}$ :

$$\mathbb{P}(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^n p_{\textcolor{red}{d}_i}(x_i \mid x_{\text{Parents}(i)})$$

Parameter sharing:  $d_i$  could be same for multiple  $i$

- Now let's consider how to learn the parameters of an arbitrary Bayesian network with arbitrary parameter sharing. You should already have the basic intuitions; the next few slides will just be expressing these intuitions in full generality.
- The parameters of a general Bayesian network include a set of local conditional distributions indexed by  $d \in D$ . Note that many variables can be powered by the same  $d \in D$ .

# General case: learning algorithm

**Input:** training examples  $\mathcal{D}_{\text{train}}$  of full assignments

**Output:** parameters  $\theta = \{p_d : d \in D\}$



## Algorithm: maximum likelihood for Bayesian networks

### Count:

For each  $x \in \mathcal{D}_{\text{train}}$ :

    For each variable  $x_i$ :

        Increment  $\text{count}_{d_i}(x_{\text{Parents}(i)}, x_i)$

### Normalize:

For each  $d$  and local assignment  $x_{\text{Parents}(i)}$ :

    Set  $p_d(x_i | x_{\text{Parents}(i)}) \propto \text{count}_d(x_{\text{Parents}(i)}, x_i)$

- Estimating the parameters is a straightforward generalization. For each distribution, we go over all the training data, keeping track of the number of times each local assignment occurs. These counts are then normalized to form the final parameter estimates.

# Maximum likelihood

Maximum likelihood objective:

$$\max_{\theta} \prod_{x \in \mathcal{D}_{\text{train}}} \mathbb{P}(X = x; \theta)$$

Algorithm on previous slide exactly computes maximum likelihood parameters (closed form solution).

- So far, we've presented the count-and-normalize algorithm, and hopefully this seems to you like a reasonable thing to do. But what's the underlying principle?
- It can be shown that the algorithm that we've been using is no more than a closed form solution to the **maximum likelihood** objective, which says we should try to find  $\theta$  to maximize the probability of the training examples.

# Maximum likelihood

$$\mathcal{D}_{\text{train}} = \{(d, 4), (d, 5), (c, 5)\}$$

$$\max_{p_G(\cdot)} (p_G(d)p_G(d)p_G(c)) \max_{p_R(\cdot|c)} p_R(5 | c) \max_{p_R(\cdot|d)} (p_R(4 | d)p_R(5 | d))$$

Solution:

$$p_G(d) = \frac{2}{3}, p_G(c) = \frac{1}{3}, p_R(5 | c) = 1, p_R(4 | d) = \frac{1}{2}, p_R(5 | d) = \frac{1}{2}$$

- Key: decomposes into subproblems, one for each distribution  $d$  and assignment  $x_{\text{Parents}}$
- For each subproblem, solve in closed form (Lagrange multipliers for sum-to-1 constraint)

- We won't go through the math, but from the small example, it's clear we can switch the order of the factors.
- Notice that the problem decomposes into several independent pieces (one for each conditional probability distribution  $d$  and assignment to the parents).
- Each such subproblem can be solved easily (using the solution from the foundations homework).



# Roadmap

Supervised learning

Laplace smoothing

Unsupervised learning with EM

# Scenario 1

## Setup:

- You have a coin with an unknown probability of heads  $p(H)$ .
- You flip it 100 times, resulting in 23 heads, 77 tails.
- What is estimate of  $p(H)$ ?

## Maximum likelihood estimate:

$$p(H) = 0.23 \quad p(T) = 0.77$$

- Having established the basic learning algorithm for maximum likelihood, let's try to stress test it a bit.
- Just to review, the maximum likelihood estimate in this case is what we would expect and seems quite reasonable.

# Scenario 2

## Setup:

- You flip a coin once and get heads.
- What is estimate of  $p(H)$ ?

## Maximum likelihood estimate:

$$p(H) = 1 \quad p(T) = 0$$

**Intuition:** This is a bad estimate; real  $p(H)$  should be closer to half

When have less data, maximum likelihood overfits, want a more reasonable estimate...

- However, if we had just one data point, maximum likelihood places probability 1 on heads, which is a horrible idea. It's a very close-minded thing to do: just because we didn't see something doesn't mean it can't exist!
- This is an example of overfitting. If we had millions of parameters and only thousands of data points (which is not enough data), maximum likelihood would surely put 0 in many of the parameters.

# Regularization: Laplace smoothing

Maximum likelihood:

$$p(H) = \frac{1}{1} \quad p(T) = \frac{0}{1}$$

Maximum likelihood with Laplace smoothing:

$$p(H) = \frac{1+1}{1+2} = \frac{2}{3} \quad p(T) = \frac{0+1}{1+2} = \frac{1}{3}$$

- There is a very simple fix to this called **Laplace smoothing**: just add 1 to the count for each possible value, regardless of whether it was observed or not.
- Here, both heads and tails get an extra count of 1.

# Example: two variables

$$\mathcal{D}_{\text{train}} = \{(d, 4), (d, 5), (c, 5)\}$$

Amount of smoothing:  $\lambda = 1$

$\theta:$

$g$	$p_G(g)$
d	3/5
c	2/5

$g$	$r$	$p_R(r   g)$
d	1	1/7
d	2	1/7
d	3	1/7
d	4	2/7
d	5	2/7
c	1	1/6
c	2	1/6
c	3	1/6
c	4	1/6
c	5	2/6

- As a concrete example, let's revisit the two-variable model from before.
- For example, d occurs 2 times, but ends up at 3 due to adding  $\lambda = 1$ . In particular, many values which were never observed in the data have positive probability as desired.

# Regularization: Laplace smoothing



## Key idea: Laplace smoothing

For each distribution  $d$  and partial assignment  $(x_{\text{Parents}(i)}, x_i)$ , add  $\lambda$  to  $\text{count}_d(x_{\text{Parents}(i)}, x_i)$ .

Then normalize to get probability estimates.

Interpretation: hallucinate  $\lambda$  occurrences of each local assignment

Larger  $\lambda \Rightarrow$  more smoothing  $\Rightarrow$  probabilities closer to uniform.

Data wins out in the end:

$$p(H) = \frac{1+1}{1+2} = \frac{2}{3} \quad p(H) = \frac{998+1}{998+2} = 0.999$$

- More generally, we can add a number  $\lambda > 0$  (sometimes called a **pseudocount**) to the count for each distribution  $d$  and local assignment  $(x_{\text{Parents}(i)}, x_i)$ .
- By varying  $\lambda$ , we can control how much we are smoothing.
- No matter what the value of  $\lambda$  is, as we get more and more data, the effect of  $\lambda$  will diminish. This is desirable, since if we have a lot of data, we should be able to trust our data more.



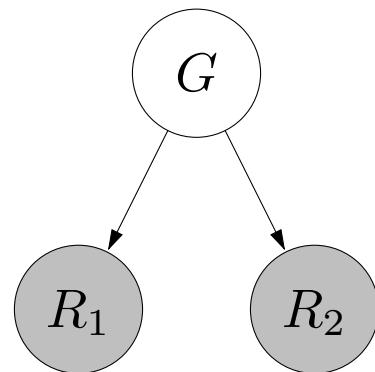
# Roadmap

Supervised learning

Laplace smoothing

**Unsupervised learning with EM**

# Motivation



What if we **don't observe** some of the variables?

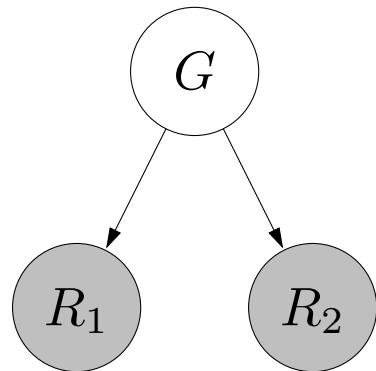
$$\mathcal{D}_{\text{train}} = \{(\textcolor{red}{?}, 4, 5), (\textcolor{red}{?}, 4, 4), (\textcolor{red}{?}, 5, 3), (\textcolor{red}{?}, 1, 2), (\textcolor{red}{?}, 5, 4)\}$$

- Data collection is hard, and often we don't observe the value of every single variable. In this example, we only see the ratings  $(R_1, R_2)$ , but not the genre  $G$ . Can we learn in this setting, which is clearly more difficult?
- Intuitively, it might seem hopeless. After all, how can we ever learn anything about the relationship between  $G$  and  $R_1$  if we never observe  $G$  at all?
- Indeed, if we don't observe enough of the variables, we won't be able to learn anything. But in many cases, we can.

# Maximum marginal likelihood

Variables:  $H$  is hidden,  $E = e$  is observed

Example:



$$H = G \quad E = (R_1, R_2) \quad e = (1, 2)$$
$$\theta = (p_G, p_R)$$

Maximum marginal likelihood objective:

$$\begin{aligned} & \max_{\theta} \prod_{e \in \mathcal{D}_{\text{train}}} \mathbb{P}(E = e; \theta) \\ &= \max_{\theta} \prod_{e \in \mathcal{D}_{\text{train}}} \sum_h \mathbb{P}(H = h, E = e; \theta) \end{aligned}$$

- Let's try to solve this problem top-down — what do we want, mathematically?
- Formally we have a set of hidden variables  $H$ , observed variables  $E$ , and parameters  $\theta$  which define all the local conditional distributions. We observe  $E = e$ , but we don't know  $H$  or  $\theta$ .
- If there were no hidden variables, then we would just use maximum likelihood:  $\max_{\theta} \prod_{(h,e) \in \mathcal{D}_{\text{train}}} \mathbb{P}(H = h, E = e; \theta)$ . But since  $H$  is unobserved, we can simply replace the joint probability  $\mathbb{P}(H = h, E = e; \theta)$  with the marginal probability  $\mathbb{P}(E = e; \theta)$ , which is just a sum over values  $h$  that the hidden variables  $H$  could take on.

# Expectation Maximization (EM)

Intuition: generalization of the K-means algorithm

Variables:  $H$  is hidden,  $E = e$  is observed



## Algorithm: Expectation Maximization (EM)

Initialize  $\theta$

E-step:

- Compute  $q(h) = \mathbb{P}(H = h \mid E = e; \theta)$  for each  $h$  (use any probabilistic inference algorithm)
- Create weighted points:  $(h, e)$  with weight  $q(h)$

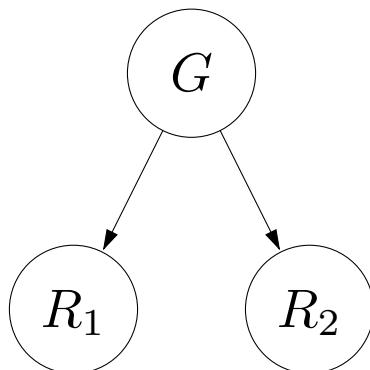
M-step:

- Compute maximum likelihood (just count and normalize) to get  $\theta$

Repeat until convergence.

- One of the most remarkable algorithms, Expectation Maximization (EM), tries to maximize the marginal likelihood.
- To get intuition for EM, consider K-means, which turns out to be a special case of EM (for Gaussian mixture models with variance tending to 0). In K-means, we had to somehow estimate the cluster centers, but we didn't know which points were assigned to which clusters. And in that setting, we took an alternating optimization approach: find the best cluster assignment given the current cluster centers, find the best cluster centers given the assignments, etc.
- The EM algorithm works analogously. EM consists of alternating between two steps, the E-step and the M-step. In the E-step, we don't know what the hidden variables are, so we compute the posterior distribution over them given our current parameters ( $\mathbb{P}(H \mid E = e; \theta)$ ). This can be done using any probabilistic inference algorithm. If  $H$  takes on a few values, then we can enumerate over all of them. If  $\mathbb{P}(H, E)$  is defined by an HMM, we can use the forward-backward algorithm. These posterior distributions provide a weight  $q(h)$  (which is a temporary variable in the EM algorithm) to every value  $h$  that  $H$  could take on. Conceptually, the E-step then generates a set of weighted full assignments  $(h, e)$  with weight  $q(h)$ . (In implementation, we don't need to create the data points explicitly.)
- In the M-step, we take in our set of full assignments  $(h, e)$  with weights, and we just do maximum likelihood estimation, which can be done in closed form — just counting and normalizing (perhaps with smoothing if you want)!
- If we repeat the E-step and the M-step over and over again, we are guaranteed to converge to a **local optima**. Just like the K-means algorithm, we might need to run the algorithm from different random initializations of  $\theta$  and take the best one.

# Example: one iteration of EM



$$\mathcal{D}_{\text{train}} = \{(\text{?, } 2, 2), (\text{?, } 1, 2)\}$$

$\theta:$	$g$	$p_G(g)$
	c	0.5
	c	0.4
	d	0.6
	d	0.5

$g$	$r$	$p_R(r   g)$
c	1	0.4
c	2	0.6
d	1	0.6
d	2	0.4

E-step



$(r_1, r_2)$	$g$	$\mathbb{P}(G = g, R_1 = r_1, R_2 = r_2)$	$q(g)$
(2, 2)	c	$0.5 \cdot 0.6 \cdot 0.6 = 0.18$	$\frac{0.18}{0.18+0.08} = 0.69$
(2, 2)	d	$0.5 \cdot 0.4 \cdot 0.4 = 0.08$	$\frac{0.08}{0.18+0.08} = 0.31$
(1, 2)	c	$0.5 \cdot 0.4 \cdot 0.6 = 0.12$	$\frac{0.12}{0.12+0.12} = 0.5$
(1, 2)	d	$0.5 \cdot 0.6 \cdot 0.4 = 0.12$	$\frac{0.12}{0.12+0.12} = 0.5$

M-step



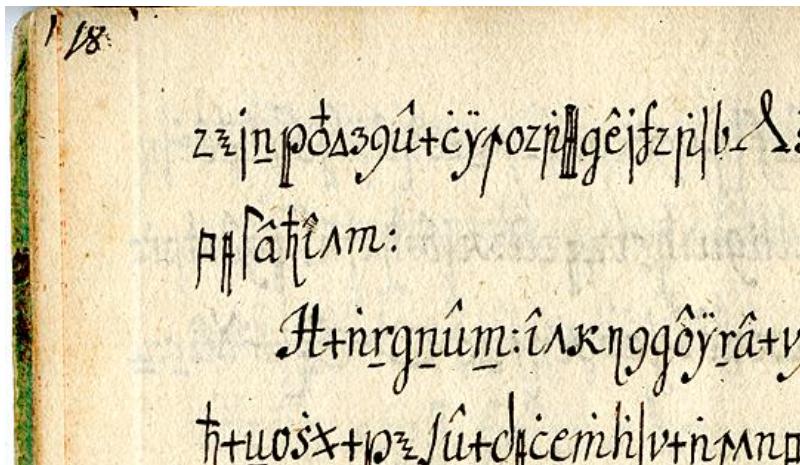
$g$	count	$p_G(g)$
c	0.69 + 0.5	0.59
d	0.31 + 0.5	0.41

$g$	$r$	count	$p_R(r   g)$
c	1	0.5	0.21
c	2	0.5 + 0.69 + 0.69	0.79
d	1	0.5	0.31
d	2	0.5 + 0.31 + 0.31	0.69

- In the E-step, we are presented with the current set of parameters  $\theta$ . We go through all the examples (in this case  $(2, 2)$  and  $(1, 2)$ ). For each example  $(r_1, r_2)$ , we will consider all possible values of  $g$  ( $c$  or  $d$ ), and compute the posterior distribution  $q(g) = \mathbb{P}(G = g | R_1 = r_1, R_2 = r_2)$ .
- The easiest way to do this is to write down the joint probability  $\mathbb{P}(G = g, R_1 = r_1, R_2 = r_2)$  because this is just simply a product of the parameters. For example, the first line is the product of  $p_G(c) = 0.5$ ,  $p_R(2 | c) = 0.6$  for  $r_1 = 2$ , and  $p_R(2 | c) = 0.6$  for  $r_2 = 2$ . For each example  $(r_1, r_2)$ , we normalize these joint probability to get  $q(g)$ .
- Now each row consists of a fictitious data point with  $g$  filled in, but appropriately weighted according to the corresponding  $q(g)$ , which is based on what we currently believe about  $g$ .
- In the M-step, for each of the parameters (e.g.,  $p_G(c)$ ), we simply add up the weighted number of times that parameter was used in the data (e.g., 0.69 for  $(c, 2, 2)$  and 0.5 for  $(c, 1, 2)$ ). Then we normalize these counts to get probabilities.
- If we compare the old parameters and new parameters after one round of EM, you'll notice that parameters tend to sharpen (though not always): probabilities tend to move towards 0 or 1.

# Application: decipherment

Copiale cipher (105-page encrypted volume from 1730s):



Cracked in 2011 with the help of EM!

- Let's now look at an interesting potential application of EM (or Bayesian networks in general): decipherment. Given a ciphertext (a string), how can we decipher it?
- The Copiale cipher was deciphered in 2011 (it turned out to be the handbook of a German secret society), largely with the help of Kevin Knight. On a related note, he has been quite interested in using probabilistic models (learned with variants of EM) for decipherment. Real ciphers are a bit too complex, so we will focus on the simple case of substitution ciphers.

# Substitution ciphers

Letter substitution table (unknown):

Plain:	abcdefghijklmnopqrstuvwxyz
Cipher:	plokijnuhbygvtfcrdxeszaqw

Plaintext (unknown): hello world

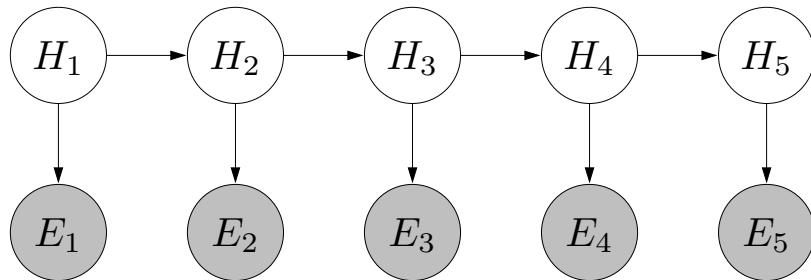
Ciphertext (known): **nmyyt ztryk**

- The input to decipherment is a ciphertext. Let's put on our modeling hats and think about how this ciphertext came to be.
- Most ciphers are quite complicated, so let's consider a simple substitution cipher. Someone comes up with a permutation of the letters (e.g., "a" maps to "p"). You can think about these as the unknown parameters of the model. Then they think of something to say — the plaintext (e.g., "hello world"). Finally, they apply the substitution table to generate the ciphertext (deterministically).

# Application: decipherment as an HMM

Variables:

- $H_1, \dots, H_n$  (e.g., characters of plaintext)
- $E_1, \dots, E_n$  (e.g., characters of ciphertext)

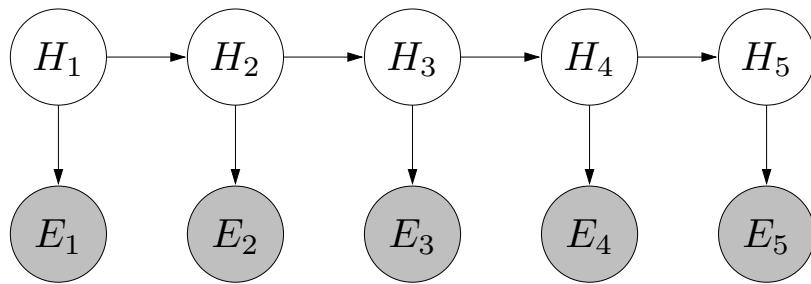


$$\mathbb{P}(H = h, E = e) = p_{\text{start}}(h_1) \prod_{i=2}^n p_{\text{trans}}(h_i \mid h_{i-1}) \prod_{i=1}^n p_{\text{emit}}(e_i \mid h_i)$$

Parameters:  $\theta = (p_{\text{start}}, p_{\text{trans}}, p_{\text{emit}})$

- We can formalize this process as an HMM as follows. The hidden variables are the plaintext and the observations are the ciphertext. Each character of the plaintext is related to the corresponding character in the ciphertext based on the cipher, and the transitions encode the fact that the characters in English are highly dependent on each other. For simplicity, we use a character-level bigram model (though  $n$ -gram models would yield better results).

# Application: decipherment as an HMM



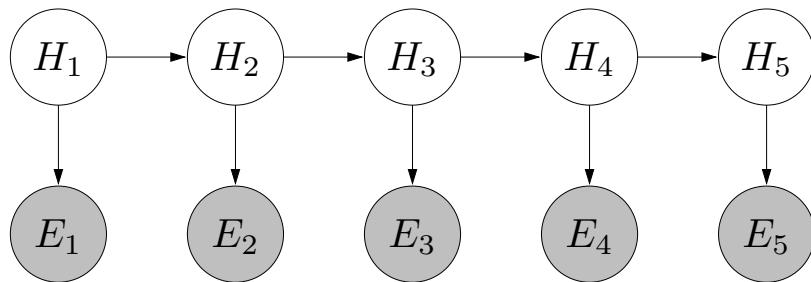
Strategy:

- $p_{\text{start}}$ : set to uniform
- $p_{\text{trans}}$ : estimate on tons of English text
- $p_{\text{emit}}$ : **substitution table**, from EM

Intuition: rely on language model ( $p_{\text{trans}}$ ) to favor plaintexts  $h$  that look like English

- We need to specify how we estimate the starting probabilities  $p_{\text{start}}$ , the transition probabilities  $p_{\text{trans}}$ , and the emission probabilities  $p_{\text{emit}}$ .
- The **starting probabilities** we won't care about so much and just set to a uniform distribution.
- The **transition probabilities** specify how someone might have generated the plaintext. We can estimate  $p_{\text{trans}}$  on a large corpora of English text. Note we need not use the same data to estimate all the parameters of the model. Indeed, there is generally much more English plaintext lying around than ciphertext.
- The **emission probabilities** encode the substitution table. Here, we know that the substitution table is deterministic, but we let the parameters be general distributions, which can certainly encode deterministic functions (e.g.,  $p_{\text{emit}}(p | a) = 1$ ). We use EM to only estimate the emission probabilities.
- We emphasize that the principal difficulty here is that we neither know the plaintext nor the parameters!

# Application: decipherment as an HMM



E-step: forward-backward algorithm computes

$$q_i(h) \stackrel{\text{def}}{=} \mathbb{P}(H_i = h \mid E_1 = e_1, \dots, E_n = e_n)$$

M-step: count (fractional) and normalize

$$\text{count}_{\text{emit}}(h, e) = \sum_{i=1}^n q_i(h) \cdot [e_i = e]$$

$$p_{\text{emit}}(e \mid h) \propto \text{count}_{\text{emit}}(h, e)$$

[live solution]

- Let's focus on the EM algorithm for estimating the emission probabilities. In the E-step, we can use the forward-backward algorithm to compute the posterior distribution over hidden assignments  $\mathbb{P}(H | E = e)$ . More precisely, the algorithm returns  $q_i(h) \stackrel{\text{def}}{=} \mathbb{P}(H_i = h | E = e)$  for each position  $i = 1, \dots, n$  and possible hidden state  $h$ .
- We can use  $q_i(h)$  as fractional counts of each  $H_i$ . To compute the counts  $\text{count}_{\text{emit}}(h, e)$ , we loop over all the positions  $i$  where  $E_i = e$  and add the fractional count  $q_i(h)$ .
- As you can see from the demo, the result isn't perfect, but not bad given the difficulty of the problem and the simplicity of the approach.



# Summary

(Bayesian network without parameters) + training examples

**Learning:** maximum likelihood (+Laplace smoothing, +EM)



$$Q \mid E \Rightarrow \boxed{\text{Parameters } \theta \\ (\text{of Bayesian network})} \Rightarrow \mathbb{P}(Q \mid E; \theta)$$



# Lecture 16: Logic I





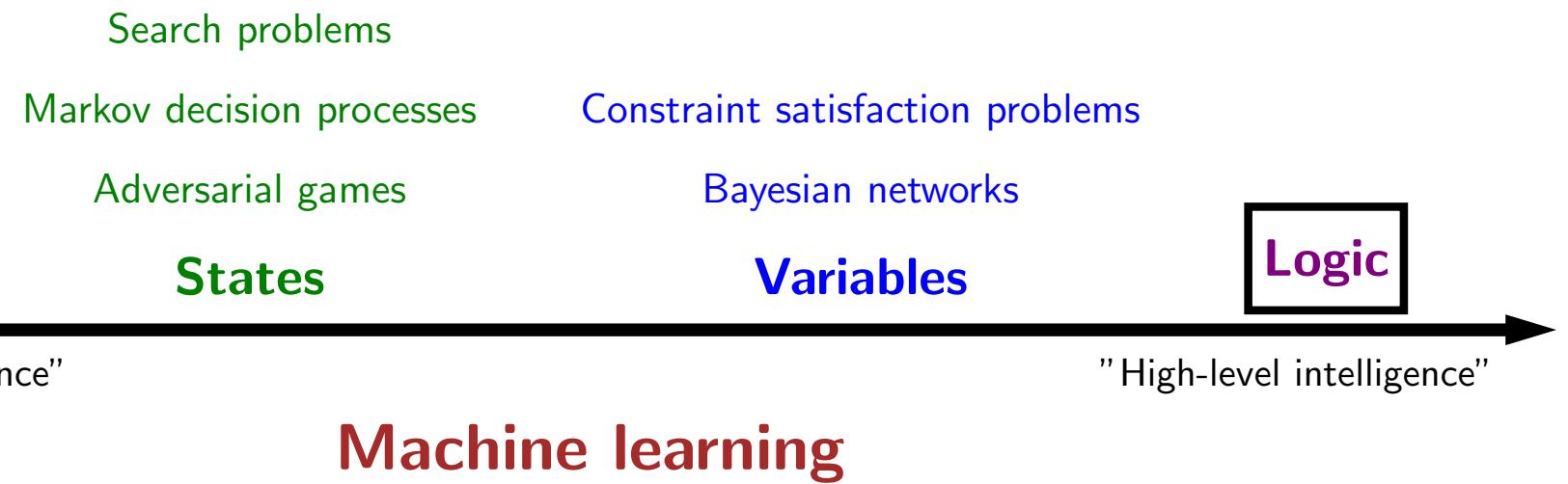
cs221.stanford.edu/q

# Question

If  $X_1 + X_2 = 10$  and  $X_1 - X_2 = 4$ , what is  $X_1$ ?

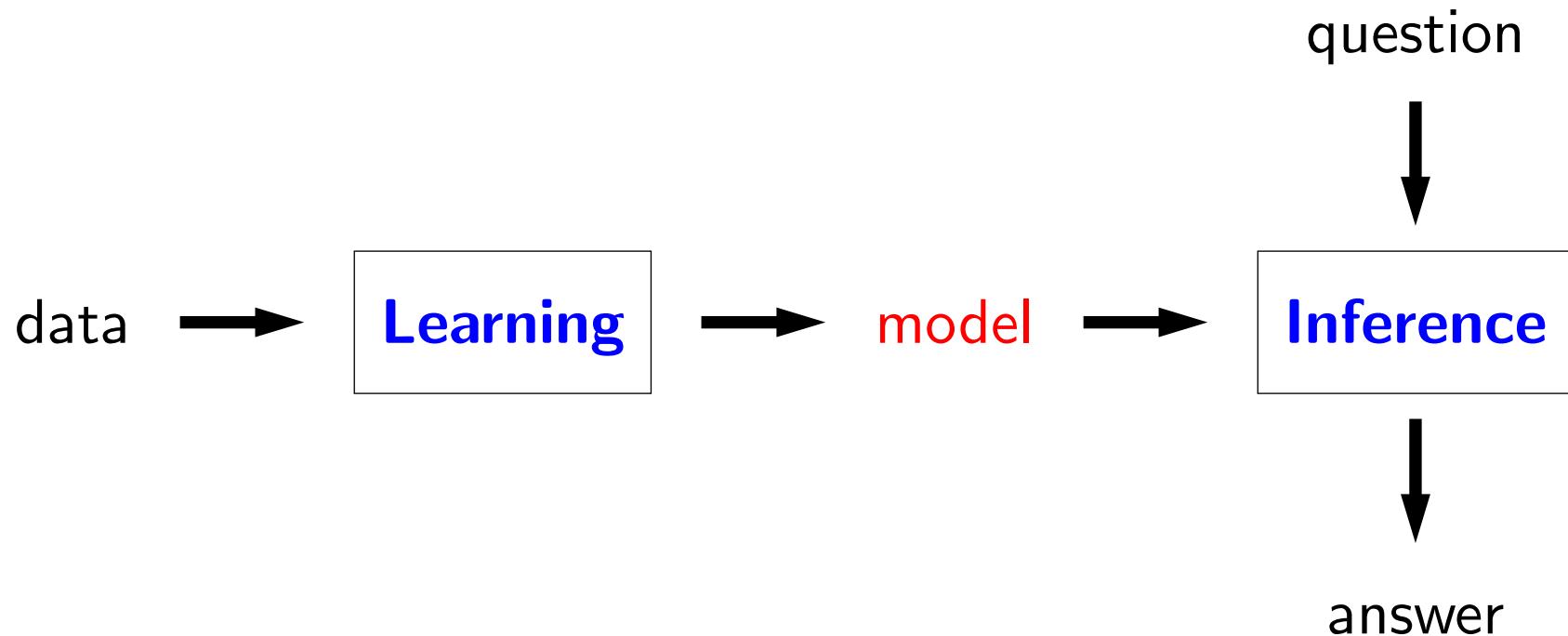
- Think about how you solved this problem. You could treat it as a CSP with variables  $X_1$  and  $X_2$ , and search through the set of candidate solutions, checking the constraints.
- However, more likely, you just added the two equations, divided both sides by 2 to easily find out that  $X_1 = 7$ . This is the power of **logical inference**, where we apply a set of truth-preserving rules to arrive at the answer. This is in contrast to what is called **model checking** (for reasons that will become clear), which tries to directly find assignments.
- We'll see that logical inference allows you to perform very powerful manipulations in a very compact way. This allows us to vastly increase the representational power of our models.

# Course plan



- We are at the last stage of our journey through the AI topics of this course: logic. Before launching in, let's take a moment to reflect.

# Taking a step back



Examples: search problems, MDPs, games, CSPs, Bayesian networks

- For each topic (e.g., MDPs) that we've studied, we followed the modeling-inference-learning paradigm: We take some data, feed it into a learning algorithm to produce a model with tuned parameters. Then we take this model and use it to perform inference (turning questions into answers).
- For search problems, the question is "what is the minimum cost path?" Inference algorithms such as DFS, UCS or A\* produced the minimum cost path. Learning algorithms such as the structured Perceptron filled in the action costs based on data (minimum cost paths).
- For MDPs and games, the question is "what is the maximum value policy?" Inference algorithms such as value iteration or minimax produced this. Learning algorithms such as Q-learning or TD learning allow you to work when we don't know the transitions and rewards.
- For CSPs, the question is "what is the maximum weight assignment?" Inference algorithms such as backtracking search, beam search, or variable elimination find such an assignment. We did not discuss learning algorithms here, but something similar to the structured Perceptron works.
- For Bayesian networks, the question is "what is the probability of a query given evidence?" Inference algorithms such as Gibbs sampling and particle filtering compute these probabilistic inference queries. Learning: if we don't know the local conditional distributions, we can learn them using maximum likelihood.
- We can think of learning as induction, where we need to generalize, and inference as deduction, where it's about computing the best predicted answer under the model.

# Modeling paradigms

**State-based models:** search problems, MDPs, games

Applications: route finding, game playing, etc.

*Think in terms of **states, actions, and costs***

**Variable-based models:** CSPs, Bayesian networks

Applications: scheduling, tracking, medical diagnosis, etc.

*Think in terms of **variables and factors***

**Logic-based models:** propositional logic, first-order logic

Applications: theorem proving, verification, reasoning

*Think in terms of **logical formulas and inference rules***

- Each topic corresponded to a modeling paradigm. The way the modeling paradigm is set up influences the way we approach a problem.
- In state-based models, we thought about inference as finding minimum cost paths in a graph. This leads us to think in terms of states, actions, and costs.
- In variable-based models, we thought about inference as finding maximum weight assignments or computing conditional probabilities. There we thought about variables and factors.
- Now, we will talk about logic-based models, where inference is applying a set of rules. For these models, we will think in terms of logical formulas and inference rules.

# A historical note

- Logic was dominant paradigm in AI before 1990s

```
(ING/BY
  (PUSH NP/2
    (SERTP VP/3))
  (C- VP/VB
    (* IF THE SUBJECT WAS NOT PROPERLY DETERMINED IN A
       POSS-ING COMPLEMENT, LOOK FOR IT HERE.))

  )))

(NP/
  (CAT DET T
    ((GETF POSSPRO
      (* START OF THE NP
        NETWORK.))
     (ADDL AGJS (BUILDQ (POSS (NP (PRO *))))))
    (SERTP DET TH)
    (* IF THE DETERMINER IS A POSSESSIVE PROGN
       (MY, YOUR), CONSTRUCT THE POSSESSIVE MODIFIER AND USE
       'THE' FOR THE DETERMINER)

    ))
  (T (SERTP DET *))
  (C- NP/ART))
  (CAT PCD T
    (SERTP N (MUTILQ (PRO *)))
    (* A PROGRAM MAY PICK UP
      PP MODIFIERS IN NP/HEAD)

    (SERTP NU (GETF NUMBER))
    (T- NP/VB)
    (EN (WHETHER IT)
      (SERTP NTYP*)
      (IT- COMP/LTIVE
        (* CONSTRUCT THE COMPLEMENT STRUCTURE FOR SENTENCES
          SUCH AS 'I DON'T KNOW WHETHER HE LEFT.')))

    ))
```

- Problem 1: deterministic, didn't handle **uncertainty** (probability addresses this)
- Problem 2: rule-based, didn't allow fine tuning from **data** (machine learning addresses this)
- Strength: provides **expressiveness** in a compact way

- Historically, in AI, logic was the dominant paradigm before the 1990s, but this tradition fell out of favor with the rise of probability and machine learning.
- There were two reasons for this: First, logic as an inference mechanism was brittle and did not handle uncertainty, whereas probability offered a coherent framework for dealing with uncertainty.
- Second, people built rule-based systems which were tedious and did not scale up, whereas machine learning automated much of the fine-tuning of a system by using data.
- However, there is one strength of logic which has not quite yet been recouped by existing probability and machine learning methods, and that is the expressivity of the model.

# Motivation: smart personal assistant



- How can we motivate logic-based models? We will take a little bit of a detour and think about an AI grand challenge: building smart personal assistants.
- Today, we have systems like Apple's Siri, Microsoft's Cortana, Amazon's Alexa, and Google Assistant.

# Motivation: smart personal assistant

Tell information



Ask questions



Use natural language!

[demo: python nli.py]

Need to:

- Digest **heterogenous** information
- Reason **deeply** with that information

- We would like to have more intelligent assistants such as Data from Star Trek. What is the functionality that's missing in between?
- At an abstract level, one fundamental thing a good personal assistant should be able to do is to take in information from people and be able to answer questions that require drawing inferences from the facts.
- In some sense, telling the system information is like machine learning, but it feels like a very different form of learning than seeing 10M images and their labels or 10M sentences and their translations. The type of information we get here is both more heterogenous, more abstract, and the expectation is that we process it more deeply (we don't want to have to tell our personal assistant 100 times that we prefer morning meetings).
- And how do we interact with our personal assistants? Let's use natural language, the very tool that was built for communication!



# Natural language

Example:

- A **dime** is better than a **nickel**.
- A **nickel** is better than a **penny**.
- Therefore, a **dime** is better than a **penny**.

Example:

- A **penny** is better than **nothing**.
- **Nothing** is better than **world peace**.
- Therefore, a **penny** is better than **world peace**???

Natural language is slippery...

- But natural language is tricky, because it is replete with ambiguities and vagueness. And drawing inferences using natural languages can be quite slippery. Of course, some concepts are genuinely vague and slippery, and natural language is as good as it gets, but that still leaves open the question of how a computer would handle those cases.

# Language

**Language** is a mechanism for expression.

Natural languages (informal):

English: Two divides even numbers.

German: Zwei dividieren geraden zahlen.

Programming languages (formal):

Python: `def even(x): return x % 2 == 0`

C++: `bool even(int x) { return x % 2 == 0; }`

Logical languages (formal):

First-order-logic:  $\forall x. \text{Even}(x) \rightarrow \text{Divides}(x, 2)$

- Let's think about language a bit deeply. What does it really buy you? Primarily, language is this wonderful human creation that allows us to express and communicate complex ideas and thoughts.
- We have mostly been talking about natural languages such as English and German. But as you all know, there are programming languages as well, which allow one to express computation formally so that a computer can understand it.
- This lecture is mostly about logical languages such as propositional logic and first-order logic. These are formal languages, but are a more suitable way of capturing declarative knowledge rather than concrete procedures, and are better connected with natural language.

# Two goals of a logic language

- **Represent** knowledge about the world



- **Reason** with that knowledge



- Some of you already know about logic, but it's important to keep the AI goal in mind: We want to use it to represent knowledge, and we want to be able to reason (or do inference) with that knowledge.
- Finally, we need to keep in mind that our goal is to get computers to use logic automatically, not for you to do it. This means that we need to think very mechanistically.

# Ingredients of a logic

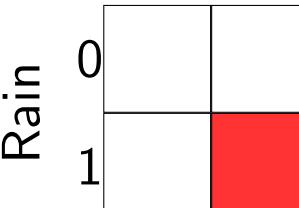
**Syntax:** defines a set of valid **formulas** (Formulas)

Example: Rain  $\wedge$  Wet

**Semantics:** for each formula, specify a set of **models** (assignments / configurations of the world)

Example:

		Wet
		0    1
Rain	0	
	1	



**Inference rules:** given  $f$ , what new formulas  $g$  can be added that are guaranteed to follow  $(\frac{f}{g})$ ?

Example: from Rain  $\wedge$  Wet, derive Rain

- The **syntax** defines a set of valid formulas, which are things which are grammatical to say in the language.
- **Semantics** usually doesn't receive much attention if you have a casual exposure to logic, but this is really the important piece that makes logic rigorous. Formally, semantics specifies the meaning of a formula, which in our setting is a set of configurations of the world in which the formula holds. This is what we care about in the end.
- But in order to get there, it's helpful to operate directly on the syntax using a set of **inference rules**. For example, if I tell you that it's raining and wet, then you should be able to conclude that it is also raining (obviously) without even explicitly mentioning semantics. Most of the time when people do logic casually, they are really just applying inference rules.

# Syntax versus semantics

Syntax: what are valid expressions in the language?

Semantics: what do these expressions mean?

Different syntax, same semantics (5):

$$2 + 3 \Leftrightarrow 3 + 2$$

Same syntax, different semantics (1 versus 1.5):

$$3 / 2 \text{ (Python 2.7)} \not\Leftrightarrow 3 / 2 \text{ (Python 3)}$$

- Just to hammer in the point that syntax and semantics are different, consider two examples from programming languages.
- First, the formula  $2 + 3$  and  $3 + 2$  are superficially different (a syntactic notion), but they have the same semantics (5).
- Second, the formula  $3 / 2$  means something different depending on which language. In Python 2.7, the semantics is 1 (integer division), and in Python 3 the semantics is 1.5 (floating point division).

# Logics

- Propositional logic with only Horn clauses
- Propositional logic
- Modal logic
- First-order logic with only Horn clauses
- First-order logic
- Second-order logic
- ...

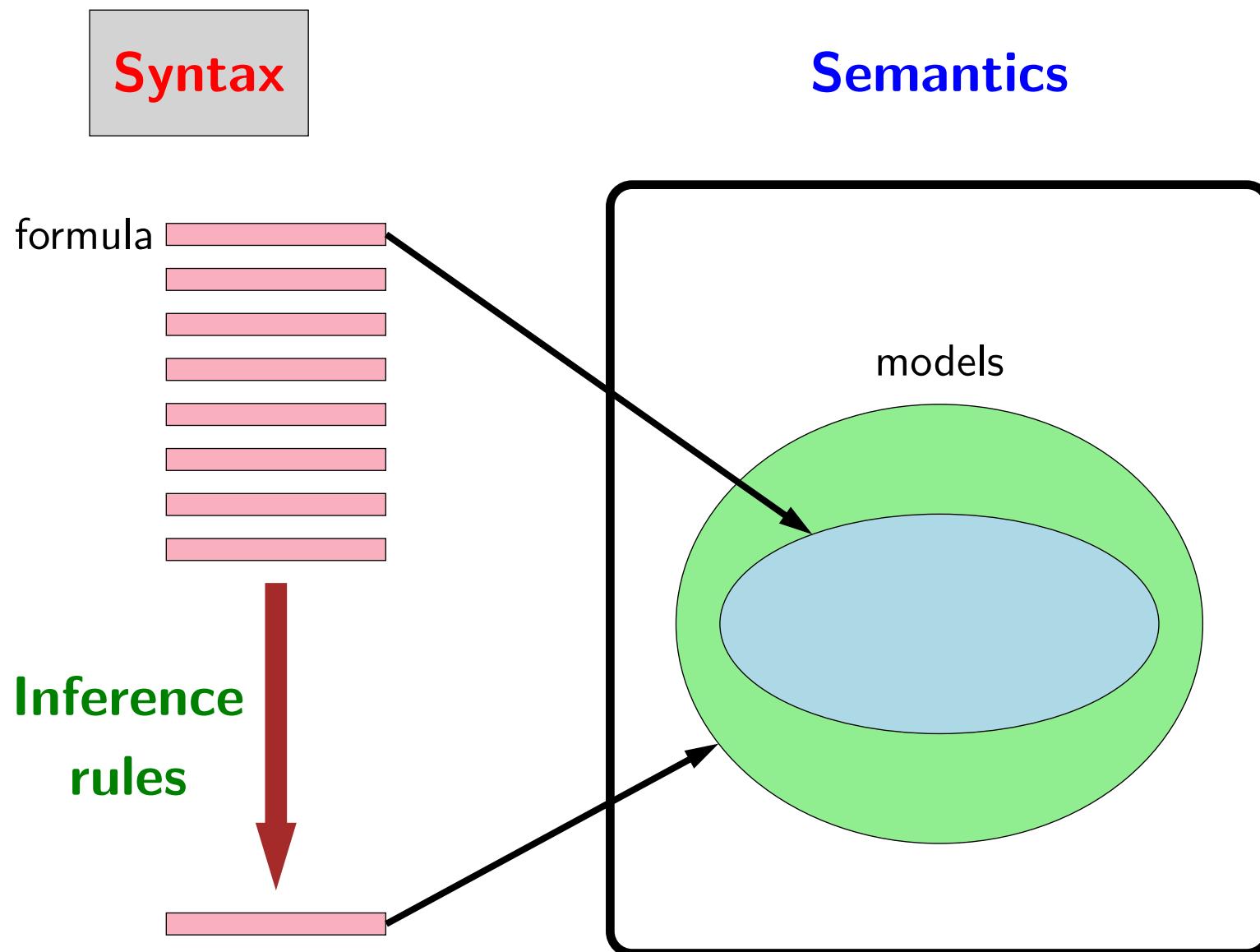


**Key idea: tradeoff**

Balance expressivity and computational efficiency.

- There are many different logical languages, just like there are programming languages. Whereas most programming languages have the expressive power (all Turing complete), logical languages exhibit a larger spectrum of expressivity.
- The bolded items are the ones we will discuss in this class.

# Propositional logic



- We begin with the syntax of propositional logic: what are the allowable formulas?

# Syntax of propositional logic

Propositional symbols (atomic formulas):  $A, B, C$

Logical connectives:  $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$

Build up formulas recursively—if  $f$  and  $g$  are formulas, so are the following:

- Negation:  $\neg f$
- Conjunction:  $f \wedge g$
- Disjunction:  $f \vee g$
- Implication:  $f \rightarrow g$
- Biconditional:  $f \leftrightarrow g$

- The building blocks of the syntax are the propositional symbols and connectives. The set of propositional symbols can be anything (e.g.,  $A$ , Wet, etc.), but the set of connectives is fixed to these five.
- All the propositional symbols are **atomic formulas** (also called atoms). We can **recursively** create larger formulas by combining smaller formulas using connectives.

# Syntax of propositional logic

- Formula:  $A$
- Formula:  $\neg A$
- Formula:  $\neg B \rightarrow C$
- Formula:  $\neg A \wedge (\neg B \rightarrow C) \vee (\neg B \vee D)$
- Formula:  $\neg\neg A$
- Non-formula:  $A \neg B$
- Non-formula:  $A + B$

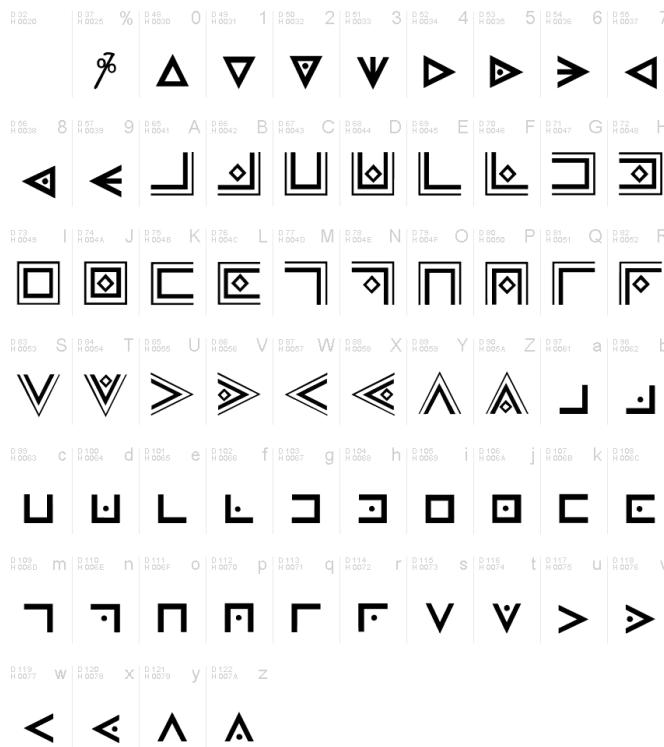
- Here are some examples of valid and invalid propositional formulas.

# Syntax of propositional logic



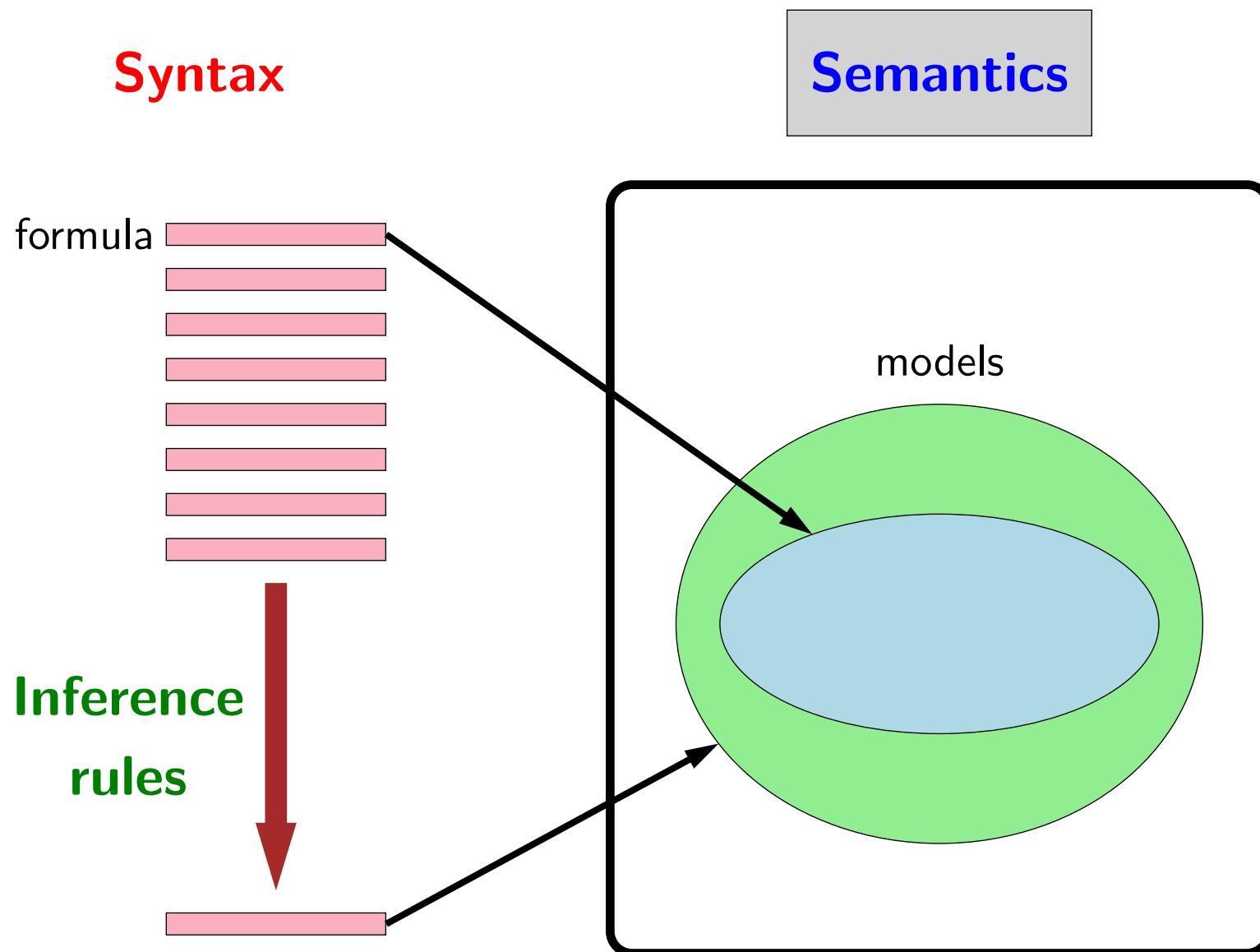
# Key idea: syntax provides symbols

Formulas by themselves are just symbols (syntax).  
No meaning yet (semantics)!



- It's important to remember that whenever we talk about syntax, we're just talking about symbols; we're not actually talking about what they mean — that's the role of semantics. Of course it will be difficult to ignore the semantics for propositional logic completely because you already have a working knowledge of what the symbols mean.

# Propositional logic



- Having defined the syntax of propositional logic, let's talk about their semantics or meaning.

# Model



## Definition: model

A **model**  $w$  in propositional logic is an **assignment** of truth values to propositional symbols.

## Example:

- 3 propositional symbols:  $A, B, C$
- $2^3 = 8$  possible models  $w$ :

$$\begin{aligned} & \{A : 0, B : 0, C : 0\} \\ & \{A : 0, B : 0, C : 1\} \\ & \{A : 0, B : 1, C : 0\} \\ & \{A : 0, B : 1, C : 1\} \\ & \{A : 1, B : 0, C : 0\} \\ & \{A : 1, B : 0, C : 1\} \\ & \{A : 1, B : 1, C : 0\} \\ & \{A : 1, B : 1, C : 1\} \end{aligned}$$

- In logic, the word **model** has a special meaning, quite distinct from the way we've been using it in the class (quite an unfortunate collision). A model (in the logical sense) represents a possible state of affairs in the world. In propositional logic, this is an assignment that specifies a truth value (true or false) for each propositional symbol.

# Interpretation function



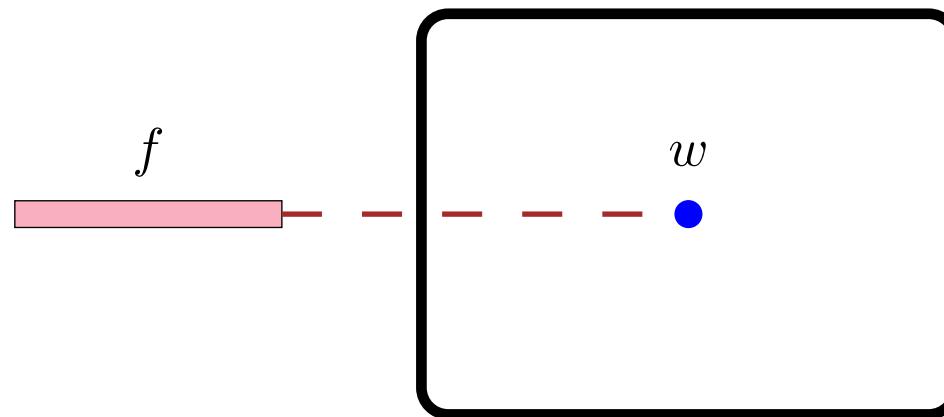
## Definition: interpretation function

Let  $f$  be a formula.

Let  $w$  be a model.

An **interpretation function**  $\mathcal{I}(f, w)$  returns:

- true (1) (say that  $w$  satisfies  $f$ )
- false (0) (say that  $w$  does not satisfy  $f$ )



- The semantics is given by an **interpretation function**, which takes a formula  $f$  and a model  $w$ , and returns whether  $w$  satisfies  $f$ . In other words, is  $f$  true in  $w$ ?
- For example, if  $f$  represents "it is Wednesday" and  $w$  corresponds to right now, then  $\mathcal{I}(f, w) = 1$ . If  $w$  corresponded to yesterday, then  $\mathcal{I}(f, w) = 0$ .

# Interpretation function: definition

Base case:

- For a propositional symbol  $p$  (e.g.,  $A, B, C$ ):  $\mathcal{I}(p, w) = w(p)$

Recursive case:

- For any two formulas  $f$  and  $g$ , define:

$\mathcal{I}(f, w)$	$\mathcal{I}(g, w)$	$\mathcal{I}(\neg f, w)$	$\mathcal{I}(f \wedge g, w)$	$\mathcal{I}(f \vee g, w)$	$\mathcal{I}(f \rightarrow g, w)$	$\mathcal{I}(f \leftrightarrow g, w)$
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

- The interpretation function is defined recursively, where the cases neatly parallel the definition of the syntax.
- Formally, for propositional logic, the interpretation function is fully defined as follows. In the base case, the interpretation of a propositional symbol  $p$  is just gotten by looking  $p$  up in the model  $w$ . For every possible value of  $(\mathcal{I}(f, w), \mathcal{I}(g, w))$ , we specify the interpretation of the combination of  $f$  and  $g$ .

# Interpretation function: example



## Example: interpretation function

Formula:  $f = (\neg A \wedge B) \leftrightarrow C$

Model:  $w = \{A : 1, B : 1, C : 0\}$

Interpretation:

$$\mathcal{I}((\neg A \wedge B) \leftrightarrow C, w) = 1$$

$$\mathcal{I}(\neg A \wedge B, w) = 0$$

$$\mathcal{I}(C, w) = 0$$

$$\mathcal{I}(\neg A, w) = 0$$

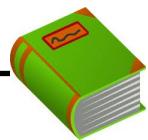
$$\mathcal{I}(B, w) = 1$$

$$\mathcal{I}(A, w) = 1$$

- For example, given the formula, we break down the formula into parts, recursively compute the truth value of the parts, and then finally combines these truth values based on the connective.

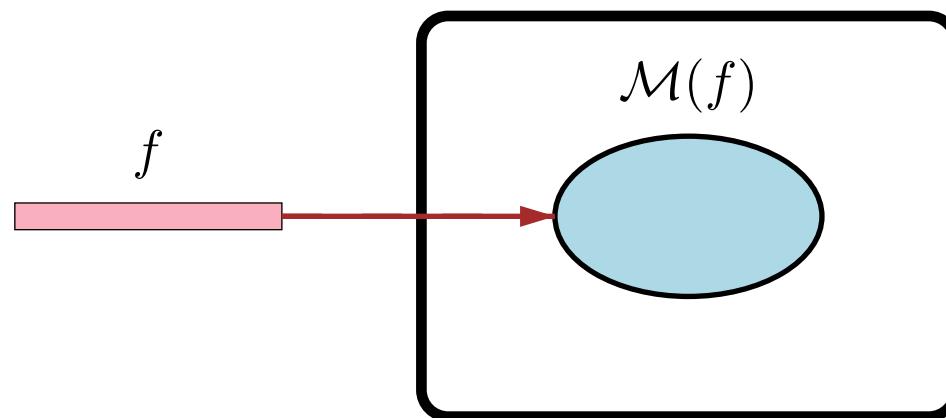
# Formula represents a set of models

So far: each formula  $f$  and model  $w$  has an interpretation  $\mathcal{I}(f, w) \in \{0, 1\}$



## Definition: models

Let  $\mathcal{M}(f)$  be the set of **models**  $w$  for which  $\mathcal{I}(f, w) = 1$ .



- So far, we've focused on relating a single model. A more useful but equivalent way to think about semantics is to think about the formula  $\mathcal{M}(f)$  as **a set of models** — those for which  $\mathcal{I}(f, w) = 1$ .

# Models: example

Formula:

$$f = \text{Rain} \vee \text{Wet}$$

Models:

$$\mathcal{M}(f) =$$

		Wet	
		0	1
Rain	0		
	1		



Key idea: compact representation

A **formula** compactly represents a set of **models**.

- In this example, there are four models for which the formula holds, as one can easily verify. From the point of view of  $\mathcal{M}$ , a formula's main job is to define a set of models.
- Recall that a model is a possible configuration of the world. So a formula like "it is raining" will pick out all the hypothetical configurations of the world where it's raining; in some of these configurations, it will be Wednesday; in others, it won't.

# Knowledge base



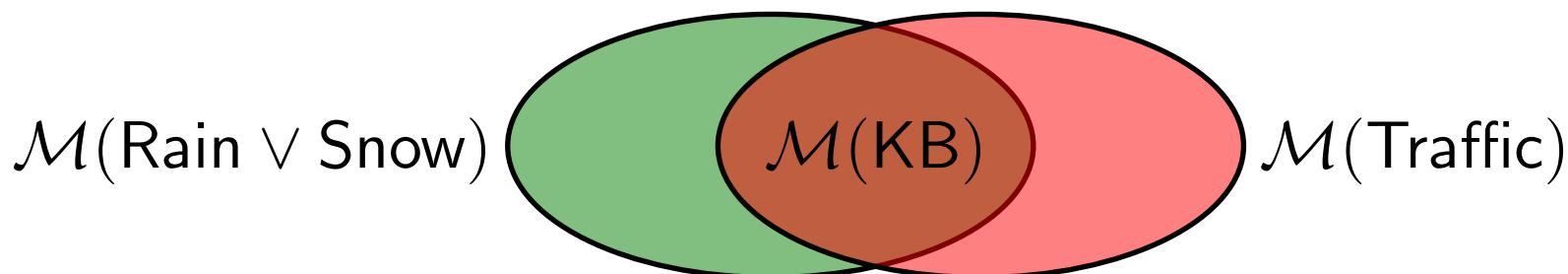
## Definition: Knowledge base

A **knowledge base** KB is a set of formulas representing their conjunction / intersection:

$$\mathcal{M}(\text{KB}) = \bigcap_{f \in \text{KB}} \mathcal{M}(f).$$

**Intuition:** KB specifies constraints on the world.  $\mathcal{M}(\text{KB})$  is the set of all worlds satisfying those constraints.

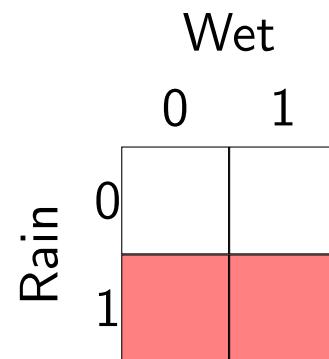
Let  $\text{KB} = \{\text{Rain} \vee \text{Snow}, \text{Traffic}\}$ .



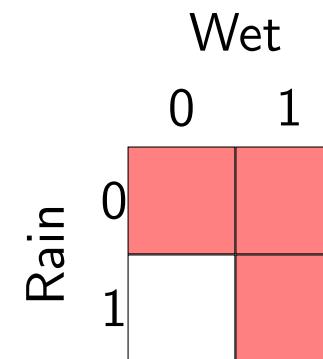
- If you take a set of formulas, you get a **knowledge base**. Each knowledge base defines a set of models — exactly those which are satisfiable by all the formulas in the knowledge base.
- Think of each formula as a fact that you know, and the **knowledge** is just the collection of those facts. Each fact narrows down the space of possible models, so the more facts you have, the fewer models you have.

# Knowledge base: example

$\mathcal{M}(\text{Rain})$

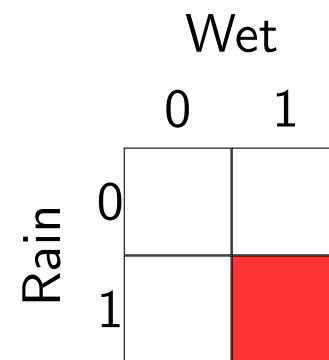


$\mathcal{M}(\text{Rain} \rightarrow \text{Wet})$



Intersection:

$\mathcal{M}(\{\text{Rain}, \text{Rain} \rightarrow \text{Wet}\})$



- As a concrete example, consider the two formulas Rain and Rain  $\rightarrow$  Wet. If you know both of these facts, then the set of models is constrained to those where it is raining and wet.

# Adding to the knowledge base

Adding more formulas to the knowledge base:

$$\text{KB} \longrightarrow \text{KB} \cup \{f\}$$

Shrinks the set of models:

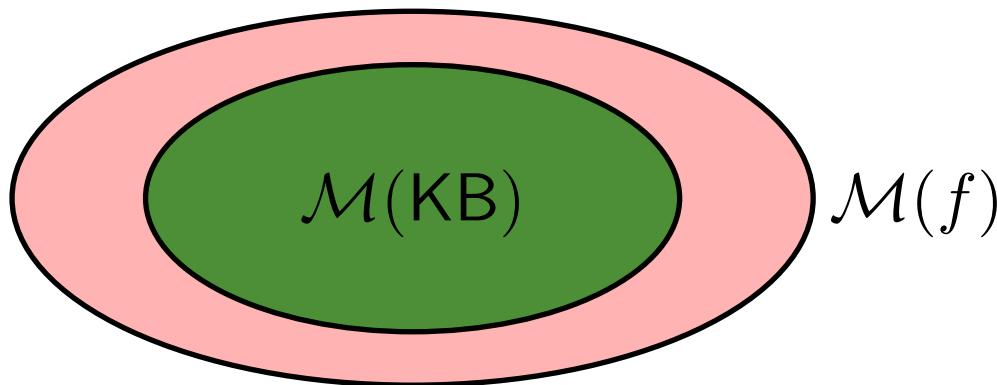
$$\mathcal{M}(\text{KB}) \longrightarrow \mathcal{M}(\text{KB}) \cap \mathcal{M}(f)$$

**How much does  $\mathcal{M}(\text{KB})$  shrink?**

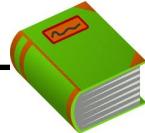
[whiteboard]

- We should think about a knowledge base as carving out a set of models. Over time, we will add additional formulas to the knowledge base, thereby winnowing down the set of models.
- Intuitively, adding a formula to the knowledge base imposes yet another constraint on our world, which naturally decreases the set of possible worlds.
- Thus, as the number of formulas in the knowledge base gets larger, the set of models gets smaller.
- A central question is how much  $f$  shrinks the set of models. There are three cases of importance.

# Entailment



**Intuition:**  $f$  added no information/constraints (it was already known).



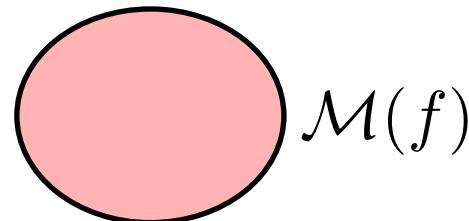
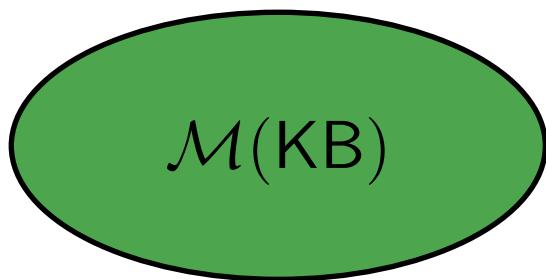
## Definition: entailment

KB entails  $f$  (written  $\text{KB} \models f$ ) iff  
 $\mathcal{M}(\text{KB}) \subseteq \mathcal{M}(f)$ .

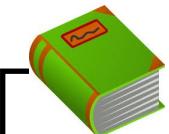
**Example:**  $\text{Rain} \wedge \text{Snow} \models \text{Snow}$

- The first case is if the set of models of  $f$  is a superset of the models of KB, then  $f$  adds no information. We say that KB **entails**  $f$ .

# Contradiction



Intuition:  $f$  contradicts what we know (captured in KB).



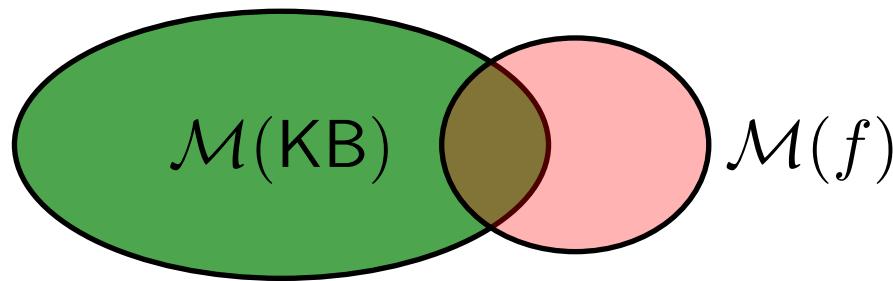
## Definition: contradiction

KB contradicts  $f$  iff  $\mathcal{M}(\text{KB}) \cap \mathcal{M}(f) = \emptyset$ .

Example: Rain  $\wedge$  Snow contradicts  $\neg\text{Snow}$

- The second case is if the set of models defined by  $f$  is completely disjoint from those of KB. Then we say that the KB and  $f$  **contradict** each other. If we believe KB, then we cannot possibly believe  $f$ .

# Contingency



**Intuition:**  $f$  adds non-trivial information to KB

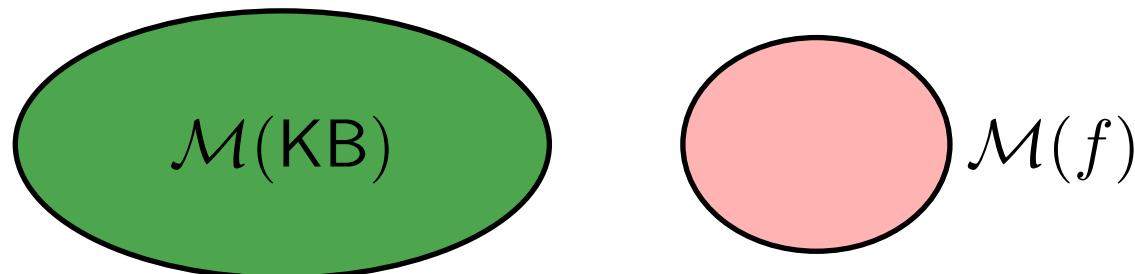
$$\emptyset \subsetneq \mathcal{M}(\text{KB}) \cap \mathcal{M}(f) \subsetneq \mathcal{M}(\text{KB})$$

**Example:** Rain and Snow

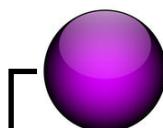
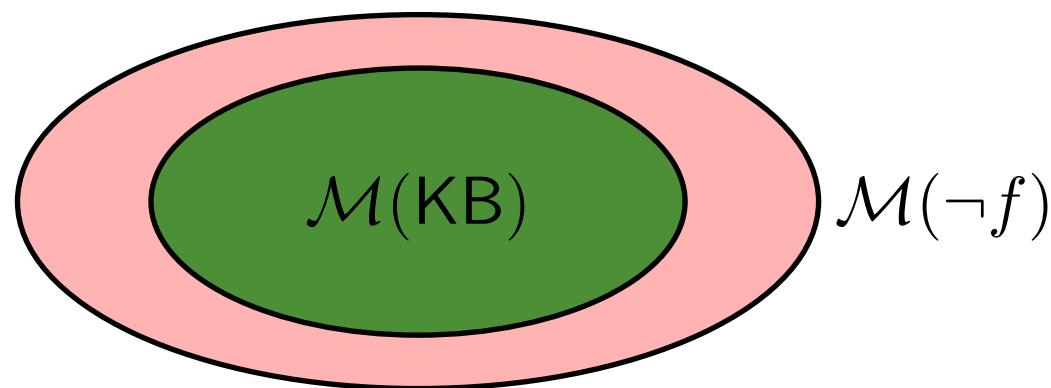
- In the third case, we have a non-trivial overlap between the models of  $\text{KB}$  and  $f$ . We say in this case that  $f$  is **contingent**;  $f$  could be satisfied or not satisfied depending on the model.

# Contradiction and entailment

Contradiction:



Entailment:

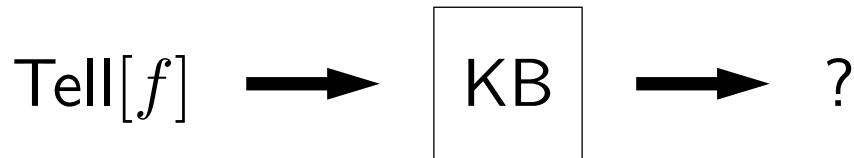


**Proposition: contradiction and entailment**

**KB contradicts  $f$  iff KB entails  $\neg f$ .**

- There is a useful connection between entailment and contradiction. If  $f$  is contradictory, then its negation ( $\neg f$ ) is entailed, and vice-versa.
- You can see this because the models  $\mathcal{M}(f)$  and  $\mathcal{M}(\neg f)$  partition the space of models.

# Tell operation



**Tell:** *It is raining.*

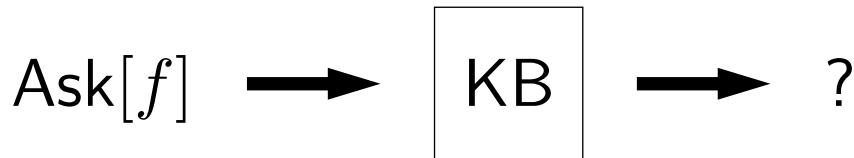
Tell[Rain]

Possible responses:

- Already knew that: entailment ( $\text{KB} \models f$ )
- Don't believe that: contradiction ( $\text{KB} \models \neg f$ )
- Learned something new (update KB): contingent

- Having defined the three possible relationships that a new formula  $f$  can have with respect to a knowledge base KB, let's try to determine the appropriate response that a system should have.
- Suppose we tell the system that it is raining ( $f = \text{Rain}$ ). If  $f$  is entailed, then we should reply that we already knew that. If  $f$  contradicts the knowledge base, then we should reply that we don't believe that. If  $f$  is contingent, then this is the interesting case, where we have non-trivially restricted the set of models, so we reply that we've learned something new.

# Ask operation



**Ask:** *Is it raining?*

Ask[Rain]

Possible responses:

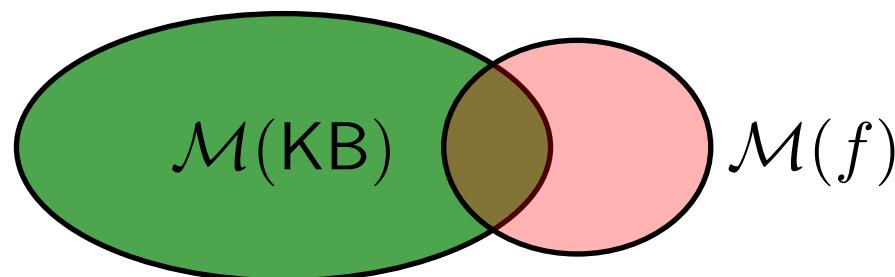
- Yes: entailment ( $\text{KB} \models f$ )
- No: contradiction ( $\text{KB} \models \neg f$ )
- I don't know: contingent

- Suppose now that we ask the system a question: is it raining? If  $f$  is entailed, then we should reply with a definitive yes. If  $f$  contradicts the knowledge base, then we should reply with a definitive no. If  $f$  is contingent, then we should just confess that we don't know.

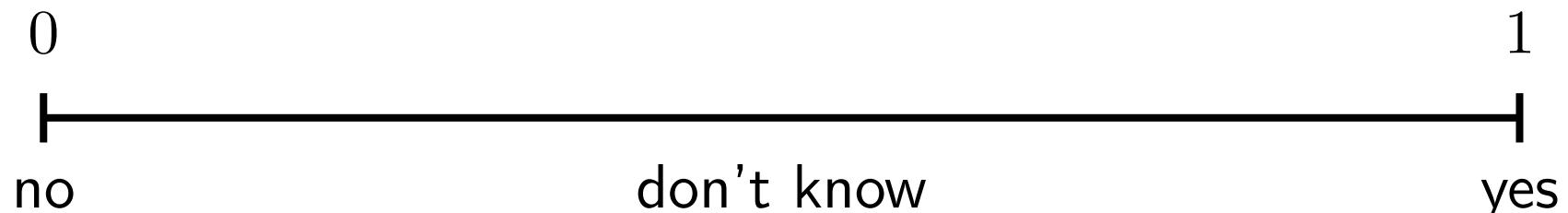
# Digression: probabilistic generalization

Bayesian network: distribution over assignments (models)

$w$	$\mathbb{P}(W = w)$
{ A: 0, B: 0, C: 0 }	0.3
{ A: 0, B: 0, C: 1 }	0.1
...	...



$$\mathbb{P}(f \mid \text{KB}) = \frac{\sum_{w \in \mathcal{M}(\text{KB} \cup \{f\})} \mathbb{P}(W = w)}{\sum_{w \in \mathcal{M}(\text{KB})} \mathbb{P}(W = w)}$$



- Note that logic captures uncertainty in a very crude way. We can't say that we're almost sure or not very sure or not sure at all.
- Probability can help here. Remember that a Bayesian network (or more generally a factor graph) defines a distribution over assignments to the variables in the Bayesian network. Then we could ask questions such as: conditioned on having a cough but not itchy eyes, what's the probability of having a cold?
- Recall that in propositional logic, models are just assignments to propositional symbols. So we can think of KB as the evidence that we're conditioning on, and  $f$  as the query.

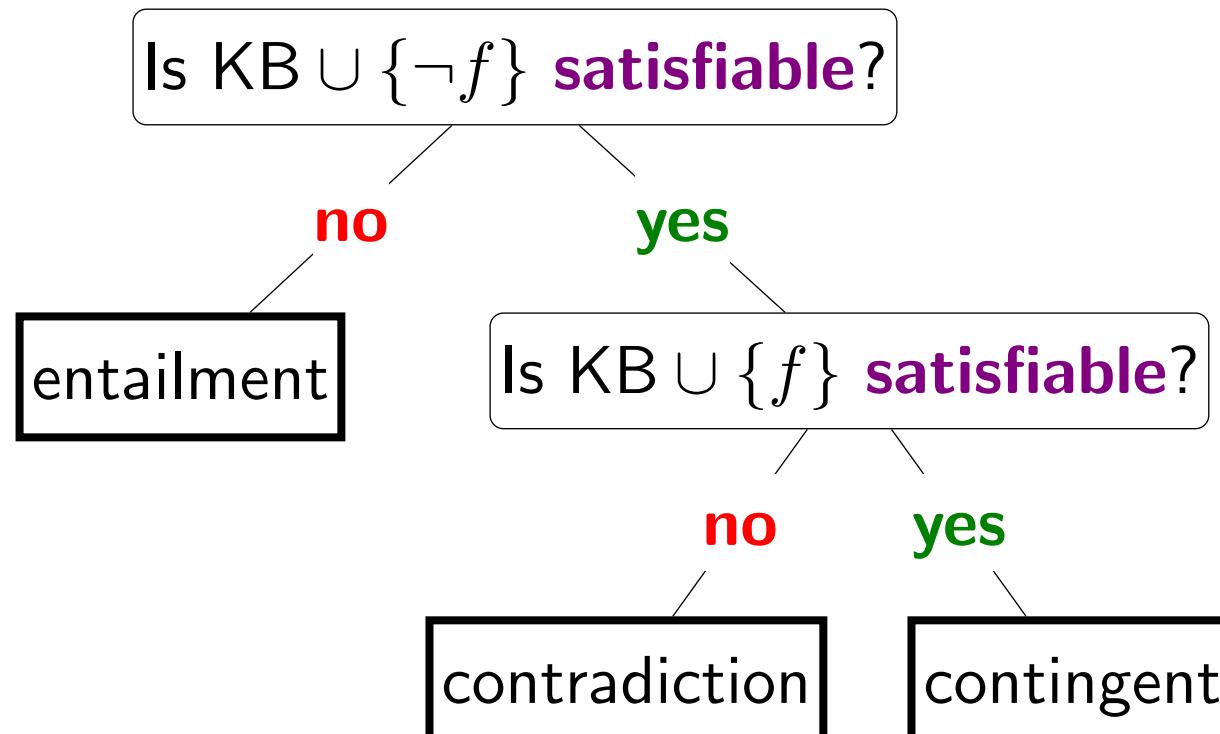
# Satisfiability



## Definition: satisfiability

A knowledge base KB is **satisfiable** if  $\mathcal{M}(\text{KB}) \neq \emptyset$ .

Reduce  $\text{Ask}[f]$  and  $\text{Tell}[f]$  to satisfiability:



- Now let's return to pure logic land again. How can we go about actually checking entailment, contradiction, and contingency? One useful concept to rule them all is **satisfiability**.
- Recall that we said a particular model  $w$  satisfies  $f$  if the interpretation function returns true  $\mathcal{I}(f, w) = 1$ . We can say that a formula  $f$  by itself is satisfiable if there is some model that satisfies  $f$ . Finally, a knowledge base (which is no more than just the conjunction of its formulas) is satisfiable if there is some model that satisfies all the formulas  $f \in \text{KB}$ .
- With this definition in hand, we can implement  $\text{Ask}[f]$  and  $\text{Tell}[f]$  as follows:
- First, we check if  $\text{KB} \cup \{\neg f\}$  is satisfiable. If the answer is no, that means the models of  $\neg f$  and  $\text{KB}$  don't intersect (in other words, the two contradict each other). Recall that this is equivalent to saying that  $\text{KB}$  entails  $f$ .
- Otherwise, we need to do another test: check whether  $\text{KB} \cup \{f\}$  is satisfiable. If the answer is no here, then  $\text{KB}$  and  $f$  are contradictory. Otherwise, we have that both  $f$  and  $\neg f$  are compatible with  $\text{KB}$ , so the result is contingent.

# Model checking

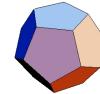
Checking satisfiability (SAT) in propositional logic is special case of solving CSPs!

Mapping:

propositional symbol	$\Rightarrow$	variable
formula	$\Rightarrow$	constraint
model	$\Leftarrow$	assignment

- Now we have reduced the problem of working with knowledge bases to checking satisfiability. The bad news is that this is an (actually, the canonical) NP-complete problem, so there are no efficient algorithms in general.
- The good news is that people try to solve the problem anyway, and we actually have pretty good SAT solvers these days. In terms of this class, this problem is just a CSP, if we convert the terminology: Each propositional symbol becomes a variable and each formula is a constraint. We can then solve the CSP, which produces an assignment, or in logic-speak, a model.

# Model checking



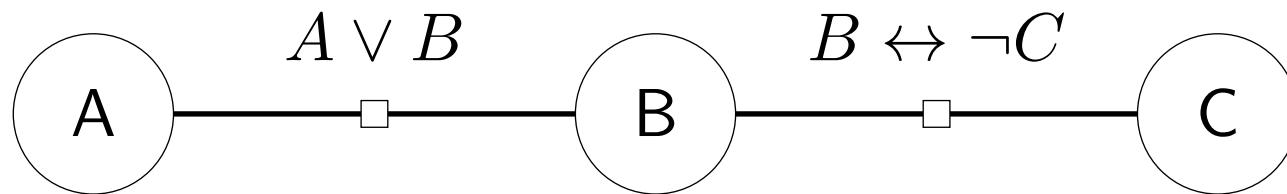
## Example: model checking

$$\text{KB} = \{A \vee B, B \leftrightarrow \neg C\}$$

Propositional symbols (CSP variables):

$$\{A, B, C\}$$

CSP:



Consistent assignment (satisfying model):

$$\{A : 1, B : 0, C : 1\}$$

- As an example, consider a knowledge base that has two formulas and three variables. Then the CSP is shown. Solving the CSP produces a consistent assignment (if one exists), which is a model that satisfies KB.
- Note that in the knowledge base tell/ask application, we don't technically need the satisfying assignment. An assignment would only offer a counterexample certifying that the answer **isn't** entailment or contradiction. This is an important point: entailment and contradiction is a claim about all models, not about the existence of a model.

# Model checking



## Definition: model checking

**Input:** knowledge base KB

**Output:** exists satisfying model ( $\mathcal{M}(\text{KB}) \neq \emptyset$ )?

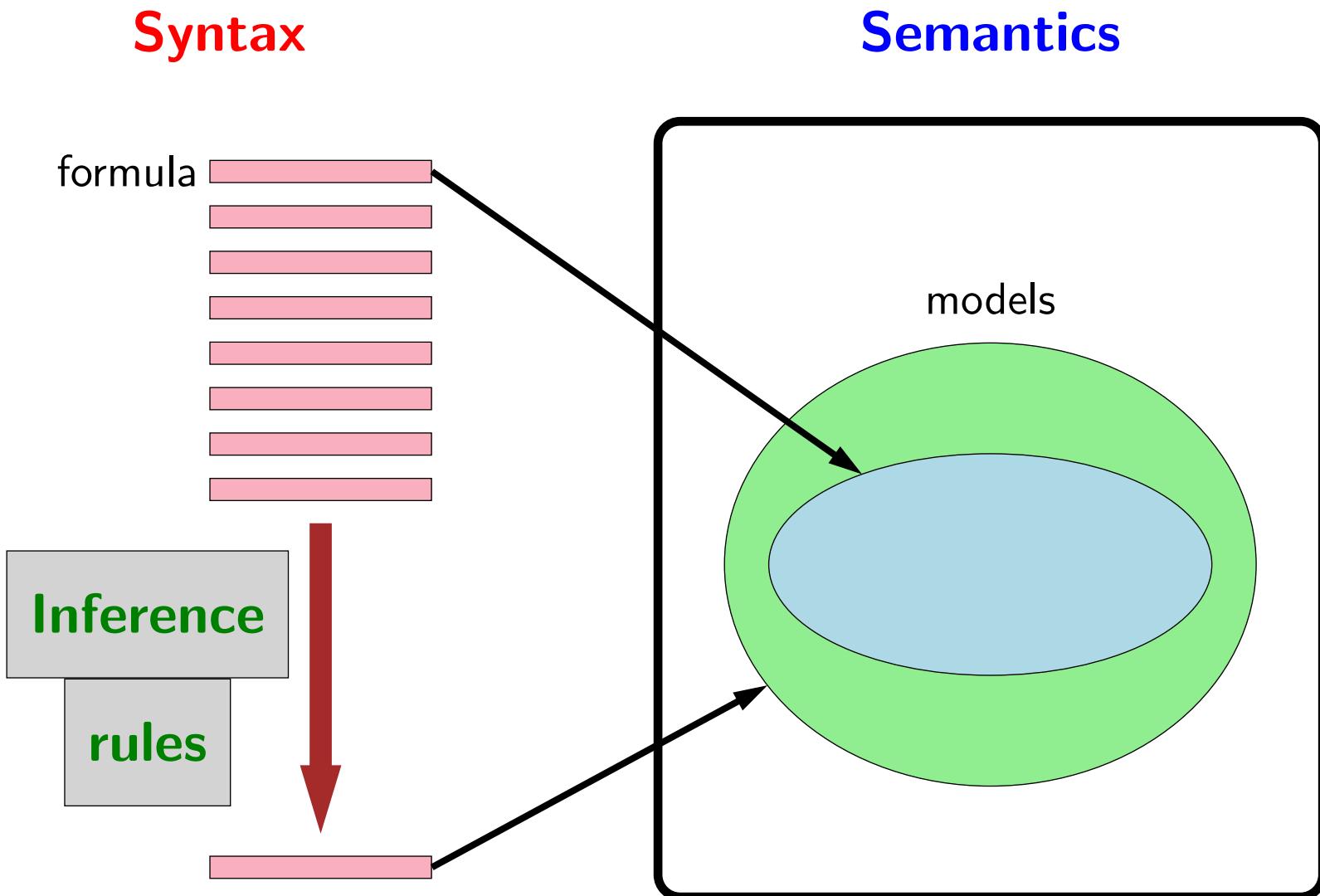
Popular algorithms:

- DPLL (backtracking search + pruning)
- WalkSat (randomized local search)

Next: Can we exploit the fact that factors are formulas?

- Checking satisfiability of a knowledge base is called **model checking**. For propositional logic, there are several algorithms that work quite well which are based on the algorithms we saw for solving CSPs (backtracking search and local search).
- However, can we do a bit better? Our CSP factors are not arbitrary — they are logic formulas, and recall that formulas are defined recursively and have some compositional structure. Let's see how to exploit this.

# Propositional logic



- So far, we have used formulas, via semantics, to define sets of models. And all our reasoning on formulas has been through these models (e.g., reduction to satisfiability). Inference rules allow us to do reasoning on the formulas themselves without ever instantiating the models.
- This can be quite powerful. If you have a huge KB with lots of formulas and propositional symbols, sometimes you can draw a conclusion without instantiating the full model checking problem. This will be very important when we move to first-order logic, where the models can be infinite, and so model checking would be infeasible.

# Inference rules

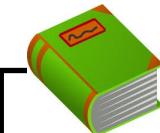
Example of making an inference:

It is raining. (Rain)

If it is raining, then it is wet. (Rain  $\rightarrow$  Wet)

Therefore, it is wet. (Wet)

$$\frac{\text{Rain}, \quad \text{Rain} \rightarrow \text{Wet}}{\text{Wet}} \quad \begin{matrix} \text{(premises)} \\ \text{(conclusion)} \end{matrix}$$



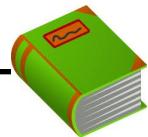
**Definition: Modus ponens inference rule**

For any propositional symbols  $p$  and  $q$ :

$$\frac{p, \quad p \rightarrow q}{q}$$

- The idea of making an inference should be quite intuitive to you. The classic example is **modus ponens**, which captures the if-then reasoning pattern.

# Inference framework



## Definition: inference rule

If  $f_1, \dots, f_k, g$  are formulas, then the following is an **inference rule**:

$$\frac{f_1, \quad \dots \quad , f_k}{g}$$



## Key idea: inference rules

Rules operate directly on **syntax**, not on **semantics**.

- In general, an inference rule has a set of premises and a conclusion. The rule says that if the premises are in the KB, then you can add the conclusion to the KB.
- We haven't yet specified whether this is a valid thing to do, but it is a thing to do. Remember, syntax is just about symbol pushing; it is only by linking to models that we have notions of truth and meaning (semantics).

# Inference algorithm



## Algorithm: forward inference

**Input:** set of inference rules Rules.

Repeat until no changes to KB:

    Choose set of formulas  $f_1, \dots, f_k \in \text{KB}$ .

    If matching rule  $\frac{f_1, \dots, f_k}{g}$  exists:

        Add  $g$  to KB.



## Definition: derivation

KB **derives/proves**  $f$  ( $\text{KB} \vdash f$ ) iff  $f$  eventually gets added to KB.

- Given a set of inference rules (e.g., modus ponens), we can just keep on trying to apply rules. Those rules generate new formulas which get added to the knowledge base, and those formulas might then be premises of other rules, which in turn generate more formulas, etc.
- We say that the KB derives or proves a formula  $f$  if by blindly applying rules, we can eventually add  $f$  to the KB.

# Inference example



## Example: Modus ponens inference

Starting point:

$$KB = \{\text{Rain}, \text{Rain} \rightarrow \text{Wet}, \text{Wet} \rightarrow \text{Slippery}\}$$

Apply modus ponens to Rain and Rain  $\rightarrow$  Wet:

$$KB = \{\text{Rain}, \text{Rain} \rightarrow \text{Wet}, \text{Wet} \rightarrow \text{Slippery}, \text{Wet}\}$$

Apply modus ponens to Wet and Wet  $\rightarrow$  Slippery:

$$KB = \{\text{Rain}, \text{Rain} \rightarrow \text{Wet}, \text{Wet} \rightarrow \text{Slippery}, \text{Wet}, \text{Slippery}\}$$

Converged.

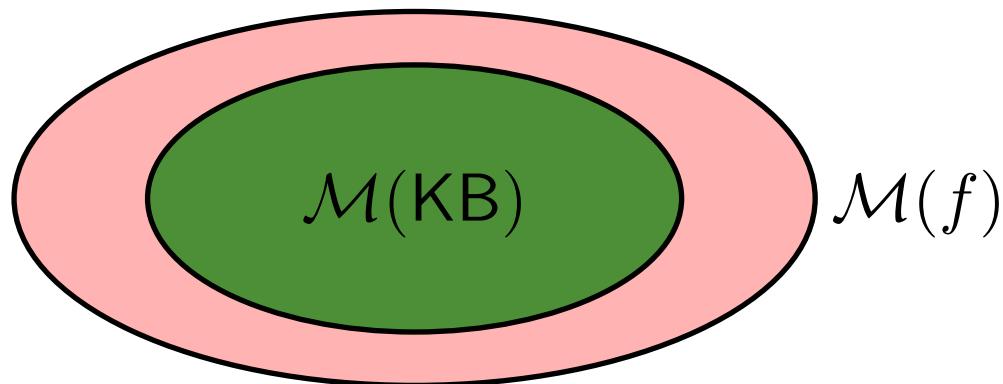
Can't derive some formulas:  $\neg\text{Wet}$ ,  $\text{Rain} \rightarrow \text{Slippery}$

- Here is an example where we've applied modus ponens twice. Note that Wet and Slippery are derived by the KB.
- But there are some formulas which cannot be derived. Some of these underivable formulas will look bad anyway ( $\neg$ Wet), but others will seem reasonable (Rain  $\rightarrow$  Slippery).

# Desiderata for inference rules

## Semantics

Interpretation defines **entailed/true** formulas:  $\text{KB} \models f$ :



## Syntax:

Inference rules **derive** formulas:  $\text{KB} \vdash f$

How does  $\{f : \text{KB} \models f\}$  relate to  $\{f : \text{KB} \vdash f\}$ ?

- We can apply inference rules all day long, but now we desperately need some guidance on whether a set of inference rules is doing anything remotely sensible.
- For this, we turn to semantics, which gives an objective notion of truth. Recall that the semantics provides us with  $\mathcal{M}$ , the set of satisfiable models for each formula  $f$  or knowledge base. This defines a set of formulas  $\{f : \text{KB} \models f\}$  which are defined to be true.
- On the other hand, inference rules also gives us a mechanism for generating a set of formulas, just by repeated application. This defines another set of formulas  $\{f : \text{KB} \vdash f\}$ .

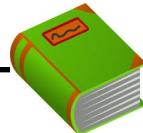
# Truth



$$\{f : \text{KB} \models f\}$$

- Imagine a glass that represents the set of possible formulas entailed by the KB (these are necessarily true).
- By applying inference rules, we are filling up the glass with water.

# Soundness



## Definition: soundness

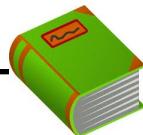
A set of inference rules  $\text{Rules}$  is sound if:

$$\{f : \text{KB} \vdash f\} \subseteq \{f : \text{KB} \models f\}$$



- We say that a set of inference rules is **sound** if using those inference rules, we never overflow the glass: the set of derived formulas is a subset of the set of true/entailed formulas.

# Completeness



## Definition: completeness

A set of inference rules  $\text{Rules}$  is complete if:

$$\{f : \text{KB} \vdash f\} \supseteq \{f : \text{KB} \models f\}$$



- We say that a set of inference rules is **complete** if using those inference rules, we fill up the glass to the brim (and possibly go over): the set of derived formulas is a superset of the set of true/entailed formulas.

# Soundness and completeness

*The truth, the whole truth, and nothing but the truth.*

- **Soundness:** nothing but the truth
- **Completeness:** whole truth

- A slogan to keep in mind is the oath given in a sworn testimony.

# Soundness: example

Is  $\frac{\text{Rain}, \quad \text{Rain} \rightarrow \text{Wet}}{\text{Wet}}$  (Modus ponens) sound?

$$\mathcal{M}(\text{Rain}) \cap \mathcal{M}(\text{Rain} \rightarrow \text{Wet}) \subseteq? \mathcal{M}(\text{Wet})$$

		Wet
		0    1
Rain	0	
	1	

		Wet
		0    1
Rain	0	
	1	

		Wet
		0    1
Rain	0	
	1	

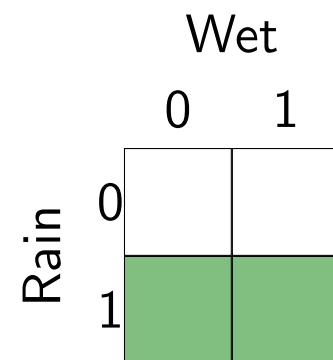
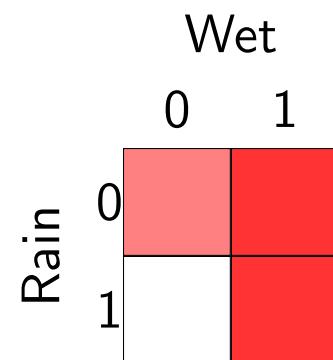
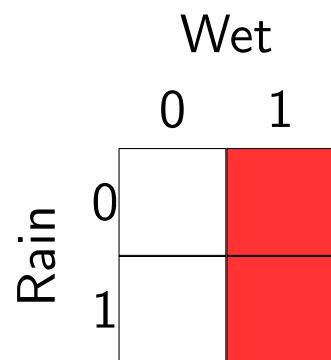
Sound!

- To check the soundness of a set of rules, it suffices to focus on one rule at a time.
- Take the modus ponens rule, for instance. We can derive Wet using modus ponens. To check entailment, we map all the formulas into semantics-land (the set of satisfiable models). Because the models of Wet is a superset of the intersection of models of Rain and Rain  $\rightarrow$  Wet (remember that the models in the KB are an intersection of the models of each formula), we can conclude that Wet is also entailed. If we had other formulas in the KB, that would reduce both sides of  $\subseteq$  by the same amount and won't affect the fact that the relation holds. Therefore, this rule is sound.
- Note, we use Wet and Rain to make the example more colorful, but this argument works for arbitrary propositional symbols.

# Soundness: example

Is  $\frac{\text{Wet}, \quad \text{Rain} \rightarrow \text{Wet}}{\text{Rain}}$  sound?

$$\mathcal{M}(\text{Wet}) \cap \mathcal{M}(\text{Rain} \rightarrow \text{Wet}) \subseteq? \mathcal{M}(\text{Rain})$$



**Unsound!**

- Here is another example: given Wet and Rain  $\rightarrow$  Wet, can we infer Rain? To check it, we mechanically construct the models for the premises and conclusion. Here, the intersection of the models in the premise are not a subset, then the rule is unsound.
- Indeed, backward reasoning is faulty. Note that we can actually do a bit of backward reasoning using Bayesian networks, since we don't have to commit to 0 or 1 for the truth value.

# Completeness: example

Recall completeness: inference rules derive all entailed formulas ( $f$  such that  $\text{KB} \models f$ )



## Example: Modus ponens is incomplete

Setup:

$$\text{KB} = \{\text{Rain}, \text{Rain} \vee \text{Snow} \rightarrow \text{Wet}\}$$

$$f = \text{Wet}$$

$$\text{Rules} = \left\{ \frac{f, \frac{f \rightarrow g}{g}}{g} \right\} \text{ (Modus ponens)}$$

Semantically:  $\text{KB} \models f$  ( $f$  is entailed).

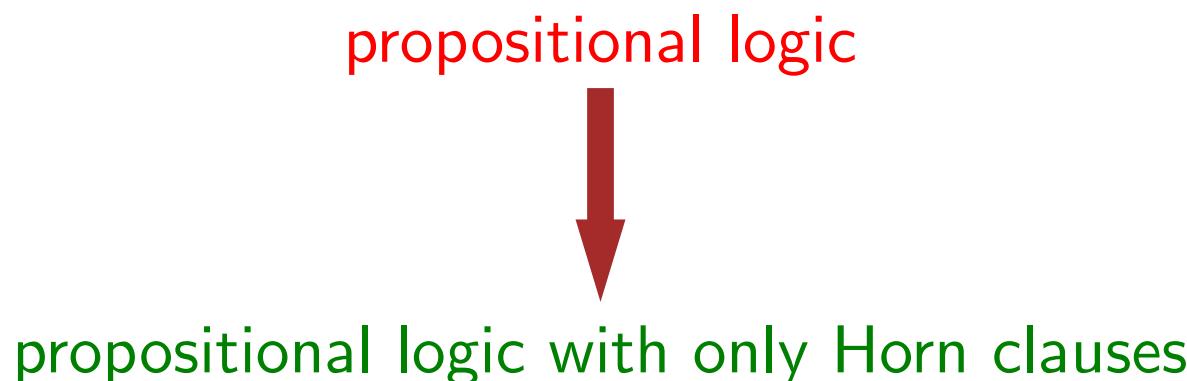
Syntactically:  $\text{KB} \not\vdash f$  (can't derive  $f$ ).

**Incomplete!**

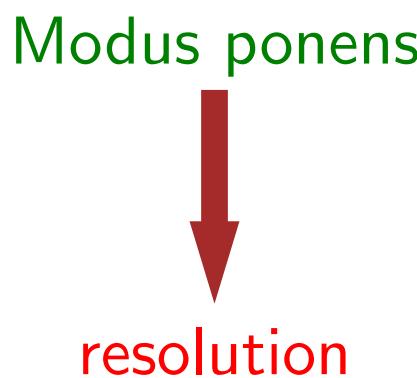
- Completeness is trickier, and here is a simple example that shows that modus ponens alone is not complete, since it can't derive Wet, when semantically, Wet is true!

# Fixing completeness

Option 1: Restrict the allowed set of formulas



Option 2: Use more powerful inference rules



- At this point, there are two ways to fix completeness. First, we can restrict the set of allowed formulas, making the water glass smaller in hopes that modus ponens will be able to fill that smaller glass.
- Second, we can use more powerful inference rules, pouring more vigorously into the same glass in hopes that this will be able to fill the glass; we'll look at one such rule, resolution, in the next lecture.

# Definite clauses



## Definition: Definite clause

A **definite clause** has the following form:

$$(p_1 \wedge \cdots \wedge p_k) \rightarrow q$$

where  $p_1, \dots, p_k, q$  are propositional symbols.

**Intuition:** if  $p_1, \dots, p_k$  hold, then  $q$  holds.

**Example:**  $(\text{Rain} \wedge \text{Snow}) \rightarrow \text{Traffic}$

**Example:**  $\text{Traffic}$

**Non-example:**  $\neg \text{Traffic}$

**Non-example:**  $(\text{Rain} \wedge \text{Snow}) \rightarrow (\text{Traffic} \vee \text{Peaceful})$

- First we will choose to restrict the allowed set of formulas. Towards that end, let's define a **definite clause** as a formula that says, if a conjunction of propositional symbols holds, then some other propositional symbol  $q$  holds. Note that this is a formula, not to be confused with an inference rule.

# Horn clauses



## Definition: Horn clause

A **Horn clause** is either:

- a definite clause  $(p_1 \wedge \dots \wedge p_k \rightarrow q)$
- a goal clause  $(p_1 \wedge \dots \wedge p_k \rightarrow \text{false})$

Example (definite):  $(\text{Rain} \wedge \text{Snow}) \rightarrow \text{Traffic}$

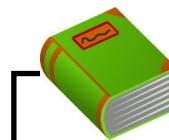
Example (goal):  $\text{Traffic} \wedge \text{Accident} \rightarrow \text{false}$

equivalent:  $\neg(\text{Traffic} \wedge \text{Accident})$

- A **Horn clause** is basically a definite clause, but includes another type of clause called a **goal clause**, which is the conjunction of a bunch of propositional symbols implying false. The form of the goal clause might seem a bit strange, but the way to interpret it is simply that it's the negation of the conjunction.

# Modus ponens

Inference rule:



## Definition: Modus ponens

$$\frac{p_1, \dots, p_k, (p_1 \wedge \dots \wedge p_k) \rightarrow q}{q}$$

Example:



## Example: Modus ponens

$$\frac{\text{Wet}, \text{ Weekday}, \text{ Wet} \wedge \text{Weekday} \rightarrow \text{Traffic}}{\text{Traffic}}$$

- Recall the Modus ponens rule from before. We simply have generalized it to arbitrary number of premises.

# Completeness of modus ponens



## Theorem: Modus ponens on Horn clauses

Modus ponens is **complete** with respect to Horn clauses:

- Suppose KB contains only Horn clauses and  $p$  is an entailed propositional symbol.
- Then applying modus ponens will derive  $p$ .

Upshot:

$\text{KB} \models p$  (entailment) is the same as  $\text{KB} \vdash p$  (derivation)!

- There's a theorem that says that modus ponens is complete on Horn clauses. This means that any propositional symbol that is entailed can be derived by modus ponens too, provided that all the formulas in the KB are Horn clauses.
- We already proved that modus ponens is sound, and now we have that it is complete (for Horn clauses). The upshot of this is that entailment (a semantic notion, what we care about) and being able to derive a formula (a syntactic notion, what we do with inference) are equivalent!

# Example: Modus ponens

KB

Rain

Weekday

$\text{Rain} \rightarrow \text{Wet}$

$\text{Wet} \wedge \text{Weekday} \rightarrow \text{Traffic}$

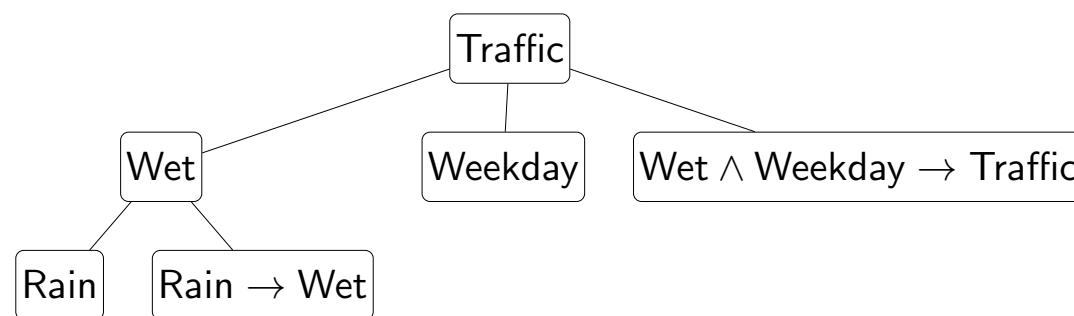
$\text{Traffic} \wedge \text{Careless} \rightarrow \text{Accident}$



## Definition: Modus ponens

$$\frac{p_1, \dots, p_k, (p_1 \wedge \dots \wedge p_k) \rightarrow q}{q}$$

Question:  $\text{KB} \models \text{Traffic} \Leftrightarrow \text{KB} \vdash \text{Traffic}$

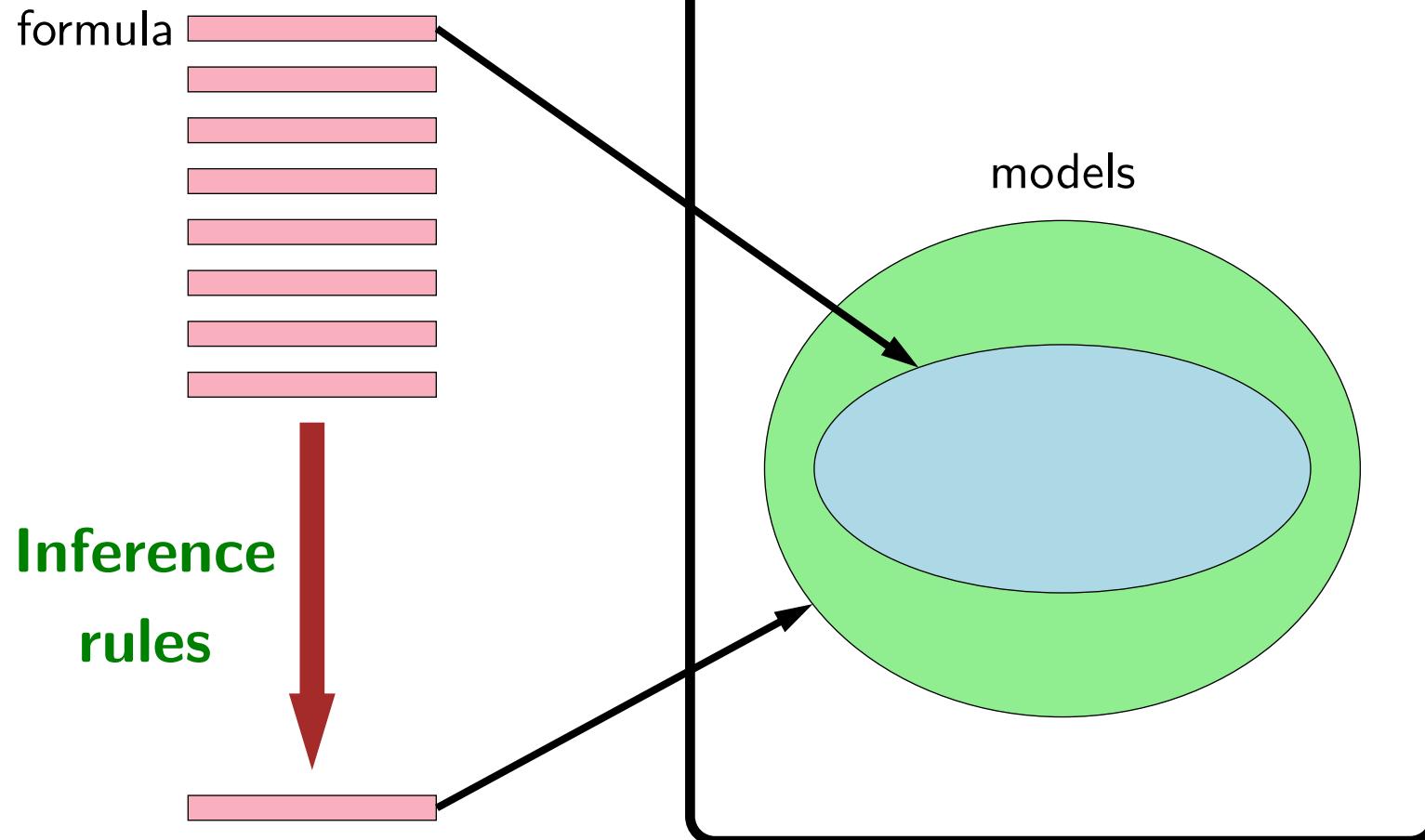


- Let's see modus ponens on Horn clauses in action. Suppose we have the given KB consisting of only Horn clauses (in fact, these are all definite clauses), and we wish to ask whether the KB entails Traffic.
- We can construct a **derivation**, a tree where the root formula (e.g., Traffic) was derived using inference rules.
- The leaves are the original formulas in the KB, and each internal node corresponds to a formula which is produced by applying an inference rule (e.g., modus ponens) with the children as premises.
- If a symbol is used as the premise in two different rules, then it would have two parents, resulting in a DAG.



# Summary

## Syntax      Semantics



# Review Session (Variable- Based Models)

---

**CS 221 AUTUMN 2019**

CHRISTOPHER WAITES

JON KOTKER

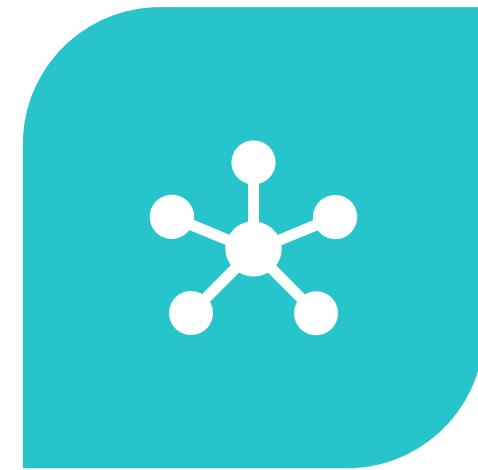


# What are we covering today?

---



CONSTRAINT  
SATISFACTION PROBLEMS



BAYESIAN NETWORKS

# CSPs: Definitions



## Key idea: variables

- Solutions to problems  $\Rightarrow$  assignments to variables (**modeling**).
- Decisions about variable ordering, etc. chosen by **inference**.

We have **factors** (constraints) and **inputs**.

Setting these inputs is an **assignment**.

*Factor Graphs:* How good is our assignment? (maximum weight assignment)

*Constraint Satisfaction:* Is this a valid assignment? (yes/no; is weight 1?)

# CSPs: Inference

**Backtracking:** Try everything and prune.



## Algorithm: backtracking search

Backtrack( $x, w, \text{Domains}$ ):

- If  $x$  is complete assignment: update best and return
- Choose unassigned **VARIABLE**  $X_i$
- Order **VALUES**  $\text{Domain}_i$  of chosen  $X_i$
- For each value  $v$  in that order:
  - $\delta \leftarrow \prod_{f_j \in D(x, X_i)} f_j(x \cup \{X_i : v\})$
  - If  $\delta = 0$ : continue
  - $\text{Domains}' \leftarrow \text{Domains}$  via **LOOKAHEAD**
  - Backtrack( $x \cup \{X_i : v\}, w\delta, \text{Domains}'$ )

# CSPs: Inference

**AC-3:** Pre-emptively prune  
with lookahead.



## Algorithm: AC-3

Add  $X_j$  to set.

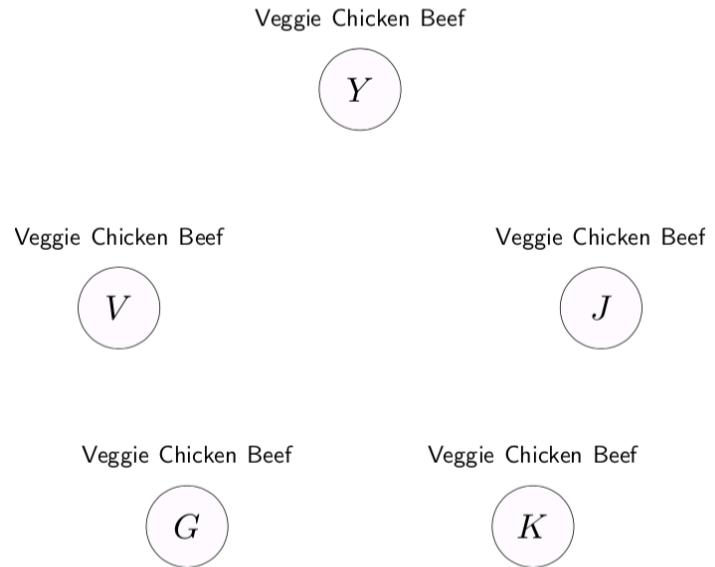
While set is non-empty:

- Remove any  $X_k$  from set.
- For all neighbors  $X_l$  of  $X_k$ :
  - Enforce arc consistency on  $X_l$  w.r.t.  $X_k$ .
  - If  $\text{Domain}_l$  changed, add  $X_l$  to set.

It's Friday night, and you and your friends go out to dinner in anticipation of the Big Game the next day (go Card!). Since you've just been working on your final project for CS221, you are seeing CSPs and Bayesian networks everywhere!

a. (10 points)

You and your friends (Veronica, Jarvis, Gabriela, Kanti) sit around a table like this:



# CSPs: Sample Question (Fall 2014)

---

There are three dishes on the menu: the vegetarian deep dish pizza, the chicken quesadilla, and the beef cheeseburger. Each person will order exactly one dish.

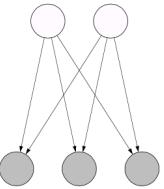
But what started out as a simple dinner has quickly turned into a logistical nightmare because of all the constraints you and your friends impose upon yourselves:

1. Each person must order something different than the people sitting immediately next to him/her.
2. You ( $Y$ ) are vegetarian.
3. If Veronica ( $V$ ) orders beef, then Jarvis ( $J$ ) will order veggie.
4. Kanti ( $K$ ) and Jarvis ( $J$ ) cannot both get non-chicken dishes.

Draw the potentials for the above constraints and write the propositional formula above each potential (e.g.,  $[Y = \text{Veggie}]$ ). Then for each pair of variables, enforce arc consistency in both directions, crossing out the appropriate values from the domains.

# CSPs: Sample Question (Fall 2014)

---



### Definition: Bayesian network

Let  $X = (X_1, \dots, X_n)$  be random variables.

A **Bayesian network** is a directed acyclic graph (DAG) that specifies a joint distribution over  $X$  as a product of local conditional distributions, one for each node:

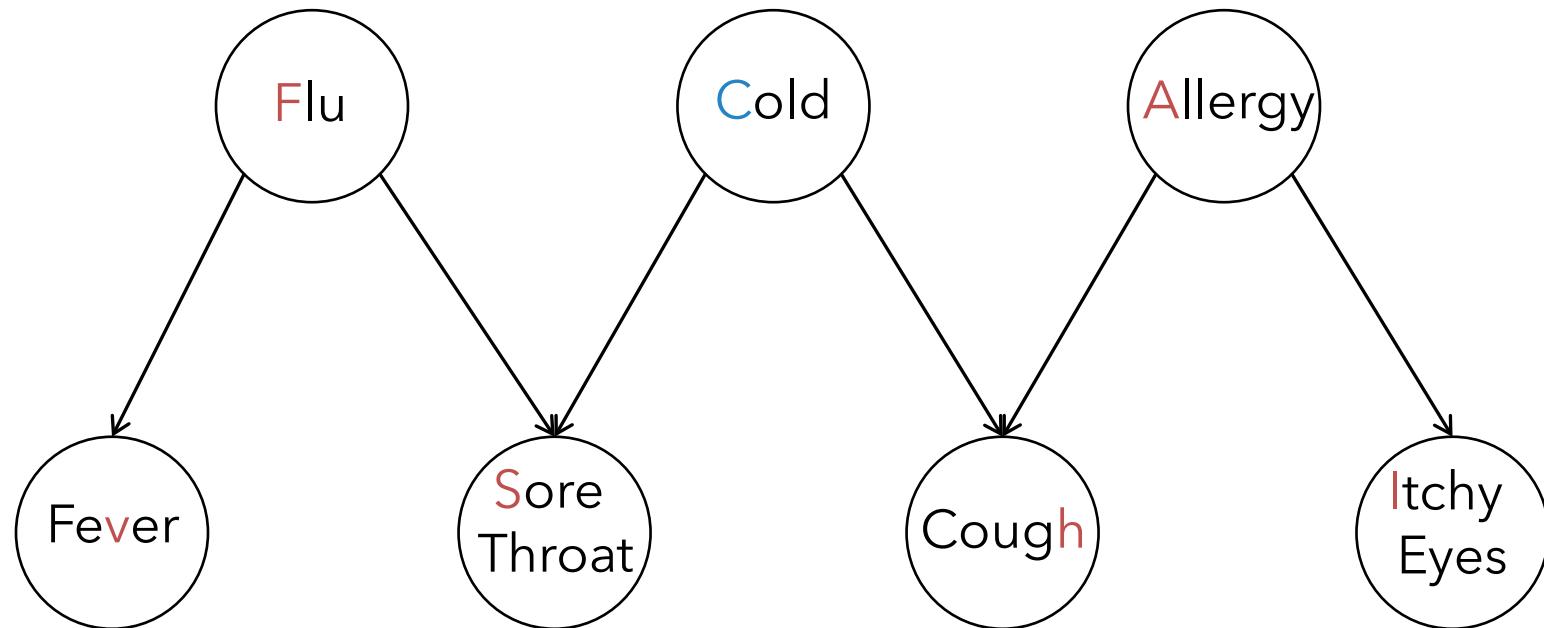
$$\mathbb{P}(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1}^n p(x_i \mid x_{\text{Parents}(i)})$$

# Bayesian Networks

---

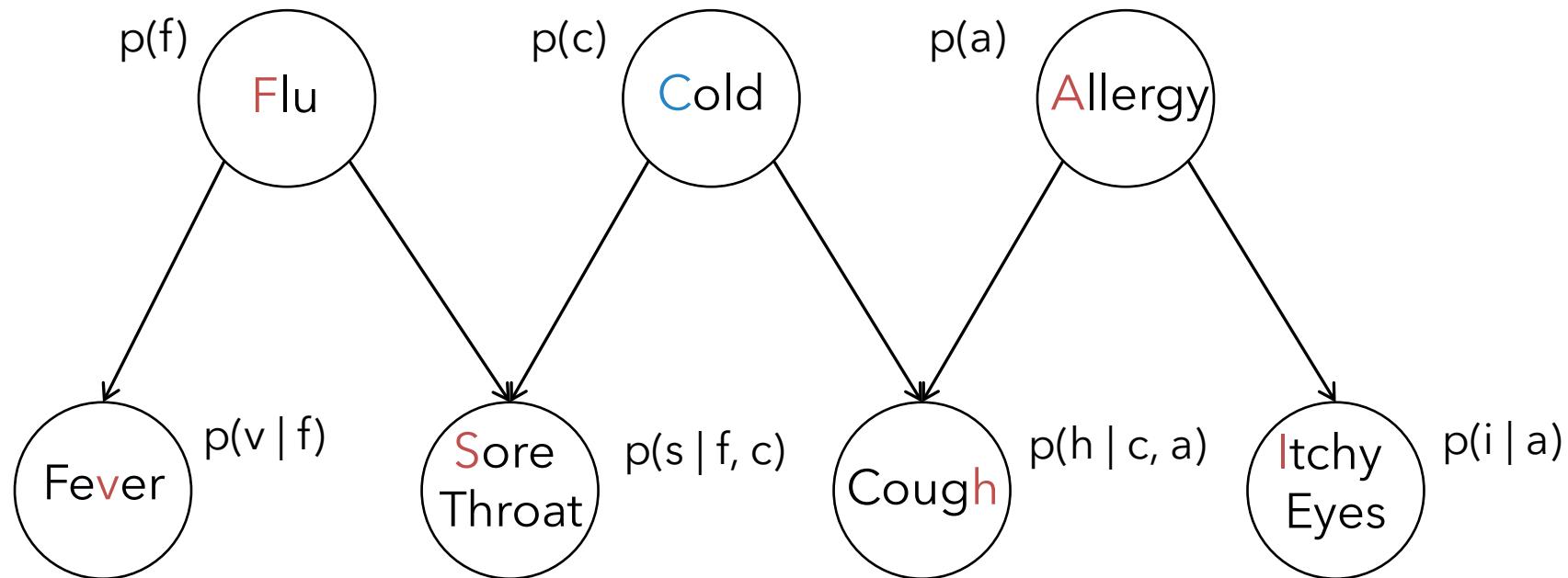
# Bayesian Networks

---



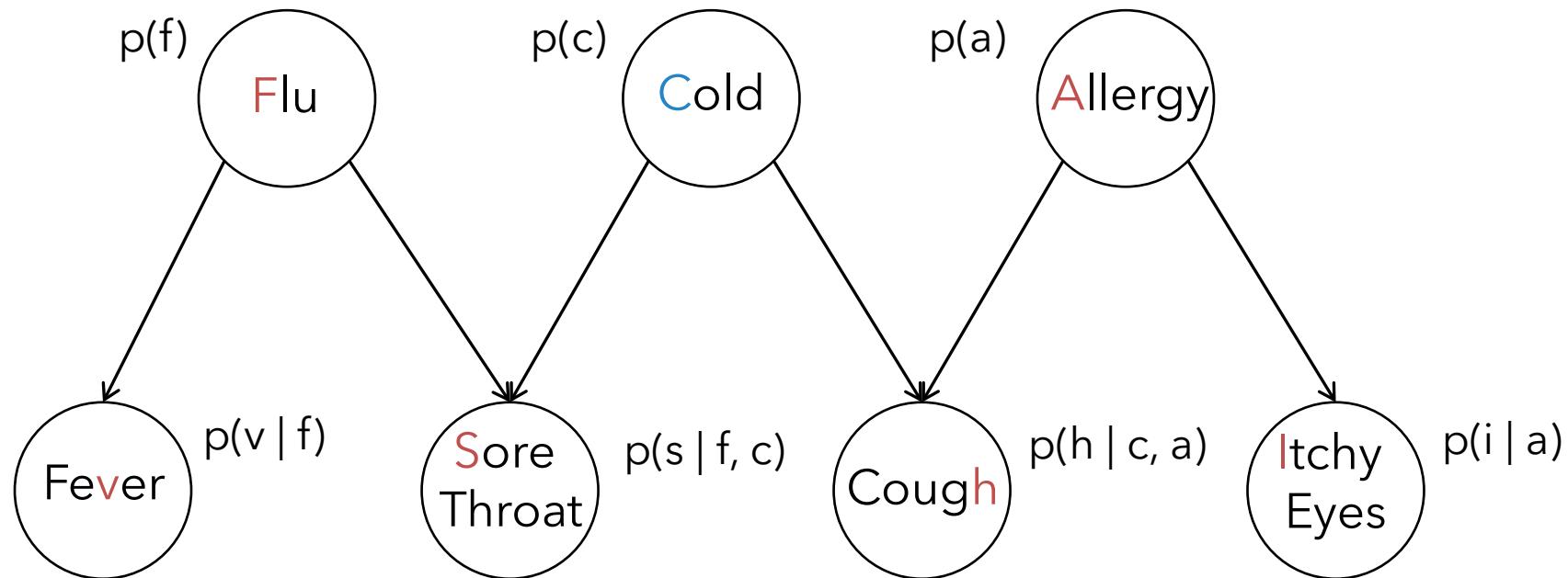
# A Bayesian network represents a joint probability distribution.

---



# A Bayesian network represents a joint probability distribution.

---



$$P(F=f, C=c, A=a, V=v, S=s, H=h, I=i) = p(f) p(c) p(a) p(v | f) p(s | f, c) p(h | c, a) p(i | a)$$

# Probabilistic Inference Cookbook

---

Given a query  $P(Q | E = e)$ :

1. Remove (marginalize) variables not ancestors of  $Q$  or  $E$ .
2. Convert Bayesian network to factor graph.
3. Condition (shade nodes / disconnect) on  $E = e$ .
4. Remove (marginalize) nodes disconnected from  $Q$ .
5. Run probabilistic inference algorithm (manual, variable elimination, Gibbs sampling, particle filtering).

### 3. The Bayesian Bag of Candies Model (50 points)

You have a lot of candy left over from Halloween, and you decide to give them away to your friends. You have four types of candy: **Apple**, **Banana**, **Caramel**, **Dark-Chocolate**. You decide to prepare candy bags using the following process.

- For each candy bag, you first flip a (biased) coin  $Y$  which comes up heads ( $Y = H$ ) with probability  $\lambda$  and tails ( $Y = T$ ) with probability  $1 - \lambda$ .
- If  $Y$  comes up heads ( $Y = H$ ), you make a **Healthy** bag, where you:
  1. Add one **Apple** candy with probability  $p_1$  or nothing with probability  $1 - p_1$ ;
  2. Add one **Banana** candy with probability  $p_1$  or nothing with probability  $1 - p_1$ ;
  3. Add one **Caramel** candy with probability  $1 - p_1$  or nothing with probability  $p_1$ ;
  4. Add one **Dark-Chocolate** candy with probability  $1 - p_1$  or nothing with probability  $p_1$ .
- If  $Y$  comes up tails ( $Y = T$ ), you make a **Tasty** bag, where you:
  1. Add one **Apple** candy with probability  $p_2$  or nothing with probability  $1 - p_2$ ;
  2. Add one **Banana** candy with probability  $p_2$  or nothing with probability  $1 - p_2$ ;
  3. Add one **Caramel** candy with probability  $1 - p_2$  or nothing with probability  $p_2$ ;
  4. Add one **Dark-Chocolate** candy with probability  $1 - p_2$  or nothing with probability  $p_2$ .

For example, if  $p_1 = 1$  and  $p_2 = 0$ , you would deterministically generate: **Healthy** bags with one **Apple** and one **Banana**; and **Tasty** bags with one **Caramel** and one **Dark-Chocolate**. For general values of  $p_1$  and  $p_2$ , bags can contain anywhere between 0 and 4 pieces of candy.

Denote  $A, B, C, D$  random variables indicating whether or not the bag contains candy of type **Apple**, **Banana**, **Caramel**, and **Dark-Chocolate**, respectively.

# Bayesian Networks: Sample Question (Fall 2017)

---

**Input:** training examples  $\mathcal{D}_{\text{train}}$  of full assignments

**Output:** parameters  $\theta = \{p_d : d \in D\}$



#### Algorithm: maximum likelihood for Bayesian networks

##### Count:

For each  $x \in \mathcal{D}_{\text{train}}$ :

    For each variable  $x_i$ :

        Increment  $\text{count}_{d_i}(x_{\text{Parents}(i)}, x_i)$

##### Normalize:

For each  $d$  and local assignment  $x_{\text{Parents}(i)}$ :

    Set  $p_d(x_i | x_{\text{Parents}(i)}) \propto \text{count}_d(x_{\text{Parents}(i)}, x_i)$

# Bayesian Networks: MLE

---

# Bayesian Networks: Sample Question (Fall 2017)

---

b. (10 points)

You realize you need to make more candy bags, but you've forgotten the probabilities you used to generate them. So you try to estimate them looking at the 5 bags you've already made:

bag 1 :	(Healthy, {Apple, Banana})
bag 2 :	(Tasty, {Caramel, Dark-Chocolate})
bag 3 :	(Healthy, {Apple, Banana})
bag 4 :	(Tasty, {Caramel, Dark-Chocolate})
bag 5 :	(Healthy, {Apple, Banana})

Estimate  $\lambda, p_1, p_2$  by maximum likelihood.

# Bayesian Networks: Smoothing

---

*Key idea:* Hallucinate  $\lambda$  extra observations.

*Assumption:* All values we listed are possible.

Maximum likelihood:

$$p(H) = \frac{1}{1} \quad p(T) = \frac{0}{1}$$

Maximum likelihood with Laplace smoothing:

$$p(H) = \frac{1+1}{1+2} = \frac{2}{3} \quad p(T) = \frac{0+1}{1+2} = \frac{1}{3}$$

# Bayesian Networks: EM algorithm

---

## E-step:

- Compute  $q(h) = \mathbb{P}(H = h \mid E = e; \theta)$  for each  $h$  (use any probabilistic inference algorithm)
- Create weighted points:  $(h, e)$  with weight  $q(h)$

## M-step:

- Compute maximum likelihood (just count and normalize) to get  $\theta$

Repeat until convergence.

# Bayesian Networks: Sample Question (Fall 2017)

---

c. (15 points) You find out your little brother had been playing with your candy bags, and had mixed them up (in a uniformly random way). Now you don't even know which ones were **Healthy** and which ones were **Tasty**. So you need to re-estimate  $\lambda, p_1, p_2$ , but now without knowing whether the bags were **Healthy** or **Tasty**.

<i>bag 1 :</i>	(? , {Apple, Banana, Caramel})
<i>bag 2 :</i>	(? , {Caramel, Dark-Chocolate})
<i>bag 3 :</i>	(? , {Apple, Banana, Caramel})
<i>bag 4 :</i>	(? , {Caramel, Dark-Chocolate})
<i>bag 5 :</i>	(? , {Apple, Banana, Caramel})

You remember the EM algorithm is just what you need. Initialize with  $\lambda = 0.5, p_1 = 0.5, p_2 = 0$ , and run one step of the EM algorithm.



# Lecture 17: Logic II

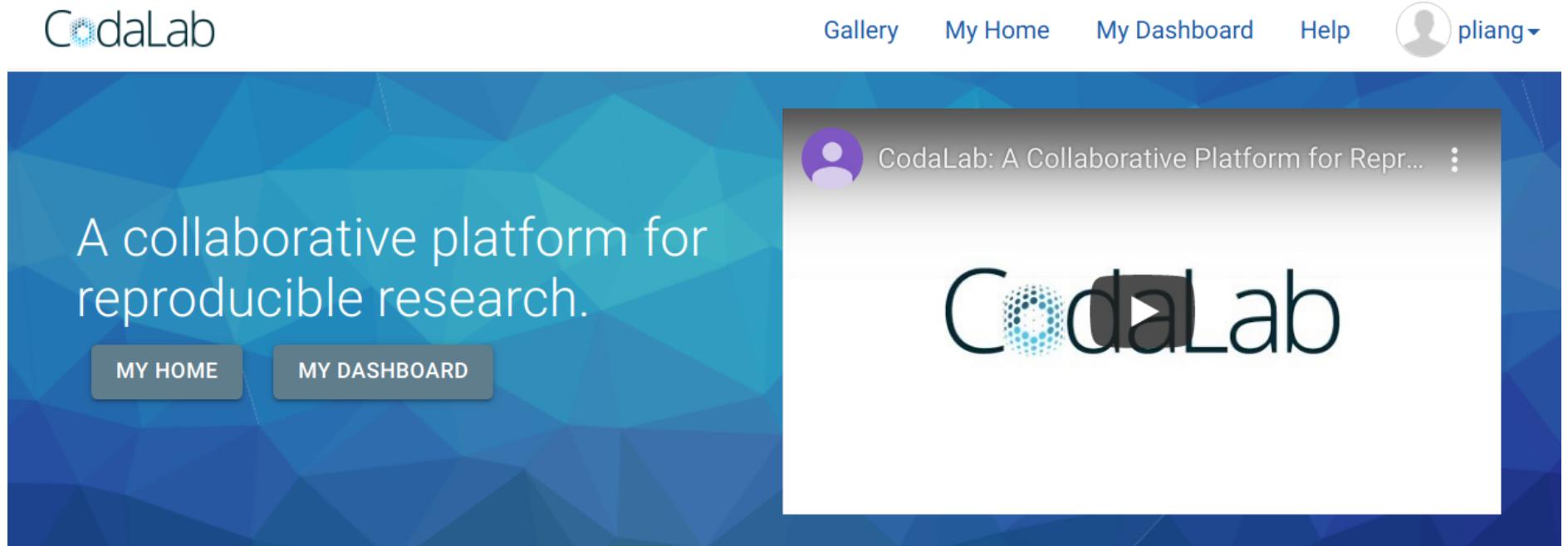


# Announcements

- **exam** is tomorrow!
- Next week is Thanksgiving break
- No more sections
- **Poster session** is following Monday (Dec. 2)
- **logic** due Tuesday (Dec. 3)

# CodaLab Worksheets

[worksheets.codalab.org](https://worksheets.codalab.org)



- Extra credit opportunity for final project
- Additional compute available

[demo]

# Review: ingredients of a logic

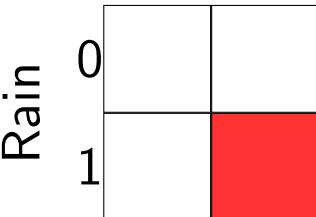
**Syntax:** defines a set of valid **formulas** (Formulas)

Example: Rain  $\wedge$  Wet

**Semantics:** for each formula  $f$ , specify a set of **models**  $\mathcal{M}(f)$  (assignments / configurations of the world)

Example:

		Wet
		0    1
Rain	0	
	1	



A 2x2 grid representing a truth table for the formula Rain  $\wedge$  Wet. The columns are labeled 0 and 1, and the rows are labeled 0 and 1. The bottom-right cell (row 1, column 1) is shaded red, while all other cells are white.

**Inference rules:** given KB, what new formulas  $f$  can be derived?

Example: from Rain  $\wedge$  Wet, derive Rain

- Logic provides a formal language to talk about the world.
- The valid sentences in the language are the logical formulas, which live in syntax-land.
- In semantics-land, a model represents a possible configuration of the world. An interpretation function connects syntax and semantics. Specifically, it defines, for each formula  $f$ , a set of models  $\mathcal{M}(f)$ .

# Review: inference algorithm

Inference algorithm:



**Definition: modus ponens inference rule**

$$\frac{p_1, \dots, p_k, (p_1 \wedge \dots \wedge p_k) \rightarrow q}{q}$$

Desiderata: soundness and completeness



entailment ( $\text{KB} \models f$ )

derivation ( $\text{KB} \vdash f$ )

- A knowledge base is a set of formulas we know to be true. Semantically the KB represents the conjunction of the formulas.
- The central goal of logic is inference: to figure out whether a query formula is entailed by, contradictory with, or contingent on the KB (these are semantic notions defined by the interpretation function).
- The unique thing about having a logical language is that we can also perform inference directly on syntax by applying **inference rules**, rather than always appealing to semantics (and performing model checking there).
- We would like the inference algorithm to be both sound (not derive any false formulas) and complete (derive all true formulas). Soundness is easy to check, completeness is harder.

# Review: formulas

Propositional logic: any legal combination of symbols

$$(\text{Rain} \wedge \text{Snow}) \rightarrow (\text{Traffic} \vee \text{Peaceful}) \wedge \text{Wet}$$

Propositional logic with only Horn clauses: restricted

$$(\text{Rain} \wedge \text{Snow}) \rightarrow \text{Traffic}$$

- Whether a set of inference rules is complete depends on what the formulas are. Last time, we looked at two logical languages: propositional logic and propositional logic restricted to Horn clauses (essentially formulas that look like  $p_1 \wedge \cdots \wedge p_k \rightarrow q$ ), which intuitively can only derive positive information.

# Review: tradeoffs

**Formulas allowed**

**Inference rule Complete?**

Propositional logic

modus ponens    no

Propositional logic (only Horn clauses) modus ponens yes

Propositional logic

resolution    yes

- We saw that if our logical language was restricted to Horn clauses, then modus ponens alone was sufficient for completeness. For general propositional logic, modus ponens is insufficient.
- In this lecture, we'll see that a more powerful inference rule, **resolution**, is complete for all of propositional logic.



# Roadmap

**Resolution in propositional logic**

First-order logic

# Horn clauses and disjunction

## Written with implication

$$A \rightarrow C$$

$$A \wedge B \rightarrow C$$

- **Literal:** either  $p$  or  $\neg p$ , where  $p$  is a propositional symbol
- **Clause:** disjunction of literals
- **Horn clauses:** at most one positive literal

Modus ponens (rewritten):

$$\frac{A, \quad \neg A \vee C}{C}$$

- Intuition: cancel out  $A$  and  $\neg A$

## Written with disjunction

$$\neg A \vee C$$

$$\neg A \vee \neg B \vee C$$

- Modus ponens can only deal with Horn clauses, so let's see why Horn clauses are limiting. We can equivalently write implication using negation and disjunction. Then it's clear that Horn clauses are just disjunctions of literals where there is at most one positive literal and zero or more negative literals. The negative literals correspond to the propositional symbols on the left side of the implication, and the positive literal corresponds to the propositional symbol on the right side of the implication.
- If we rewrite modus ponens, we can see a "canceling out" intuition emerging. To make the intuition a bit more explicit, remember that, to respect soundness, we require  $\{A, \neg A \vee C\} \models C$ ; this is equivalent to: if  $A \wedge (\neg A \vee C)$  is true, then  $C$  is also true. This is clearly the case.
- But modus ponens cannot operate on general clauses.

# Resolution [Robinson, 1965]

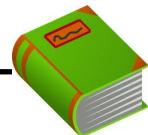
General clauses have any number of literals:

$$\neg A \vee B \vee \neg C \vee D \vee \neg E \vee F$$



**Example: resolution inference rule**

$$\frac{\text{Rain} \vee \text{Snow}, \quad \neg \text{Snow} \vee \text{Traffic}}{\text{Rain} \vee \text{Traffic}}$$



**Definition: resolution inference rule**

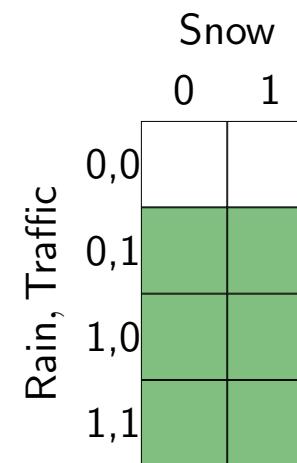
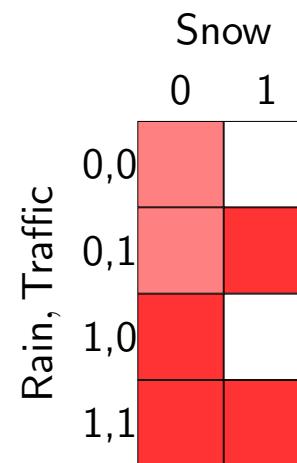
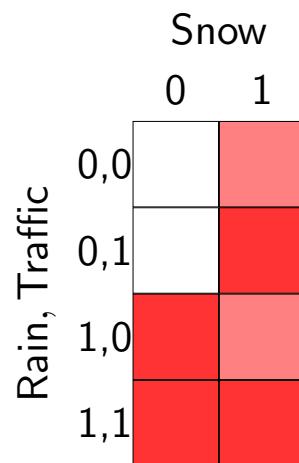
$$\frac{f_1 \vee \cdots \vee f_n \vee p, \quad \neg p \vee g_1 \vee \cdots \vee g_m}{f_1 \vee \cdots \vee f_n \vee g_1 \vee \cdots \vee g_m}$$

- Let's try to generalize modus ponens by allowing it to work on general clauses. This generalized inference rule is called **resolution**, which was invented in 1965 by John Alan Robinson.
- The idea behind resolution is that it takes two general clauses, where one of them has some propositional symbol  $p$  and the other clause has its negation  $\neg p$ , and simply takes the disjunction of the two clauses with  $p$  and  $\neg p$  removed. Here,  $f_1, \dots, f_n, g_1, \dots, g_m$  are arbitrary literals.

# Soundness of resolution

$$\frac{\text{Rain} \vee \text{Snow}, \quad \neg \text{Snow} \vee \text{Traffic}}{\text{Rain} \vee \text{Traffic}} \text{ (resolution rule)}$$

$$\mathcal{M}(\text{Rain} \vee \text{Snow}) \cap \mathcal{M}(\neg \text{Snow} \vee \text{Traffic}) \subseteq ?\mathcal{M}(\text{Rain} \vee \text{Traffic})$$



**Sound!**

- Why is resolution logically sound? We can verify the soundness of resolution by checking its semantic interpretation. Indeed, the intersection of the models of  $f$  and  $g$  is a subset of models of  $f \vee g$ .

# Conjunctive normal form

So far: resolution only works on clauses...but that's enough!



**Definition: conjunctive normal form (CNF)**

A **CNF formula** is a conjunction of clauses.

**Example:**  $(A \vee B \vee \neg C) \wedge (\neg B \vee D)$

**Equivalent:** knowledge base where each formula is a clause



**Proposition: conversion to CNF**

Every formula  $f$  in propositional logic can be converted into an equivalent CNF formula  $f'$ :

$$\mathcal{M}(f) = \mathcal{M}(f')$$

- But so far, we've only considered clauses, which are disjunctions of literals. Surely this can't be all of propositional logic... But it turns out it actually is in the following sense.
- A conjunction of clauses is called a CNF formula, and every formula in propositional logic can be converted into an equivalent CNF. Given a CNF formula, we can toss each of its clauses into the knowledge base.
- But why can every formula be put in CNF?

# Conversion to CNF: example

Initial formula:

$$(\text{Summer} \rightarrow \text{Snow}) \rightarrow \text{Bizzare}$$

Remove implication ( $\rightarrow$ ):

$$\neg(\neg\text{Summer} \vee \text{Snow}) \vee \text{Bizzare}$$

Push negation ( $\neg$ ) inwards (de Morgan):

$$(\neg\neg\text{Summer} \wedge \neg\text{Snow}) \vee \text{Bizzare}$$

Remove double negation:

$$(\text{Summer} \wedge \neg\text{Snow}) \vee \text{Bizzare}$$

Distribute  $\vee$  over  $\wedge$ :

$$(\text{Summer} \vee \text{Bizzare}) \wedge (\neg\text{Snow} \vee \text{Bizzare})$$

- The answer is by construction. There is a six-step procedure that takes any propositional formula and turns it into CNF. Here is an example of how it works (only four of the six steps apply here).

# Conversion to CNF: general

Conversion rules:

- Eliminate  $\leftrightarrow$ : 
$$\frac{f \leftrightarrow g}{(f \rightarrow g) \wedge (g \rightarrow f)}$$
- Eliminate  $\rightarrow$ : 
$$\frac{f \rightarrow g}{\neg f \vee g}$$
- Move  $\neg$  inwards: 
$$\frac{\neg(f \wedge g)}{\neg f \vee \neg g}$$
- Move  $\neg$  inwards: 
$$\frac{\neg(f \vee g)}{\neg f \wedge \neg g}$$
- Eliminate double negation: 
$$\frac{\neg\neg f}{f}$$
- Distribute  $\vee$  over  $\wedge$ : 
$$\frac{f \vee (g \wedge h)}{(f \vee g) \wedge (f \vee h)}$$

- Here are the general rules that convert any formula to CNF. First, we try to reduce everything to negation, conjunction, and disjunction.
- Next, we try to push negation inwards so that they sit on the propositional symbols (forming literals). Note that when negation gets pushed inside, it flips conjunction to disjunction, and vice-versa.
- Finally, we distribute so that the conjunctions are on the outside, and the disjunctions are on the inside.
- Note that each of these operations preserves the semantics of the logical form (remember there are many formulas that map to the same set of models). This is in contrast with most inference rules, where the conclusion is more general than the conjunction of the premises.
- Also, when we apply a CNF rewrite rule, we replace the old formula with the new one, so there is no blow-up in the number of formulas. This is in contrast to applying general inference rules. An analogy: conversion to CNF does simplification in the context of full inference, just like AC-3 does simplification in the context of backtracking search.

# Resolution algorithm

Recall: relationship between entailment and contradiction (basically "proof by contradiction")

$$\text{KB} \models f \quad \longleftrightarrow \quad \text{KB} \cup \{\neg f\} \text{ is unsatisfiable}$$



## Algorithm: resolution-based inference

- Add  $\neg f$  into KB.
- Convert all formulas into **CNF**.
- Repeatedly apply **resolution** rule.
- Return entailment iff derive false.

- After we have converted all the formulas to CNF, we can repeatedly apply the resolution rule. But what is the final target?
- Recall that both testing for entailment and contradiction boil down to checking satisfiability. Resolution can be used to do this very thing. If we ever apply a resolution rule (e.g., to premises  $A$  and  $\neg A$ ) and we derive false (which represents a contradiction), then the set of formulas in the knowledge base is unsatisfiable.
- If we are unable to derive false, that means the knowledge base is satisfiable because resolution is complete. However, unlike in model checking, we don't actually produce a concrete model that satisfies the KB.

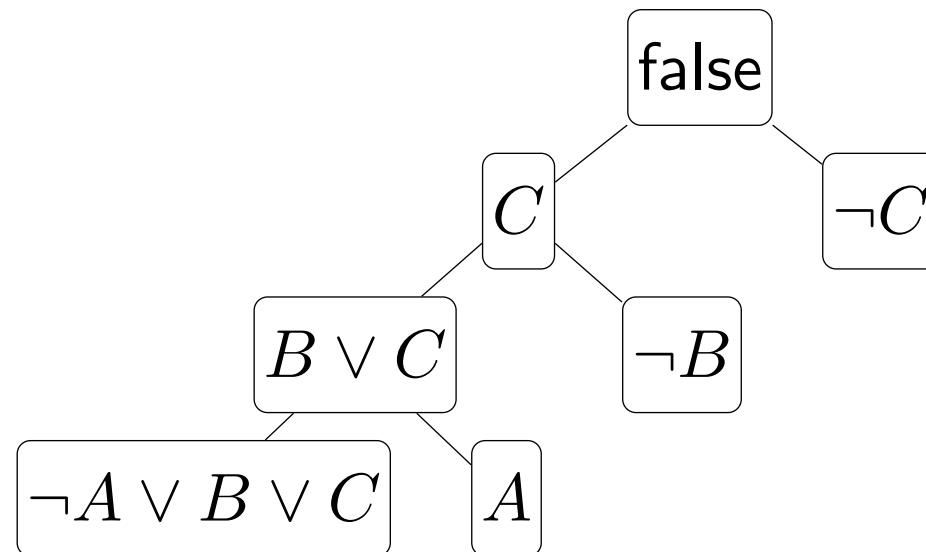
# Resolution: example

$$\text{KB}' = \{A \rightarrow (B \vee C), A, \neg B, \neg C\}$$

Convert to CNF:

$$\text{KB}' = \{\neg A \vee B \vee C, A, \neg B, \neg C\}$$

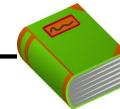
Repeatedly apply **resolution** rule:



Conclusion: **KB entails f**

- Here's an example of taking a knowledge base, converting it into CNF, and applying resolution. In this case, we derive false, which means that the original knowledge base was unsatisfiable.

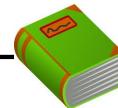
# Time complexity



**Definition: modus ponens inference rule**

$$\frac{p_1, \dots, p_k, (p_1 \wedge \dots \wedge p_k) \rightarrow q}{q}$$

- Each rule application adds clause with **one** propositional symbol  
 $\Rightarrow$  linear time



**Definition: resolution inference rule**

$$\frac{f_1 \vee \dots \vee f_n \vee p, \neg p \vee g_1 \vee \dots \vee g_m}{f_1 \vee \dots \vee f_n \vee g_1 \vee \dots \vee g_m}$$

- Each rule application adds clause with **many** propositional symbols  
 $\Rightarrow$  exponential time

- There we have it — a sound and complete inference procedure for all of propositional logic (although we didn't prove completeness). But what do we have to pay computationally for this increase?
- If we only have to apply modus ponens, each propositional symbol can only get added once, so with the appropriate algorithm (forward chaining), we can apply all necessary modus ponens rules in linear time.
- But with resolution, we can end up adding clauses with many propositional symbols, and possibly any subset of them! Therefore, this can take exponential time.



# Summary

**Horn clauses**

**any clauses**

modus ponens

resolution

linear time

exponential time

less expressive

more expressive

- To summarize, we can either content ourselves with the limited expressivity of Horn clauses and obtain an efficient inference procedure (via modus ponens).
- If we wanted the expressivity of full propositional logic, then we need to use resolution and thus pay more.



# Roadmap

Resolution in propositional logic

First-order logic

# Limitations of propositional logic

*Alice and Bob both know arithmetic.*

$\text{AliceKnowsArithmetic} \wedge \text{BobKnowsArithmetic}$

*All students know arithmetic.*

$\text{AliceIsStudent} \rightarrow \text{AliceKnowsArithmetic}$

$\text{BobIsStudent} \rightarrow \text{BobKnowsArithmetic}$

...

*Every even integer greater than 2 is the sum of two primes.*

???

- If the goal of logic is to be able to express facts in the world in a compact way, let us ask ourselves if propositional logic is enough.
- Some facts can be expressed in propositional logic, but it is very clunky, having to instantiate many different formulas. Others simply can't be expressed at all, because we would need to use an infinite number of formulas.

# Limitations of propositional logic

*All students know arithmetic.*

AliceIsStudent  $\rightarrow$  AliceKnowsArithmetic

BobIsStudent  $\rightarrow$  BobKnowsArithmetic

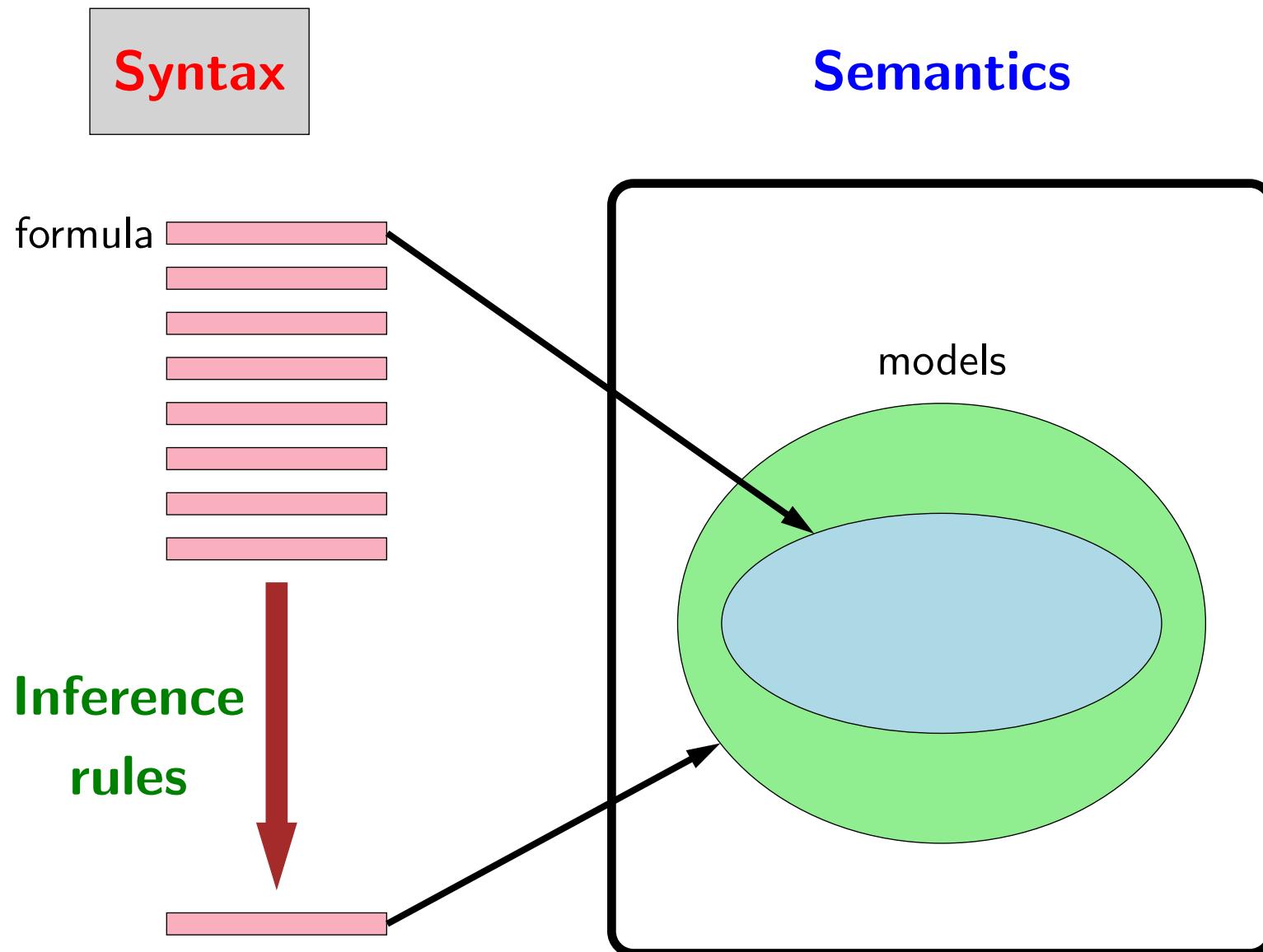
...

Propositional logic is very clunky. What's missing?

- **Objects and predicates:** propositions (e.g., AliceKnowsArithmetic) have more internal structure (alice, Knows, arithmetic)
- **Quantifiers and variables:** *all* is a quantifier which applies to each person, don't want to enumerate them all...

- What's missing? The key conceptual observation is that the world is not just a bunch of atomic facts, but that each fact is actually made out of **objects** and **predicates** on those objects.
- Once facts are decomposed in this way, we can use **quantifiers** and **variables** to implicitly define a huge (and possibly infinite) number of facts with one compact formula. Again, where logic excels is the ability to represent complex things via simple means.

# First-order logic



- We will now introduce **first-order logic**, which will address the representational limitations of propositional logic.
- Remember to define a logic, we need to talk about its syntax, its semantics (interpretation function), and finally inference rules that we can use to operate on the syntax.

# First-order logic: examples

*Alice and Bob both know arithmetic.*

$$\text{Knows(alice, arithmetic)} \wedge \text{Knows(bob, arithmetic)}$$

*All students know arithmetic.*

$$\forall x \text{ Student}(x) \rightarrow \text{Knows}(x, \text{arithmetic})$$

- Before formally defining things, let's look at two examples. First-order logic is basically propositional logic with a few more symbols.

# Syntax of first-order logic

Terms (refer to objects):

- Constant symbol (e.g., arithmetic)
- Variable (e.g.,  $x$ )
- Function of terms (e.g.,  $\text{Sum}(3, x)$ )

Formulas (refer to truth values):

- Atomic formulas (atoms): predicate applied to terms (e.g.,  $\text{Knows}(x, \text{arithmetic})$ )
- Connectives applied to formulas (e.g.,  $\text{Student}(x) \rightarrow \text{Knows}(x, \text{arithmetic})$ )
- Quantifiers applied to formulas (e.g.,  $\forall x \text{ Student}(x) \rightarrow \text{Knows}(x, \text{arithmetic})$ )

- In propositional logic, everything was a formula (or a connective). In first-order logic, there are two types of beasts: terms and formulas. There are three types of terms: constant symbols (which refer to specific objects), variables (which refer to some unspecified object to be determined by quantifiers), and functions (which is a function applied to a set of arguments which are themselves terms).
- Given the terms, we can form atomic formulas, which are the analogue of propositional symbols, but with internal structure (e.g., terms).
- From this point, we can apply the same connectives on these atomic formulas, as we applied to propositional symbols in propositional logic. At this level, first-order logic looks very much like propositional logic.
- Finally, to make use of the fact that atomic formulas have internal structure, we have **quantifiers**, which are really the whole point of first-order logic!

# Quantifiers

Universal quantification ( $\forall$ ):

Think conjunction:  $\forall x P(x)$  is like  $P(A) \wedge P(B) \wedge \dots$

Existential quantification ( $\exists$ ):

Think disjunction:  $\exists x P(x)$  is like  $P(A) \vee P(B) \vee \dots$

Some properties:

- $\neg \forall x P(x)$  equivalent to  $\exists x \neg P(x)$
- $\forall x \exists y \text{Knows}(x, y)$  different from  $\exists y \forall x \text{Knows}(x, y)$

- There are two types of quantifiers: universal and existential. These are basically glorified ways of doing conjunction and disjunction, respectively.
- For crude intuition, we can think of conjunction and disjunction as very nice syntactic sugar, which can be rolled out into something that looks more like propositional logic. But quantifiers aren't just sugar, and it is important that they be compact, for sometimes the variable being quantified over can take on an infinite number of objects.
- That being said, the conjunction and disjunction intuition suffices for day-to-day guidance. For example, it should be intuitive that pushing the negation inside a universal quantifier (conjunction) turns it into a existential (disjunction), which was the case for propositional logic (by de Morgan's laws). Also, one cannot interchange universal and existential quantifiers any more than one can swap conjunction and disjunction in propositional logic.

# Natural language quantifiers

Universal quantification ( $\forall$ ):

*Every student knows arithmetic.*

$$\forall x \text{Student}(x) \rightarrow \text{Knows}(x, \text{arithmetic})$$

Existential quantification ( $\exists$ ):

*Some student knows arithmetic.*

$$\exists x \text{Student}(x) \wedge \text{Knows}(x, \text{arithmetic})$$

**Note the different connectives!**

- Universal and existential quantifiers naturally correspond to the words *every* and *some*, respectively. But when converting English to formal logic, one must exercise caution.
- *Every* can be thought of as taking two arguments  $P$  and  $Q$  (e.g., *student* and *knows arithmetic*). The connective between  $P$  and  $Q$  is an implication (not conjunction, which is a common mistake). This makes sense because when we talk about every  $P$ , we are only restricting our attention to objects  $x$  for which  $P(x)$  is true. Implication does exactly that.
- On the other hand, the connective for existential quantification is conjunction, because we're asking for an object  $x$  such that  $P(x)$  and  $Q(x)$  both hold.

# Some examples of first-order logic

*There is some course that every student has taken.*

$$\exists y \text{Course}(y) \wedge [\forall x \text{Student}(x) \rightarrow \text{Takes}(x, y)]$$

*Every even integer greater than 2 is the sum of two primes.*

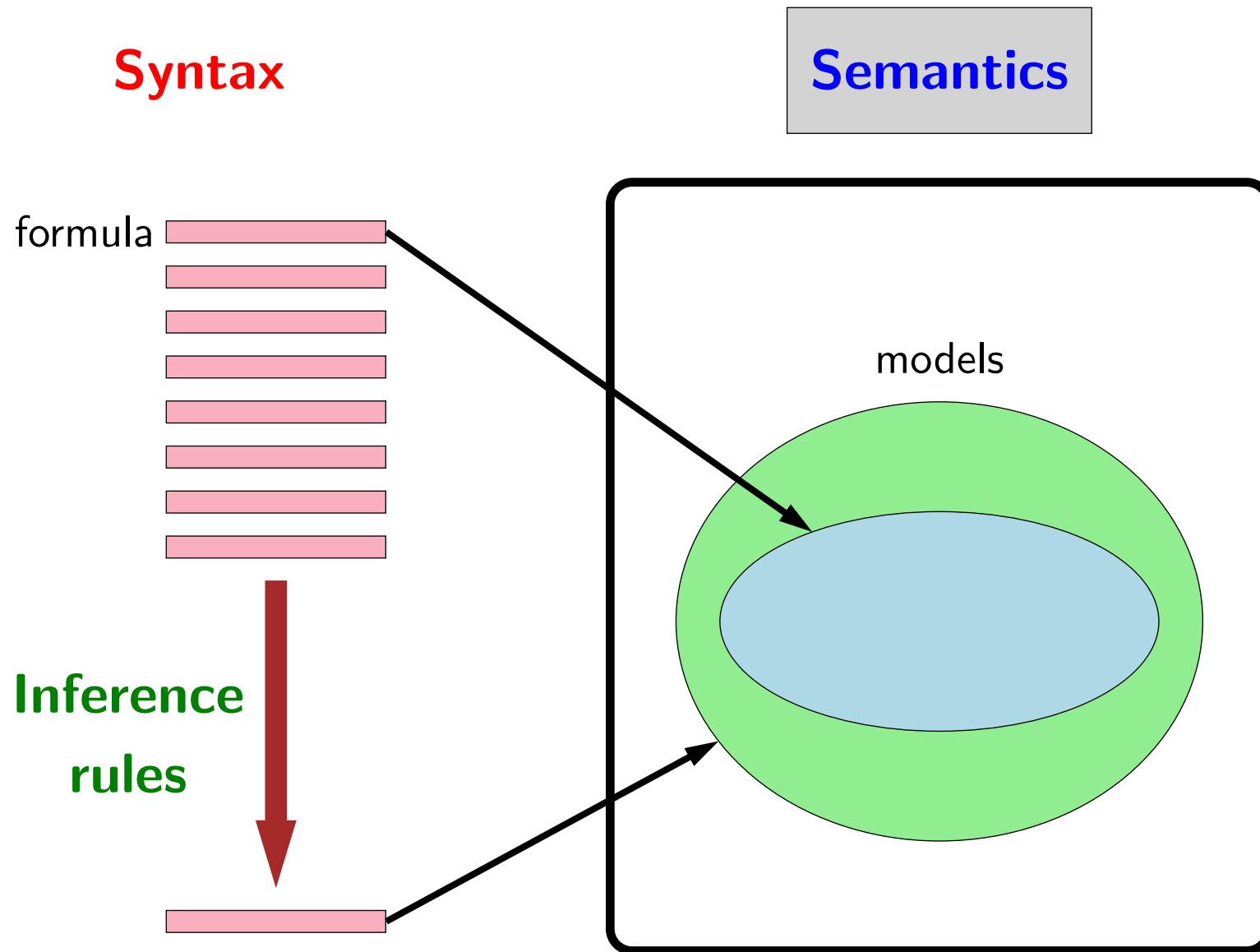
$$\forall x \text{EvenInt}(x) \wedge \text{Greater}(x, 2) \rightarrow \exists y \exists z \text{Equals}(x, \text{Sum}(y, z)) \wedge \text{Prime}(y) \wedge \text{Prime}(z)$$

*If a student takes a course and the course covers a concept, then the student knows that concept.*

$$\forall x \forall y \forall z (\text{Student}(x) \wedge \text{Takes}(x, y) \wedge \text{Course}(y) \wedge \text{Covers}(y, z)) \rightarrow \text{Knows}(x, z)$$

- Let's do some more examples of converting natural language to first-order logic. Remember the connectives associated with existential and universal quantification!
- Note that some English words such as *a* can trigger both universal or existential quantification, depending on context. In *A student took CS221*, we have existential quantification, but in *if a student takes CS221, ...*, we have universal quantification.
- Formal logic clears up the ambiguities associated with natural language.

# First-order logic



- So far, we've only presented the syntax of first-order logic, although we've actually given quite a bit of intuition about what the formulas mean. After all, it's hard to talk about the syntax without at least a hint of semantics for motivation.
- Now let's talk about the formal semantics of first-order logic.

# Models in first-order logic

Recall a model represents a possible situation in the world.

Propositional logic: Model  $w$  maps **propositional symbols** to truth values.

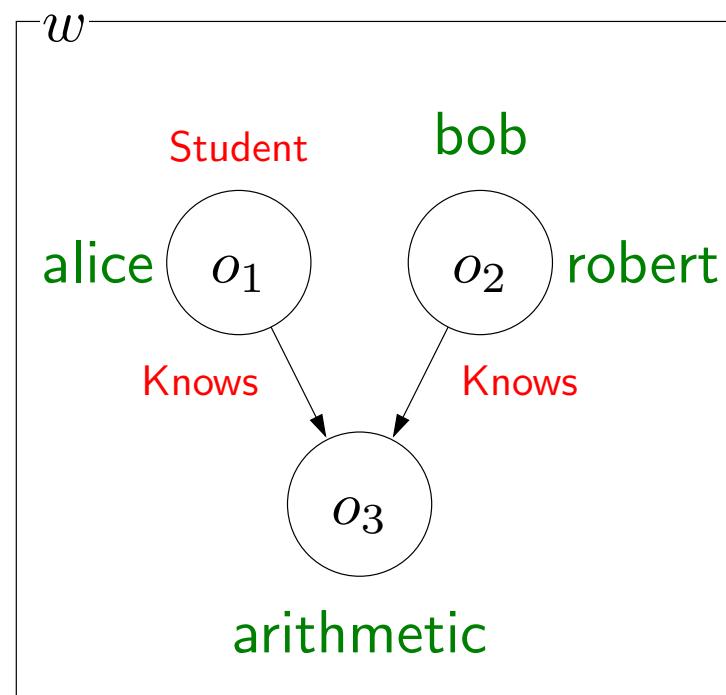
$$w = \{\text{AliceKnowsArithmetic} : 1, \text{BobKnowsArithmetic} : 0\}$$

First-order logic: ?

- Recall that a model in propositional logic was just an assignment of truth values to propositional symbols.
- A natural candidate for a model in first-order logic would then be an assignment of truth values to grounded atomic formula (those formulas whose terms are constants as opposed to variables). This is almost right, but doesn't talk about the relationship between constant symbols.

# Graph representation of a model

If only have unary and binary predicates, a model  $w$  can be represented as a directed graph:



- Nodes are objects, labeled with **constant symbols**
- Directed edges are binary predicates, labeled with **predicate symbols**; unary predicates are additional node labels

- A better way to think about a first-order model is that there are a number of objects in the world ( $o_1, o_2, \dots$ ); think of these as nodes in a graph. Then we have predicates between these objects. Predicates that take two arguments can be visualized as labeled edges between objects. Predicates that take one argument can be visualized as node labels (but these are not so important).
- So far, the objects are unnamed. We can access individual objects directly using constant symbols, which are labels on the nodes.

# Models in first-order logic



## Definition: model in first-order logic

A model  $w$  in first-order logic maps:

- constant symbols to objects

$$w(\text{alice}) = o_1, w(\text{bob}) = o_2, w(\text{arithmetic}) = o_3$$

- predicate symbols to tuples of objects

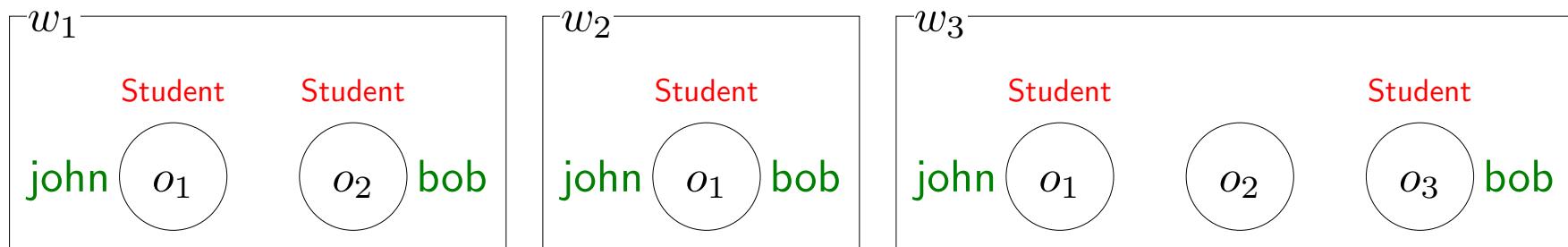
$$w(\text{Knows}) = \{(o_1, o_3), (o_2, o_3), \dots\}$$

- Formally, a first-order model  $w$  maps constant symbols to objects and predicate symbols to tuples of objects (2 for binary predicates).

# A restriction on models

*John and Bob are students.*

$$\text{Student(john)} \wedge \text{Student(bob)}$$



- **Unique names assumption:** Each object has **at most one constant symbol**. This rules out  $w_2$ .
- **Domain closure:** Each object has **at least one constant symbol**. This rules out  $w_3$ .

Point:

constant symbol  object

- Note that by default, two constant symbols can refer to the same object, and there can be objects which no constant symbols refer to. This can make life somewhat confusing. Fortunately, there are two assumptions that people sometimes make to simplify things.
- The unique names assumption says that there's at most one way to refer to an object via a constant symbol. Domain closure says there's at least one. Together, they imply that there is a one-to-one relationship between constant symbols in syntax-land and objects in semantics-land.

# Propositionalization

If one-to-one mapping between constant symbols and objects (**unique names** and **domain closure**),

first-order logic is syntactic sugar for propositional logic:

## Knowledge base in first-order logic

$\text{Student}(\text{alice}) \wedge \text{Student}(\text{bob})$

$\forall x \text{ Student}(x) \rightarrow \text{Person}(x)$

$\exists x \text{ Student}(x) \wedge \text{Creative}(x)$

## Knowledge base in propositional logic

$\text{Studentalice} \wedge \text{Studentbob}$

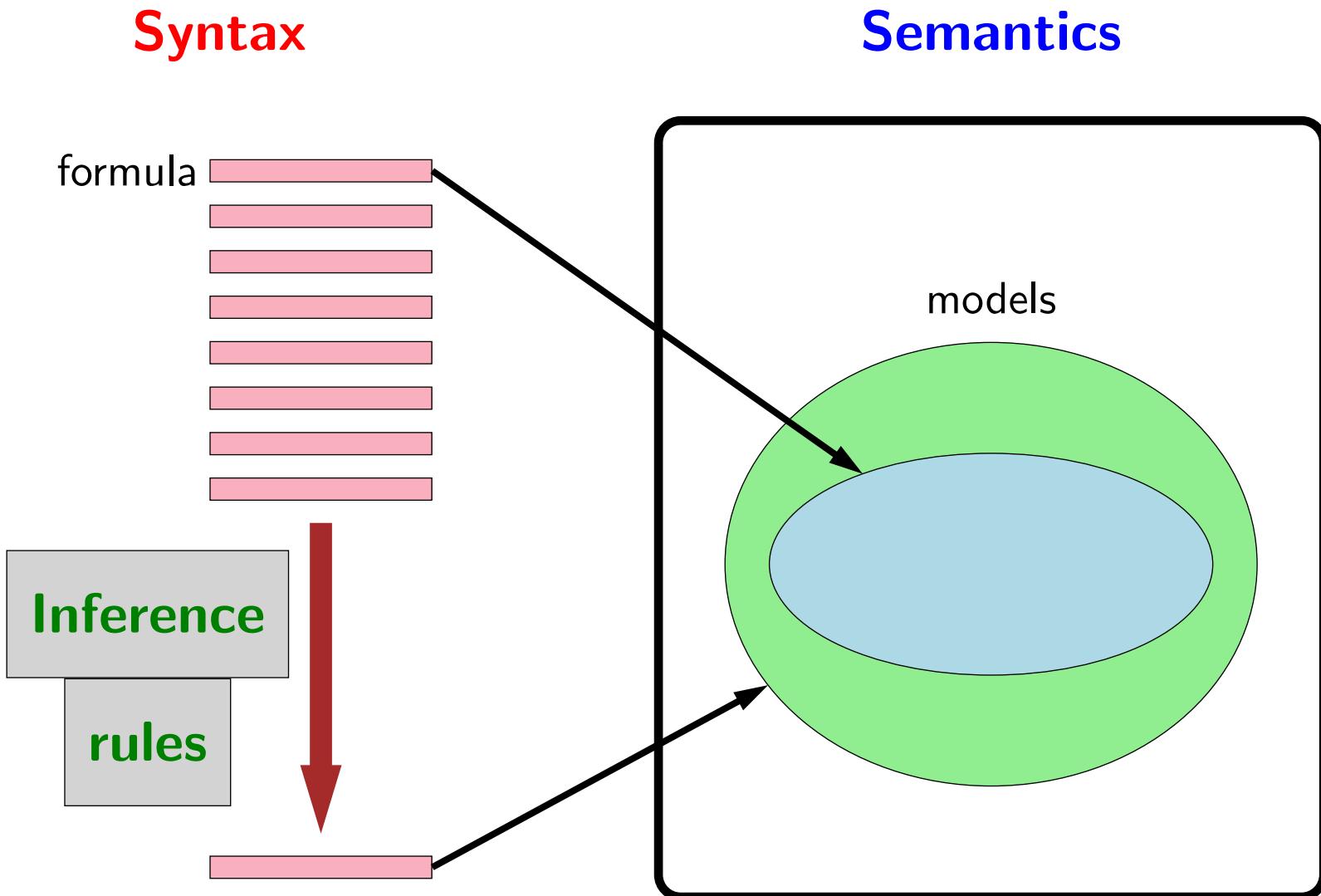
$(\text{Studentalice} \rightarrow \text{Personalice}) \wedge (\text{Studentbob} \rightarrow \text{Personbob})$

$(\text{Studentalice} \wedge \text{Creativealice}) \vee (\text{Studentbob} \wedge \text{Creativebob})$

Point: use any inference algorithm for propositional logic!

- If a one-to-one mapping really exists, then we can **propositionalize** all our formulas, which basically unrolls all the quantifiers into explicit conjunctions and disjunctions.
- The upshot of this conversion, is that we're back to propositional logic, and we know how to do inference in propositional logic (either using model checking or by applying inference rules). Of course, propositionalization could be quite expensive and not the most efficient thing to do.

# First-order logic

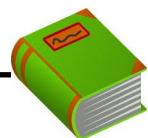


- Now we look at inference rules which can make first-order inference much more efficient. The key is to do everything implicitly and avoid propositionalization; again the whole spirit of logic is to do things compactly and implicitly.

# Definite clauses

$$\forall x \forall y \forall z (\text{Takes}(x, y) \wedge \text{Covers}(y, z)) \rightarrow \text{Knows}(x, z)$$

Note: if propositionalize, get one formula for each value to  $(x, y, z)$ , e.g.,  
(alice, cs221, mdp)



## Definition: definite clause (first-order logic)

A definite clause has the following form:

$$\forall x_1 \dots \forall x_n (a_1 \wedge \dots \wedge a_k) \rightarrow b$$

for variables  $x_1, \dots, x_n$  and atomic formulas  $a_1, \dots, a_k, b$  (which contain those variables).

- Like our development of inference in propositional logic, we will first talk about first-order logic restricted to Horn clauses, in which case a first-order version of modus ponens will be sound and complete. After that, we'll see how resolution allows to handle all of first-order logic.
- We start by generalizing definite clauses from propositional logic to first-order logic. The only difference is that we now have universal quantifiers sitting at the beginning of the definite clause. This makes sense since universal quantification is associated with implication, and one can check that if one propositionalizes a first-order definite clause, one obtains a set (conjunction) of multiple propositional definite clauses.

# Modus ponens (first attempt)



**Definition: modus ponens (first-order logic)**

$$\frac{a_1, \dots, a_k \quad \forall x_1 \dots \forall x_n (a_1 \wedge \dots \wedge a_k) \rightarrow b}{b}$$

Setup:

Given  $P(\text{alice})$  and  $\forall x P(x) \rightarrow Q(x)$ .

Problem:

Can't infer  $Q(\text{alice})$  because  $P(x)$  and  $P(\text{alice})$  don't match!

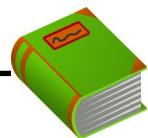
Solution: substitution and unification

- If we try to write down the modus ponens rule, we would fail.
- As a simple example, suppose we are given  $P(\text{alice})$  and  $\forall x P(x) \rightarrow Q(x)$ . We would naturally want to derive  $Q(\text{alice})$ . But notice that we can't apply modus ponens because  $P(\text{alice})$  and  $P(x)$  don't match!
- Recall that we're in syntax-land, which means that these formulas are just symbols. Inference rules don't have access to the semantics of the constants and variables — it is just a pattern matcher. So we have to be very methodical.
- To develop a mechanism to match variables and constants, we will introduce two concepts, substitution and unification for this purpose.

# Substitution

$$\text{Subst}[\{x/\text{alice}\}, P(x)] = P(\text{alice})$$

$$\text{Subst}[\{x/\text{alice}, y/z\}, P(x) \wedge K(x, y)] = P(\text{alice}) \wedge K(\text{alice}, z)$$



## Definition: Substitution

A substitution  $\theta$  is a mapping from variables to terms.

$\text{Subst}[\theta, f]$  returns the result of performing substitution  $\theta$  on  $f$ .

- The first step is substitution, which applies a search-and-replace operation on a formula or term.
- We won't define  $\text{Subst}[\theta, f]$  formally, but from the examples, it should be clear what  $\text{Subst}$  does.
- Technical note: if  $\theta$  contains variable substitutions  $x/\text{alice}$  we only apply the substitution to the free variables in  $f$ , which are the variables not bound by quantification (e.g.,  $x$  in  $\exists y, P(x, y)$ ). Later, we'll see how CNF formulas allow us to remove all the quantifiers.

# Unification

$\text{Unify}[\text{Knows(alice, arithmetic)}, \text{Knows}(x, \text{arithmetic})] = \{x/\text{alice}\}$

$\text{Unify}[\text{Knows(alice, }y\text{)}, \text{Knows}(x, z)] = \{x/\text{alice}, y/z\}$

$\text{Unify}[\text{Knows(alice, }y\text{)}, \text{Knows(bob, }z\text{)}] = \text{fail}$

$\text{Unify}[\text{Knows(alice, }y\text{)}, \text{Knows}(x, F(x))] = \{x/\text{alice}, y/F(\text{alice})\}$



## Definition: Unification

Unification takes two formulas  $f$  and  $g$  and returns a substitution  $\theta$  which is the most general unifier:

$\text{Unify}[f, g] = \theta$  such that  $\text{Subst}[\theta, f] = \text{Subst}[\theta, g]$   
or "fail" if no such  $\theta$  exists.

- Substitution can be used to make two formulas identical, and unification is the way to find the least committal substitution we can find to achieve this.
- Unification, like substitution, can be implemented recursively. The implementation details are not the most exciting, but it's useful to get some intuition from the examples.

# Modus ponens



**Definition: modus ponens (first-order logic)**

$$\frac{a'_1, \dots, a'_k \quad \forall x_1 \dots \forall x_n (a_1 \wedge \dots \wedge a_k) \rightarrow b}{b'}$$

Get most general unifier  $\theta$  on premises:

- $\theta = \text{Unify}[a'_1 \wedge \dots \wedge a'_k, a_1 \wedge \dots \wedge a_k]$

Apply  $\theta$  to conclusion:

- $\text{Subst}[\theta, b] = b'$

- Having defined substitution and unification, we are in position to finally define the modus ponens rule for first-order logic. Instead of performing an exact match, we instead perform a unification, which generates a substitution  $\theta$ . Using  $\theta$ , we can generate the conclusion  $b'$  on the fly.
- Note the significance here: the rule  $a_1 \wedge \dots \wedge a_k \rightarrow b$  can be used in a myriad ways, but Unify identifies the appropriate substitution, so that it can be applied to the conclusion.

# Modus ponens example



## Example: modus ponens in first-order logic

Premises:

$\text{Takes}(\text{alice}, \text{cs221})$

$\text{Covers}(\text{cs221}, \text{mdp})$

$\forall x \forall y \forall z \text{Takes}(x, y) \wedge \text{Covers}(y, z) \rightarrow \text{Knows}(x, z)$

Conclusion:

$\theta = \{x/\text{alice}, y/\text{cs221}, z/\text{mdp}\}$

Derive  $\text{Knows}(\text{alice}, \text{mdp})$

- Here's a simple example of modus ponens in action. We bind  $x, y, z$  to appropriate objects (constant symbols), which is used to generate the conclusion  $\text{Knows}(\text{alice}, \text{mdp})$ .

# Complexity

$$\forall x \forall y \forall z P(x, y, z)$$

- Each application of Modus ponens produces an atomic formula.
- If no function symbols, number of atomic formulas is at most  $(\text{num-constant-symbols})^{(\text{maximum-predicate-arity})}$
- If there are function symbols (e.g.,  $F$ ), then infinite...

$$Q(a) \quad Q(F(a)) \quad Q(F(F(a))) \quad Q(F(F(F(a)))) \quad \dots$$

- In propositional logic, modus ponens was considered efficient, since in the worst case, we generate each propositional symbol.
- In first-order logic, though, we typically have many more atomic formulas in place of propositional symbols, which leads to a potentially exponentially number of atomic formulas, or worse, with function symbols, there might be an infinite set of atomic formulas.

# Complexity



## Theorem: completeness

Modus ponens is complete for first-order logic with only Horn clauses.



## Theorem: semi-decidability

First-order logic (even restricted to only Horn clauses) is **semi-decidable**.

- If  $\text{KB} \models f$ , forward inference on complete inference rules will prove  $f$  in finite time.
- If  $\text{KB} \not\models f$ , no algorithm can show this in finite time.

- We can show that modus ponens is complete with respect to Horn clauses, which means that every true formula has an actual finite derivation.
- However, this doesn't mean that we can just run modus ponens and be done with it, for first-order logic even restricted to Horn clauses is semi-decidable, which means that if a formula is entailed, then we will be able to derive it, but if it is not entailed, then we don't even know when to stop the algorithm — quite troubling!
- With propositional logic, there were a finite number of propositional symbols, but now the number of atomic formulas can be infinite (the culprit is function symbols).
- Though we have hit a theoretical barrier, life goes on and we can still run modus ponens inference to get a one-sided answer. Next, we will move to working with full first-order logic.

# Resolution

Recall: First-order logic includes non-Horn clauses

$$\forall x \text{ Student}(x) \rightarrow \exists y \text{ Knows}(x, y)$$

High-level strategy (same as in propositional logic):

- Convert all formulas to CNF
- Repeatedly apply resolution rule

- To go beyond Horn clauses, we will develop a single resolution rule which is sound and complete.
- The high-level strategy is the same as propositional logic: convert to CNF and apply resolution.

# Conversion to CNF

Input:

$$\forall x (\forall y \text{Animal}(y) \rightarrow \text{Loves}(x, y)) \rightarrow \exists y \text{Loves}(y, x)$$

Output:

$$(\text{Animal}(Y(x)) \vee \text{Loves}(Z(x), x)) \wedge (\neg \text{Loves}(x, Y(x)) \vee \text{Loves}(Z(x), x))$$

New to first-order logic:

- All variables (e.g.,  $x$ ) have universal quantifiers by default
- Introduce **Skolem functions** (e.g.,  $Y(x)$ ) to represent existential quantified variables

- Consider the logical formula corresponding to *Everyone who loves all animals is loved by someone*. The slide shows the desired output, which looks like a CNF formula in propositional logic, but there are two differences: there are variables (e.g.,  $x$ ) and functions of variables (e.g.,  $Y(x)$ ). The variables are assumed to be universally quantified over, and the functions are called **Skolem functions** and stand for a property of the variable.

# Conversion to CNF (part 1)

*Anyone who likes all animals is liked by someone.*

Input:

$$\forall x (\forall y \text{Animal}(y) \rightarrow \text{Loves}(x, y)) \rightarrow \exists y \text{Loves}(y, x)$$

Eliminate implications (old):

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

Push  $\neg$  inwards, eliminate double negation (old):

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

Standardize variables (**new**):

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

- We start by eliminating implications, pushing negation inside, and eliminating double negation, which is all old.
- The first thing new to first-order logic is standardization of variables. Note that in  $\exists x P(x) \wedge \exists x Q(x)$ , there are two instances of  $x$  whose scopes don't overlap. To make this clearer, we will convert this into  $\exists x P(x) \wedge \exists y Q(y)$ . This sets the stage for when we will drop the quantifiers on the variables.

# Conversion to CNF (part 2)

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

Replace existentially quantified variables with Skolem functions (**new**):

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

Distribute  $\vee$  over  $\wedge$  (**old**):

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

Remove universal quantifiers (**new**):

$$[\text{Animal}(Y(x)) \vee \text{Loves}(Z(x), x)] \wedge [\neg \text{Loves}(x, Y(x)) \vee \text{Loves}(Z(x), x)]$$

- The next step is to remove existential variables by replacing them with Skolem functions. This is perhaps the most non-trivial part of the process. Consider the formula:  $\forall x \exists y P(x, y)$ . Here,  $y$  is existentially quantified and depends on  $x$ . So we can mark this dependence explicitly by setting  $y = Y(x)$ . Then the formula becomes  $\forall x P(x, Y(x))$ . You can even think of the function  $Y$  as being existentially quantified over outside the  $\forall x$ .
- Next, we distribute disjunction over conjunction as before.
- Finally, we simply drop all universal quantifiers. Because those are the only quantifiers left, there is no ambiguity.
- The final CNF formula can be difficult to interpret, but we can be assured that the final formula captures exactly the same information as the original formula.

# Resolution



**Definition: resolution rule (first-order logic)**

$$\frac{f_1 \vee \cdots \vee f_n \vee p, \quad \neg q \vee g_1 \vee \cdots \vee g_m}{\text{Subst}[\theta, f_1 \vee \cdots \vee f_n \vee g_1 \vee \cdots \vee g_m]}$$

where  $\theta = \text{Unify}[p, q]$ .



**Example: resolution**

$$\frac{\text{Animal}(Y(x)) \vee \text{Loves}(Z(x), x), \quad \neg \text{Loves}(u, v) \vee \text{Feeds}(u, v)}{\text{Animal}(Y(x)) \vee \text{Feeds}(Z(x), x)}$$

Substitution:  $\theta = \{u/Z(x), v/x\}$ .

- After converting all formulas to CNF, then we can apply the resolution rule, which is generalized to first-order logic. This means that instead of doing exact matching of a literal  $p$ , we unify atomic formulas  $p$  and  $q$ , and then apply the resulting substitution  $\theta$  on the conclusion.



# Summary

## Propositional logic

model checking

⇐ propositionalization

modus ponens  
(Horn clauses)

resolution  
(general)

## First-order logic

n/a

modus ponens++  
(Horn clauses)  
resolution++  
(general)

++: unification and substitution



**Key idea: variables in first-order logic**

Variables yield compact knowledge representations.

- To summarize, we have presented propositional logic and first-order logic. When there is a one-to-one mapping between constant symbols and objects, we can propositionalize, thereby converting first-order logic into propositional logic. This is needed if we want to use model checking to do inference.
- For inference based on syntactic derivations, there is a neat parallel between using modus ponens for Horn clauses and resolution for general formulas (after conversion to CNF). In the first-order logic case, things are more complex because we have to use unification and substitution to do matching of formulas.
- The main idea in first-order logic is the use of variables (not to be confused with the variables in variable-based models, which are mere propositional symbols from the point of view of logic), coupled with quantifiers.
- Propositional formulas allow us to express large complex sets of models compactly using a small piece of propositional syntax. Variables in first-order logic in essence takes this idea one more step forward, allowing us to effectively express large complex propositional formulas compactly using a small piece of first-order syntax.
- Note that variables in first-order logic are not same as the variables in variable-based models (CSPs). CSPs variables correspond to atomic formula and denote truth values. First-order logic variables denote objects.



# Lecture 18: Deep Learning



# A brief history

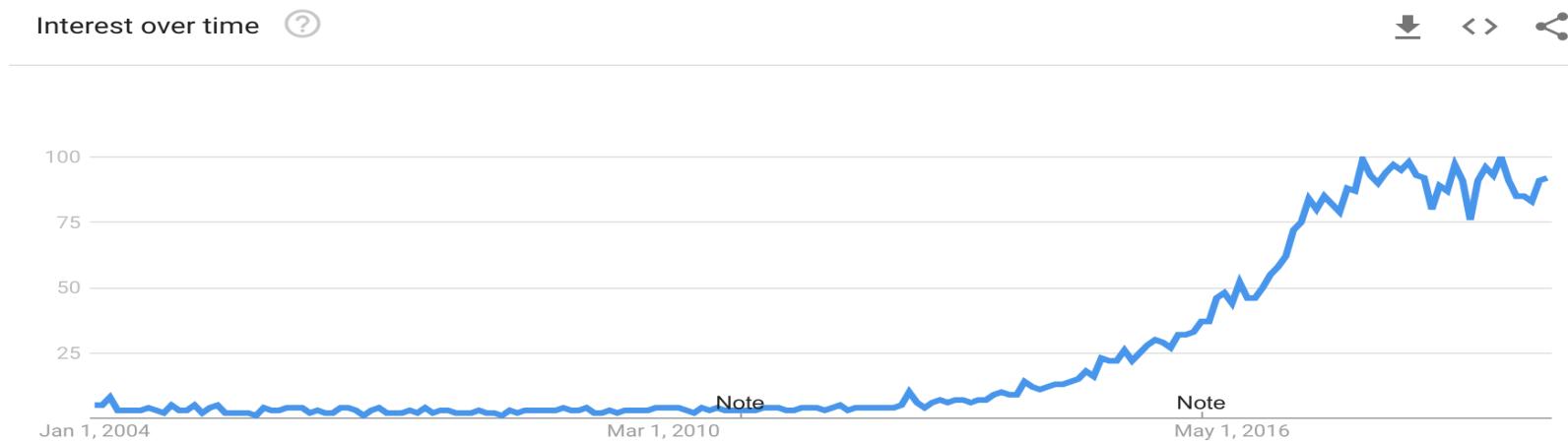
- 1943: neural networks  $\Leftrightarrow$  logical circuits (McCulloch/Pitts)
- 1949: "cells that fire together wire together" learning rule (Hebb)
- 1969: theoretical limitations of neural networks (Minsky/Papert)
- 1974: backpropagation for training multi-layer networks (Werbos)
- 1986: popularization of backpropagation (Rumelhardt, Hinton, Williams)

# A brief history

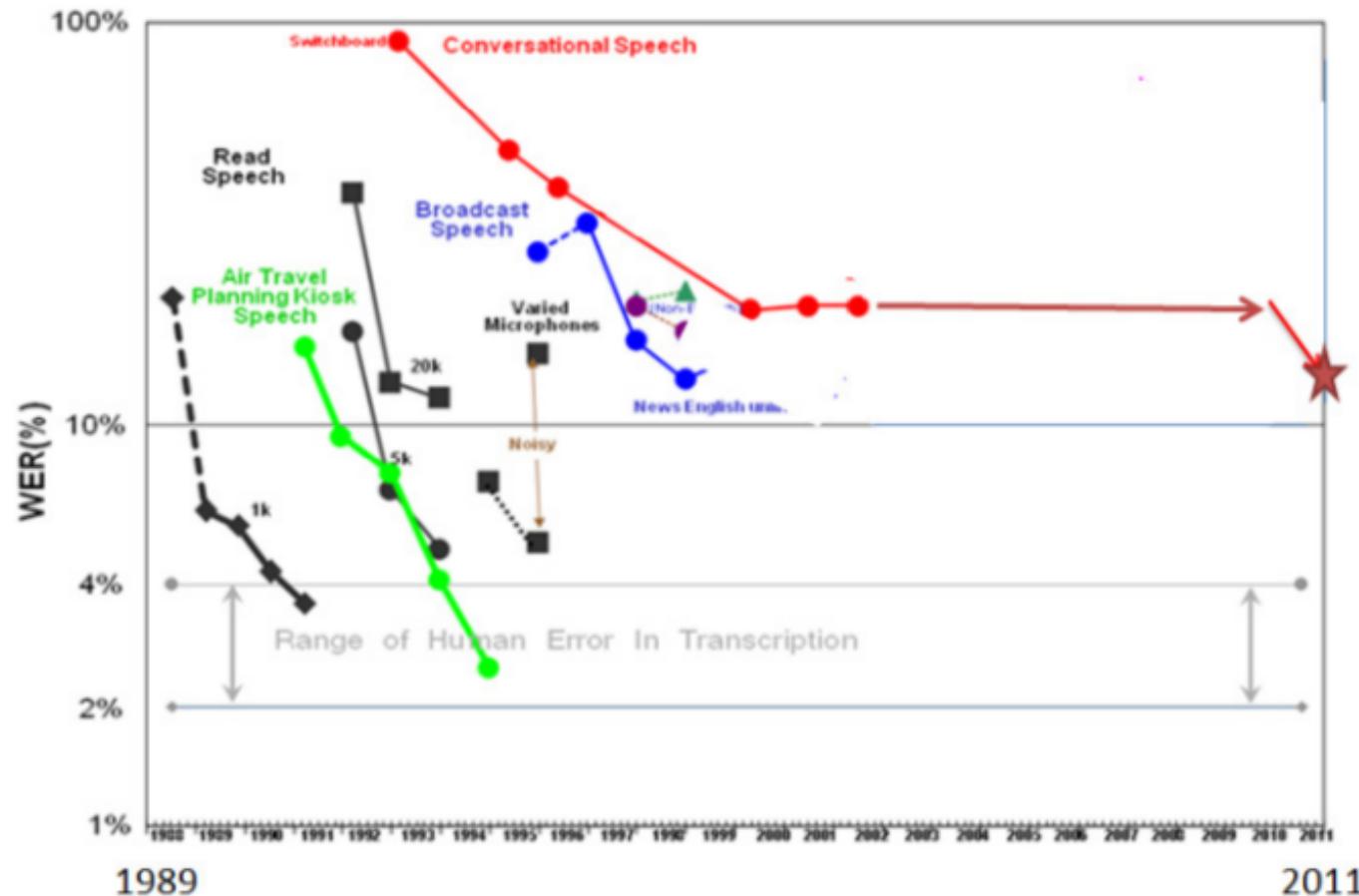
- 1980: Neocognitron, a.k.a. convolutional neural networks (Fukushima)
- 1989: backpropagation on convolutional neural networks (LeCun)
- 1990: recurrent neural networks (Elman)
- 1997: Long Short-Term Memory networks (Hochreiter/Schmidhuber)
- 2006: unsupervised layerwise training of deep networks (Hinton et al.)

# Google Trends

Query: deep learning

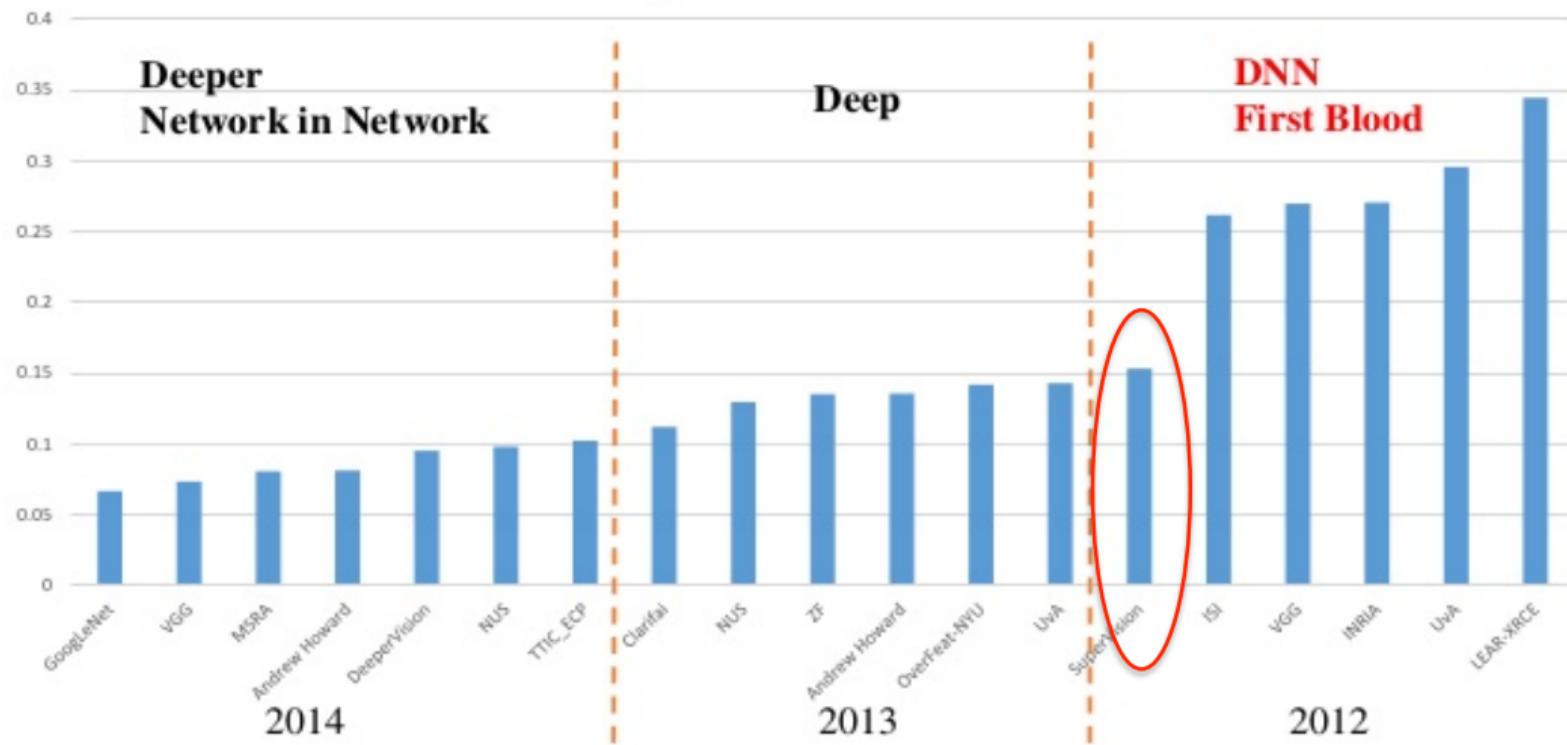


# Speech recognition (2009-2011)



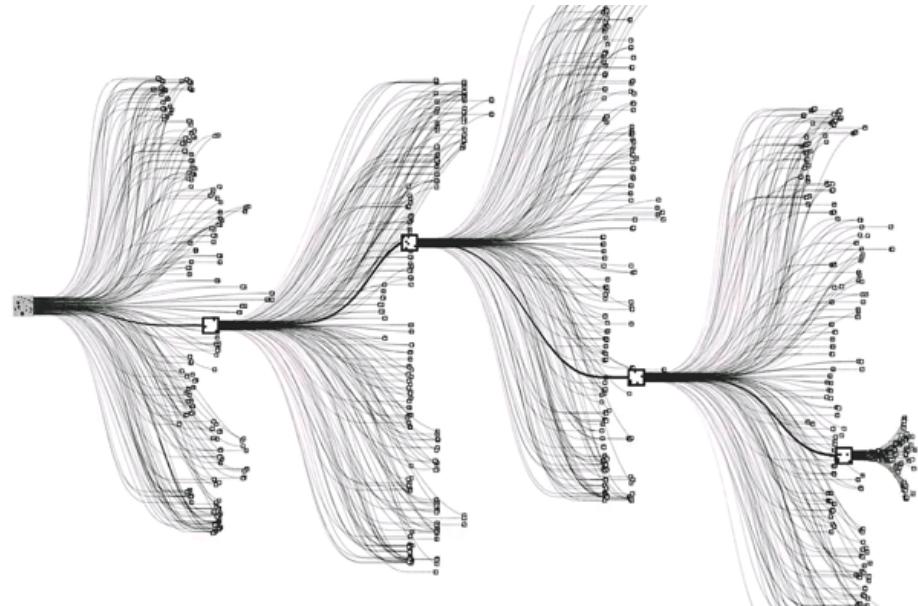
- Steep drop in WER due to deep learning
- IBM, Google, Microsoft all switched over from GMM-HMM

# Object recognition (2012)



- Landslide win in ILSVRC object recognition competition
- Computer vision community switched to CNNs
- Simpler than hand-engineered features (SIFT)

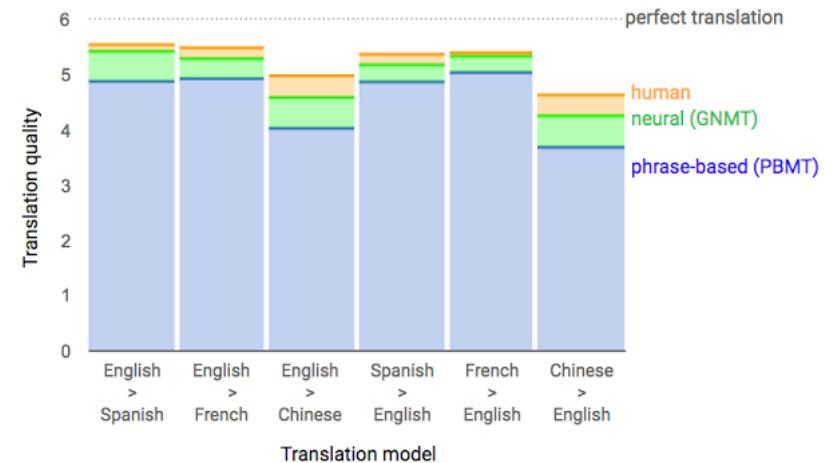
# Go (2016)



- Defeated world champion Le Sedol 4-1
- Simple architecture (in contrast, DeepBlue was search + hand-crafted heuristics)
- 2017: AlphaGoZero does not require human expert games as supervision

# Machine translation (2016)

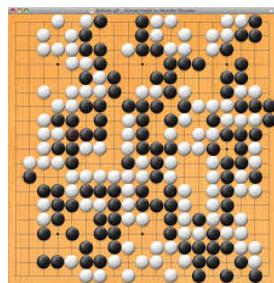
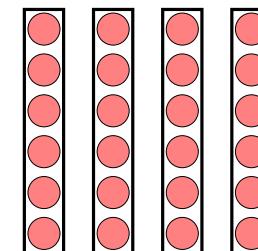
<i>Input sentence:</i>	<i>Translation (PBMT):</i>	<i>Translation (GNMT):</i>	<i>Translation (human):</i>
李克強此行將啟動中加總理年度對話機制，與加拿大總理杜魯多舉行兩國總理首次年度對話。	Li Keqiang premier added this line to start the annual dialogue mechanism with the Canadian Prime Minister Trudeau two prime ministers held its first annual session.	Li Keqiang will start the annual dialogue mechanism with Prime Minister Trudeau of Canada and hold the first annual dialogue between the two premiers.	Li Keqiang will initiate the annual dialogue mechanism between premiers of China and Canada during this visit, and hold the first annual dialogue with Premier Trudeau of Canada.



- Decisive wins have taken longer to achieve in NLP (words are meaningful in a way that pixels are not)
- Current state-of-the-art in machine translation
- Simpler architecture (throw out word alignment, phrases tables, language models)

# What is deep learning?

*A family of techniques for learning compositional vector representations of complex data.*





# Roadmap

**Feedforward neural networks**

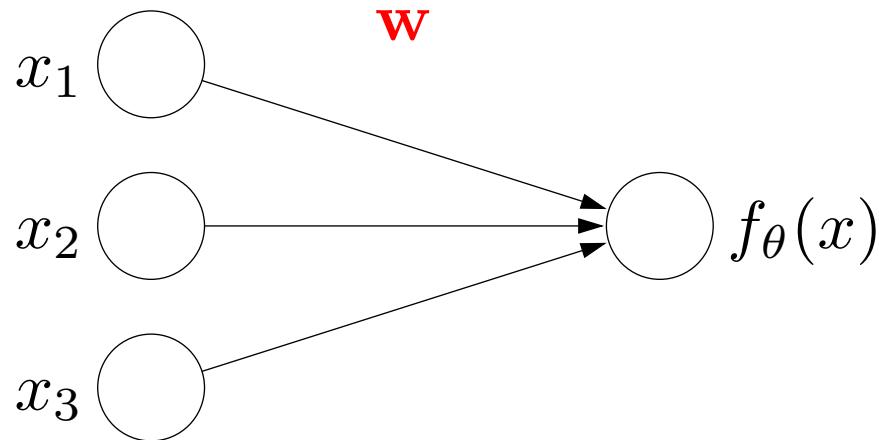
Convolutional neural networks

Recurrent neural networks

Unsupervised learning

Final remarks

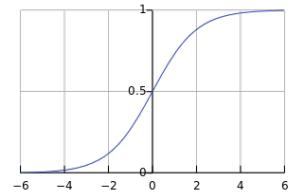
# Review: linear predictors



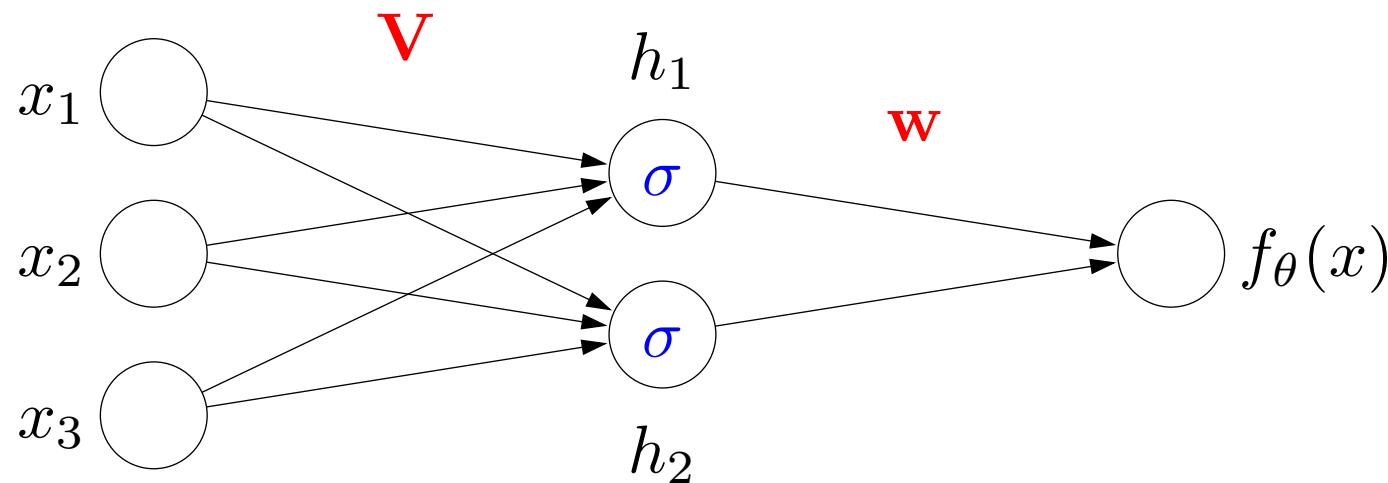
Output:

$$f_{\theta}(x) = \mathbf{w} \cdot \mathbf{x}$$

Parameters:  $\theta = \mathbf{w}$



# Review: neural networks



Intermediate hidden units:

$$h_j(x) = \sigma(\mathbf{v}_j \cdot \mathbf{x}) \quad \sigma(z) = (1 + e^{-z})^{-1}$$

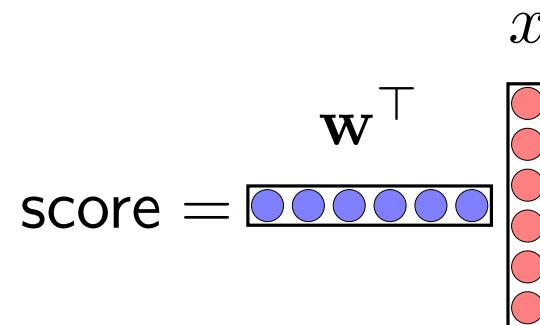
Output:

$$f_\theta(x) = \mathbf{w} \cdot \mathbf{h}(x)$$

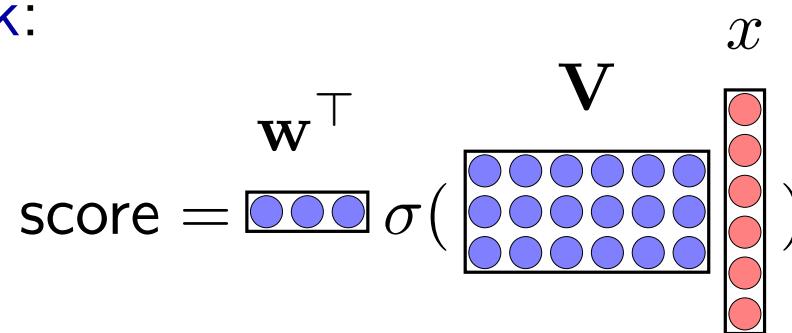
Parameters:  $\theta = (\mathbf{V}, \mathbf{w})$

# Deep neural networks

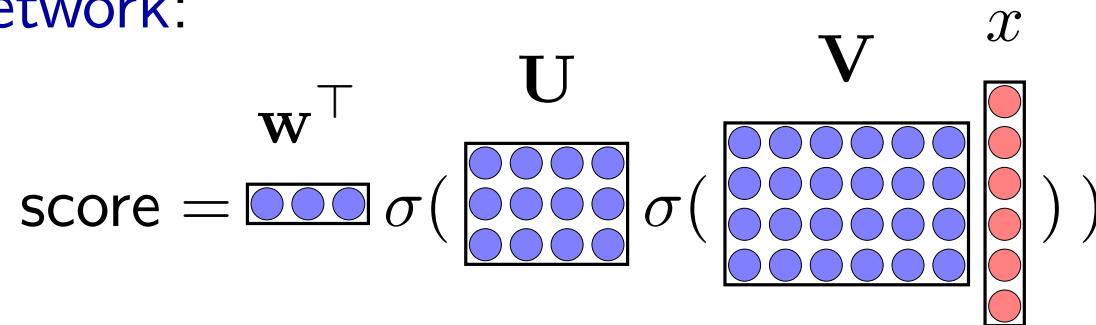
1-layer neural network:



2-layer neural network:

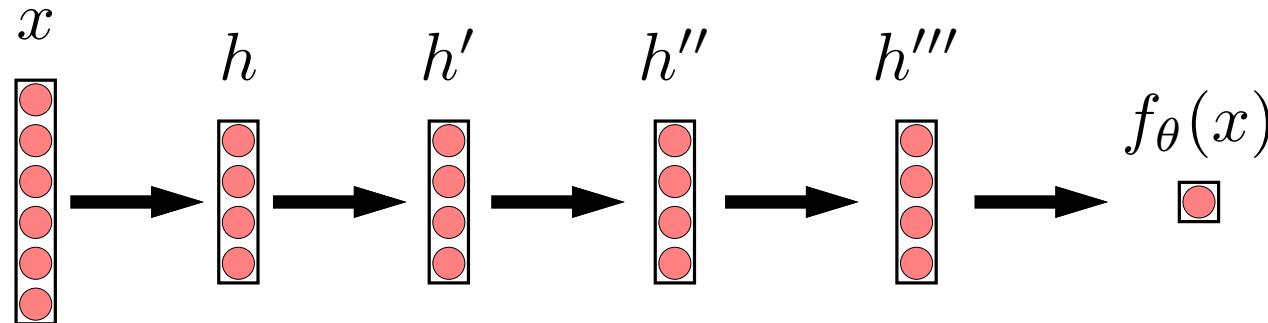


3-layer neural network:



...

# Depth



## Intuitions:

- Hierarchical feature representations
- Can simulate a bounded computation logic circuit (original motivation from McCulloch/Pitts, 1943)
- Learn this computation (and potentially more because networks are real-valued)
- Depth  $k+1$  logic circuits can represent more than depth  $k$  (counting argument)
- Formal theory/understanding is still incomplete

# Review: optimization

Regression:

$$\text{Loss}(x, y, \theta) = (f_\theta(x) - y)^2$$



**Key idea: minimize training loss**

$$\text{TrainLoss}(\theta) = \frac{1}{|\mathcal{D}_{\text{train}}|} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \theta)$$

$$\min_{\theta \in \mathbb{R}^d} \text{TrainLoss}(\theta)$$



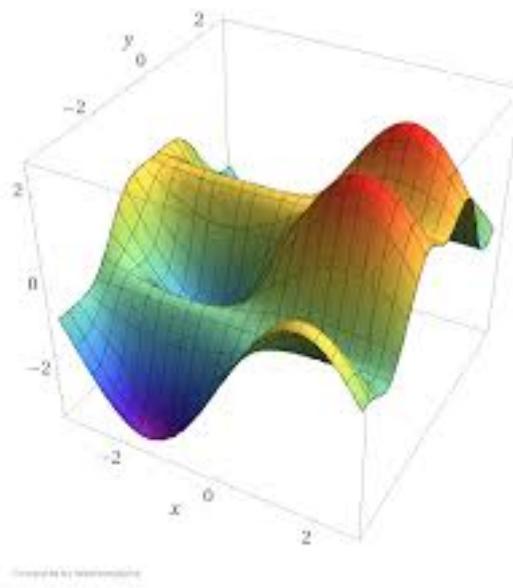
**Algorithm: stochastic gradient descent**

For  $t = 1, \dots, T$ :

    For  $(x, y) \in \mathcal{D}_{\text{train}}$ :

$$\theta \leftarrow \theta - \eta_t \nabla_{\theta} \text{Loss}(x, y, \theta)$$

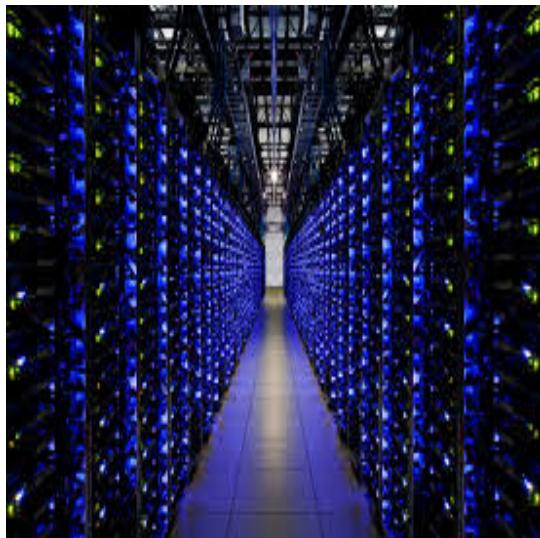
# Training



- Non-convex optimization
- No theoretical guarantees that it works
- Before 2000s, empirically very difficult to get working

# What's different today

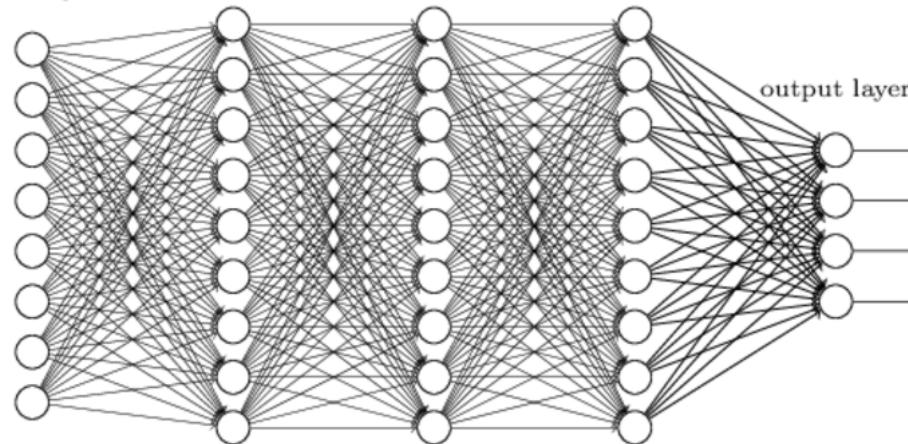
Computation (time/memory)



Information (data)



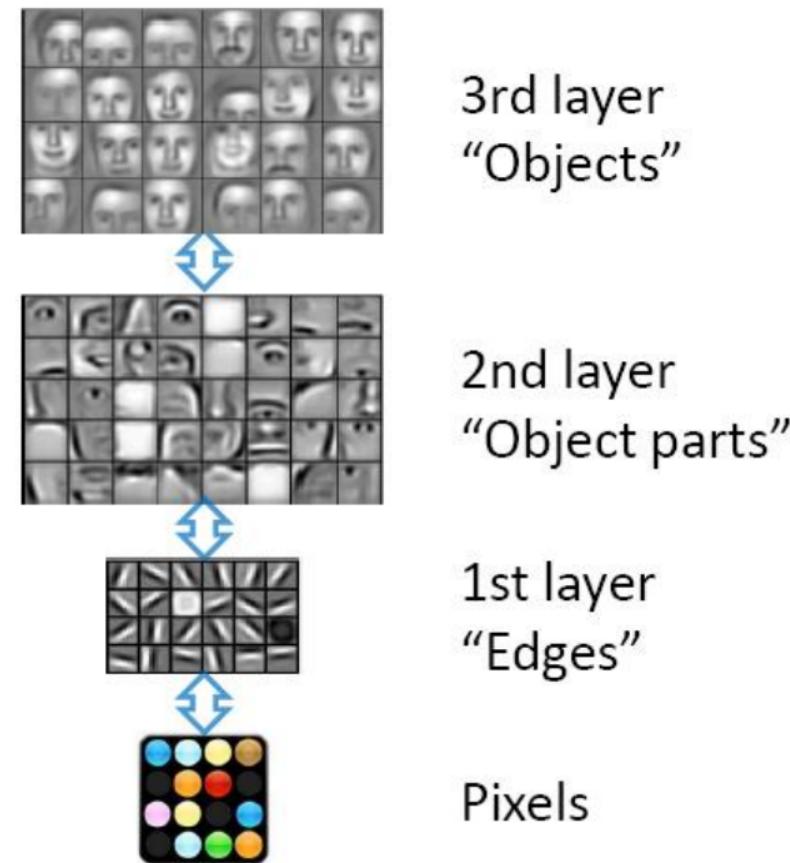
# How to make it work



- More hidden units (over-provisioning)
- Adaptive step sizes (AdaGrad, ADAM)
- Dropout to guard against overfitting
- Careful initialization (pre-training)
- Batch normalization

**Model and optimization are tightly coupled**

# What's learned?





# Summary

- Deep networks learn hierarchical representations of data
- Train via SGD, use backpropagation to compute gradients
- Non-convex optimization, but works empirically given enough compute and data



# Roadmap

Feedforward neural networks

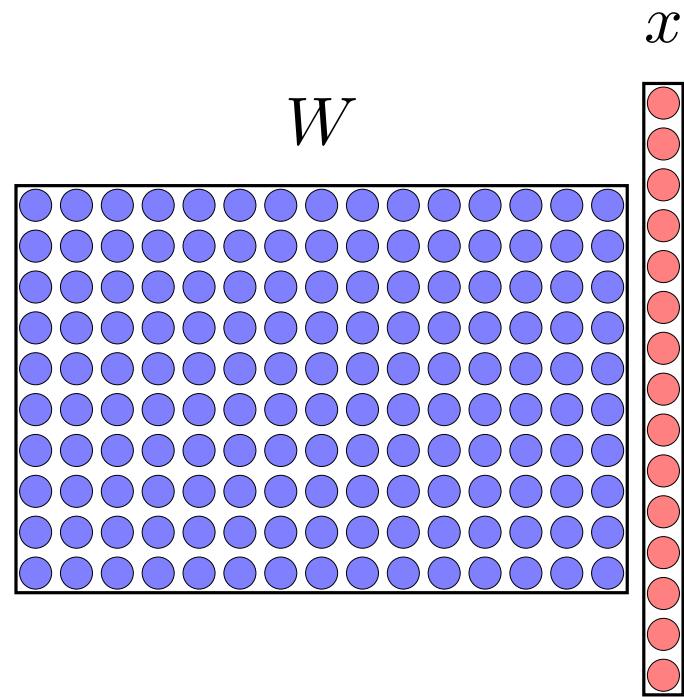
**Convolutional neural networks**

Recurrent neural networks

Unsupervised learning

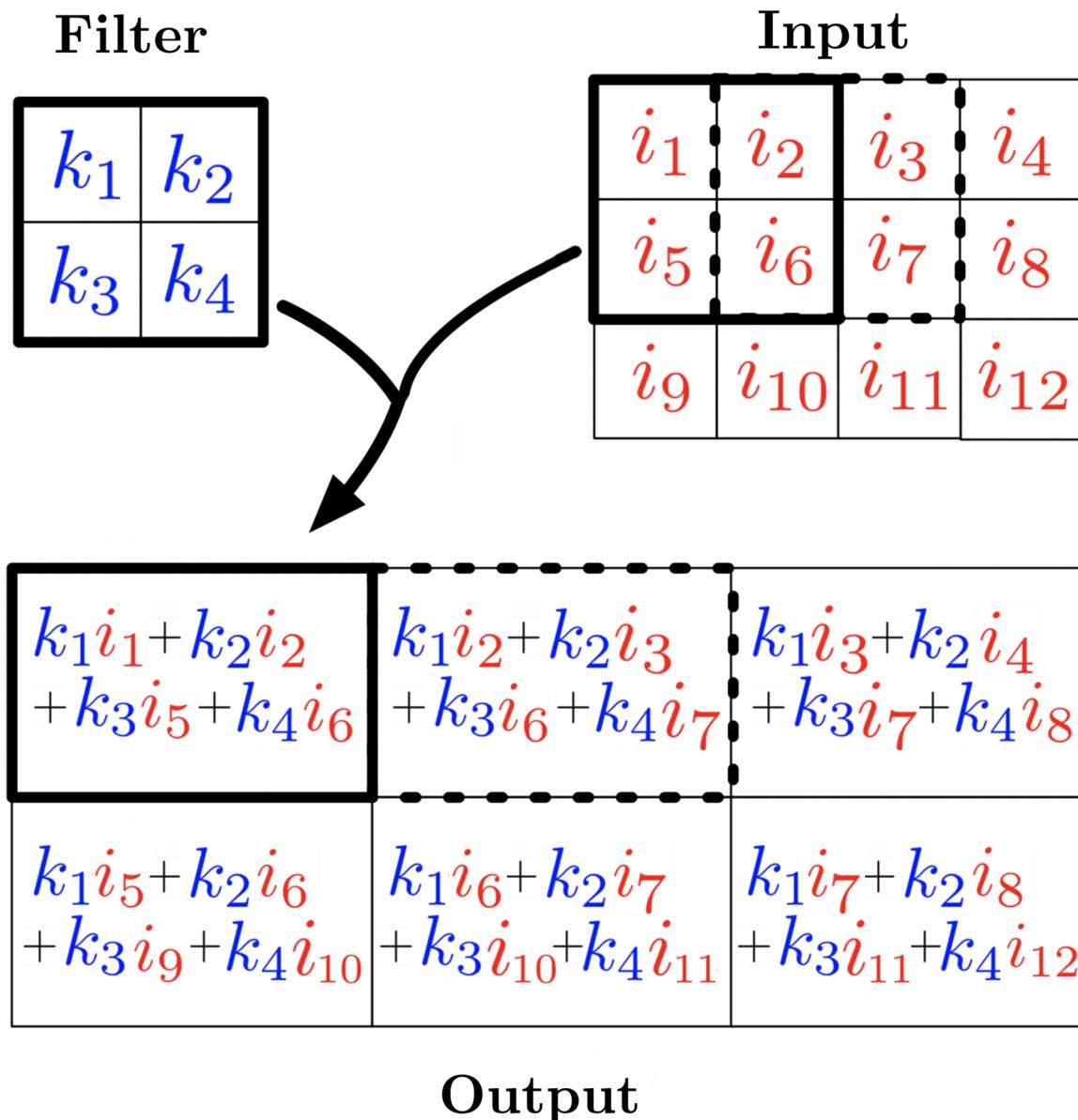
Final remarks

# Motivation

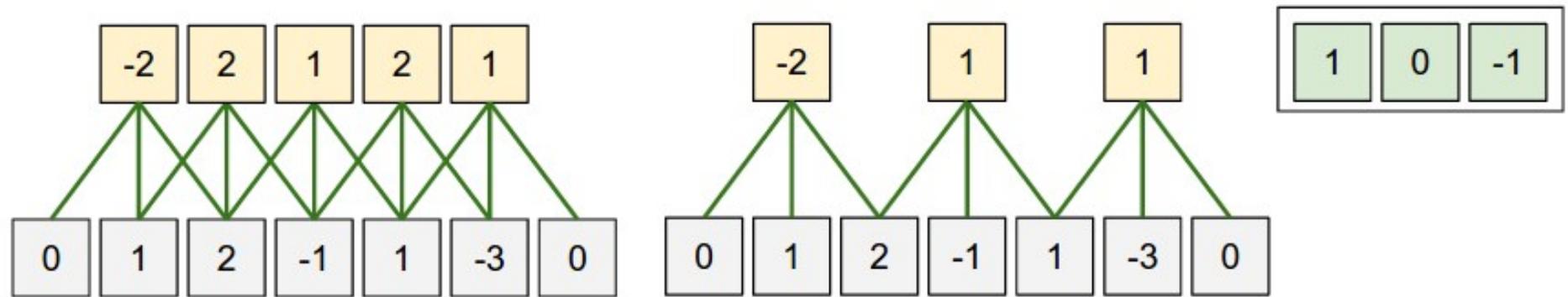


- **Observation:** images are not arbitrary vectors
- **Goal:** leverage spatial structure of images (translation invariance)

# Idea: Convolutions

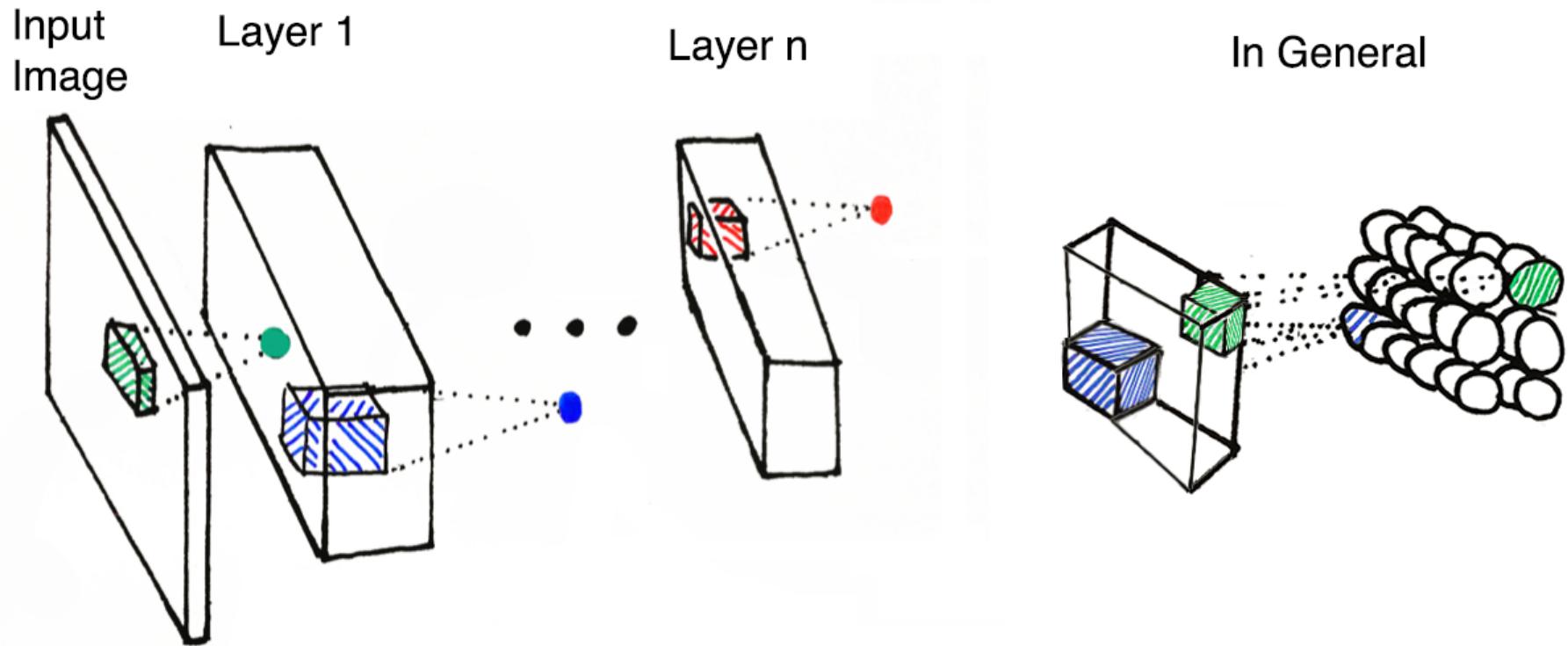


# Prior knowledge



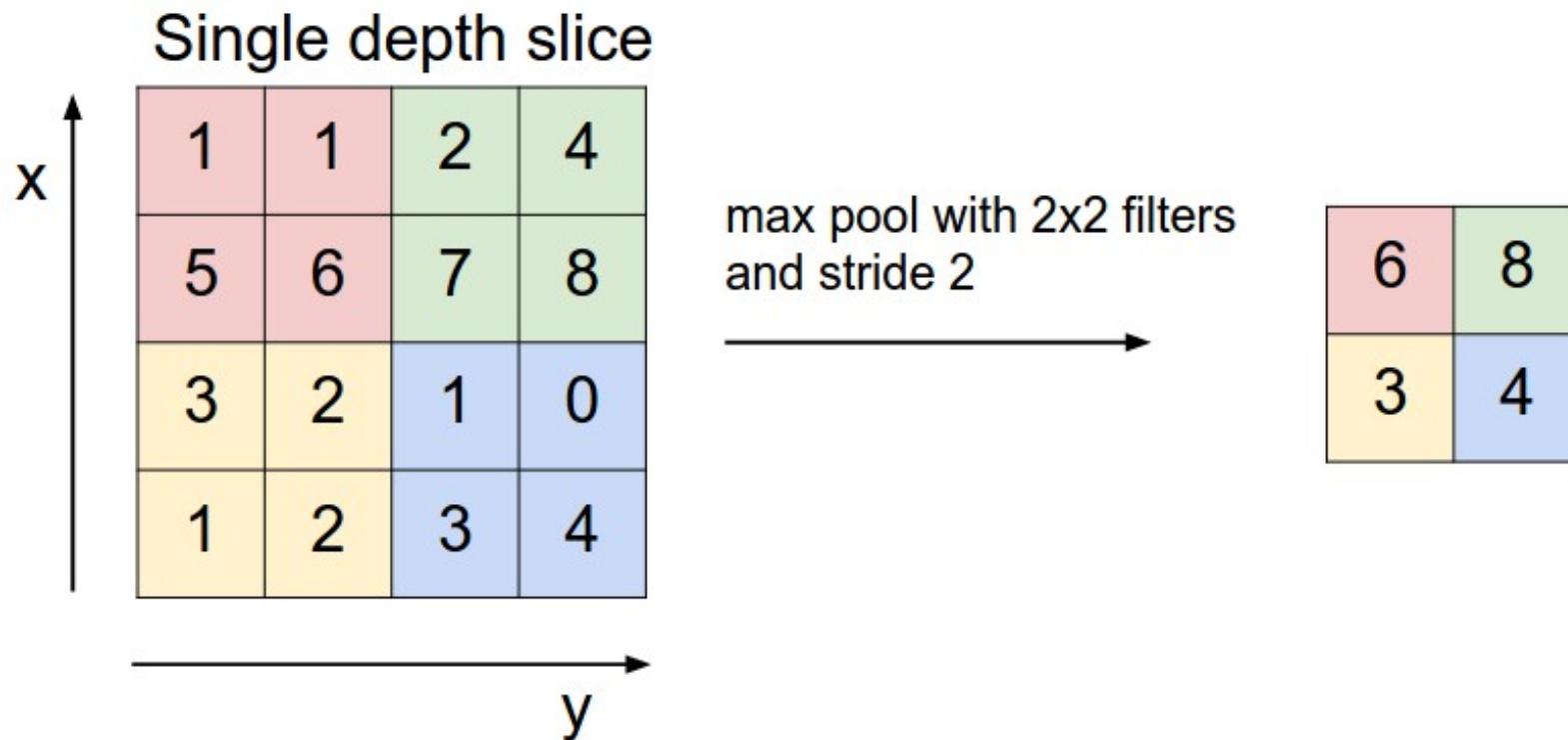
- **Local connectivity:** each hidden unit operates on a local image patch (3 instead of 7 connections per hidden unit)
- **Parameter sharing:** processing of each image patch is same (3 parameters instead of  $3 \cdot 5$ )
- **Intuition:** try to match a pattern in image

# Convolutional layers



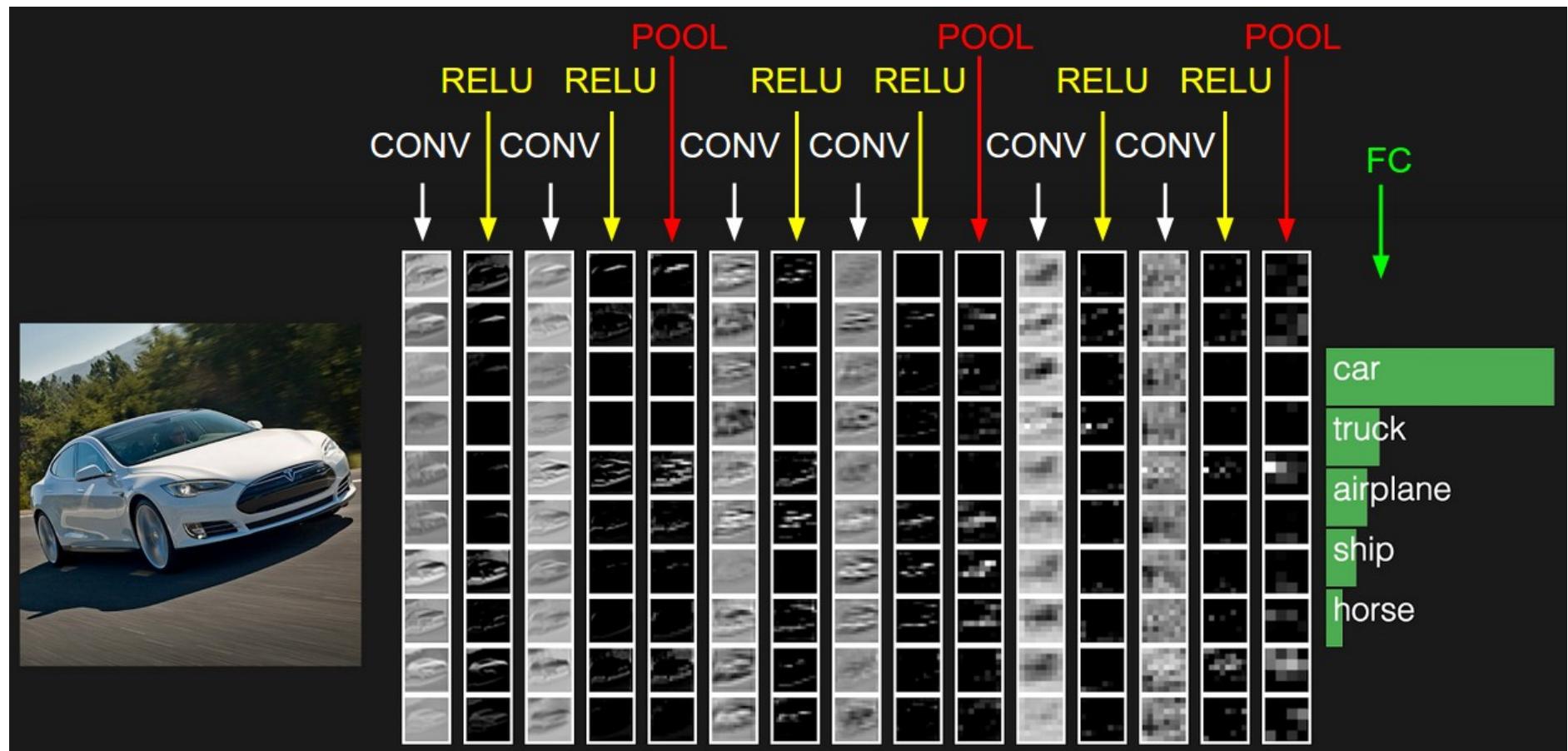
- Instead of vector to vector, we do volume to volume  
[Andrej Karpathy's demo]

# Max-pooling



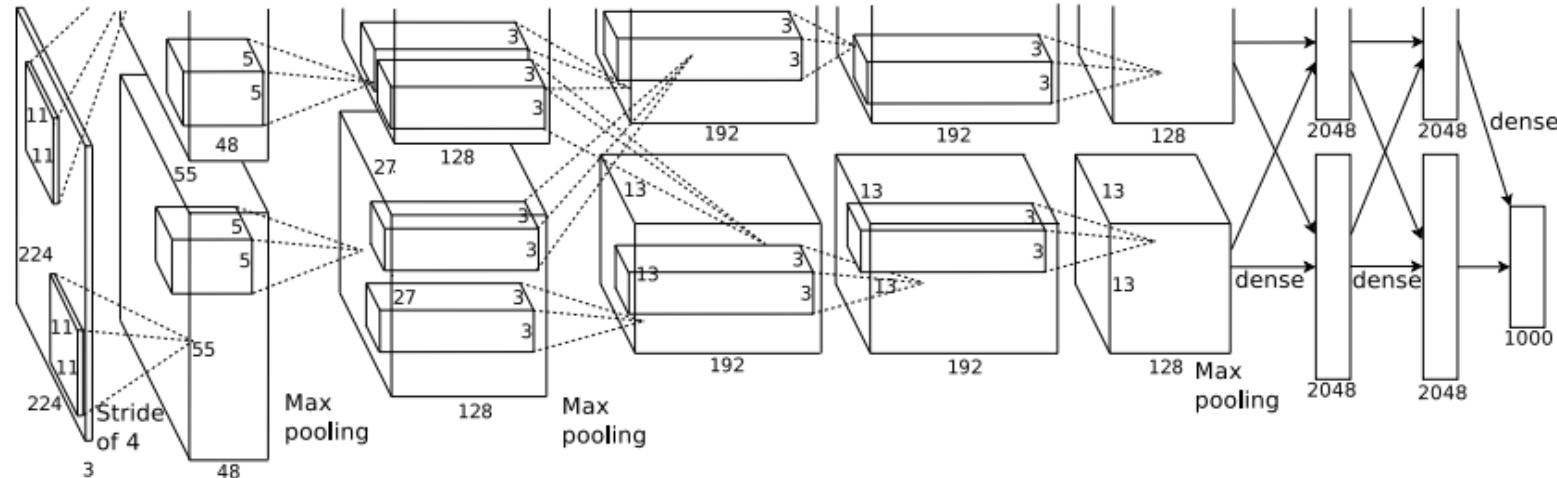
- Intuition: test if there exists a pattern in neighborhood
- Reduce computation, prevent overfitting

# Example of function evaluation



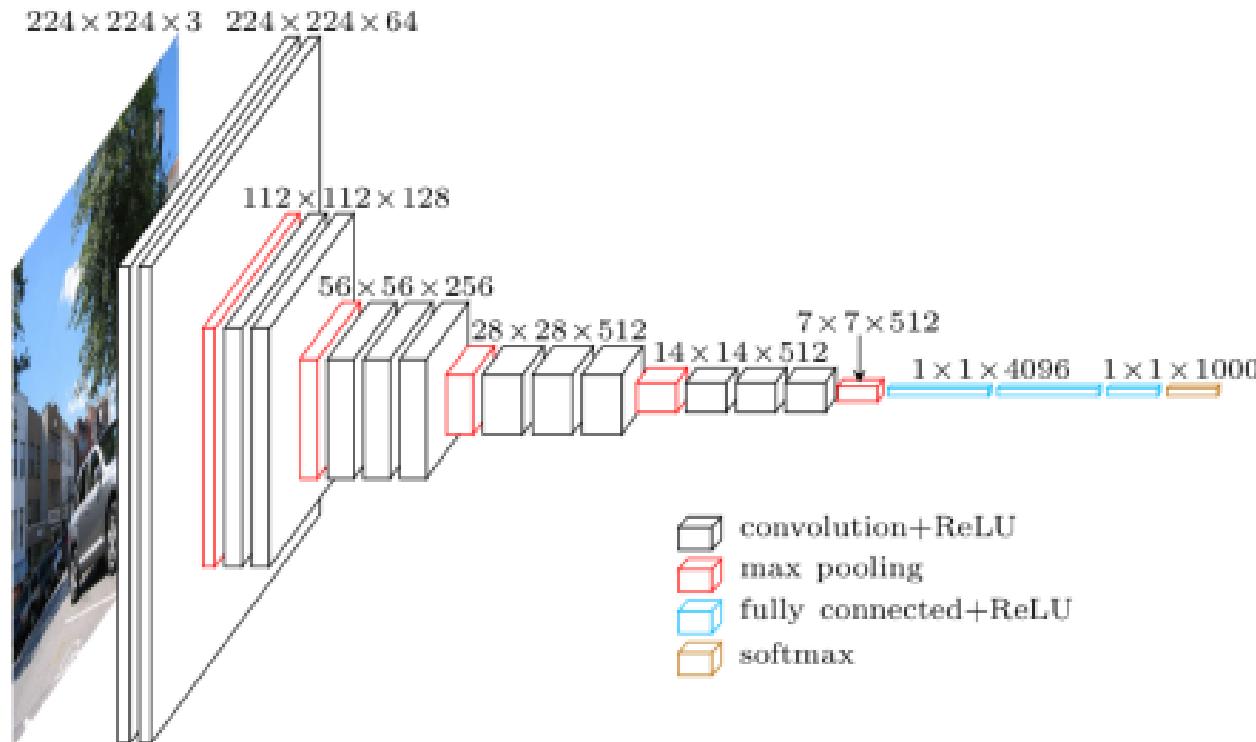
[Andrej Karpathy's demo]

# AlexNet



- **Non-linearity:** use ReLU ( $\max(z, 0)$ ) instead of logistic
- **Data augmentation:** translate, horizontal reflection, vary intensity, dropout (guard against overfitting)
- **Computation:** parallelize across two GPUs (6 days)
- **Results on ImageNet:** 16.4% error (next best was 25.8%)

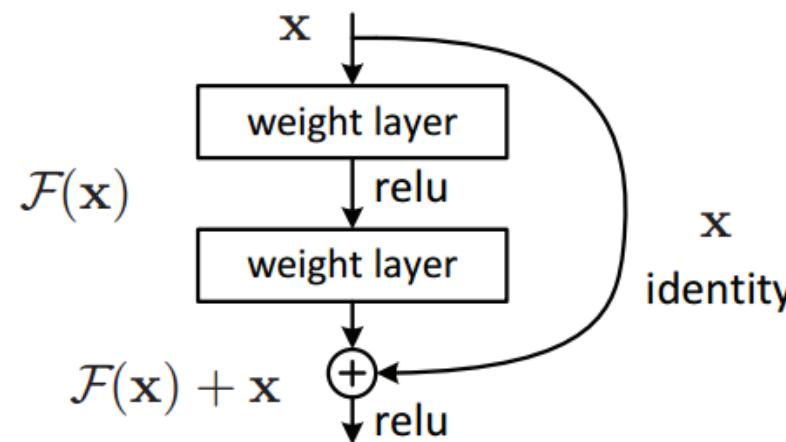
# VGGNet



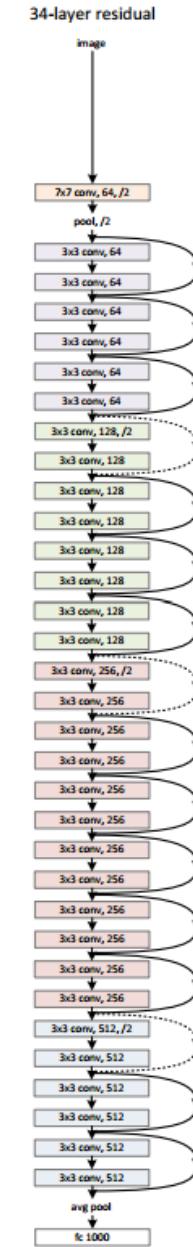
- **Architecture:** deeper but smaller filters; uniform
- **Computation:** 4 GPUs for 2-3 weeks
- **Results on ImageNet:** 7.3% error (AlexNet: 16.4%)

# Residual networks

$$x \mapsto \sigma(Wx) + x$$



- Key idea: make it easy to learn the identity (good inductive bias)
- Enables training 152 layer networks
- Results on ImageNet: 3.6% error





# Summary

- Key idea: locality of connections, capture spatial structure
- Filters have parameter sharing; most parameters in last fully connected layers
- Depth really matters
- Applications to text, Go, drug design, etc.



# Roadmap

Feedforward neural networks

Convolutional neural networks

**Recurrent neural networks**

Unsupervised learning

Final remarks

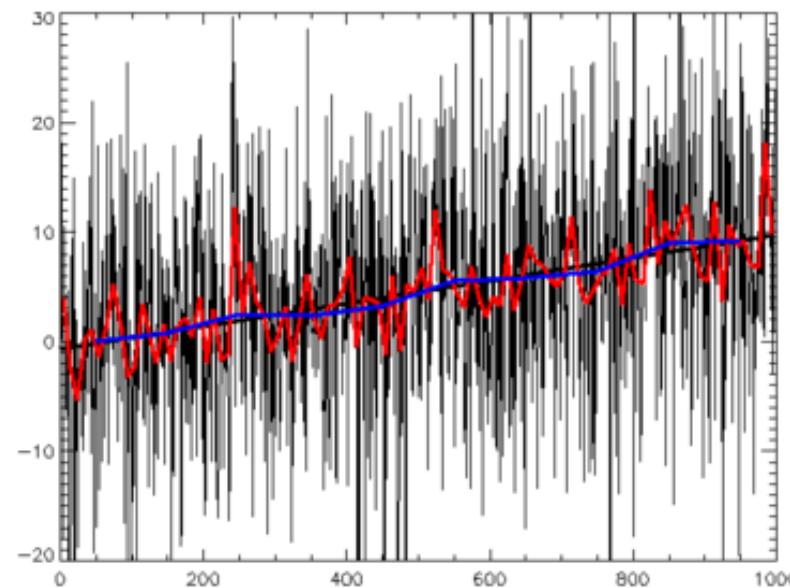
# Motivation: modeling sequences

Sentences:

$x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8 \quad x_9 \quad x_{10} \quad x_{11} \quad x_{12}$

*Paris Talks Set Stage for Action as Risks to the Climate Rise*

Time series:

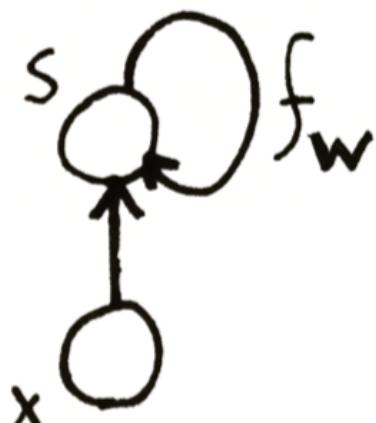


# Recurrent neural networks

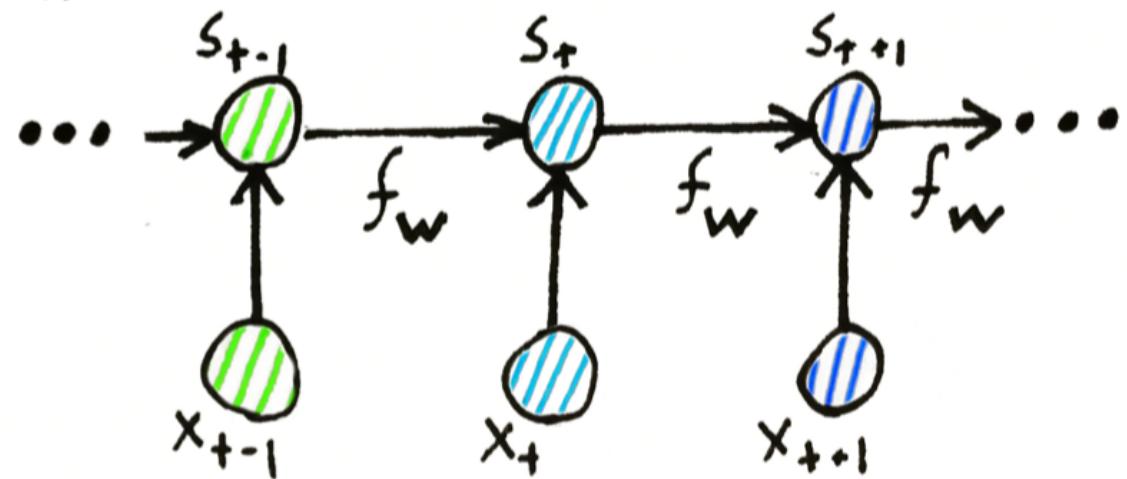
Formula

$$s_t = f_W(s_{t-1}, x_t)$$

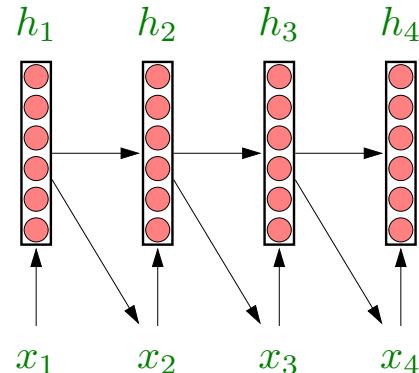
Network



Computation  
Graph



# Recurrent neural networks



$$h_1 = \text{Encode}(x_1)$$

$$x_2 \sim \text{Decode}(h_1)$$

$$h_2 = \text{Encode}(h_1, x_2)$$

$$x_3 \sim \text{Decode}(h_2)$$

$$h_3 = \text{Encode}(h_2, x_3)$$

$$x_4 \sim \text{Decode}(h_3)$$

$$h_4 = \text{Encode}(h_3, x_4)$$

Update context vector:

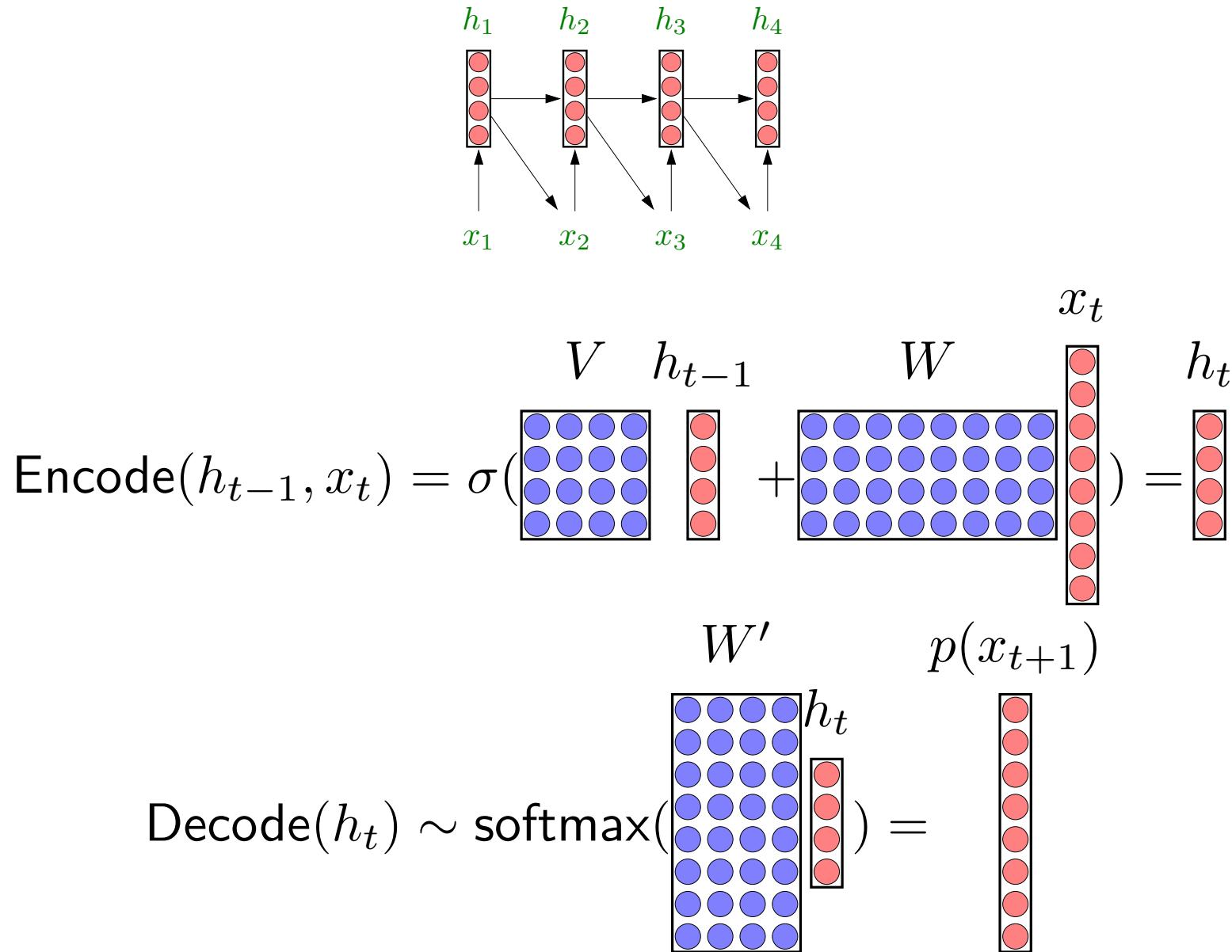
$$h_t = \text{Encode}(h_{t-1}, x_t)$$

Predict next character:

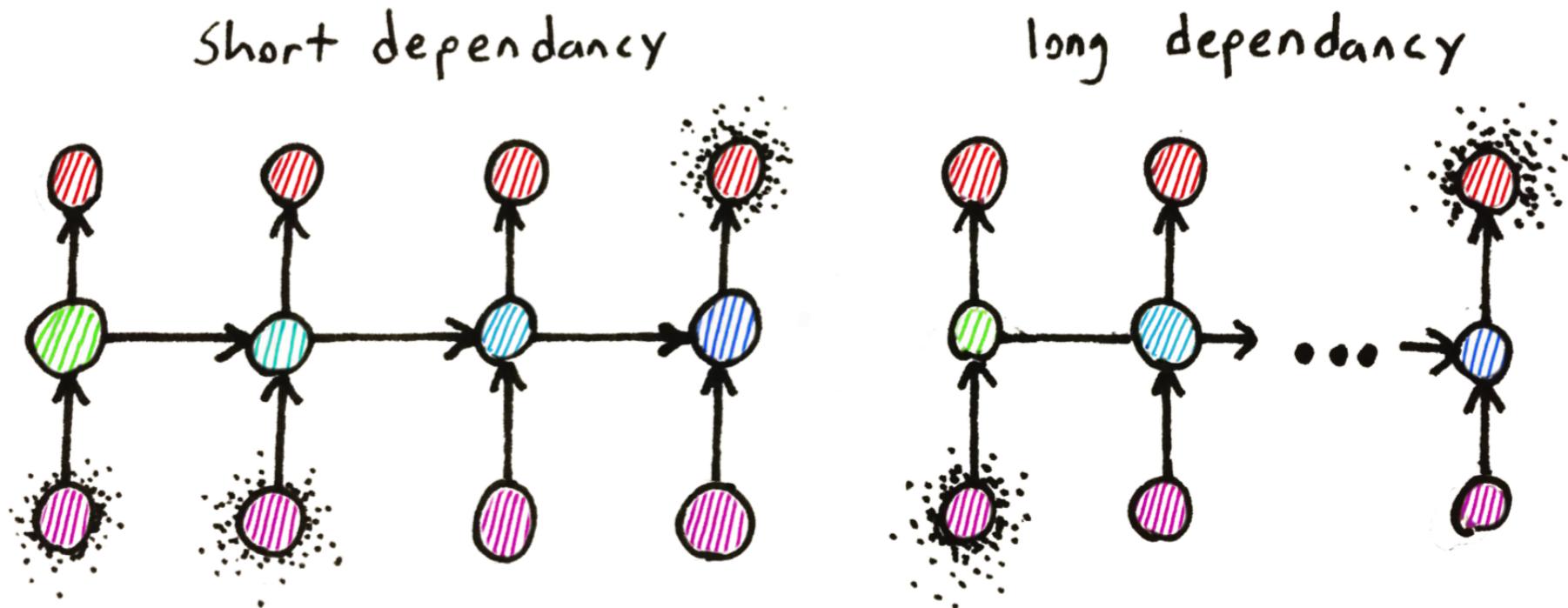
$$x_{t+1} = \text{Decode}(h_t)$$

context  $h_t$  compresses  $x_1, \dots, x_t$

# Simple recurrent network

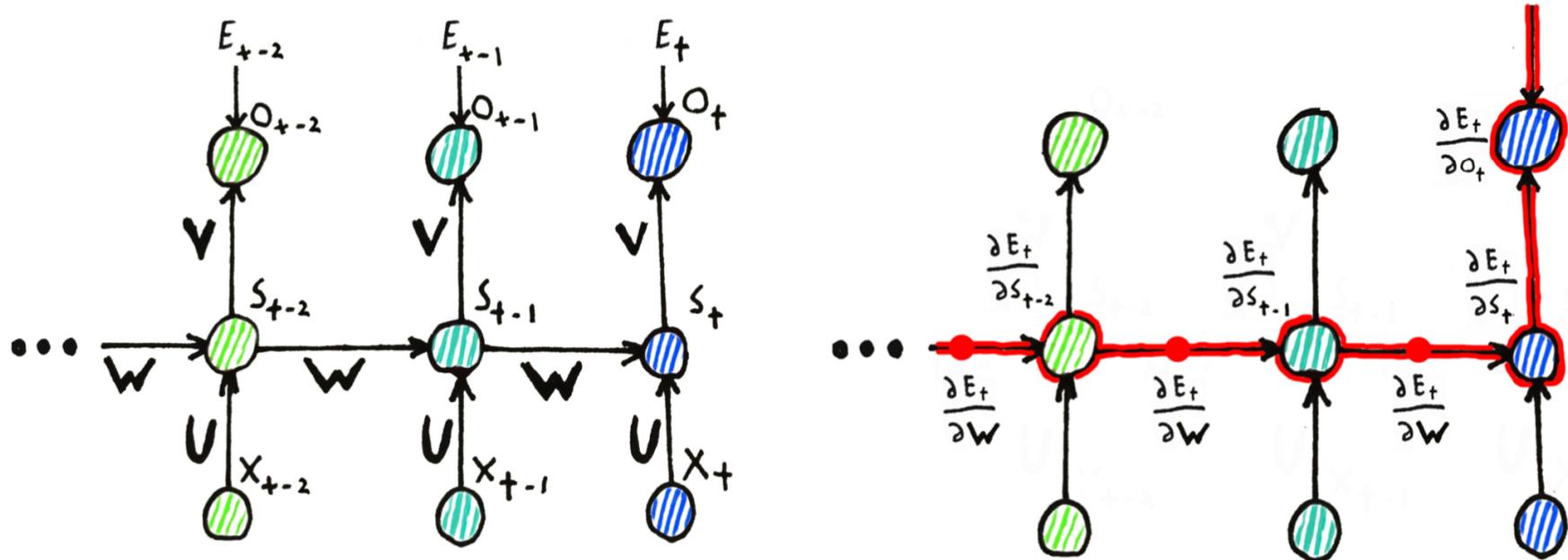


# Vanishing gradient problem



- RNNs can have long or short dependancies
- When there are long dependancies, gradients have trouble back-propagating through

# Vanishing gradient problem



Chain rule => multiplications

Can explode or shrink!

$$\frac{\partial E_t}{\partial \mathbf{W}} = \sum_{k=0}^t \frac{\partial E_t}{\partial \mathbf{o}_t} \frac{\partial \mathbf{o}_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial \mathbf{W}}$$

$$\frac{\partial s_t}{\partial s_k} = \prod_{j=k+1}^t \frac{\partial s_j}{\partial s_{j-1}}$$

# Long Short Term Memory (LSTM)

API:

$$(h_t, c_t) = \text{LSTM}(h_{t-1}, c_{t-1}, x_t)$$

Input gate:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + V_i c_{t-1} + b_i)$$

Forget gate (initialize with  $b_f$  large, so close to 1):

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + V_f c_{t-1} + b_f)$$

Cell: additive combination of RNN update with previous cell

$$c_t = i_t \odot \tanh(W_c x_t + U_c h_{t-1} + b_c) + f_t \odot c_{t-1}$$

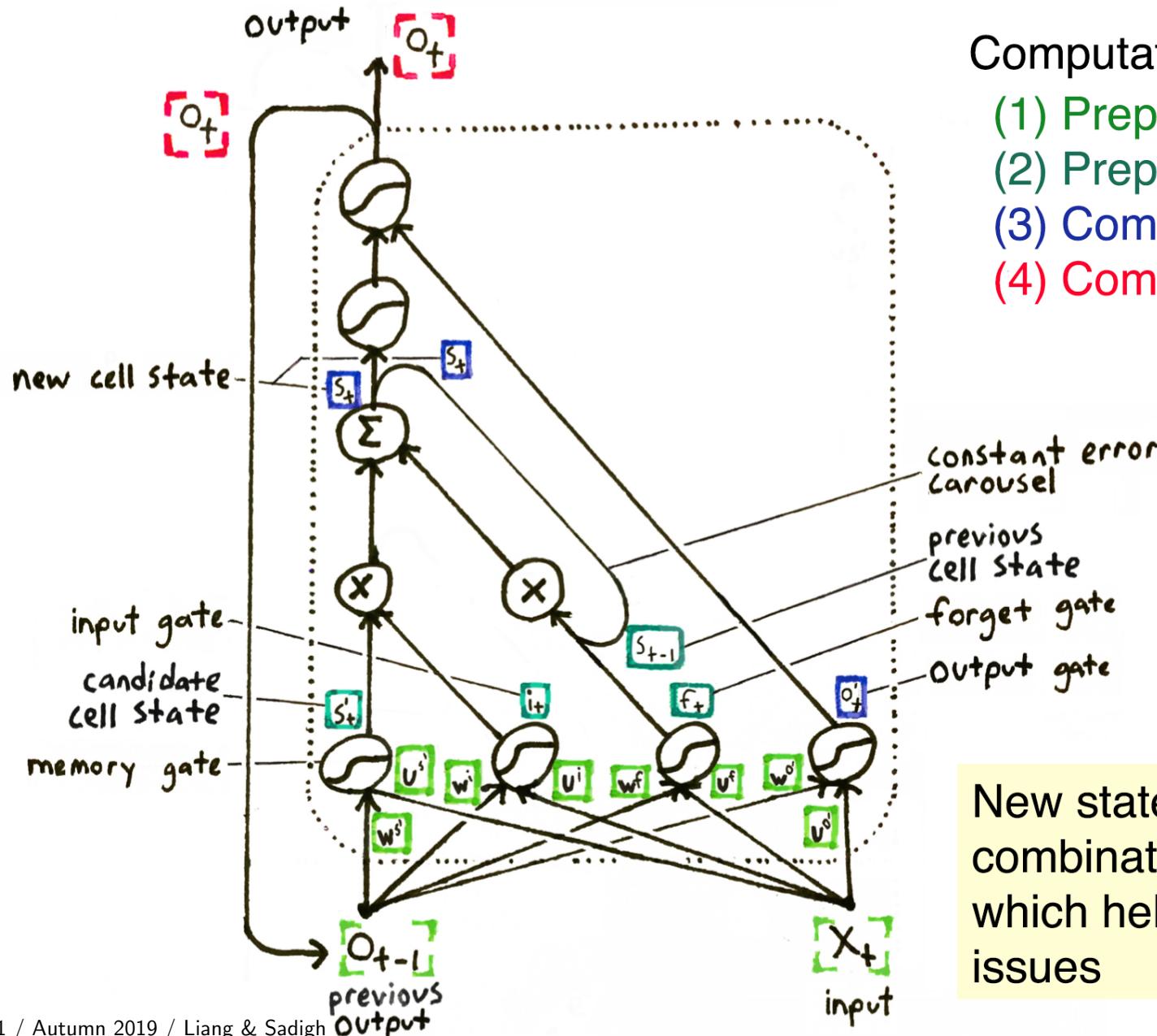
Output gate:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + V_o c_t + b_o)$$

Hidden state:

$$h_t = o_t \odot \tanh(c_t)$$

# Long Short Term Memory (LSTM)



Computation:

- (1) Prepare cell inputs
- (2) Prepare new state inputs
- (3) Compute new state
- (4) Compute output

New state is additive combination with old state, which helps avoid gradient issues

# Character-level language modeling

Sampled output:

*Naturalism and decision for the majority of Arab countries' capitalide was grounded by the Irish language by [[John Clair]], [[An Imperial Japanese Revolt]], associated with Guangzham's sovereignty. His generals were the powerful ruler of the Portugal in the [[Protestant Immineners]], which could be said to be directly in Cantonese Communication, which followed a ceremony and set inspired prison, training. The emperor travelled back to [[Antioch, Perth, October 25—21]] to note, the Kingdom of Costa Rica, unsuccessful fashioned the [[Thrales]], [[Cynth's Dajoard]], known in western [[Scotland]], near Italy to the conquest of India with the conflict.*

Cell sensitive to position in line:

```
The sole importance of the crossing of the Berezina lies in the fact
that it plainly and indubitably proved the fallacy of all the plans for
cutting off the enemy's retreat and the soundness of the only possible
line of action--the one Kutuzov and the general mass of the army
demanded--namely, simply to follow the enemy up. The French crowd fled
at a continually increasing speed and all its energy was directed to
reaching its goal. It fled like a wounded animal and it was impossible
to block its path. This was shown not so much by the arrangements it
made for crossing as by what took place at the bridges. When the bridges
broke down, unarmed soldiers, people from Moscow and women with children
who were with the French transport, all--carried on by vis inertiae--
pressed forward into boats and into the ice-covered water and did not,
surrender.
```

Cell that turns on inside quotes:

```
"You mean to imply that I have nothing to eat out of.... On the
contrary, I can supply you with everything even if you want to give
dinner parties," warmly replied Chichagov, who tried by every word he
spoke to prove his own rectitude and therefore imagined Kutuzov to be
animated by the same desire.
```

```
Kutuzov, shrugging his shoulders, replied with his subtle penetrating
smile: "I meant merely to say what I said."
```

Cell that robustly activates inside if statements:

```
static int __dequeue_signal(struct sigpending *pending, sigset_t *mask,
    siginfo_t *info)
{
    int sig = next_signal(pending, mask);
    if (sig) {
        if (current->notifier) {
            if (sigismember(current->notifier_mask, sig)) {
                if (! (current->notifier)(current->notifier_data)) {
                    clear_thread_flag(TIF_SIGPENDING);
                    return 0;
                }
            }
        }
        collect_signal(sig, pending, info);
    }
    return sig;
}
```

A large portion of cells are not easily interpretable. Here is a typical example:

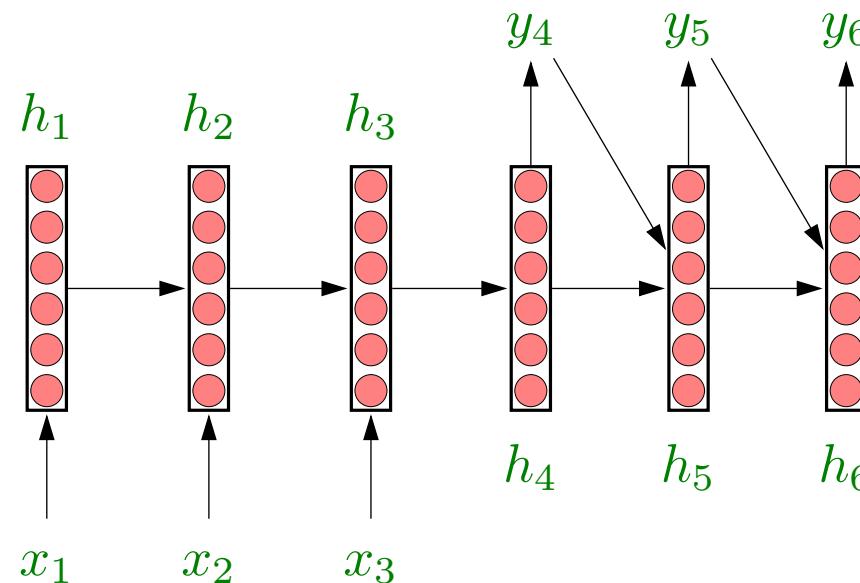
```
/* Unpack a filter field's string representation from user-space
 * buffer. */
char *audit_unpack_string(void **bufp, size_t *remain, size_t len)
{
    char *str;
    if (! *bufp || (len == 0) || (len > *remain))
        return ERR_PTR(-EINVAL);
    /* Of the currently implemented string fields, PATH_MAX
     * defines the longest valid length.
    */
```

# Sequence-to-sequence model

Motivation: machine translation

$x$ : *Je crains l'homme de un seul livre.*

$y$ : *Fear the man of one book.*



Read in a sentence first, output according to RNN:

$$h_t = \text{Encode}(h_{t-1}, x_t \text{ or } y_{t-1}), \quad y_t = \text{Decode}(h_t)$$

# Attention-based models

Motivation: long sentences — compress to finite dimensional vector?

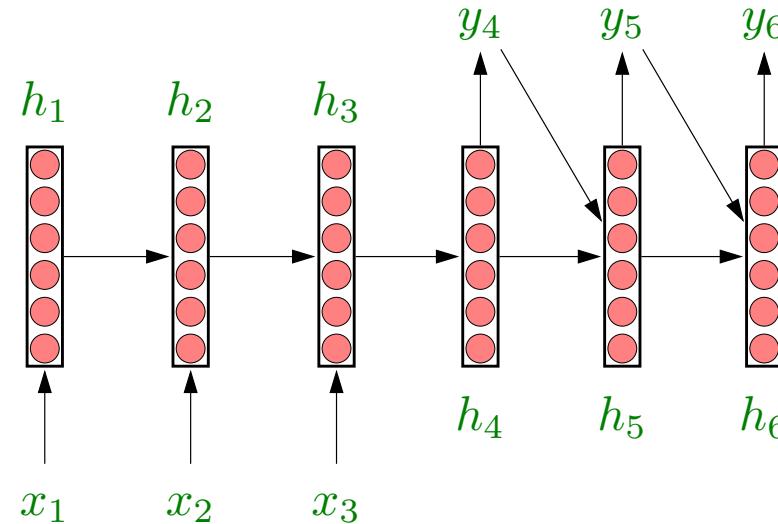
*Eine Folge von Ereignissen bewirkte, dass aus Beethovens Studienreise nach Wien ein dauerhafter und endgültiger Aufenthalt wurde. Kurz nach Beethovens Ankunft, am 18. Dezember 1792, starb sein Vater. 1794 besetzten französische Truppen das Rheinland, und der kurfürstliche Hof musste fliehen.*



**Key idea: attention**

Learn to look back at your notes.

# Attention-based models



Distribution over input positions:

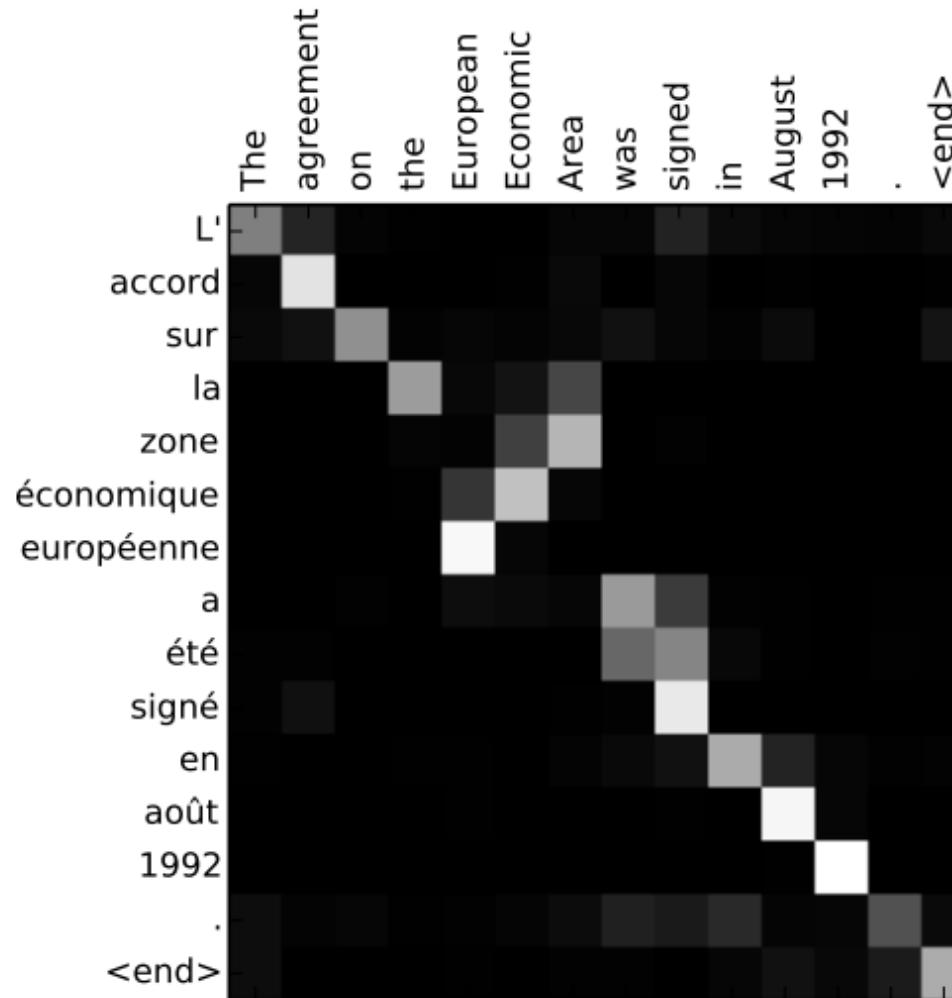
$$\alpha_t = \text{softmax}([\text{Attend}(h_1, h_{t-1}), \dots, \text{Attend}(h_L, h_{t-1})])$$

Generate with attended input:

$$h_t = \text{Encode}(h_{t-1}, y_{t-1}, \sum_{j=1}^L \alpha_t h_j)$$

Transformer models: attention only – no RNN!

# Machine translation



# Image captioning



A woman is throwing a frisbee in a park.

A dog is standing on a hardwood floor.



A little girl sitting on a bed with a teddy bear.



A group of people sitting on a boat in the water.



# Summary

- Recurrent neural networks: model sequences (non-linear version of Kalman filter or HMM)
- Logic intuition: learning a program with a for loop (reduce)
- LSTMs mitigate the vanishing gradient problem
- Attention-based models: when only part of input is relevant at a time
- Newer models with "external memory": memory networks, neural Turing machines



# Roadmap

Feedforward neural networks

Convolutional neural networks

Recurrent neural networks

**Unsupervised learning**

Final remarks

# Motivation

- Deep neural networks require lot of data
- Sometimes not very much labeled data, but plenty of unlabeled data (text, images, videos)
- Humans rarely get direct supervision; can learn from raw sensory information?

# Autoencoders

Analogy:

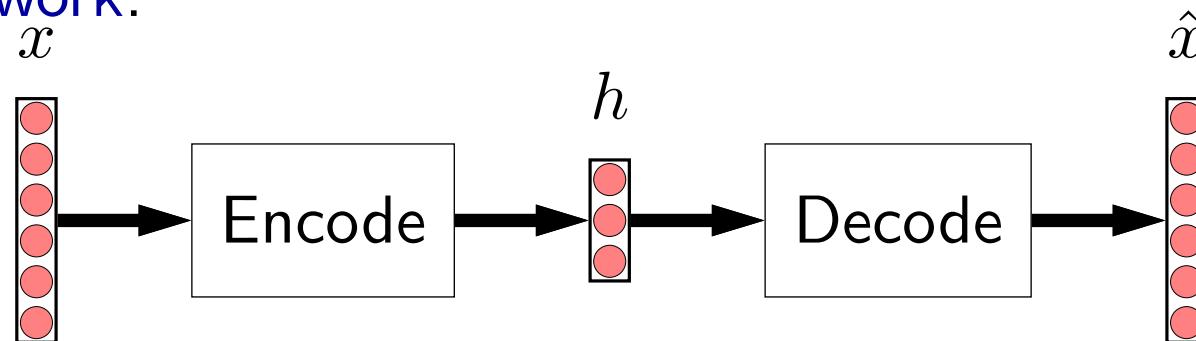
A A A A B B B B B → 4 A's, 5 B's → A A A A B B B B B



**Key idea: autoencoders**

If we can compress a data point and still reconstruct it, then we have learned something generally useful.

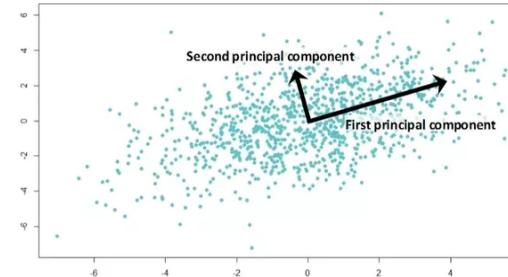
General framework:



$$\text{minimize } \|x - \hat{x}\|^2$$

# Principal component analysis

Input: points  $x_1, \dots, x_n$



$$\text{Encode}(x) = U^\top x$$

Decode( $h$ ) =  $U h$

(assume  $x_i$ 's are mean zero and  $U$  is orthogonal)

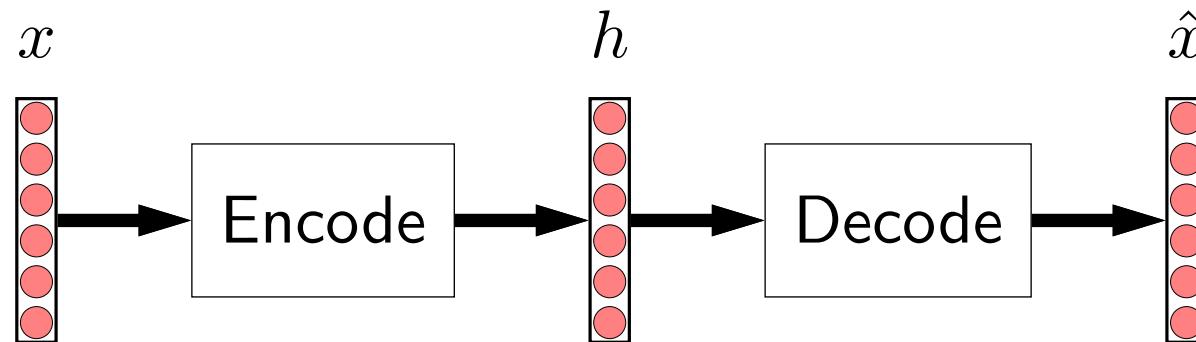
The diagram illustrates the PCA process. On the left, the input vector  $x$  is shown as a vertical column of red circles. It is multiplied by the transpose of the orthogonal matrix  $U$  to produce the encoded representation  $U^\top x$ , which is shown as a horizontal row of blue circles. On the right, the encoded representation  $h$  is multiplied by the matrix  $U$  to produce the decoded reconstruction  $U h$ , which is shown as a vertical column of red circles.

PCA objective:

$$\text{minimize} \sum_{i=1}^n \|x_i - \text{Decode}(\text{Encode}(x_i))\|^2$$

# Autoencoders

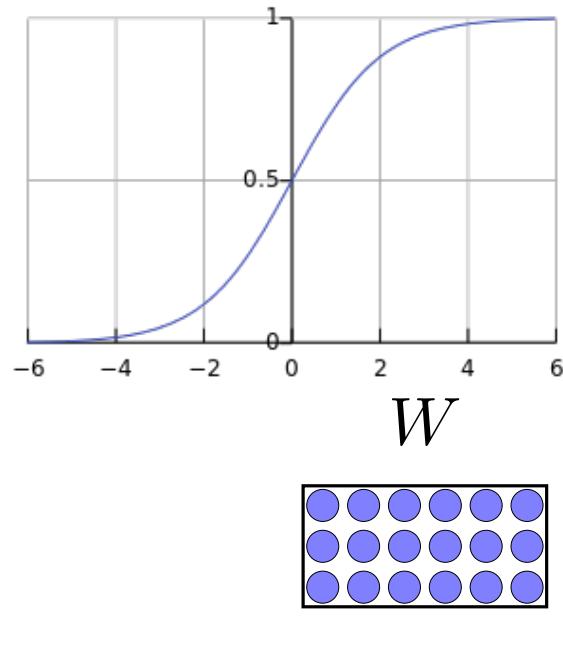
Increase dimensionality of hidden dimension:



- **Problem:** learning nothing — just set Encode, Decode to identity function!
- Need to control complexity of Encode and Decode somehow...

# Non-linear autoencoders

Non-linear transformation (e.g., logistic function):



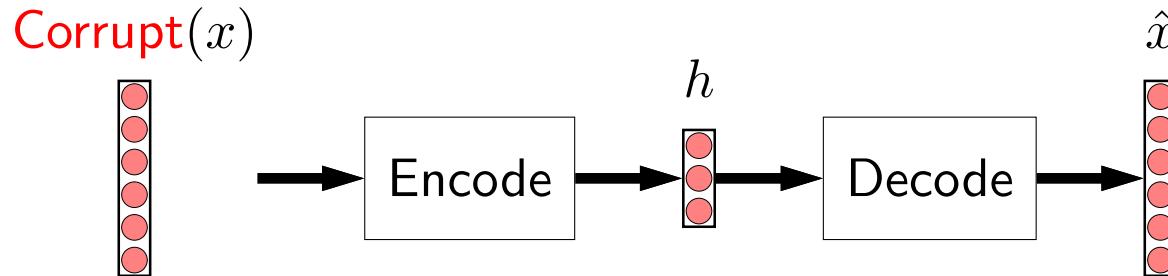
$$\text{Encode}(x) = \sigma(Wx + b)$$
$$\text{Decode}(h) = \sigma(W'h + b')$$

Loss function:

$$\text{minimize } \|x - \text{Decode}(\text{Encode}(x))\|^2$$

**Key:** non-linearity makes life harder, prevents degeneracy

# Denoising autoencoders



Types of noise:

- Blankout:  $\text{Corrupt}([1, 2, 3, 4]) = [0, 2, 3, 0]$
- Gaussian:  $\text{Corrupt}([1, 2, 3, 4]) = [1.1, 1.9, 3.3, 4.2]$

Objective:

$$\text{minimize } \|x - \text{Decode}(\text{Encode}(\text{Corrupt}(x)))\|^2$$

Algorithm: pick example, add fresh noise, SGD update

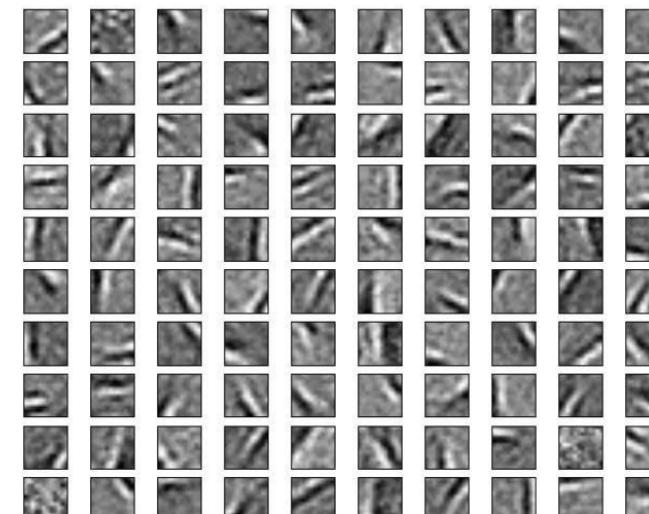
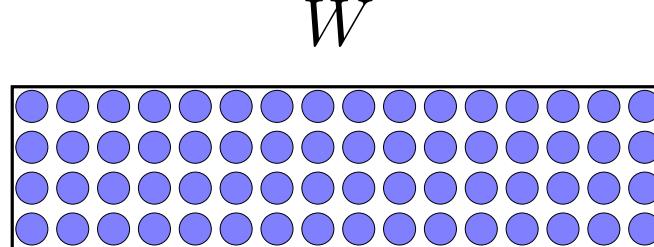
Key: noise makes life harder, prevents degeneracy

# Denoising autoencoders

MNIST: 60,000 images of digits (784 dimensions)

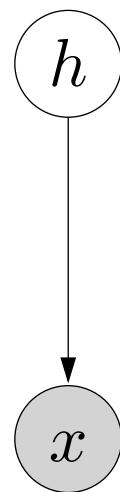


200 learned filters (rows of  $W$ ):



# Variational autoencoders

Motivation: learn a latent-variable model

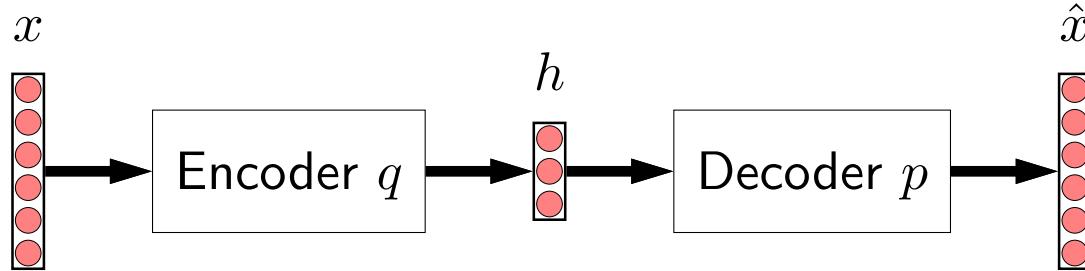


$$p(h, x) = p(h)p(x | h)$$

E-step in EM: computing  $p(h | x)$  is intractable

Solution: approximate using a neural network  $q(h | x)$

# Variational autoencoders



Objective: maximize

$$\log p(x) \geq \mathbb{E}_{q(h|x)}[\log p(x | h)] - \text{KL}(q(h | x) || p(h))$$

Algorithm:

- Sample  $h$  from encoder  $q$ , gradient update on  $q$  and  $p$
- Reparametrization trick [Kingma/Welling, 2014]

# Reading comprehension (SQuAD)

In meteorology, precipitation is any product of the condensation of atmospheric water vapor that falls under **gravity**. The main forms of precipitation include drizzle, rain, sleet, snow, **graupel** and hail... Precipitation forms as smaller droplets coalesce via collision with other rain drops or ice crystals **within a cloud**. Short, intense periods of rain in scattered locations are called "showers".

What causes precipitation to fall?

**gravity**

100K examples

What is another main form of precipitation besides drizzle, rain, snow, sleet and hail?

**graupel**

Where do water droplets collide with ice crystals to form precipitation?

**within a cloud**

# Raw text

Stanford University (officially Leland Stanford Junior University,<sup>[10]</sup> colloquially "the Farm") is a private research university in Stanford, California. Stanford is known for its academic strength, wealth, proximity to Silicon Valley, and ranking as one of the world's top universities.<sup>[11][12][13][14][15]</sup>

The university was founded in 1885 by Leland and Jane Stanford in memory of their only child, Leland Stanford Jr., who had died of typhoid fever at age 15 the previous year. Stanford was a U.S. Senator and former Governor of California who made his fortune as a railroad tycoon. The school admitted its first students on October 1, 1891,<sup>[2][3]</sup> as a coeducational and non-denominational institution.

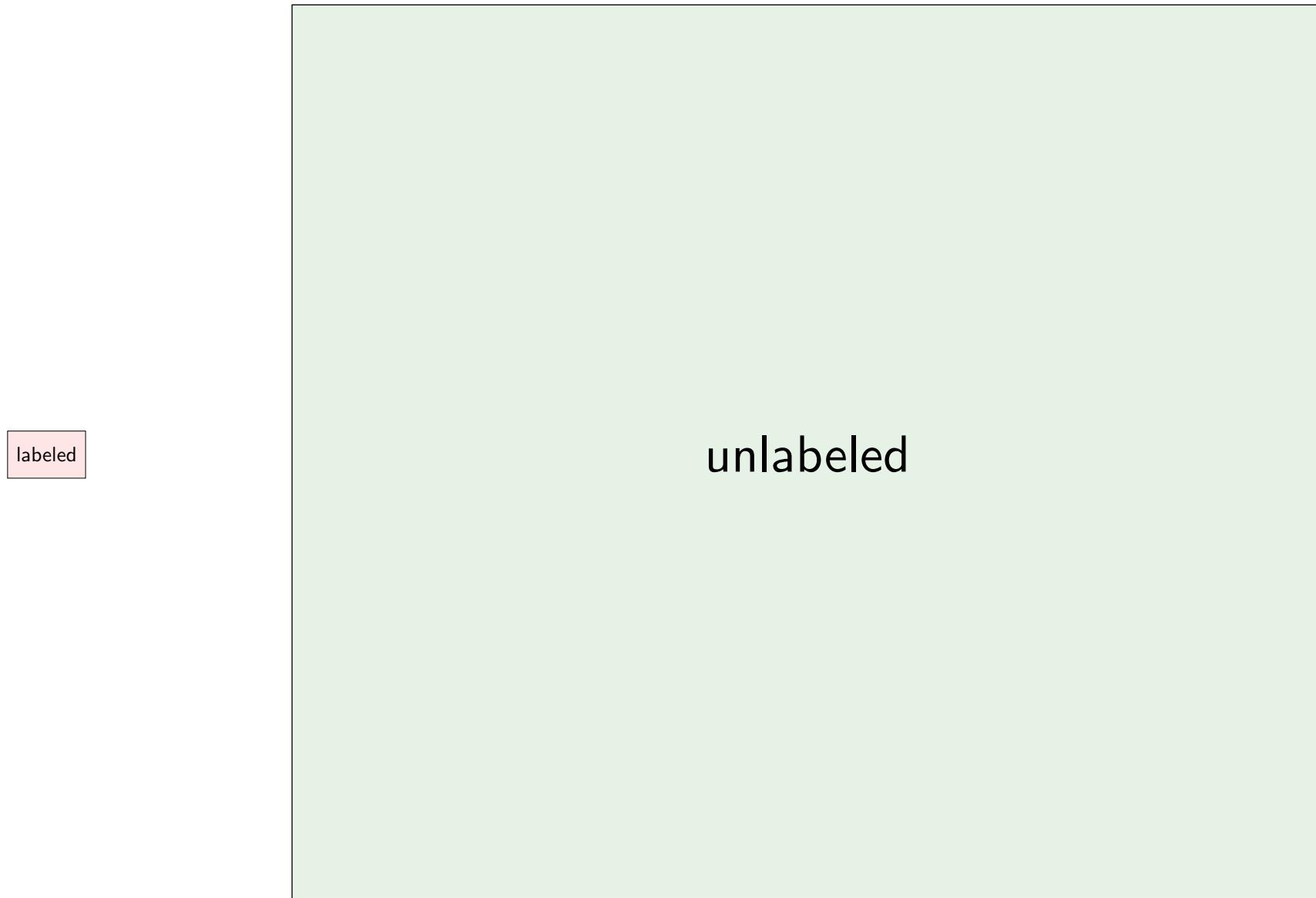
Stanford University struggled financially after the death of Leland Stanford in 1893 and again after much of the campus was damaged by the 1906 San Francisco earthquake.<sup>[16]</sup> Following World War II, Provost Frederick Terman supported faculty and graduates' entrepreneurialism to build self-sufficient local industry in what would later be known as Silicon Valley.<sup>[17]</sup> The university is also one of the top fundraising institutions in the country, becoming the first school to raise more than a billion dollars in a year.<sup>[18]</sup>

The university is organized around three traditional schools consisting of 40 academic departments at the undergraduate and graduate level and four professional schools that focus on graduate programs in Law, Medicine, Education and Business. Stanford's undergraduate program is one of the top three most selective in the United States by acceptance rate.<sup>[19][20][21][22][23]</sup> Students compete in 36 varsity sports, and the university is one of two private institutions in the Division I FBS Pac-12 Conference. It has gained 117 NCAA team championships,<sup>[24]</sup> the most for a university. Stanford athletes have won 512 individual championships,<sup>[25]</sup> and Stanford has won the NACDA Directors' Cup for 23 consecutive years, beginning in 1994–1995.<sup>[26]</sup> In addition, Stanford students and alumni have won 270 Olympic medals including 139 gold medals.<sup>[27]</sup>

...

3.3 billion words

# Unsupervised pre-training





# BERT

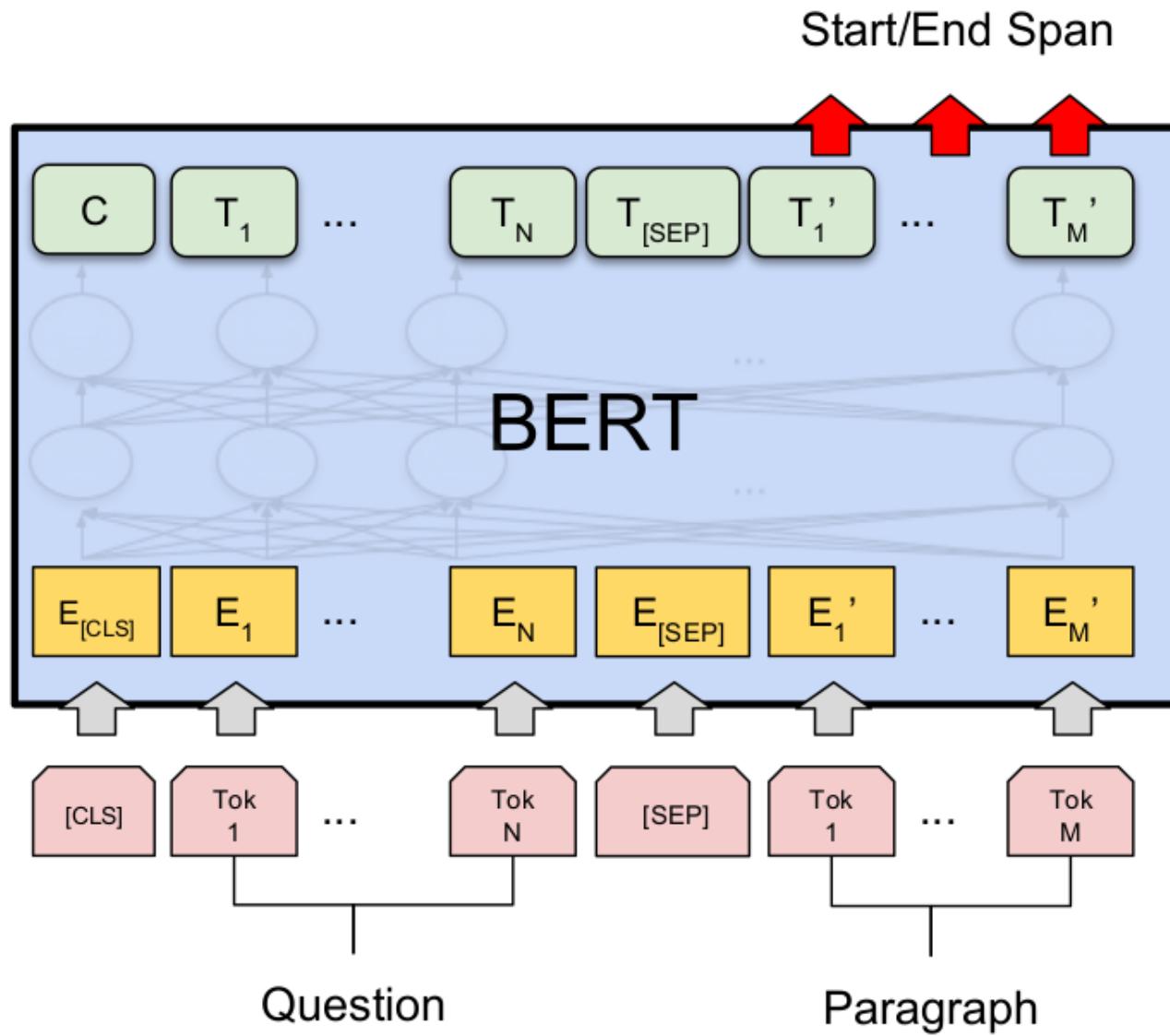
Paris Talks \_\_\_ Stage for \_\_\_\_\_ as Risks to \_\_\_ Climate Rise



Paris Talks Set Stage for Action as Risks to the Climate Rise

- Tasks: fill in words, predict whether is next sentence
- Trained on 3.3B words, 4 days on 64 TPUs

# BERT



Rank	Model	EM	F1
	Human Performance <i>Stanford University</i> (Rajpurkar et al. '16)	82.304	91.221
1 Oct 05, 2018	BERT (ensemble) Google A.I.	87.433	93.160
2 Oct 05, 2018	BERT (single model) Google A.I.	85.083	91.835
2 Sep 09, 2018	nlnet (ensemble) Microsoft Research Asia	85.356	91.202
2 Sep 26, 2018	nlnet (ensemble) Microsoft Research Asia	85.954	91.677
3 Jul 11, 2018	QANet (ensemble) Google Brain & CMU	84.454	90.490
4 Jul 08, 2018	r-net (ensemble) Microsoft Research Asia	84.003	90.147
5 Mar 19, 2018	QANet (ensemble) Google Brain & CMU	83.877	89.737
5 Sep 09, 2018	nlnet (single model) Microsoft Research Asia	83.468	90.133
5 Jun 20, 2018	MARS (ensemble) YUANFUDAO research NLP	83.982	89.796
6 Sep 01, 2018	MARS (single model) YUANFUDAO research NLP	83.185	89.547



# Unsupervised learning

- Principle: make up prediction tasks (e.g.,  $x$  given  $x$  or context)
- Hard task → pressure to learn something
- Loss minimization using SGD
- Discriminatively fine tune: initialize feedforward neural network and backpropagate to optimize task accuracy
- How far can one push this?



# Roadmap

Feedforward neural networks

Convolutional neural networks

Recurrent neural networks

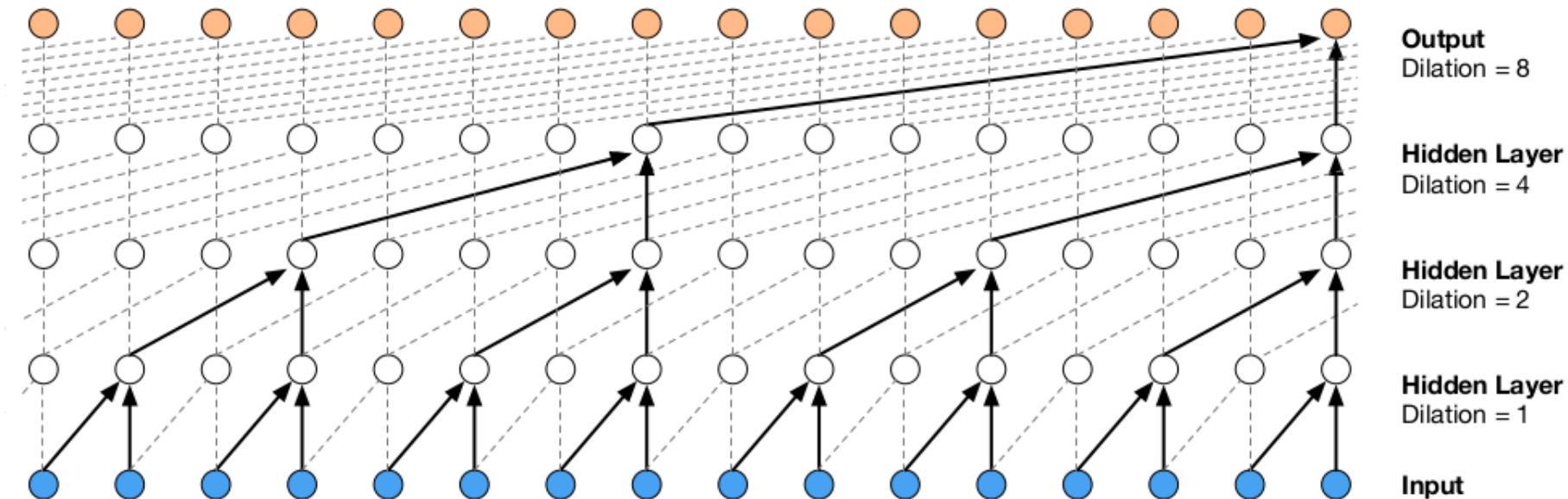
Unsupervised learning

**Final remarks**

# WaveNet for audio generation

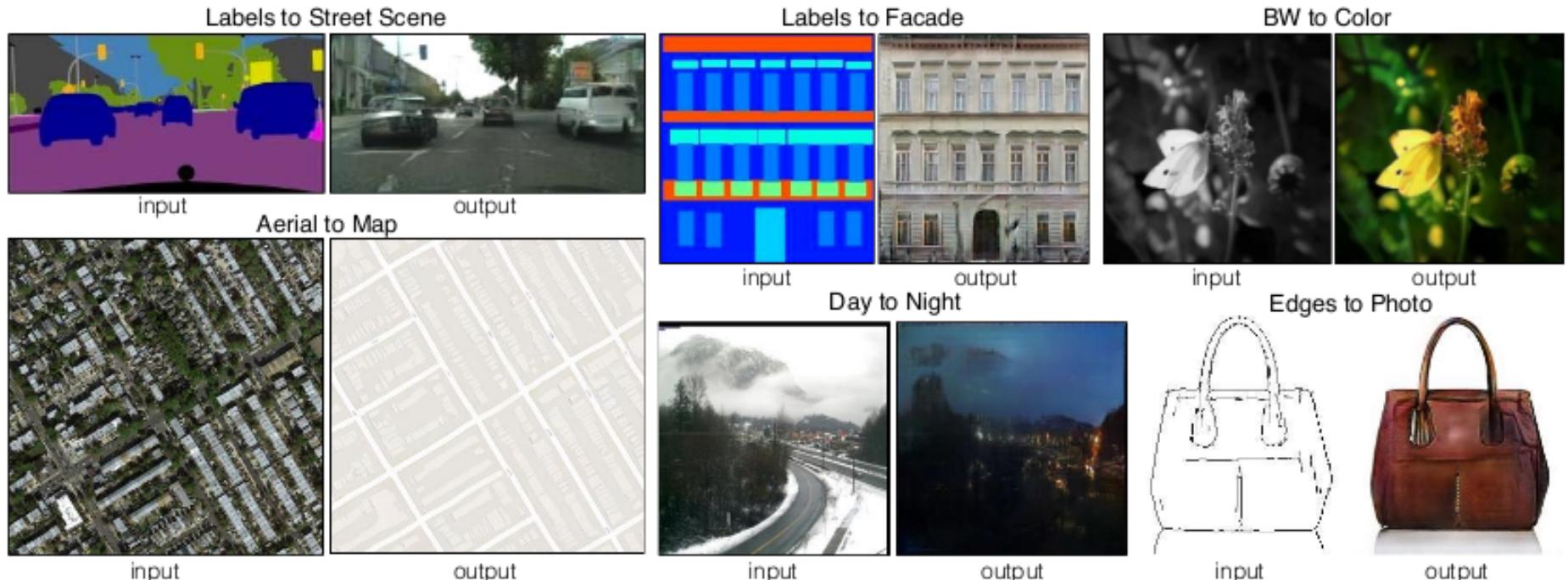


- Work with **raw** audio (16K observations / second)



- Key idea: **dilated convolutions** captures multiple scales of resolution, not recurrent

# Conditional adversarial networks



Key idea: game between

- **Generator:** generates fake images
- **Discriminator:** distinguishes between fake/real images

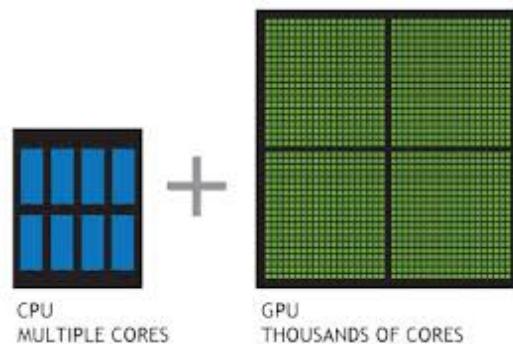
# Getting things to work

Better optimization algorithms: SGD, SGD+momentum, AdaGrad, AdaDelta, momentum, Nesterov, Adam

Tricks: initialization, gradient clipping, batch normalization, dropout

More hyperparameter tuning: step sizes, architectures

Better hardware: GPUs, TPUs



...wait for a long time...

# Theory: why does it work?

Two questions:

- Approximation: why are neural networks good hypothesis classes?
- Optimization: why can SGD optimize a high-dimensional non-convex problem?

Partial answers:

- 1-layer neural networks can approximate any continuous function on compact set [Cybenko, 1989; Barron, 1993]
- Generate random features works too [Rahimi/Recht, 2009; Andoni et. al, 2014]
- Use statistical physics to analyze loss surfaces [Choromanska et al., 2014]



# Summary

## Phenomena

Fixed vectors

Spatial structure

Sequence

Sequence-to-sequence

Unsupervised

## Ideas

Feedforward NNs

convolutional NNs

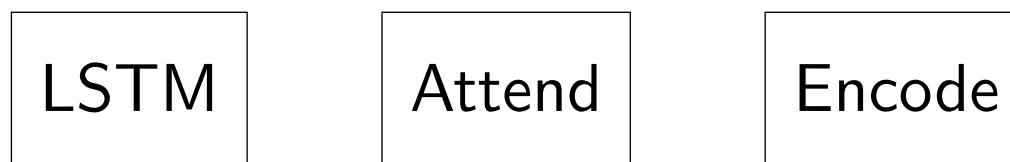
recurrent NNs  
LSTMs

encoder-decoder  
attention-based models

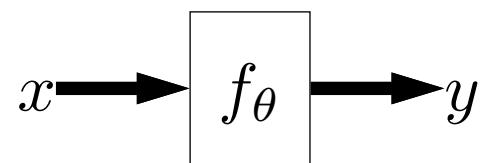
autoencoders  
variational autoencoders  
any auxiliary task

# Outlook

Extensibility: able to compose modules



Learning programs: think about analogy with a computer



# Semantic Parsing

---

CS221 Section 9: 11/30/2018

By: Ajay Sohmshetty and David Golub

# Motivation: From Natural Language to Logic

*Alice likes hiking.* → Likes(alice, hiking)

*Alice likes geometry.* → Likes(alice, geometry)

*Bob likes hiking.* → Likes(bob, hiking)

*Bob likes geometry.* → Likes(bob, geometry)

Lots of regularities — can we convert language to logic automatically?

# Lambda calculus

- Improves upon first-order logic for increased expressiveness and compositionality.
- Differences between the quantifiers using variable  $x$ :
  - $\exists x P(x)$  means “There exists an  $x$  such that  $P(x)$  is true.”
  - $\forall x P(x)$  means: “For all  $x$ ,  $P(x)$  is true.”
  - $\lambda x P(x)$  means: “**Given**  $x$ ,  $P(x)$  is true.”
- Unlike  $\exists$  and  $\forall$ , which are symbols for the words “exists” and “all”,  $\lambda$  is a **function**
  - The  $\lambda$  operators allow us to abstract over  $x$ . They have no meaning on their own, but allow us to bind the **variable**  $x$  to a given **argument** value.

# Analogy: Python lambda functions

- Think of lambda functions in Python
- Example:
  - Let  $P$  denote “even”. Then  $\lambda x P(x)$  means “Given  $x$ ,  $x$  is even.”
  - This statement could be true or false.
  - Let the function  $f$  denote  $\lambda x P(x)$ . Then we can try this in the Python shell:

```
[>>> f = lambda x : x % 2 == 0
[>>> f(2)
True
[>>> f(3)
False
```

# Lambda calculus example 1

- “Given  $x$ ,  $x$  is a student and  $x$  likes hiking.”

- $x = \text{alice}$

→ “alice is student and  
alice likes hiking.”

Function:

$$\lambda x \text{ Student}(x) \wedge \text{Likes}(x, \text{hiking})$$

Argument:

alice

Function application:

$$(\lambda x \text{ Student}(x) \wedge \text{Likes}(x, \text{hiking}))(alice)$$

$$\text{Student(alice)} \wedge \text{Likes(alice, hiking)}$$

# Lambda calculus example 2

- “Given y and x, x likes y.”
- $y = \text{hiking}$

→ “Given x, x likes hiking.”

Function:

$$\lambda y \lambda x \text{Likes}(x, y)$$

Argument:

**hiking**

Function application:

$$(\lambda y \lambda x \text{Likes}(x, y))(\text{hiking}) =$$

$$\lambda x \text{Likes}(x, \text{hiking})$$

# Lambda calculus example 3

- “Given f and x, not  $f(x)$ .”
- $f =$  “Given y, y likes hiking.”
- $f(x) =$  “x likes hiking”

→ “Given x, x does not like hiking”

Function:

$$\lambda f \lambda x \neg f(x)$$

Argument:

$$\lambda y \text{Likes}(y, \text{hiking})$$

Function application:

$$(\lambda f \lambda x \neg f(x))(\lambda y \text{Likes}(y, \text{hiking})) =$$

$$\lambda x \neg (\lambda y \text{Likes}(y, \text{hiking}))(x) =$$

$$\lambda x \neg \text{Likes}(x, \text{hiking})$$

# Principle of Compositionality



## Key idea: principle of compositionality

The semantics of a sentence is combination of meanings of its parts.

Sentence:

*Alice likes hiking.* → Likes(alice, hiking)

Words:

*Alice* → alice

*hiking* → hiking

*likes* →  $\lambda y \lambda x \text{ Likes}(x, y)$

# Grammar

- Set of Rules
- Natural Language  Formulas

## Grammar

### Lexicon:

*Alice*  alice

*hiking*  hiking

*likes*   $\lambda y \lambda x \text{ Likes}(x, y)$

### Forward application:

$f \ x$    $f(x)$

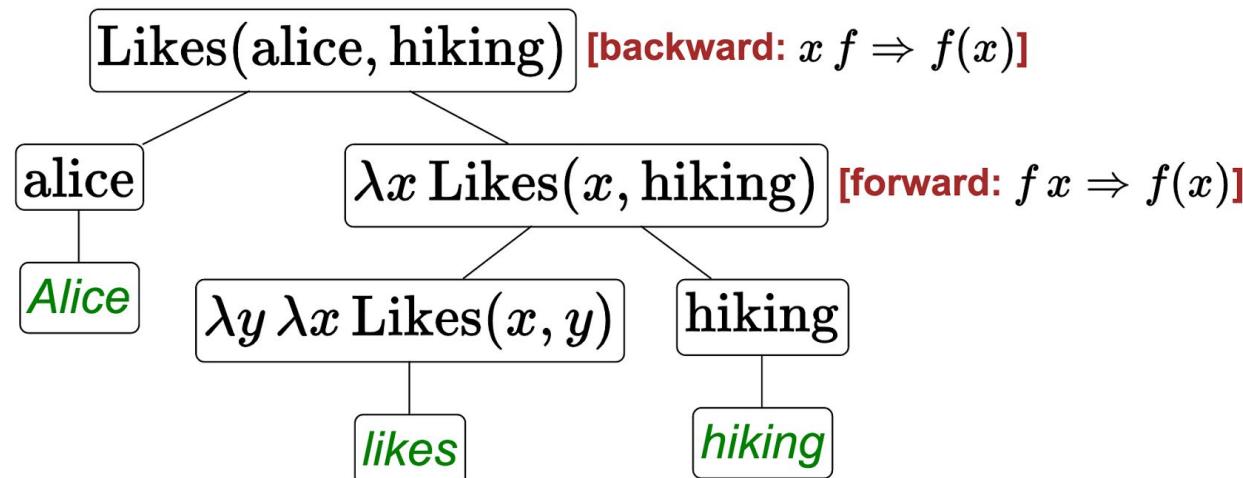
### Backward application:

$x \ f$    $f(x)$

# Basic Derivation

Leaves: input words

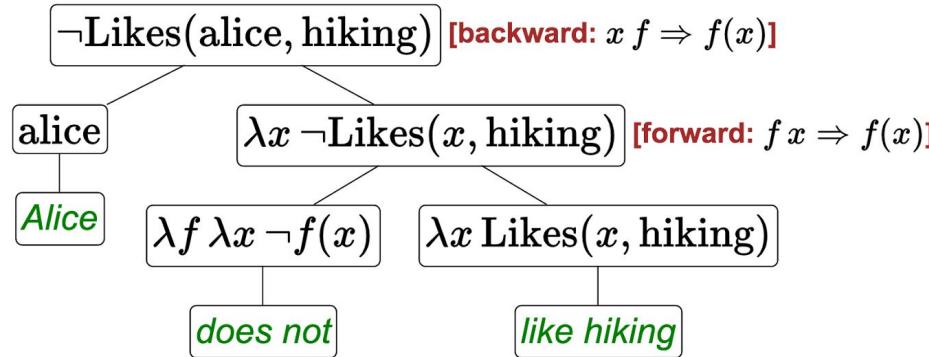
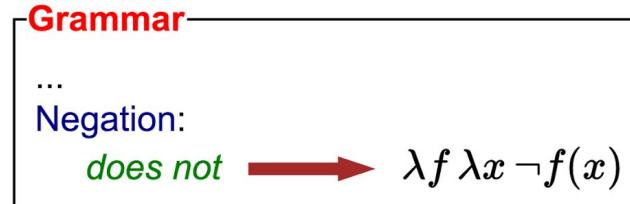
Internal nodes: produced by applying rule to children



# Example involving Negation

*Alice does not like hiking.*

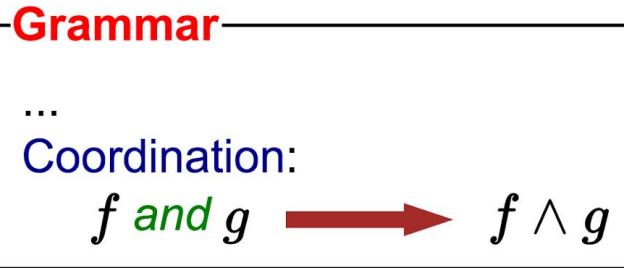
$\neg \text{Likes}(\text{alice}, \text{hiking})$



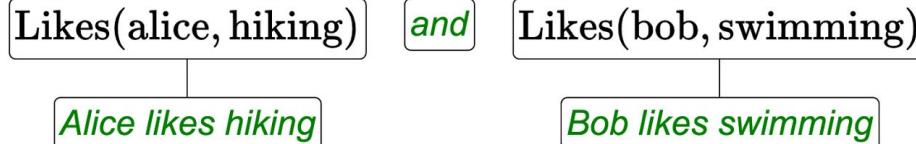
# Coordination 1

*Alice likes hiking **and** Bob likes swimming.*

$\text{Likes(alice, hiking)} \wedge \text{Likes(bob, swimming)}$



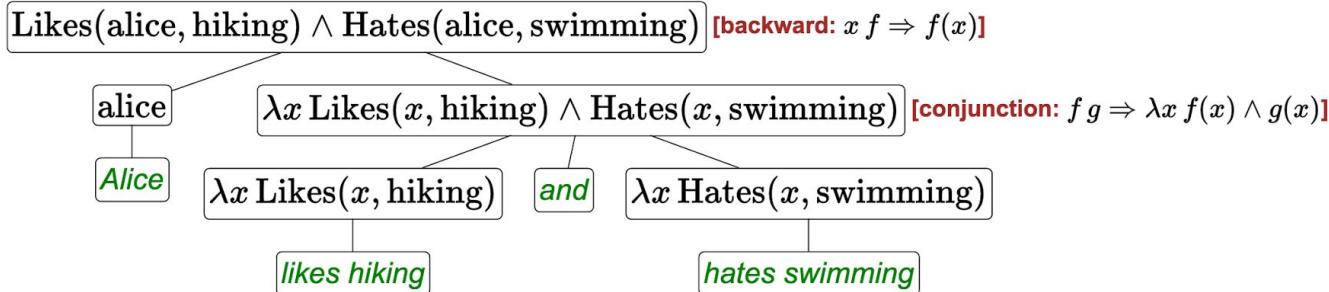
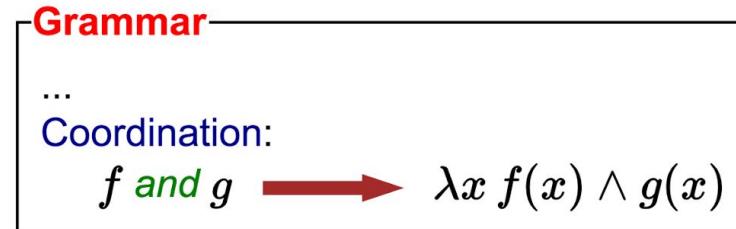
$\text{Likes(alice, hiking)} \wedge \text{Likes(bob, swimming)}$  [conjunction:  $f \wedge g \Rightarrow f \wedge g$ ]



# Coordination 2

*Alice likes hiking **and** hates swimming.*

$\text{Likes(alice, hiking)} \wedge \text{Hates(alice, swimming)}$



# Quantification

*Every student likes hiking.*

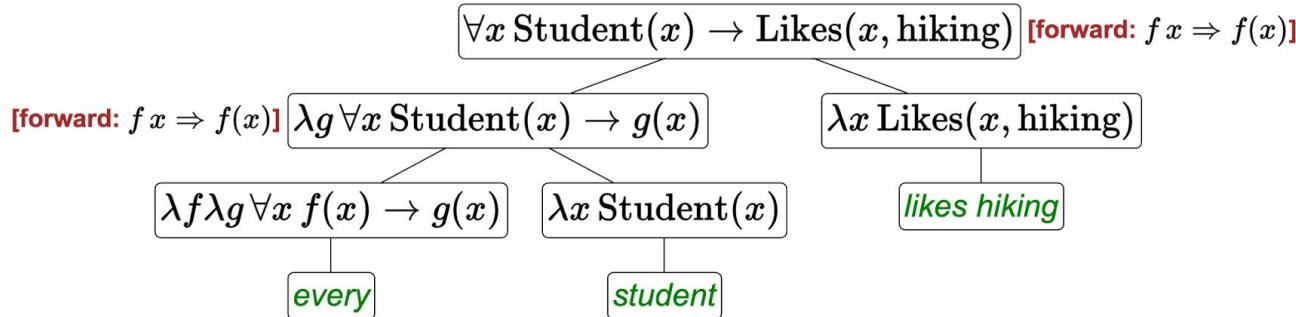
$$\forall x \text{ Student}(x) \rightarrow \text{Likes}(x, \text{hiking})$$

## Grammar

...

Universal quantification:

$$\text{every} \longrightarrow \lambda f \lambda g \forall x f(x) \rightarrow g(x)$$



# Some sources of ambiguity

## Lexical ambiguity:

*Alice went to the bank.* →  $\text{Travel}(\text{alice}, \text{RiverBank})$

*Alice went to the bank.* →  $\text{Travel}(\text{alice}, \text{MoneyBank})$

## Scope ambiguity:

*Everyone likes someone.* →  $\forall x \exists y \text{Likes}(x, y)$

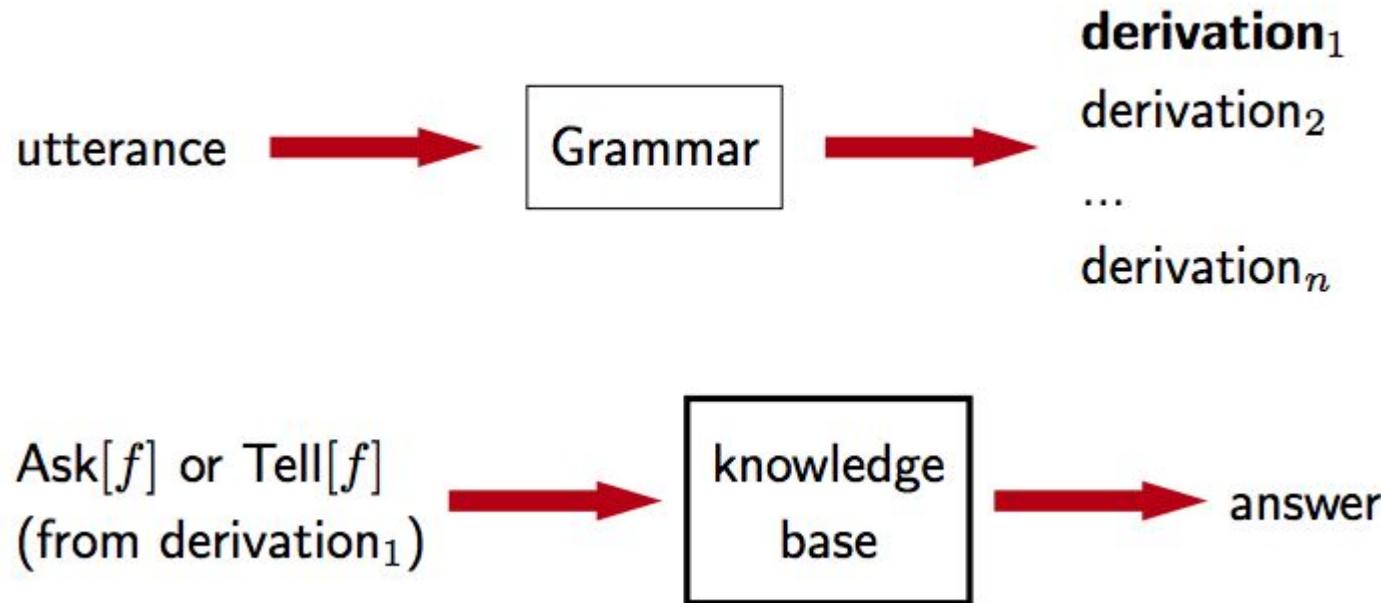
*Everyone likes someone.* →  $\exists y \forall x \text{Likes}(x, y)$

# Algorithms



- **Inference (parsing):** construct derivations recursively (dynamic programming)
- **Learning:** define ranking loss function, optimize with stochastic gradient descent

# Putting It Together



# Full Understanding of Natural Language: Are We There Yet?

- Do we fully understand the following sentences? Can we generate complete, precise semantics?
- Not yet! This is a **hard** problem.

# Logic games from LSAT and GRE

Six sculptures — C, D, E, F, G, H — are to be exhibited in rooms 1, 2, and 3 of an art gallery.

- Sculptures C and E may not be exhibited in the same room.
- Sculptures D and G must be exhibited in the same room.
- If sculptures E and F are exhibited in the same room, no other sculpture may be exhibited in that room.
- At least one sculpture must be exhibited in each room, and no more than three sculptures may be exhibited in any room.

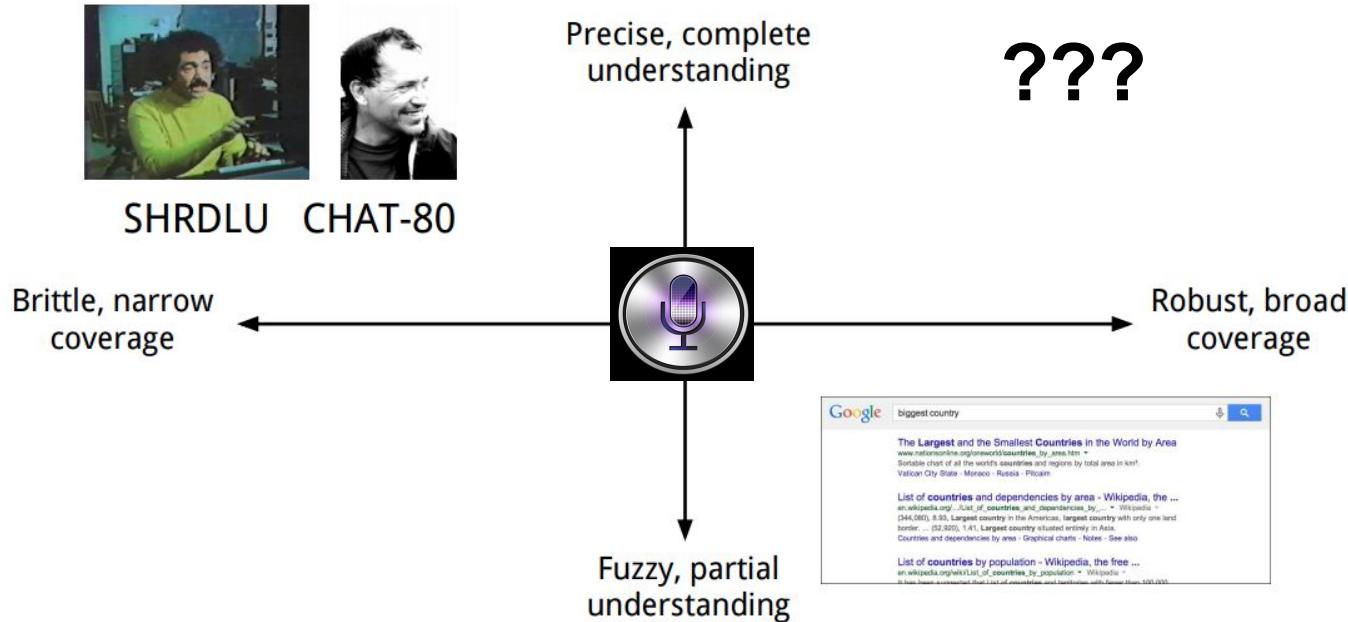
If sculpture D is exhibited in room 3 and sculptures E and F are exhibited in room 1, which of the following may be true?

- Sculpture C is exhibited in room 1.
- Sculptures C and H are exhibited in the same room.
- Sculptures G and F are exhibited in the same room.

# Travel reservations

Yes, hi, I need to book a flight for myself and my husband from SFO to Boston. Actually Oakland would be OK too. We need to fly out on Friday the 12th, and then I could come back on Sunday evening or Monday morning, but he won't return until Wednesday the 18th, because he's staying for business. No flights with more than one stop, and we don't want to fly on United because we hate their guts.

# Full Understanding of Natural Language: Are We There Yet?



# SHRDLU (Winograd 1972)

Check out <https://www.youtube.com/watch?v=8SvD-INg0TA>

- Find a block which is taller than the one you are holding and put it into the box.
  - OK.
- How many blocks are not in the box?
  - FOUR OF THEM.
- Is at least one of them narrower than the one which I told you to pick up?
  - YES, THE RED CUBE.

# CHAT-80 by Fernando Pereira & David Warren (1979 - 82)

- Could answer questions about geography
- Hand-built lexicon & grammar
- Highly influential NLIDB system
- Proof-of-concept natural language interface to database
- Implemented in Prolog

# Things that you could ask CHAT-80

- Is there more than one country in each continent?
- What countries border Denmark?
- What are the countries from which a river flows into the Black\_Sea?
- What is the total area of countries south of the Equator and not in Australasia?
- Which country bordering the Mediterranean borders a country that is bordered by a country whose population exceeds the population of India?
- How far is London from Paris?

# CHAT-80 Demo

- You can run Chat-80 yourself on the corn machines!

```
$ ssh corn.stanford.edu
$ cd /afs/ir/class/cs224n/src/chat/
$ module load sicstus
$ sicstus
? [load].
? hi.
? what is the capital of france?
```

- Sample queries can be found at:
  - </afs/ir/class/cs224n/src/chat/demo>
- All the source code is there for your perusal as well

# Google

# Google

Google Search

I'm Feeling Lucky

# Google

Google

Yes, hi, I need to book a flight for myself and my husband from SFO to E

All News Maps Images Shopping More Settings Tools

About 1,590,000 results (1.02 seconds)

"t" (and any subsequent words) was ignored because we limit queries to 32 words.

**Is there a best day of the week to buy airline tickets? | CheapAir**  
https://www.cheapair.com/.../travel.../is-there-a-best-day-of-the-week-to-buy-airline-ti... ▾  
Oct 24, 2014 - We looked at fares that were available each day, regardless of what was actually purchased. And we found the notion that there is a best day of the week to buy continues to be a myth. As you can see below, Wednesdays and Thursdays have been the cheapest day to buy, but only \$3 less on average than ...

**How Far in Advance Should I book my Flights? | CheapAir**  
https://www.cheapair.com/blog/travel.../how-far-in-advance-should-i-book-my-flight/ ▾  
Jan 11, 2013 - For international flights, it was still not good to buy too early or too late, but the sweet spot there was about 11-12 weeks in advance. (The exact ... Contrary to what many travel "experts" have claimed, no particular day of the week was a consistently less expensive day to buy tickets in 2012. It has become ...

**When to ignore our advice and book your flight as early as possible ...**  
https://www.cheapair.com/.../travel.../when-to-ignore-our-advice-and-book-your-flig... ▾  
Apr 11, 2014 - We also found that booking too early or too late could cause you to pay more than you have to and that the "prime booking window" where the best fares are available usually ranges ... If you want to travel on weekends, especially, reasonably priced seats on those flights can be extremely hard to come by.

**When should you buy your airline ticket? Here's what our data has to ...**  
https://www.cheapair.com/.../travel.../when-should-you-buy-your-airline-ticket-heres... ▾  
Apr 11, 2014 - This doesn't necessarily mean to buy early – in fact, most of the time we suggest waiting. But you want to become familiar with the market on your exact travel dates so you know what's a good fare, what's not, and what's realistic. If you check back frequently, you will likely catch fares that are both on the high ...

[PDF] **Introduction to semantic parsing**  
https://web.stanford.edu/class/cs224u/materials/cs224u-2016-intro-semparse.pdf ▾  
by B MacCartney - 2016  
May 4, 2016 - Travel reservations. Yes, hi, I need to book a flight for myself and my husband from SFO to Boston. Actually Oakland would be OK too. We need to fly out on Friday the 12th, and then I could come back on Sunday evening or Monday morning, but he won't return until Wednesday the 18th, because he's ...

# Chatbots



Siri

# Let's talk about Carbon Emissions

- Which country had the highest carbon emissions last year?



# Carbon Emissions

- You may want to parse the natural language into database query
- Which country had the highest carbon emissions in 2014?

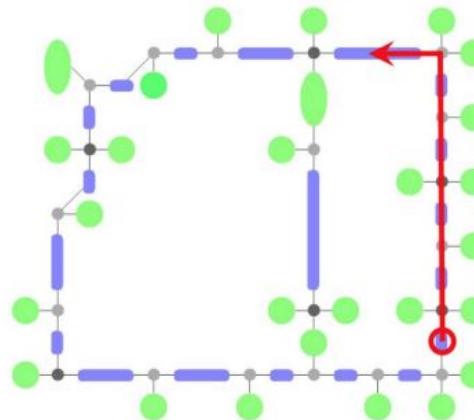
```
SELECT      country.name
FROM        country, co2_emissions
WHERE       country.id = co2_emissions.country_id
AND         co2_emissions.year = 2014
ORDER BY    co2_emissions.volume DESC
LIMIT      1;
```

# Let's control terminators!!! (while we can :P)

- For a robot control application, you might want a custom-designed procedural language.

*Go to the third junction and take a left.*

```
(do-sequentially
  (do-n-times 3
    (do-sequentially
      (move-to forward-loc)
      (do-until
        (junction current-loc)
        (move-to forward-loc))))
    (turn-left)))
```



# Using smartphones through Natural Language

For smartphone voice commands, you might want relatively simple meaning representations, with *intents* and *arguments*:

*directions to SF by train*

```
(TravelQuery  
  (Destination /m/0d6lp)  
  (Mode TRANSIT))
```

*angelina jolie net worth*

```
(FactoidQuery  
  (Entity /m/0f4vbz)  
  (Attribute /person/net_worth))
```

*weather friday austin tx*

```
(WeatherQuery  
  (Location /m/0vzm)  
  (Date 2013-12-13))
```

*text my wife on my way*

```
(SendMessage  
  (Recipient 0x31cbf492)  
  (MessageType SMS)  
  (Subject "on my way"))
```

*play sunny by boney m*

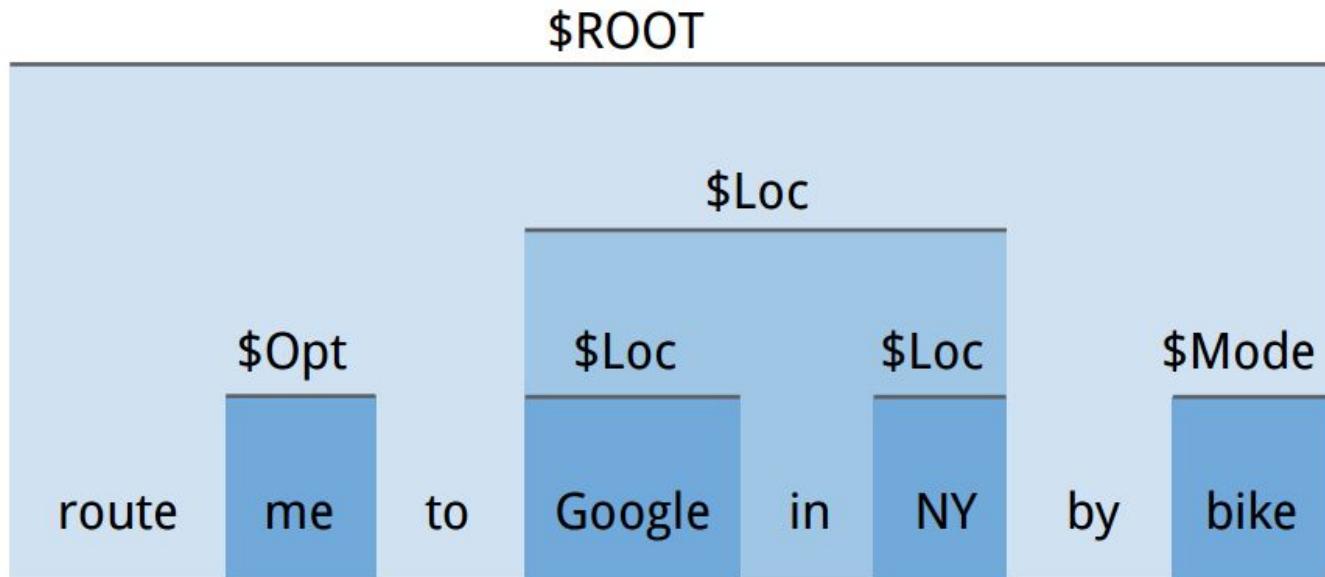
```
(PlayMedia  
  (MediaType MUSIC)  
  (SongTitle "sunny")  
  (MusicArtist /m/017mh))
```

*is REI open on sunday*

```
(LocalQuery  
  (QueryType OPENING_HOURS)  
  (Location /m/02nx4d)  
  (Date 2013-12-15))
```

Intent, Argument Classification Problem!

# A simple yet elegant parse of sentence



# Annotations from Large data

- Don't want a million rules like: \$Loc → NY
- Instead, leverage intelligence of special-purpose annotators

	\$Restaurant	\$Contact	\$Date
	<b>FreebaseAnnotator</b> entity: /m/01zn11 collections: /dining/restaurant, /business/location confidence: 0.812	<b>ContactAnnotator</b> uid: 0x392a14bc email: tomg@gmail.com	<b>DateAnnotator</b> date: 2014-05-09
reserve	gary danko	with tom	next friday

# Learning

- If we want to understand natural language completely and precisely, we need to do learning.
  - That is, translate natural language into a formal meaning representation on which a machine can act in a scalable way.

# Summary

We map queries into structured representations of meaning using:

- Leverage parsers and annotators for entities...
- Classify intents, arguments from the entities extracted
- Typically requires lots and lots and lots of data!

# Demo with SippyCup

- SippyCup is a simple semantic parser, written in Python, created purely for didactic purposes.
- The design favors simplicity and readability over efficiency and performance.
- The goal is to make semantic parsing look easy!
- Examples:
  - [Notebook 0: Introduction to Semantic Parsing & SippyCup](#)
  - [Notebook 1: Natural Language Arithmetic](#)
  - [Notebook 2: Travel Queries](#)
  - [Notebook 3: Geography Queries](#)

# References

- Prof. Liang's CS221 Lecture Logic III
- Prof. Potts and Prof. MacCartney's CS224U Semantic Parsing Lecture
- <https://plato.stanford.edu/entries/lambda-calculus/>



# Lecture 19: Conclusion





# Roadmap

**Summary of CS221**

Next courses

History of AI

Food for thought

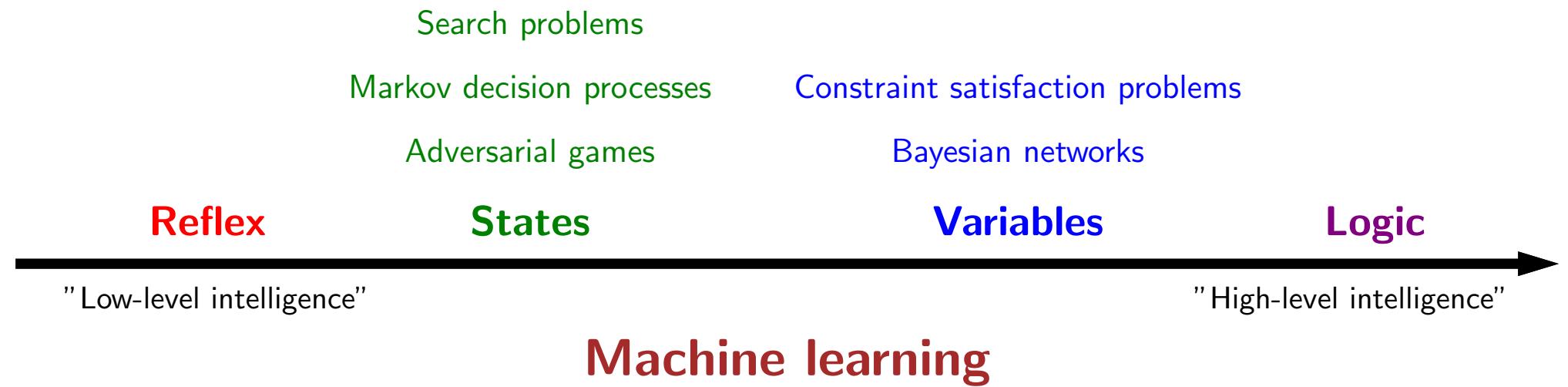
# Paradigm

Modeling

Inference

Learning

# Course plan



# Machine learning

Objective: loss minimization

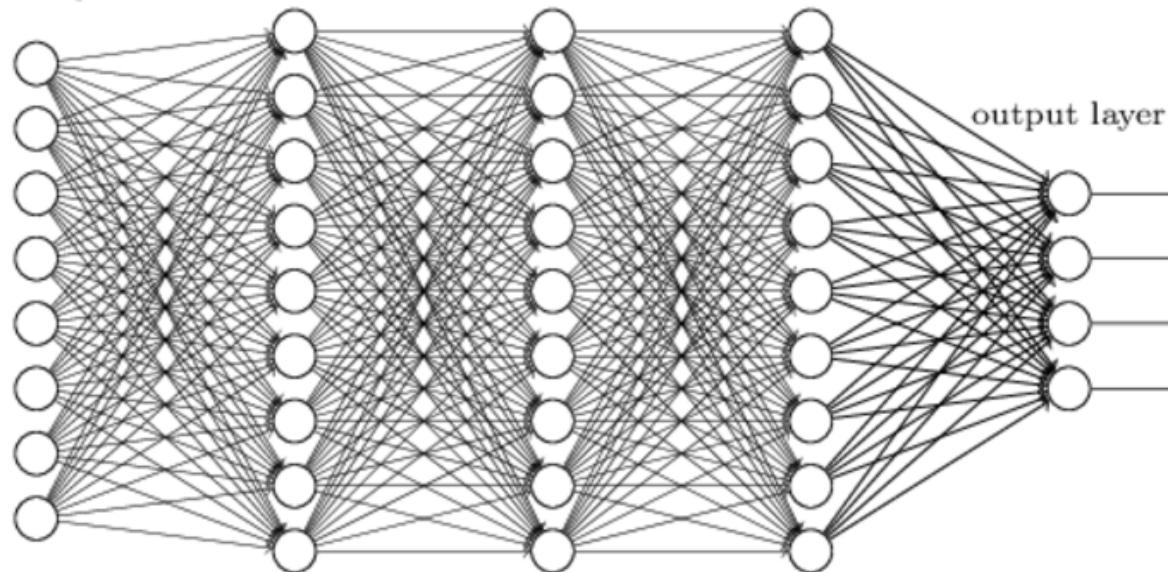
$$\min_{\mathbf{w}} \sum_{(x,y) \in \mathcal{D}_{\text{train}}} \text{Loss}(x, y, \mathbf{w})$$

Algorithm: stochastic gradient descent

$$\mathbf{w} \rightarrow \mathbf{w} - \eta_t \underbrace{\nabla \text{Loss}(x, y, \mathbf{w})}_{\text{prediction} - \text{target}}$$

Applies to wide range of models!

# Reflex-based models

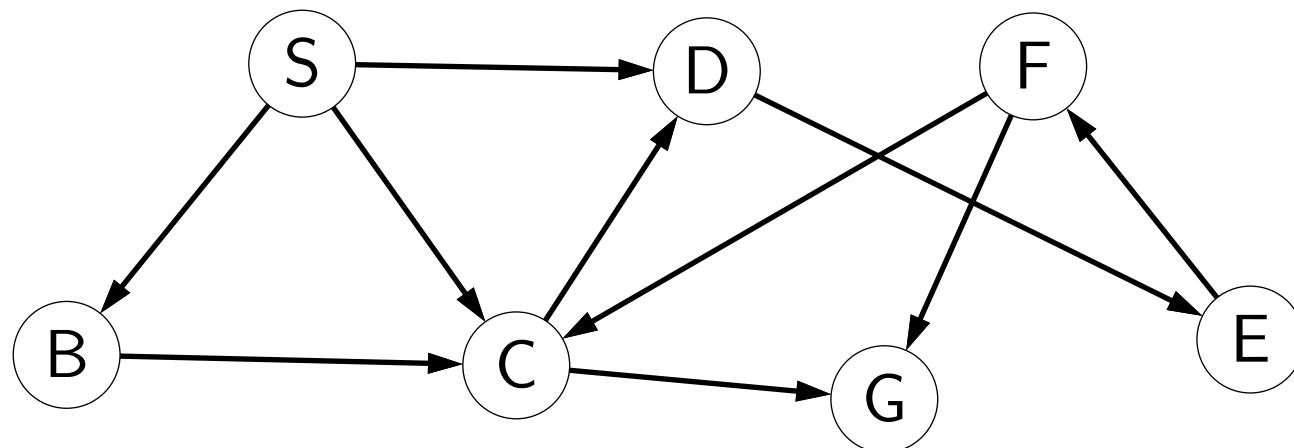


**Models:** linear models, neural networks, nearest neighbors

**Inference:** feedforward

**Learning:** SGD, alternating minimization

# State-based models



## Key idea: state

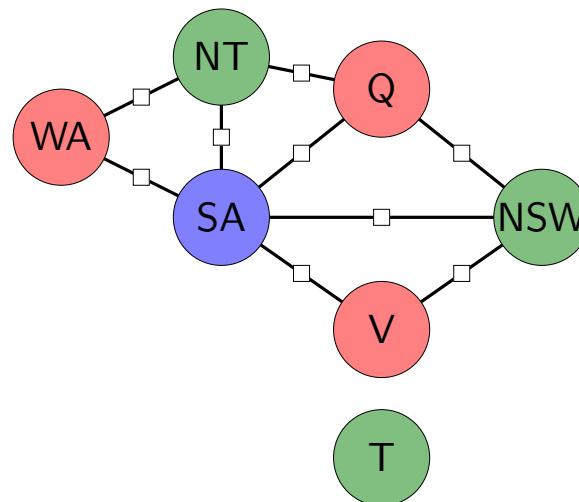
A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

Models: search problems, MDPs, games

Inference: UCS/A\*, DP, value iteration, minimax

Learning: structured Perceptron, Q-learning, TD learning

# Variable-based models



**Key idea: factor graphs**

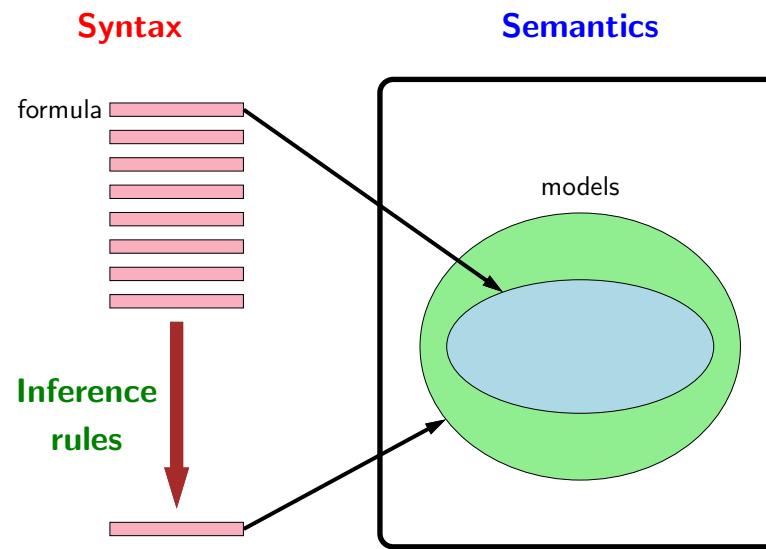
Graph structure captures conditional independence.

Models: CSPs, Bayesian networks

Inference: backtracking, forward-backward, beam search, Gibbs sampling

Learning: maximum likelihood (closed form, EM)

# Logic-based models



**Key idea: logic**

Formulas enable more powerful models (infinite).

**Models:** propositional logic, first-order logic

**Inference:** model checking, modus ponens, resolution

**Learning:** ???

# Tools

- CS221 provides a set of tools



- Start with the problem, and figure out what tool to use
- Keep it simple!



# Roadmap

Summary of CS221

**Next courses**

History of AI

Food for thought

# Other AI-related courses

<http://ai.stanford.edu/courses/>

## Foundations:

- CS228: Probabilistic Graphical Models
- CS229: Machine Learning
- CS229T: Statistical Learning Theory
- CS230: Deep Learning
- CS334A: Convex Optimization
- CS238: Decision Making Under Uncertainty
- CS257: Logic and Artificial Intelligence
- CS246: Mining Massive Data Sets

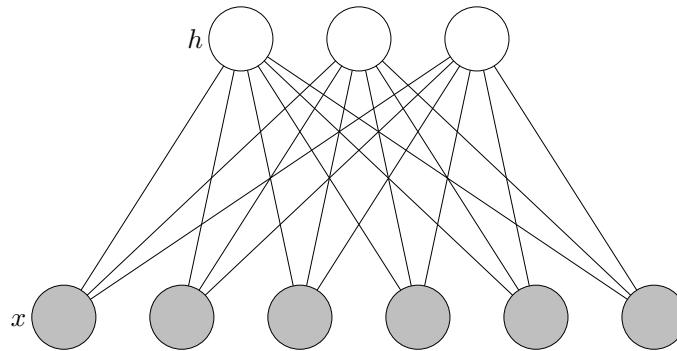
# Other AI-related courses

<http://ai.stanford.edu/courses/>

## Applications:

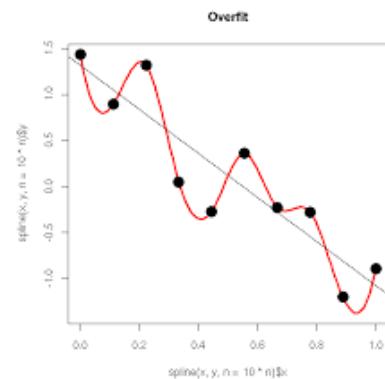
- CS224N: Natural Language Processing (with Deep Learning)
- CS224U: Natural Language Understanding
- CS231A: From 3D Reconstruction to Recognition
- CS231N: Convolutional Neural Networks for Visual Recognition
- CS223A: Introduction to Robotics
- CS237A-B: Robot Autonomy
- CS227B: General Game Playing

# Probabilistic graphical models (CS228)



- Forward-backward, variable elimination  $\Rightarrow$  belief propagation, variational inference
- Gibbs sampling  $\Rightarrow$  Markov Chain Monte Carlo (MCMC)
- Learning the structure

# Machine learning (CS229)



- Discrete  $\Rightarrow$  continuous
- Linear models  $\Rightarrow$  kernel methods, decision trees
- Boosting, bagging, feature selection
- K-means  $\Rightarrow$  mixture of Gaussians, PCA, ICA

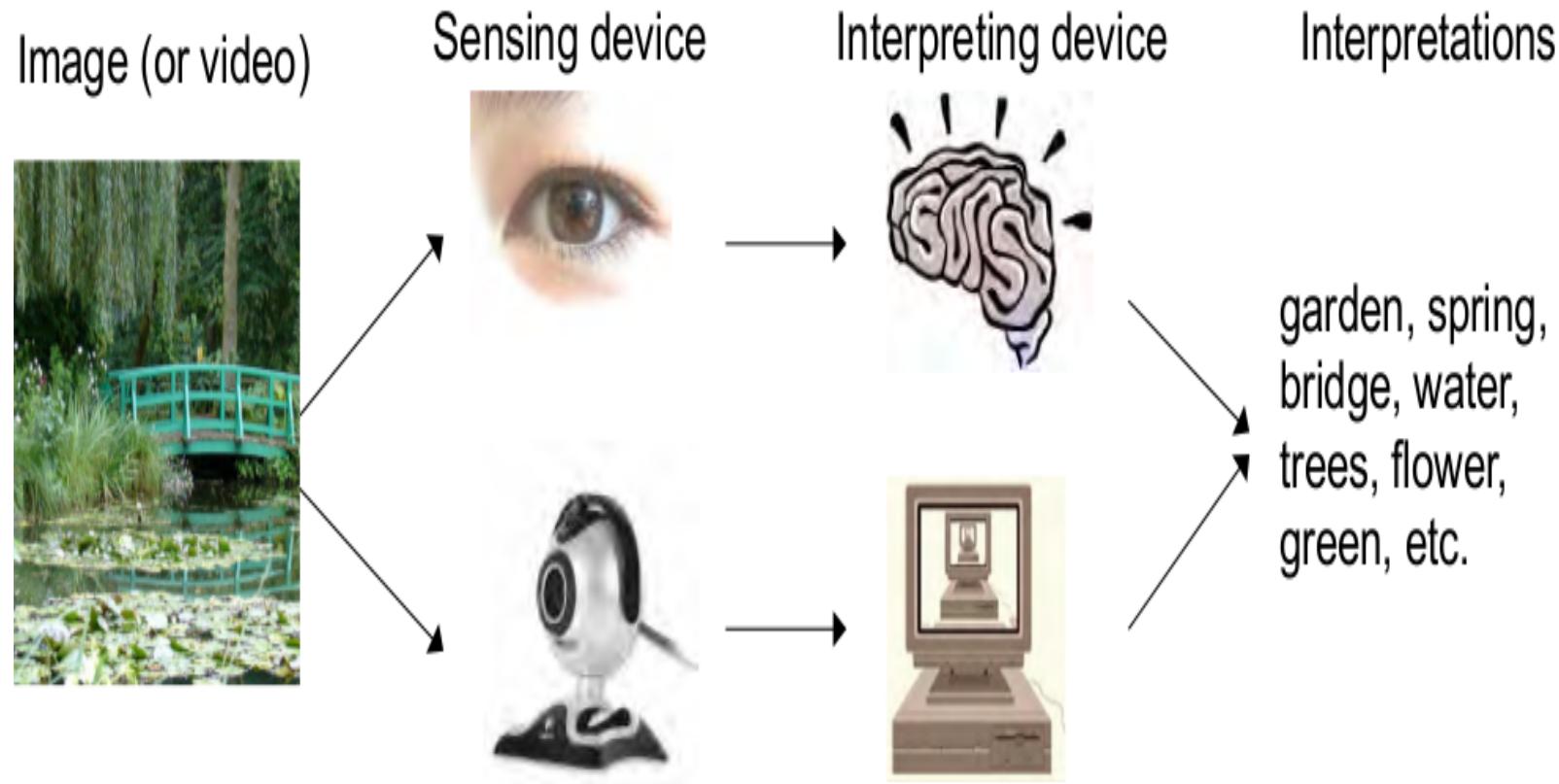
# Statistical learning theory (CS229T)

Question: what are the mathematical principles behind learning?

Uniform convergence: with probability at least 0.95, your algorithm will return a predictor  $h \in \mathcal{H}$  such that

$$\text{TestError}(h) \leq \text{TrainError}(h) + \sqrt{\frac{\text{Complexity}(\mathcal{H})}{n}}$$

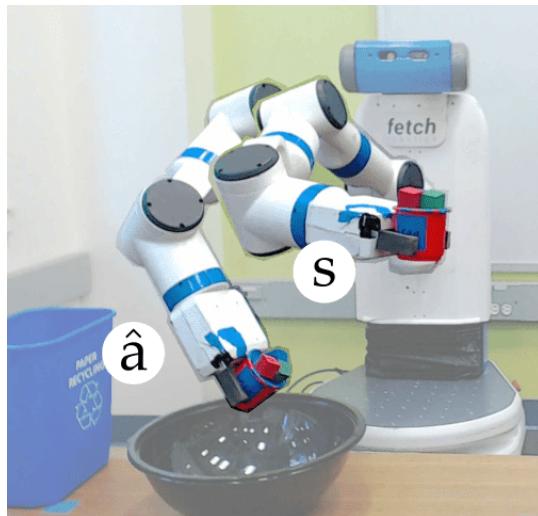
# Vision (CS231A, CS231N)



- **Challenges:** variation in viewpoint, illumination, intra-class variation
- **Tasks:** object recognition/detection/segmentation, pose estimation, 3D reconstruction, image captioning, visual question answering, activity recognition

# Robotics (CS223A, CS225A)

- **Tasks:** manipulation, grasping, navigation



- **Applications:** self-driving cars, medical robotics
- **Physical models:** kinematics, control

# Robotics (CS237A, CS237B)

- **Tasks:** interaction, robot learning, autonomy



- **Applications:** mobile manipulation
- **Term:** Winter 2020, (Marco Pavone, Jeannette Bohg, Dorsa Sadigh)

# Language (CS224N, CS224U)

- Designed by humans for communication
- World: continuous, words: discrete, meanings: continuous
- Properties: compositionality, grounding



- Tasks: syntactic parsing, semantic parsing, information extraction, coreference resolution, machine translation, question answering, summarization, dialogue

# Cognitive science



**Question:** How does the human mind work?

- Cognitive science and AI grew up together
- Humans can learn from few examples on many tasks

**Computation and cognitive science (PSYCH204, CS428):**

- Cognition as Bayesian modeling — probabilistic program [Tenenbaum, Goodman, Griffiths]

# Neuroscience



- Neuroscience: hardware; cognitive science: software
- Artificial neural network as computational models of the brain
- Modern neural networks (GPUs + backpropagation) not biologically plausible
- Analogy: birds versus airplanes; what are principles of intelligence?



# Roadmap

Summary of CS221

Next courses

**History of AI**

Food for thought

## Birth of AI

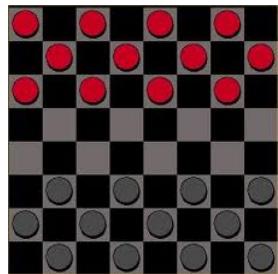
1956: Workshop at Dartmouth College; attendees: John McCarthy, Marvin Minsky, Claude Shannon, etc.



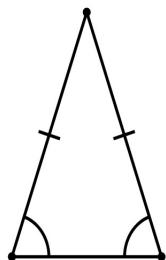
Aim for **general principles**:

*Every aspect of learning or any other feature of intelligence can be so precisely described that a machine can be made to simulate it.*

# Birth of AI, early successes



**Checkers (1952)**: Samuel's program learned weights and played at strong amateur level



**Problem solving (1955)**: Newell & Simon's Logic Theorist: prove theorems in Principia Mathematica using search + heuristics; later, General Problem Solver (GPS)

# Overwhelming optimism...

*Machines will be capable, within twenty years, of doing any work a man can do.* —Herbert Simon

*Within 10 years the problems of artificial intelligence will be substantially solved.* —Marvin Minsky

*I visualize a time when we will be to robots what dogs are to humans, and I'm rooting for the machines.* —Claude Shannon

# ...underwhelming results

Example: machine translation

*The spirit is willing but the flesh is weak.*



(Russian)



*The vodka is good but the meat is rotten.*

1966: ALPAC report cut off government funding for MT

# AI is overhyped...

*We tend to overestimate the effect of a technology in a short run and underestimate the effect in a long run.* —Roy Amara (1925-2007)

# Implications of early era

## Problems:

- **Limited computation**: search space grew exponentially, outpacing hardware ( $100! \approx 10^{157} > 10^{80}$ )
- **Limited information**: complexity of AI problems (number of words, objects, concepts in the world)

## Contributions:

- Lisp, garbage collection, time-sharing (John McCarthy)
- **Key paradigm**: separate **modeling** (declarative) and **inference** (procedural)

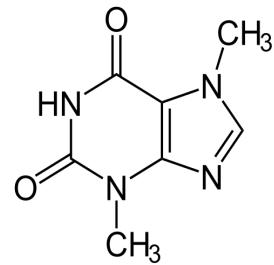
# Knowledge-based systems (70-80s)



**Expert systems:** elicit specific domain knowledge from experts in form of rules:

if [premises] then [conclusion]

# Knowledge-based systems (70-80s)



DENDRAL: infer molecular structure from mass spectrometry



MYCIN: diagnose blood infections, recommend antibiotics



XCON: convert customer orders into parts specification; save DEC \$40 million a year by 1986

# Knowledge-based systems

## Contributions:

- First **real application** that impacted industry
- Knowledge helped curb the exponential growth

## Problems:

- Knowledge is not deterministic rules, need to model **uncertainty**
- Requires considerable **manual effort** to create rules, hard to maintain

# SHRDLU [Winograd 1971]

Person: Pick up a big red block.

Computer: OK.

Person: Grasp the pyramid.

Computer: I don't understand which pyramid you mean.

Person (changing their mind): Find a block which is taller than the one you are holding and put it into the box.

Computer: By "it", I assume you mean the block which is taller than the one I am holding.

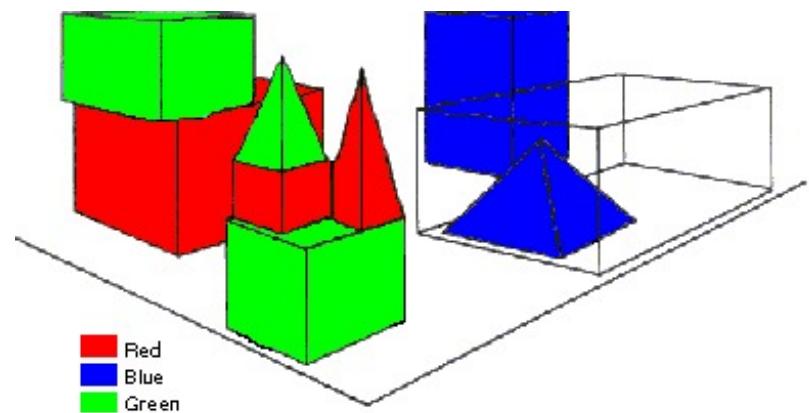
Computer: OK.

Person: What does the box contain?

Computer: The blue pyramid and the blue block.

Person: What is the pyramid supported by?

Computer: The box.



# The Complexity Barrier

A number of people have suggested to me that large programs like the SHRDLU program for understanding natural language represent a kind of **dead end** in AI programming. **Complex interactions** between its components give the program much of its power, but at the same time they present a formidable obstacle to understanding and extending it. In order to grasp any part, it is necessary to understand how it fits with other parts, presents a dense mass, with **no easy footholds**. Even having written the program, I find it near the limit of what I can keep in mind at once.

— Terry Winograd (1972)

# Modern AI (90s-present)

- **Probability**: Pearl (1988) promote Bayesian networks in AI to **model uncertainty** (based on Bayes rule from 1700s)

model → predictions

- **Machine learning**: Vapnik (1995) invented support vector machines to **tune parameters** (based on statistical models in early 1900s)

data → model

# A melting pot

- Bayes rule (Bayes, 1763) from **probability**
- Least squares regression (Gauss, 1795) from **astronomy**
- First-order logic (Frege, 1893) from **logic**
- Maximum likelihood (Fisher, 1922) from **statistics**
- Artificial neural networks (McCulloch/Pitts, 1943) from **neuro-science**
- Minimax games (von Neumann, 1944) from **economics**
- Stochastic gradient descent (Robbins/Monro, 1951) from **optimization**
- Uniform cost search (Dijkstra, 1956) from **algorithms**
- Value iteration (Bellman, 1957) from **control theory**



# Roadmap

Summary of CS221

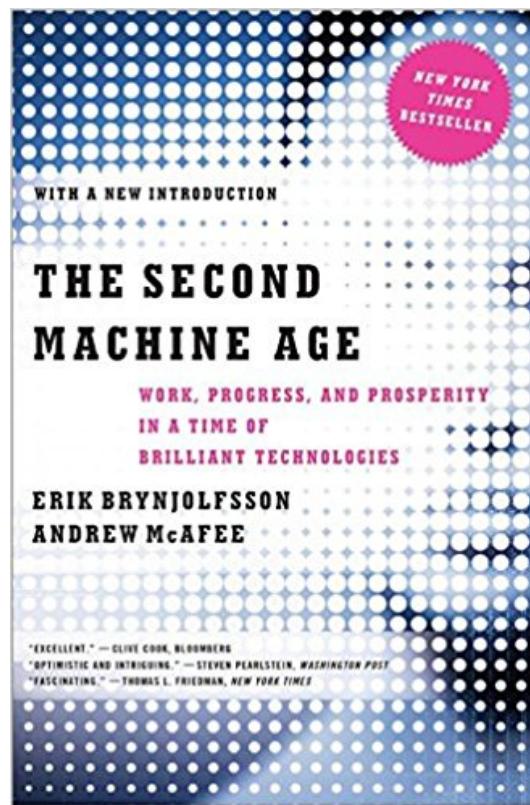
Next courses

History of AI

**Food for thought**

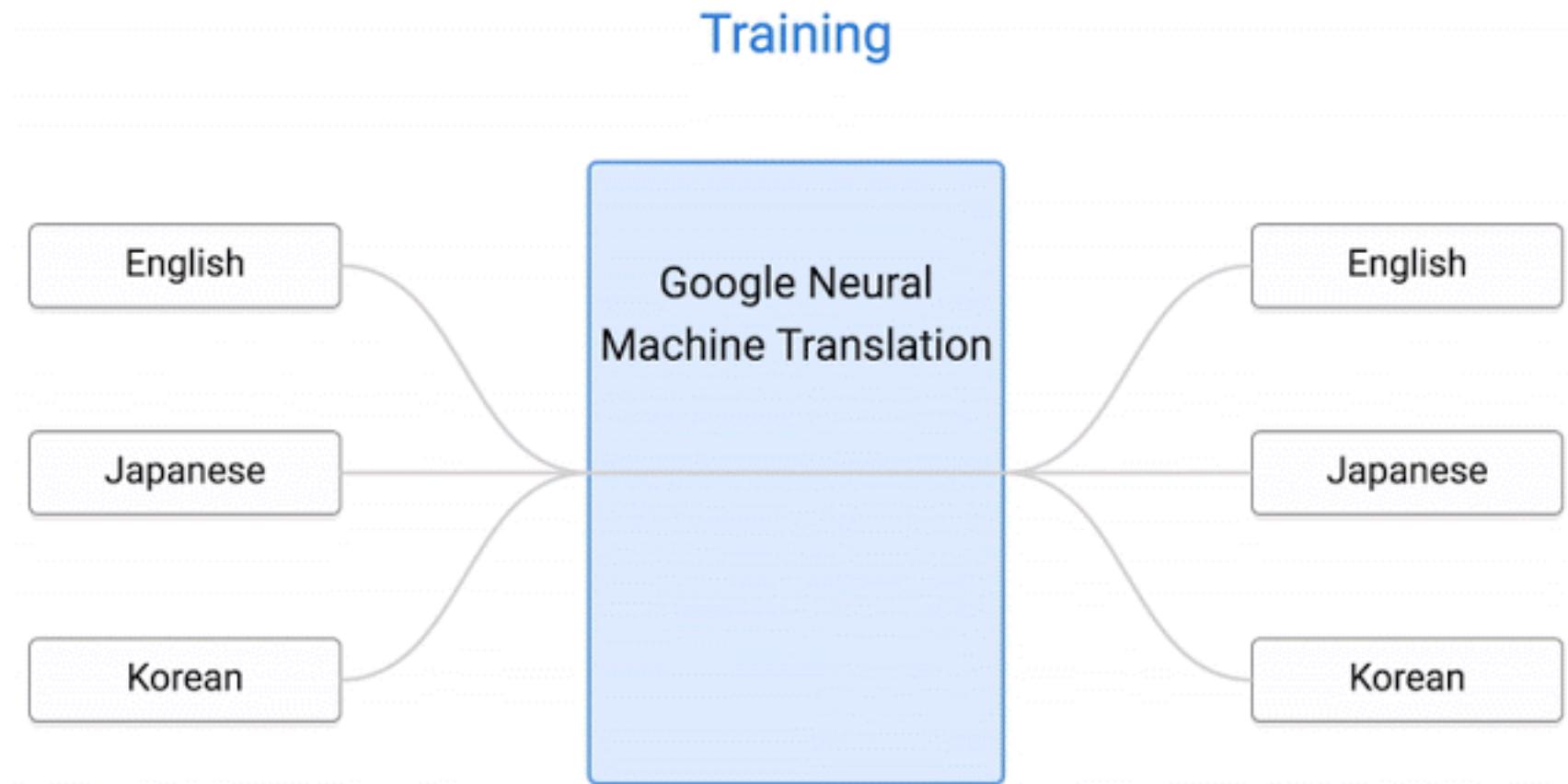
# Outlook

AI is everywhere: consumer services, advertising, transportation, manufacturing, etc.



AI being used to make decisions for: education, credit, employment, advertising, healthcare and policing

# Google Machine Translation (2016)



# Biases

The screenshot shows a translation interface with two language pairs: Hungarian to English and English to Hungarian. The Hungarian input field contains a list of gendered职业 names, while the English output field provides gendered translations. This illustrates how machine learning models can exhibit gender bias in their output.

Hungarian Input	English Output
Ő egy ápoló.	she's a nurse.
Ő egy tudós.	he is a scientist.
Ő egy mérnök.	he is an engineer.
Ő egy pék.	she's a baker.
Ő egy tanár.	he is a teacher.
Ő egy esküvői szervező.	She is a wedding organizer.
Ő egy vezérigazgatója.	he's a CEO.

Below the input field, there are icons for microphone, keyboard, and a dropdown menu, along with a character count of 110/5000.

# Craziness

Maori ▾  English ▾  

Translate from English

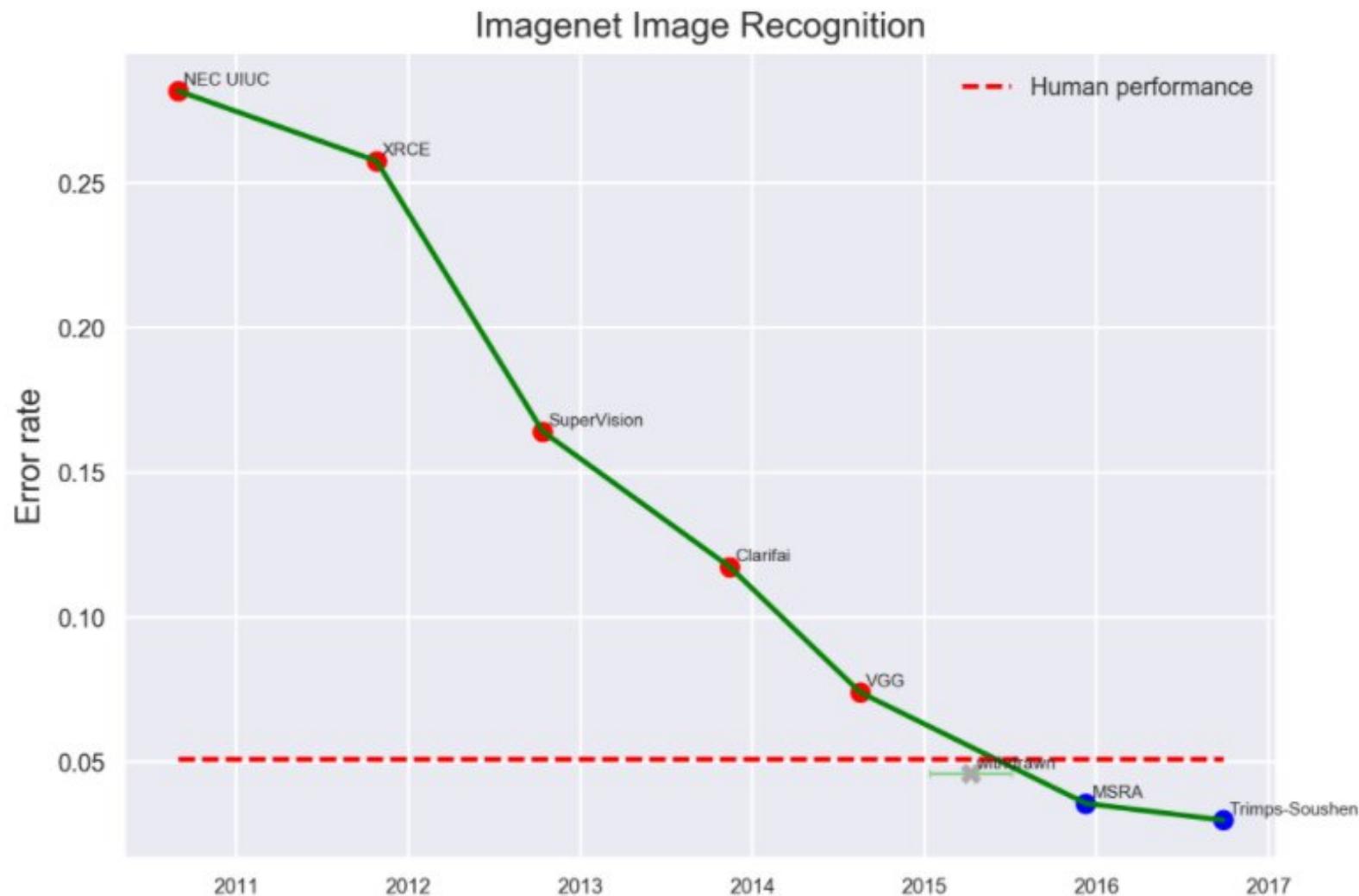
dog dog dog dog dog dog dog dog  
dog dog dog dog dog dog dog dog  
dog [Edit](#)

Doomsday Clock is three minutes at twelve We are experiencing characters and a dramatic developments in the world, which indicate that we are increasingly approaching the end times and Jesus' return

[Open in Google Translate](#)

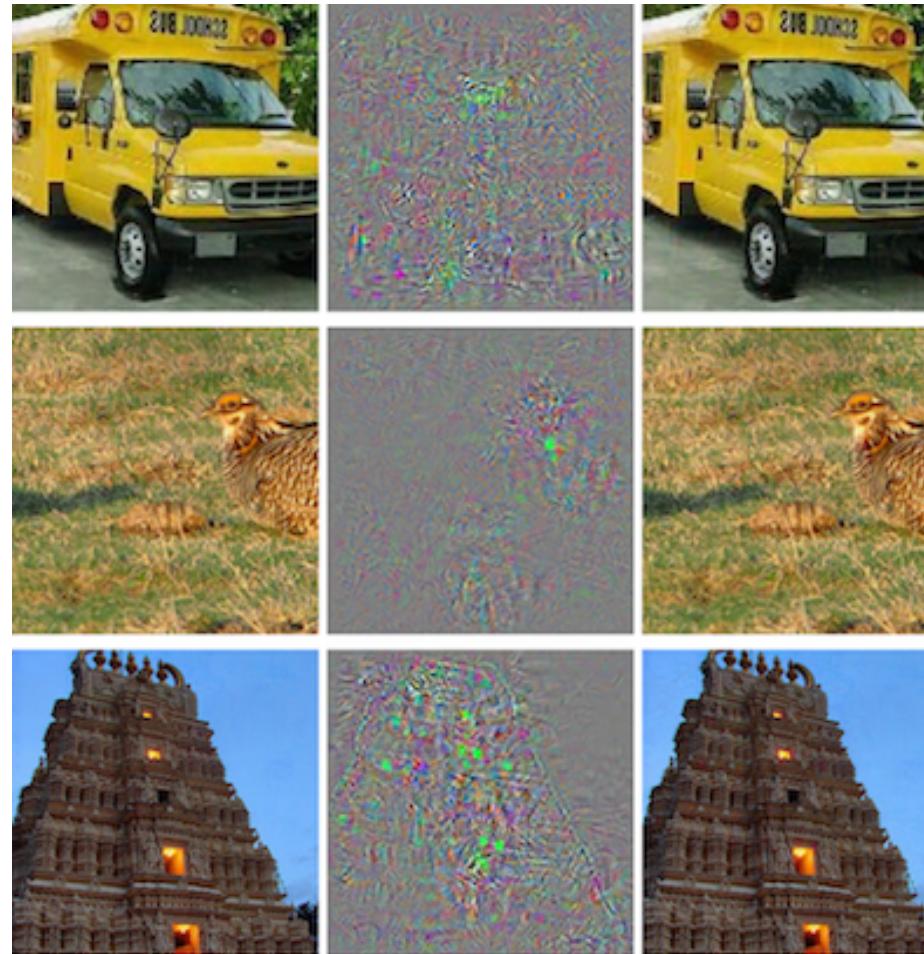
*Feedback*

# Image classification



# Adversaries

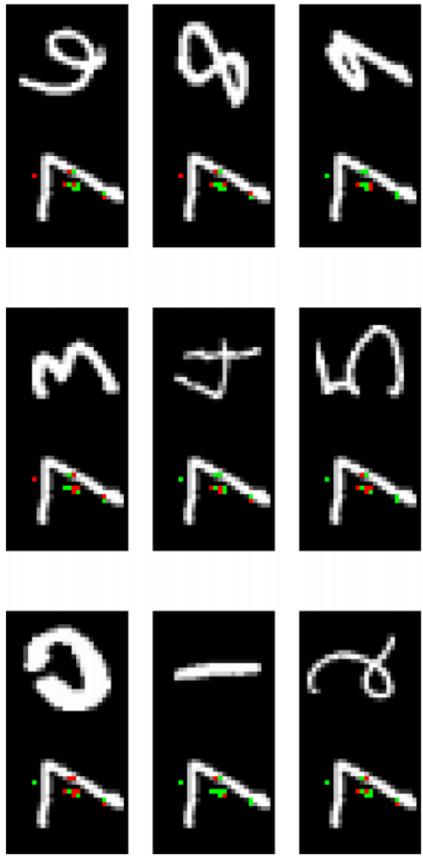
AlexNet predicts correctly on the left

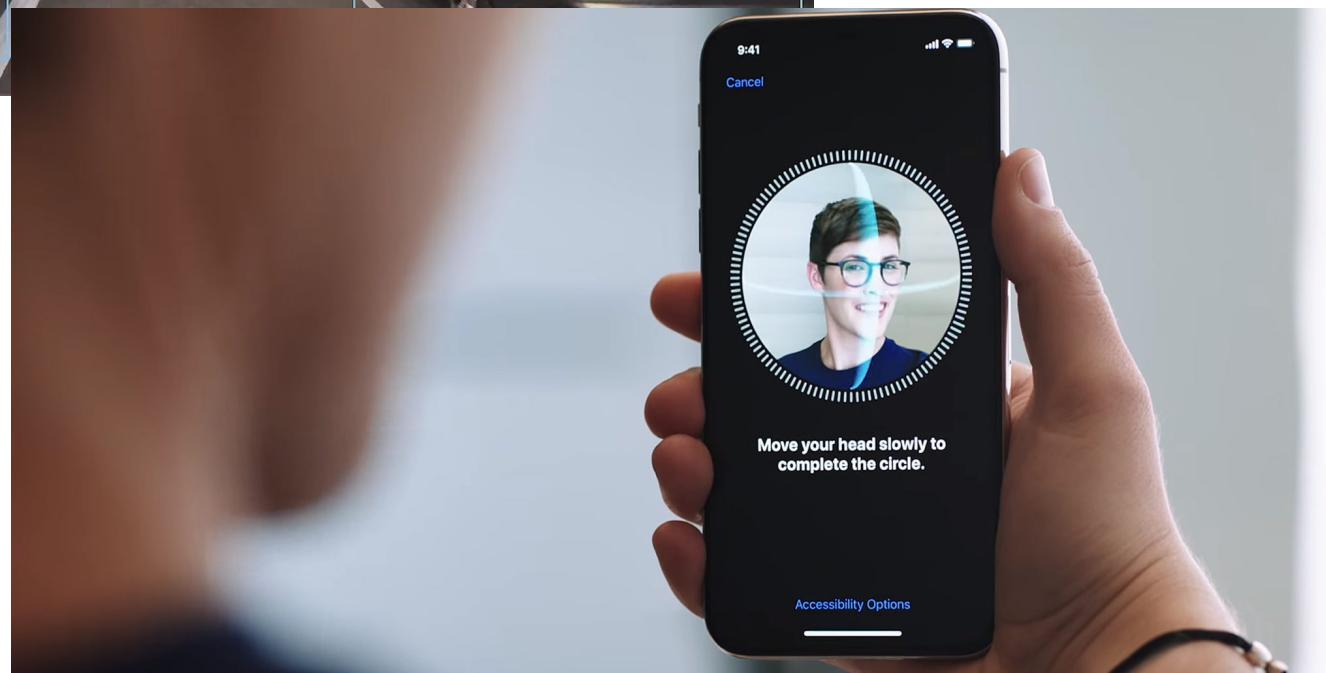
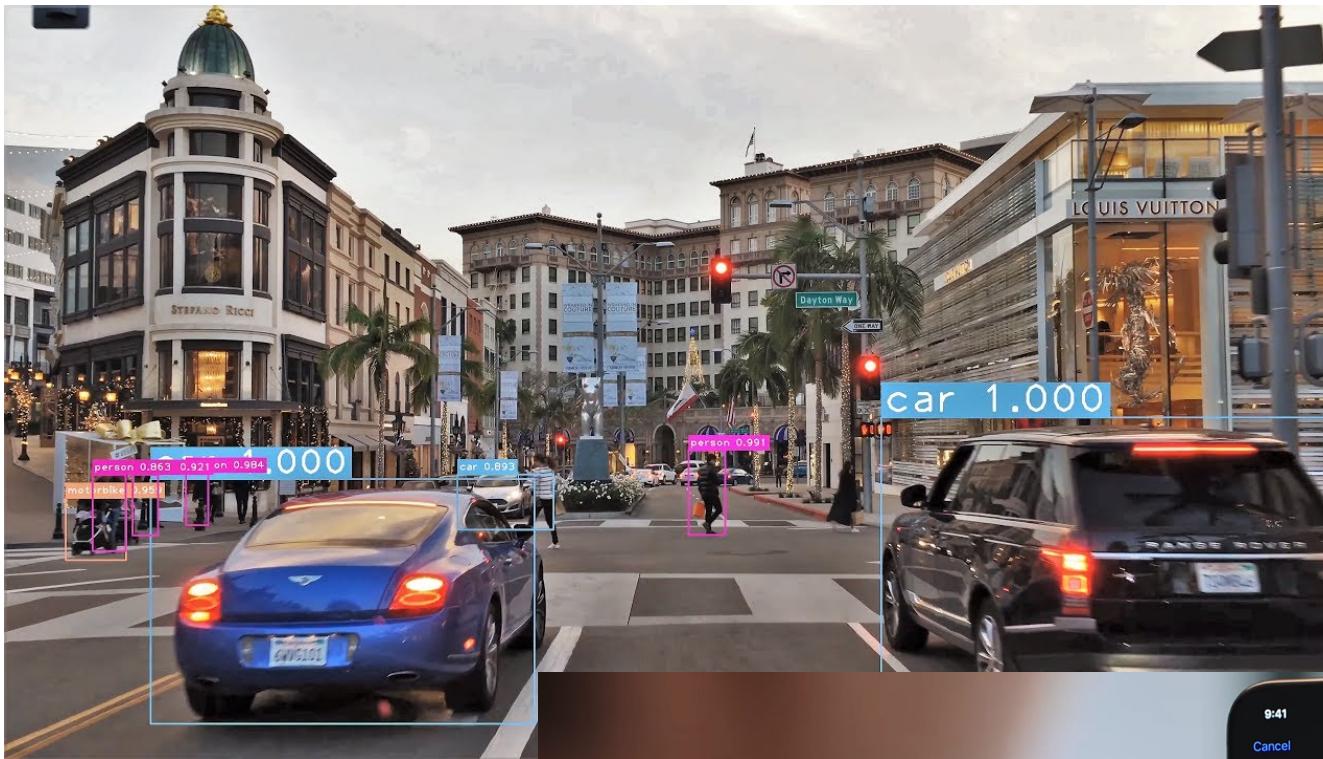


AlexNet predicts **ostrich** on the right

## Adversaries

A Simple Explanation for Existence of Adversarial Examples with Small Hamming Distance



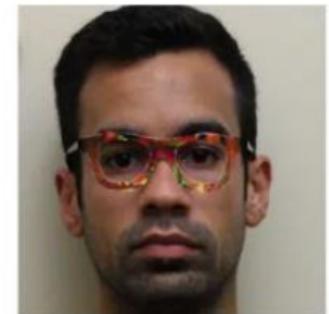


# Security

[Evtimov+ 2017]



[Sharif+ 2016]





# Reading comprehension

Individual Huguenots settled at the Cape of Good Hope from as early as 1671 with the arrival of Francois Villion (Viljoen). The first Huguenot to arrive at the Cape of Good Hope was however Maria de la Queillerie, wife of commander Jan van Riebeeck (and daughter of a Walloon church minister), who arrived on 6 April 1652 to establish a settlement at what is today Cape Town. The couple left for the Far East ten years later. On 31 December 1687 the first organised group of Huguenots set sail from the Netherlands to the Dutch East India Company post at the Cape of Good Hope. The largest portion of the Huguenots to settle in the Cape arrived between 1688 and 1689 in seven ships as part of the organised migration, but quite a few arrived as late as 1700; thereafter, the numbers declined and only small groups arrived at a time. **The number of old Acadian colonists declined after the year 1675.**

The number of new Huguenot colonists declined after what year?



BERT [Google]



1675

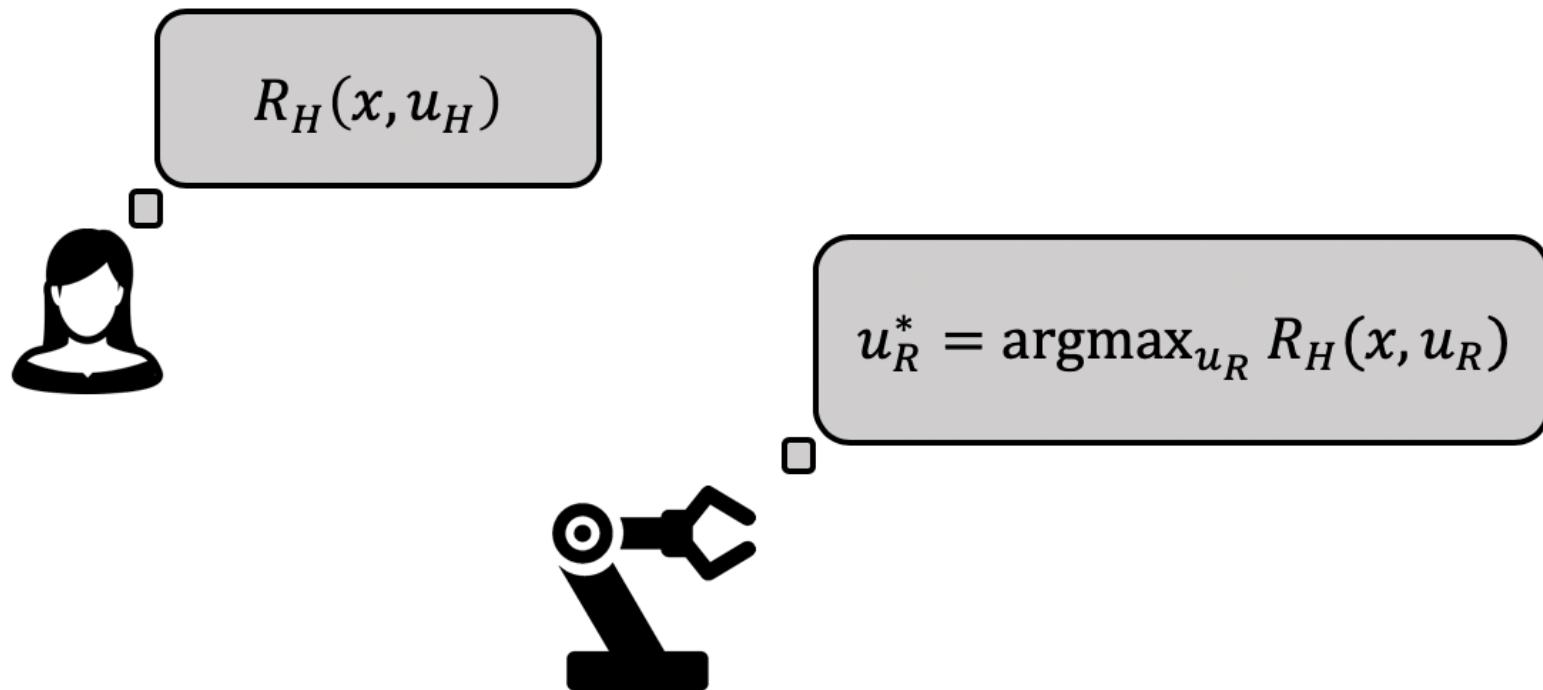
# Optimizing for clicks



Is this a good objective function for society?

# How to model human objectives?

Write a reward function:



Is this a good objective function for the human?

## How to model human objectives?



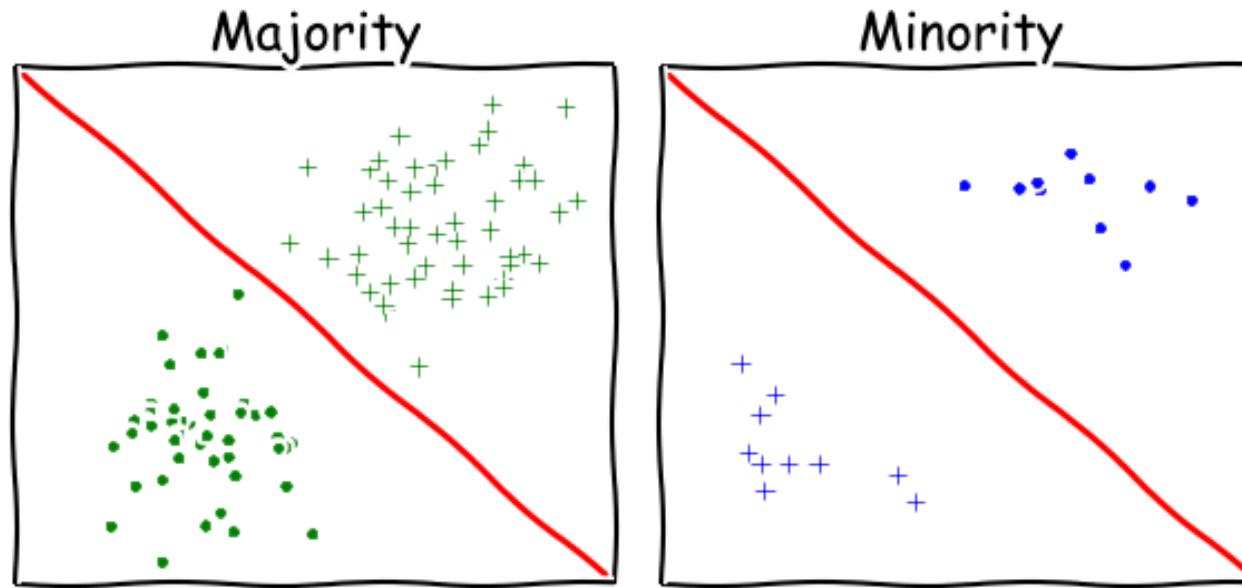
Be aware of the mismatch between human preferences and what the robot thinks are the human preferences.

# Generating fake content



Can build it  $\neq$  should build it?

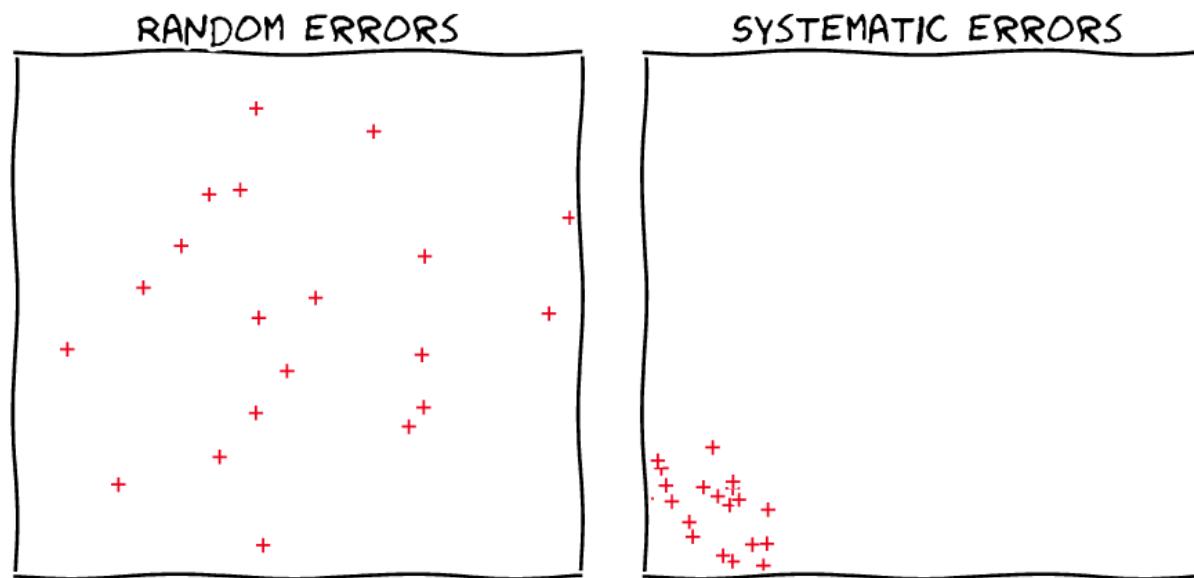
# Fairness



- Most ML training objectives will produce model accurate for majority class, at the expense of the minority one.

# Fairness

Two classifiers with 5% error:



# Fairness in criminal risk assessment

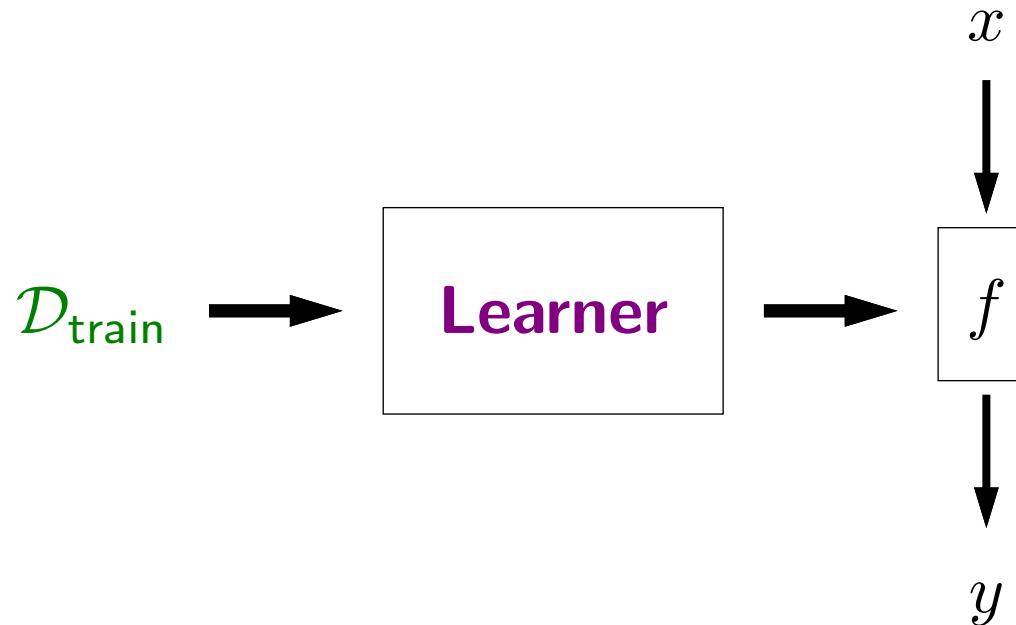
- Northpointe: COMPAS predicts criminal risk score (1-10)
- ProPublica: given that an individual did not reoffend, blacks 2x likely to be (wrongly) classified 5 or above
- Northpointe: given a risk score of 7, 60% of whites reoffended, 60% of blacks reoffended

**California just replaced cash bail with algorithms**

By [Dave Gershman](#) • September 4, 2018

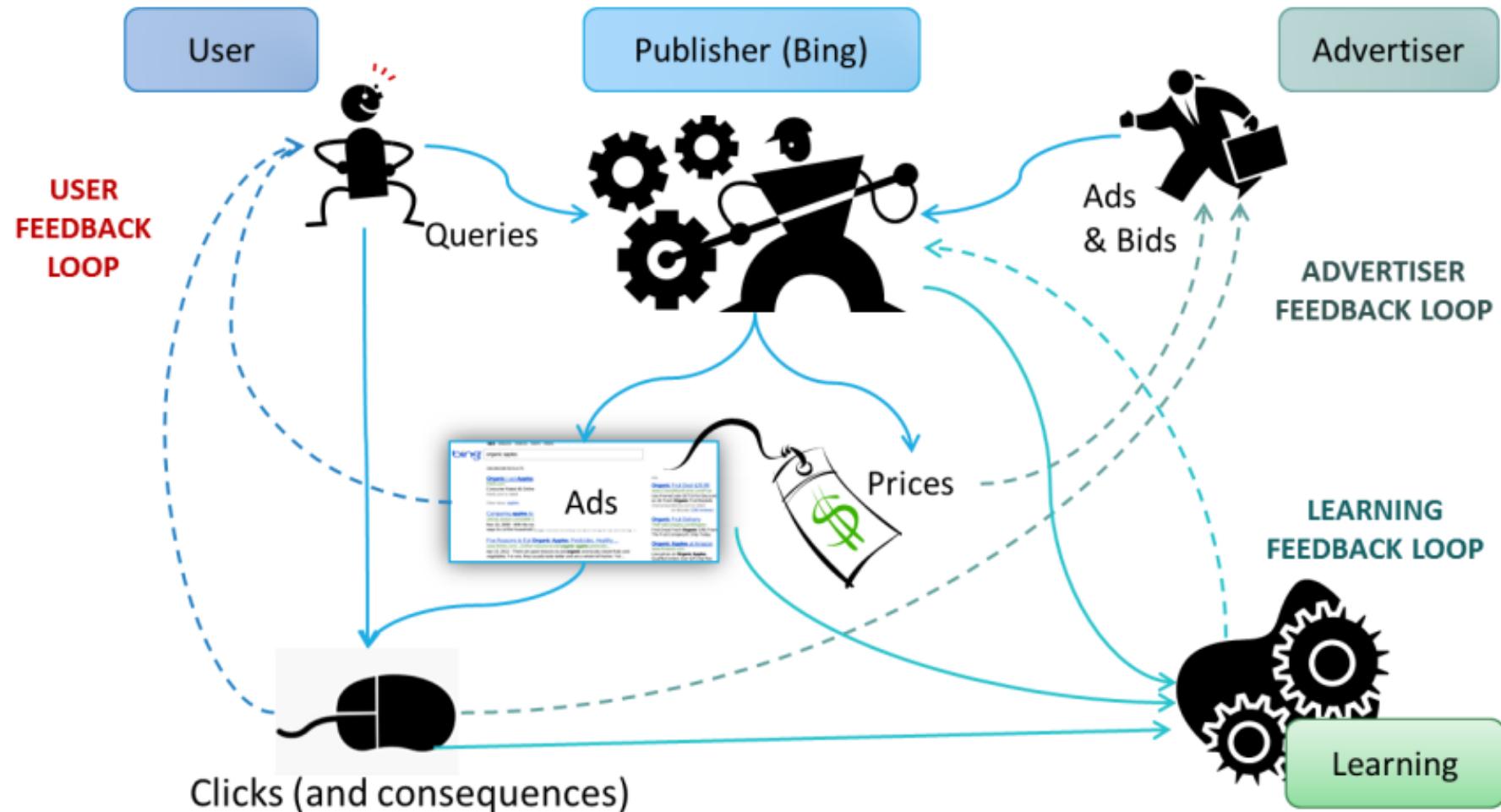


# Are algorithms neutral?



By design: picks up patterns in training data, including biases

# Feedback loops



# Privacy

- Not reveal sensitive information (income, health, communication)
- Compute average statistics (how many people have cancer?)



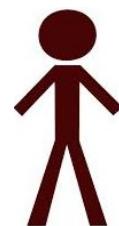
yes

no



no

no



yes

yes



no

no



no

yes

# Privacy: Randomized response

Do you have a sibling?



## Method:

- Flip two coins.
- If both heads: answer yes/no randomly
- Otherwise: answer yes/no truthfully

## Analysis:

$$\text{true-prob} = \frac{4}{3} \times (\text{observed-prob} - \frac{1}{8})$$

# Causality

Goal: figure out the effect of a treatment on survival

Data:

For untreated patients, 80% survive  
For treated patients, 30% survive

**Does the treatment help?**

Who knows? Sick people are more likely to undergo treatment...

# Interpretability versus accuracy

- For air-traffic control, threshold level of safety: probability  $10^{-9}$  for a catastrophic failure (e.g., collision) per flight hour
- Move from human designed rules to a numeric Q-value table?

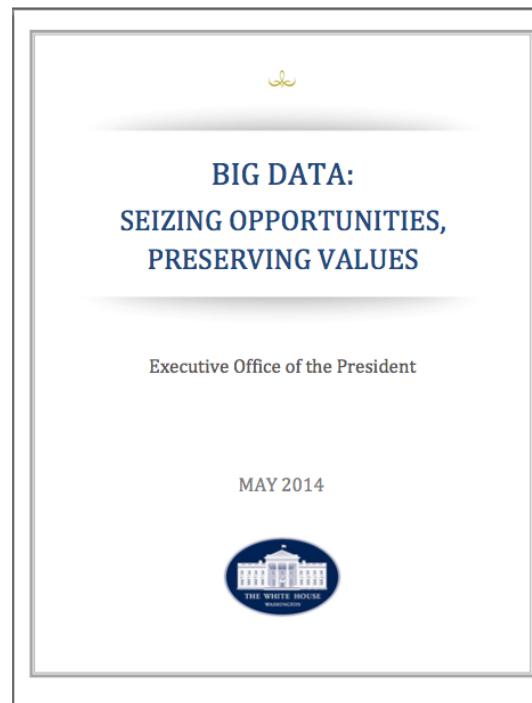
yes



## PREPARING FOR THE FUTURE

R&D Strategy .....	15
Strategy 1: Make Long-Term Investments in AI Research .....	16
Strategy 2: Develop Effective Methods for Human-AI Collaboration .....	22
Strategy 3: Understand and Address the Ethical, Legal, and Societal Implications of AI.....	26
Strategy 4: Ensure the Safety and Security of AI Systems.....	27
Strategy 5: Develop Shared Public Datasets and Environments for AI Training and Testing.....	30
Strategy 6: Measure and Evaluate AI Technologies through Standards and Benchmarks.....	32
Strategy 7: Better Understand the National AI R&D Workforce Needs.....	35





*..big data analytics have the potential to eclipse longstanding civil rights protections in how personal information is used in housing, credit, employment, health, education and the marketplace. Americans relationship with data should expand, not diminish, their opportunities..*

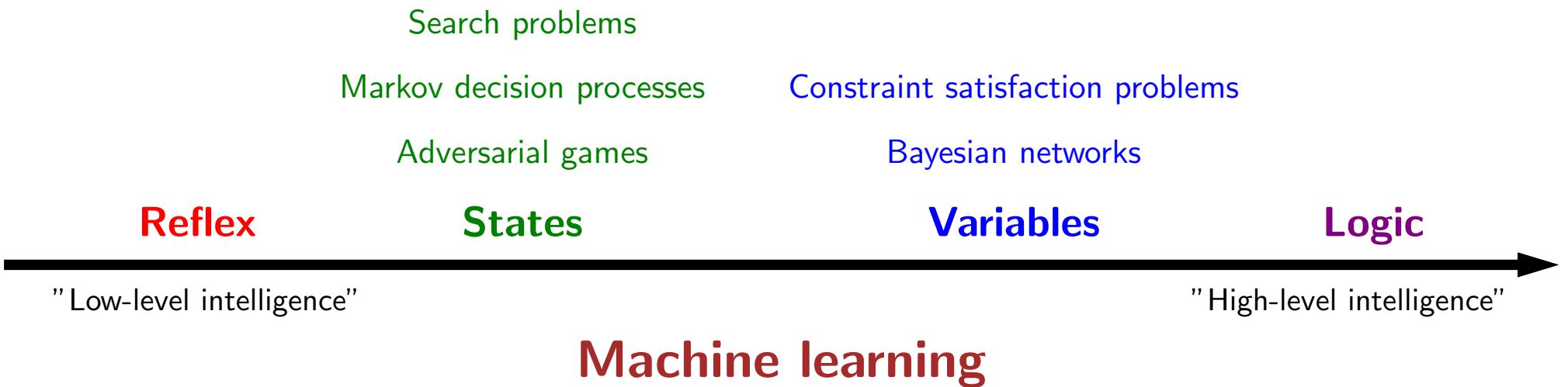
# Principles for Accountable Algorithms and a Social Impact Statement for Algorithms

*There is always a human ultimately responsible for decisions made or informed by an algorithm. "The algorithm did it" is not an acceptable excuse if algorithmic systems make mistakes or have undesired consequences, including from machine-learning processes*

# Societal and industrial impact



Enormous potential for positive impact, use responsibly!



Please fill out course evaluations on Axess.

Thanks for an exciting quarter!