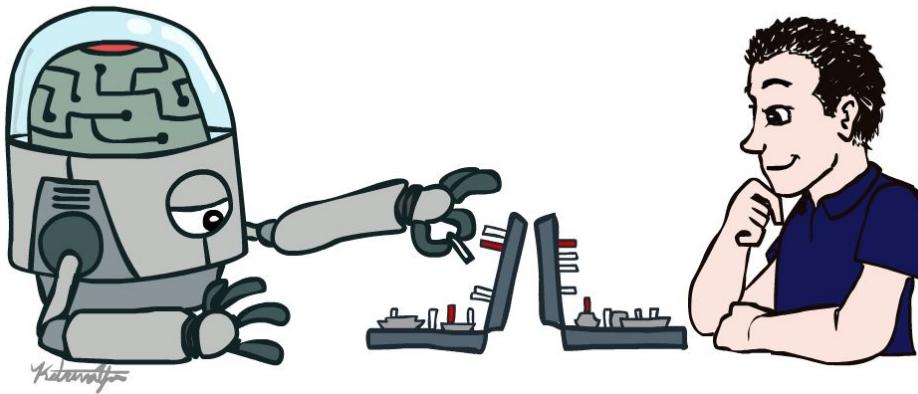
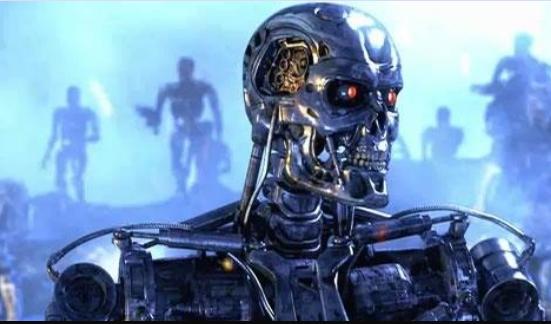


Artificial Intelligence



Sci-Fi AI?





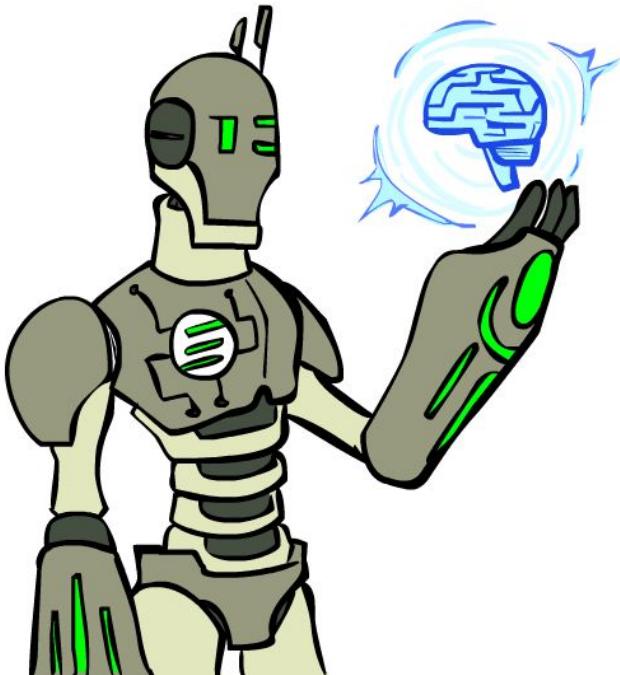
TUG
CAUTION
MAY CONTAIN
CHEMOTHERAPY DRUG

CAUTION
MAY CONTAIN
CHEMOTHERAPY DRUG



Today

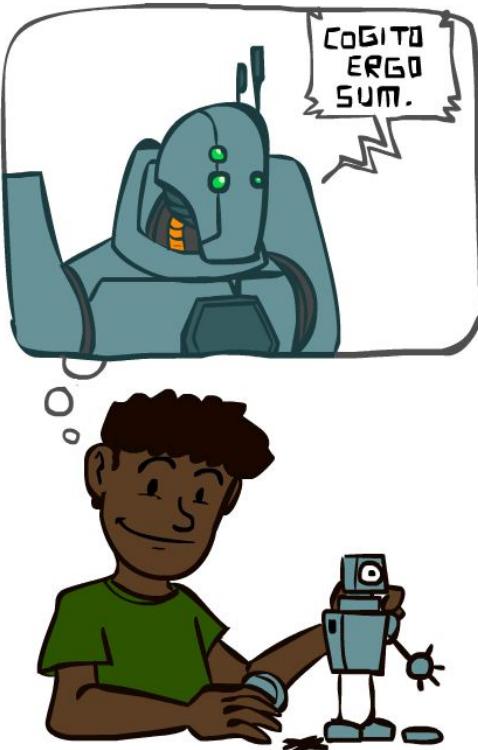
- What is this course?
- What is artificial intelligence?
- History of AI
- What can AI do?



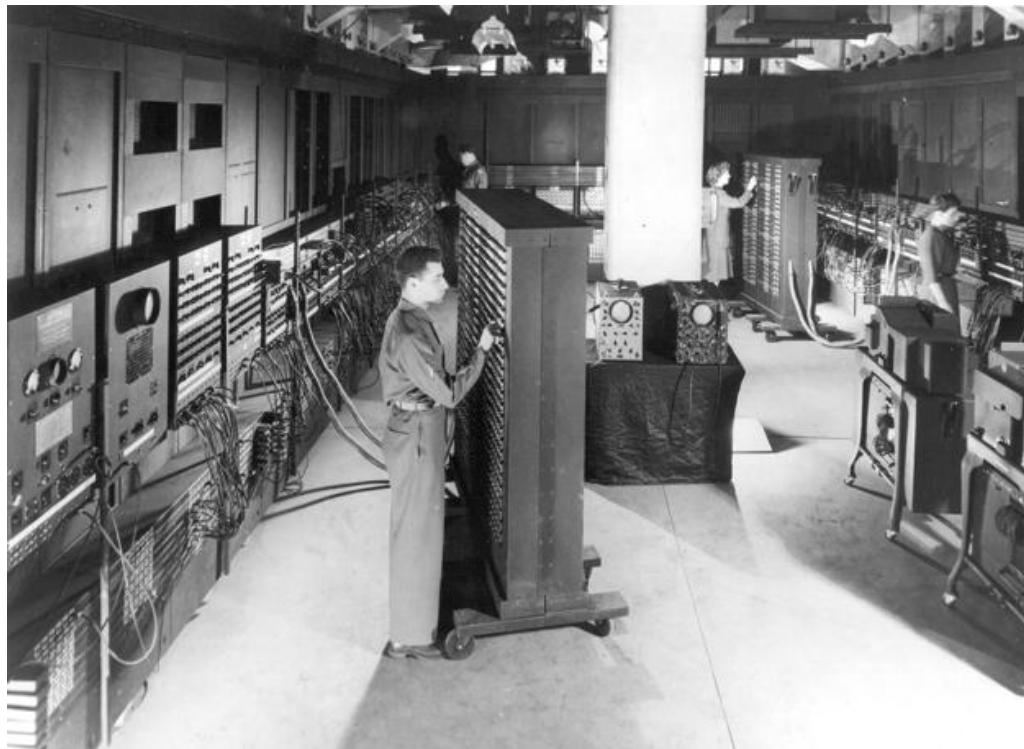
About Course

- *Text Book:*
 - *Artificial Intelligence: A Modern Approach. Third Edition.* – Stuart Russell and Peter Norvig (**AIMA**)
- *References:*
 - *Building Problem Solvers* – K.D.Forbus and J.D.Kleer
 - *Knowledge Representation and Reasoning*– R. Brachman & H. Levesque
 - *Artificial Intelligence. Third Edition* – Patrick Winston

A (Short) History of AI



ENIAC (1940s)



Early AI: 1940-1950

- 1943- McCulloch and Walter Pitts- Neural Network

Turing Test-1950s

1950: Turing asks the question....



I propose to consider the question:
“Can machines think?”

--Alan Turing, 1950

Early AI: 1950—70

- 1950s: Early AI programs, including Samuel's checkers program, Newell & Simon's Logic Theorist, Gelernter's Geometry Engine
- 1956: Dartmouth meeting: “Artificial Intelligence” adopted as a separate field
 - 4 people: John McCarthy, Marvin Minsky, Claude Shannon and Nathaniel Rochester.
 - The Dartmouth Summer Research Project on Artificial Intelligence was a 1956 summer workshop widely considered to be the founding event of artificial intelligence as a field.
- 1958: LISP by John McCarthy. Became a dominant AI language.

Early AI: 1950—70

- 1964: First Chat bot- Eliza-psychotherapist
- 1965: Robinson's complete algorithm for logical reasoning
- 1966: Shakey- the general purpose mobile robot at Stanford research institute

“Look ma, no hands!” era- John McCarthy

—

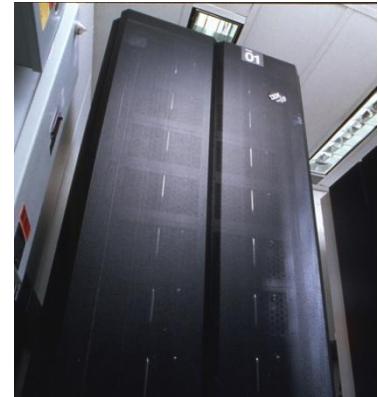
—

Knowledge-based approaches: 1970—90

- 1969—79: Early development of knowledge-based systems
 - Failure of neural network
 - Poor speech understanding
 - Failed machine translation (Russian to English)
 - Caused cancellation of govt. funds
- 1980—88: Expert systems industry booms
 - Decline of LISP
- 1988—93: Expert systems industry busts: “AI Winter”
- 1988- Backpropagation

Statistical approaches + subfield expertise: 1990—2012

- 1996: EQP (an algorithm) proves Robbins algebra are Booleans
- 1997: Deep blue wins chess vs Gary Kasparov
 - “I could feel human level intelligence across room- Gary Kasparov”
 - “Deep Blue hasn't proven anything.”
 - “If it works it is not AI”
 - But now can a human defeat the machine in a series of game??????
- Early 2000s: Resurgence of probability, focus on uncertainty
- 2005: DARPA- Driverless car (first working demonstration)
- Agents and learning systems... “AI Spring”?



Present AI: 2010s to present

- 2011: IBM Watson won the Jeopardy vs Ken Jennings and Brad Rutter
 - “I, for one, welcome our new computer overlords,” - Ken Jennings
- Big data, big compute, neural networks
 - Some re-unification of subfields
 - AI used in many industries



2016:Alpha Go



Lee Se-dol while playing with the AI

Computer Vision



"man in black shirt is playing guitar."



"construction worker in orange safety vest is working on road."



"two young girls are playing with lego toy."



"boy is doing backflip on wakeboard."



"girl in pink dress is jumping in air."



"black and white dog jumps over bar."



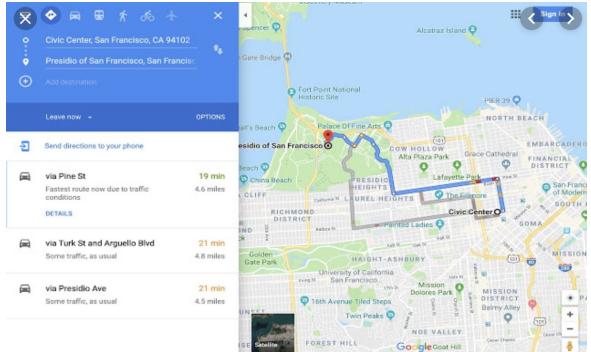
"young girl in pink shirt is swinging on swing."



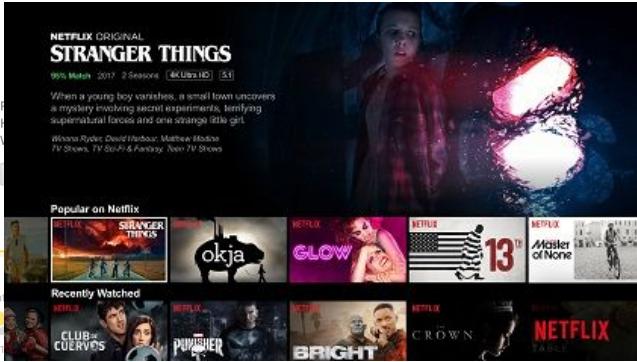
"man in blue wetsuit is surfing on wave."

Karpathy & Fei-Fei, 2015; Donahue et al., 2015; Xu et al, 2015; many more

Tools for Predictions & Decisions

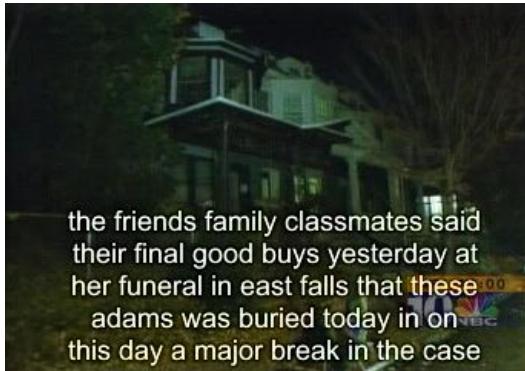


Berkeley, CA 94709
Tuesday 2:00 PM
Mostly Sunny



Natural Language

- Speech technologies (e.g. Siri)
 - Automatic speech recognition (ASR)
 - Text-to-speech synthesis (TTS)
 - Dialog systems
- Language processing technologies
 - Question answering
 - Machine translation



"Il est impossible aux journalistes de rentrer dans les régions tibétaines"

Bruno Philip, correspondant du "Monde" en Chine, estime que les journalistes de l'AFP qui ont été expulsés de la province tibétaine du Qinghai "n'étaient pas dans l'ilégalité".



Les faits Le dalaï-lama dénonce l'"enfer" imposé au Tibet depuis sa fuite, en 1959
Vidéo Anniversaire de la rébellion

"It is impossible for journalists to enter Tibetan areas"

Philip Bruno, correspondent for "World" in China, said that journalists of the AFP who have been deported from the Tibetan province of Qinghai "were not illegal."

Fact The Dalai Lama denounces the "hell" imposed since he fled Tibet in 1959
Video Anniversary of the Tibetan rebellion: China on guard

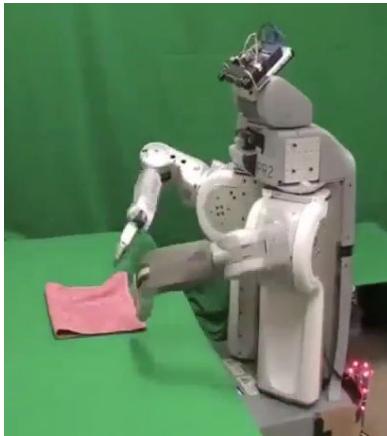


- Web search
- Text classification, spam filtering, etc...

<https://play.aidungeon.io/>

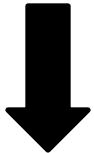
Robotics

- Robotics
 - Part mech. eng.
 - Part AI
 - Reality much harder than simulations!
- Technologies
 - Vehicles
 - Rescue
 - Help in the home
 - Lots of automation...
- In this class:
 - We ignore mechanical aspects
 - Methods for planning
 - Methods for control



Images from UC Berkeley, Boston Dynamics, RoboCup, Google

-
- If it works its not AI

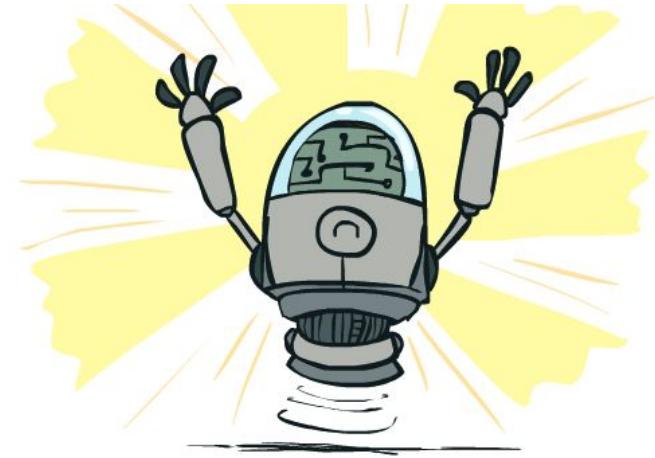


- It's all AI

What Can AI Do?

Quiz: Which of the following can be done at present?

- Play a decent game of Jeopardy?
- Win against any human at chess?
- Win against the best humans at Go?
- Play a decent game of tennis?
- Grab a particular cup and put it on a shelf?
- Unload any dishwasher in any home?
- Drive safely along the highway?
- Drive safely along Telegraph Avenue?
- Buy a week's worth of groceries on the web?
- Buy a week's worth of groceries at Berkeley Bowl?
- Discover and prove a new mathematical theorem?
- Perform a surgical operation?
- Translate spoken Chinese into spoken English in real time?
- Write an intentionally funny story?



How so much increase???

- DATA
- Computational Power
- Algorithms

Artificial Intelligence

- AI: What is the nature of intelligent thought?
- What is intelligence????
 - *Dictionary meaning: capacity for learning, reasoning, understanding and similar for of **mental activity***
- Ability to perceive and act in the world
- Reasoning: Proving theorems, Medical Diagnosis
- Planning: Take decisions
- Learning and Adaptation: Recommend Movies, learn traffic Patterns
- Understanding: Text, speech, visual scene

-
- Are human intelligent??????
 - Are human *always* intelligent???????????
 - Can non-human behavior be intelligent??????

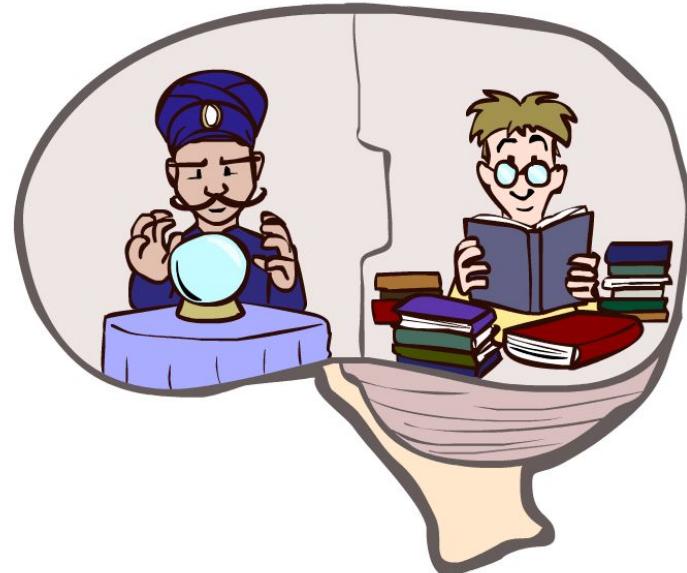
What is AI?

The science of making machines that:

- Acting humanly
 - Turning Test
 - Do you want machine to make human like errors????
- Thinking humanly
 - Cognitive modeling
 - (very hard to understand how human think)
 - Intelligent tutor
 - Elderly healthcare bot
- Thinking Rationally
 - Laws of thought (Logic and reasoning, inferences and conclusion)
 - Purposeful thinking?
- Acting Rationally
 - Rational behavior: Doing right thing
 - What is rationality??

What About the Brain?

- Brains (human minds) are very good at making rational decisions, but not perfect
- Brains aren't as modular as software, so hard to reverse engineer!
- "Brains are to intelligence as wings are to flight"
- Lessons learned from the brain: memory (data) and simulation (computation) are key to decision making



Rational Agents

- An agents should strive **to do right thing**, based on what it can perceive and the actions it can perform.
- The right action is the one that will cause the agent to be the most successful
- Performance measure: An objective criterion for success of an agent's behavior
- For example: performance measure of a vacuum cleaner agent could be the amount of dirt cleaned up, time taken, electricity consumed, etc.

Ideal Rational Agents

- “For each possible percept sequence, does whatever action is **expected to maximize its performance measure** on the basis of **evidence perceived** so far and **built in knowledge**”

- RATIONALITY vs OMNISCIENCE???
- Acting in order to obtain information

Rational Decisions

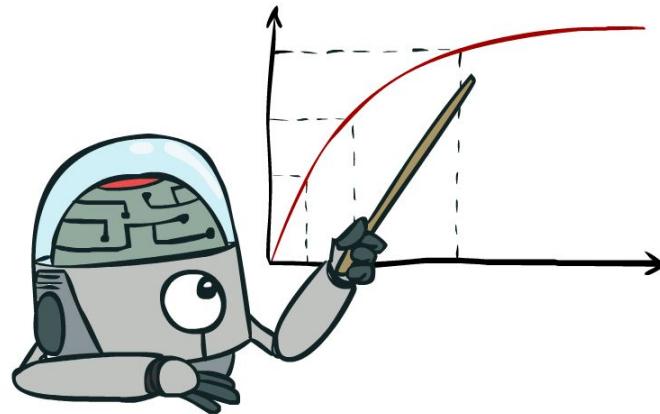
We'll use the term **rational** in a very specific, technical way:

- Rational: maximally achieving pre-defined goals
- Rationality only concerns what decisions are made
(not the thought process behind them)
- Goals are expressed in terms of the **utility** of outcomes
- Being rational means **maximizing your expected utility**

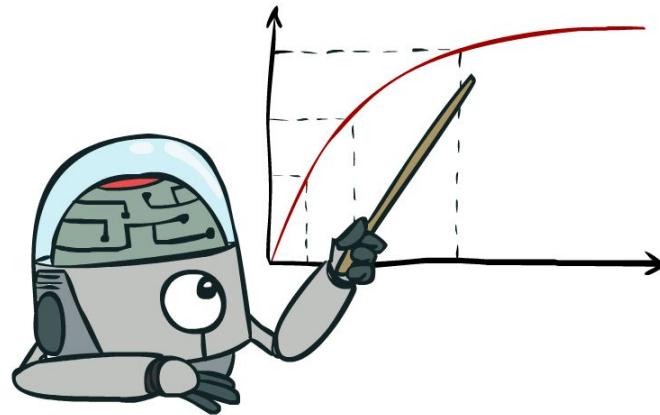
A better title for this course would be:

Computational Rationality

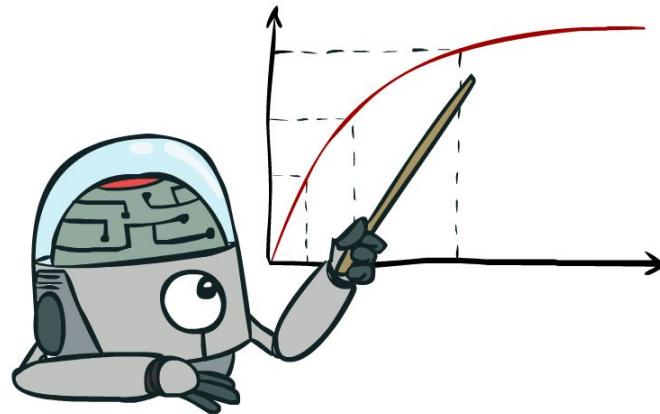
Maximize Your Expected Utility



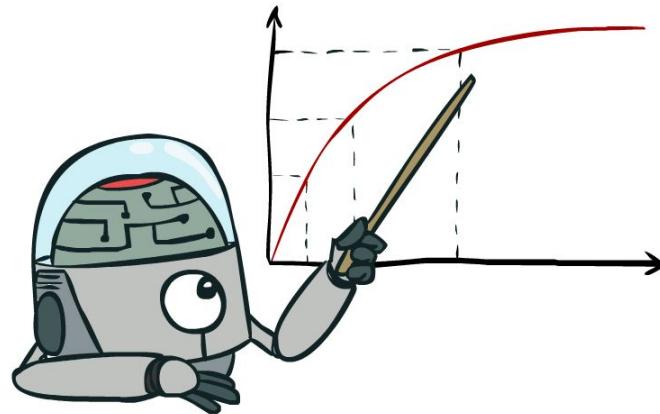
Maximize Your Expected Utility



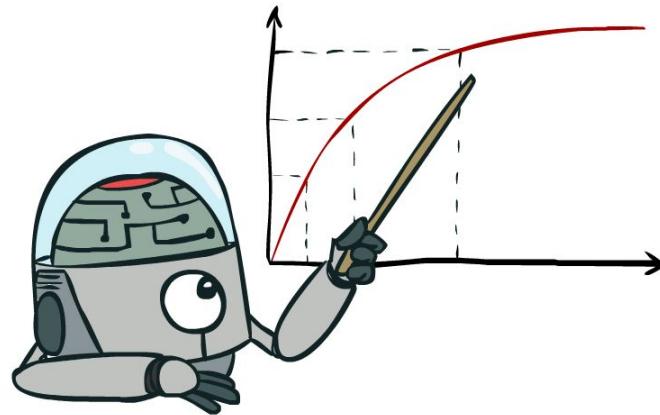
Maximize Your Expected Utility



Maximize Your Expected Utility

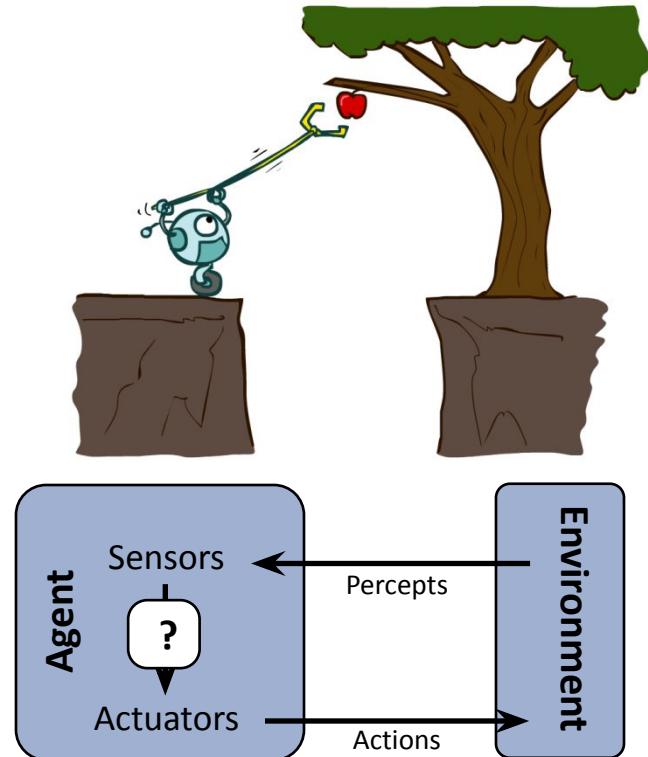


Maximize Your Expected Utility



Designing Rational Agents

- An **agent** is an entity that *perceives* and *acts*.
- A **rational agent** selects actions that maximize its (expected) **utility**.
- Characteristics of the **percepts**, **environment**, and **action space** dictate techniques for selecting rational actions
- **This course is about:**
 - General AI techniques for a variety of problem types
 - Learning to recognize when and how a new problem can be solved with an existing technique



INTELLIGENT AGENTS

CHAPTER 2

Reminders

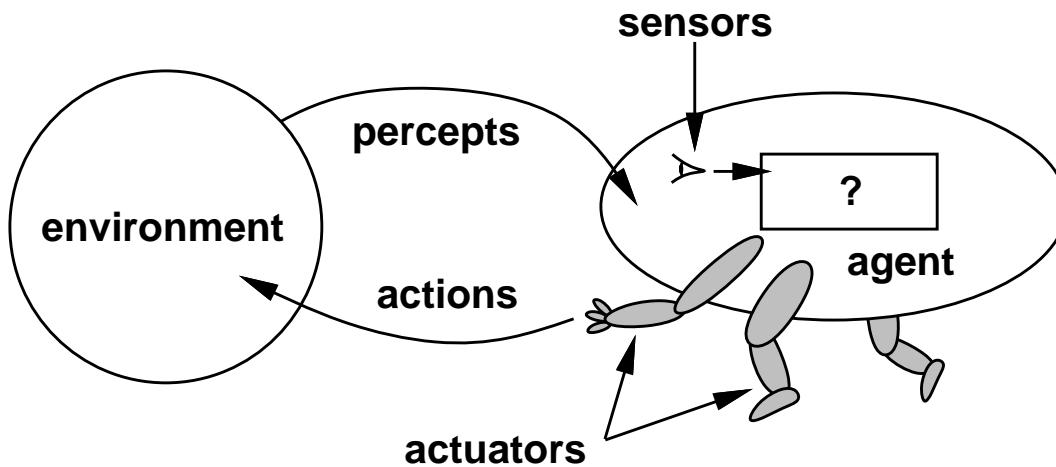
Assignment 0 (lisp refresher) due 1/28

Lisp/emacs/AIMA tutorial: 11-1 today and Monday, 271 Soda

Outline

- ◊ Agents and environments
- ◊ Rationality
- ◊ PEAS (Performance measure, Environment, Actuators, Sensors)
- ◊ Environment types
- ◊ Agent types

Agents and environments



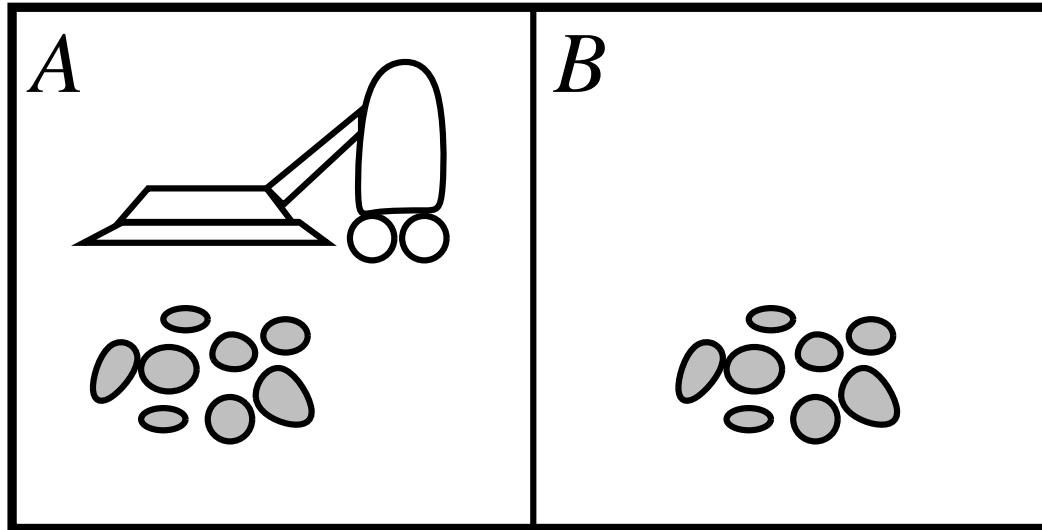
Agents include humans, robots, softbots, thermostats, etc.

The agent function maps from percept histories to actions:

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

The agent program runs on the physical architecture to produce f

Vacuum-cleaner world



Percepts: location and contents, e.g., [*A*, *Dirty*]

Actions: *Left*, *Right*, *Suck*, *NoOp*

A vacuum-cleaner agent

Percept sequence	Action
$[A, Clean]$	<i>Right</i>
$[A, Dirty]$	<i>Suck</i>
$[B, Clean]$	<i>Left</i>
$[B, Dirty]$	<i>Suck</i>
$[A, Clean], [A, Clean]$	<i>Right</i>
$[A, Clean], [A, Dirty]$	<i>Suck</i>
:	:

```
function REFLEX-VACUUM-AGENT( [location,status] ) returns an action
    if status = Dirty then return Suck
    else if location = A then return Right
    else if location = B then return Left
```

What is the **right** function?
Can it be implemented in a small agent program?

Rationality

Fixed **performance measure** evaluates the **environment sequence**

- one point per square cleaned up in time T ?
- one point per clean square per time step, minus one per move?
- penalize for $> k$ dirty squares?

A **rational agent** chooses whichever action maximizes the **expected value** of the performance measure **given the percept sequence to date**

Rational \neq omniscient

- percepts may not supply all relevant information

Rational \neq clairvoyant

- action outcomes may not be as expected

Hence, rational \neq successful

Rational \Rightarrow exploration, learning, autonomy

PEAS

To design a rational agent, we must specify the **task environment**

Consider, e.g., the task of designing an automated taxi:

Performance measure??

Environment??

Actuators??

Sensors??

PEAS

To design a rational agent, we must specify the **task environment**

Consider, e.g., the task of designing an automated taxi:

Performance measure?? safety, destination, profits, legality, comfort, . . .

Environment?? US streets/freeways, traffic, pedestrians, weather, . . .

Actuators?? steering, accelerator, brake, horn, speaker/display, . . .

Sensors?? video, accelerometers, gauges, engine sensors, keyboard, GPS, . . .

Internet shopping agent

Performance measure??

Environment??

Actuators??

Sensors??

Internet shopping agent

Performance measure?? price, quality, appropriateness, efficiency

Environment?? current and future WWW sites, vendors, shippers

Actuators?? display to user, follow URL, fill in form

Sensors?? HTML pages (text, graphics, scripts)

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u> <u>Deterministic??</u> <u>Episodic??</u> <u>Static??</u> <u>Discrete??</u> <u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>				
<u>Episodic??</u>				
<u>Static??</u>				
<u>Discrete??</u>				
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>				
<u>Static??</u>				
<u>Discrete??</u>				
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>				
<u>Discrete??</u>				
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>	Yes	Semi	Semi	No
<u>Discrete??</u>				
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>	Yes	Semi	Semi	No
<u>Discrete??</u>	Yes	Yes	Yes	No
<u>Single-agent??</u>				

Environment types

	Solitaire	Backgammon	Internet shopping	Taxi
<u>Observable??</u>	Yes	Yes	No	No
<u>Deterministic??</u>	Yes	No	Partly	No
<u>Episodic??</u>	No	No	No	No
<u>Static??</u>	Yes	Semi	Semi	No
<u>Discrete??</u>	Yes	Yes	Yes	No
<u>Single-agent??</u>	Yes	No	Yes (except auctions)	No

The environment type largely determines the agent design

The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent

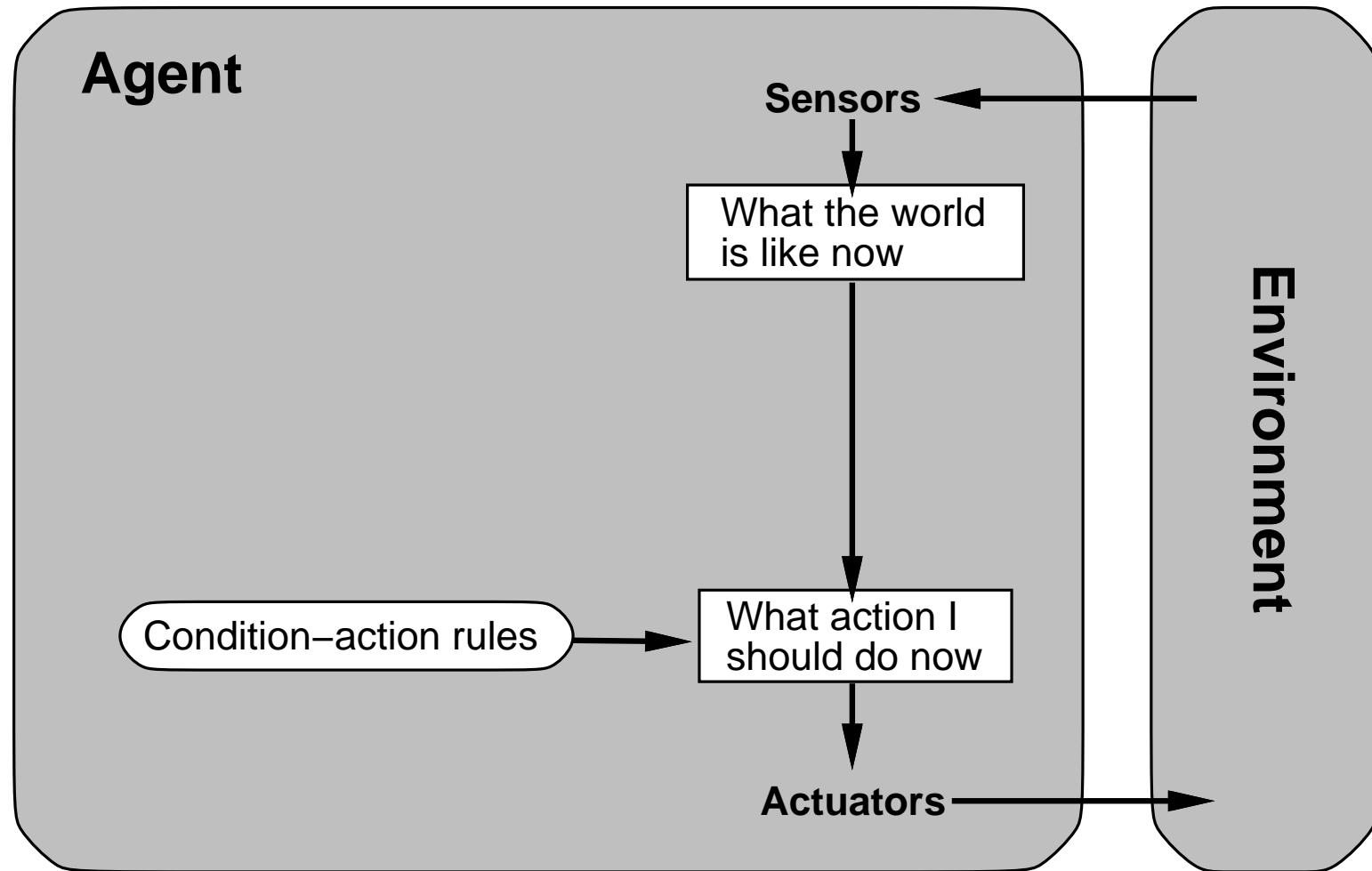
Agent types

Four basic types in order of increasing generality:

- simple reflex agents
- reflex agents with state
- goal-based agents
- utility-based agents

All these can be turned into learning agents

Simple reflex agents



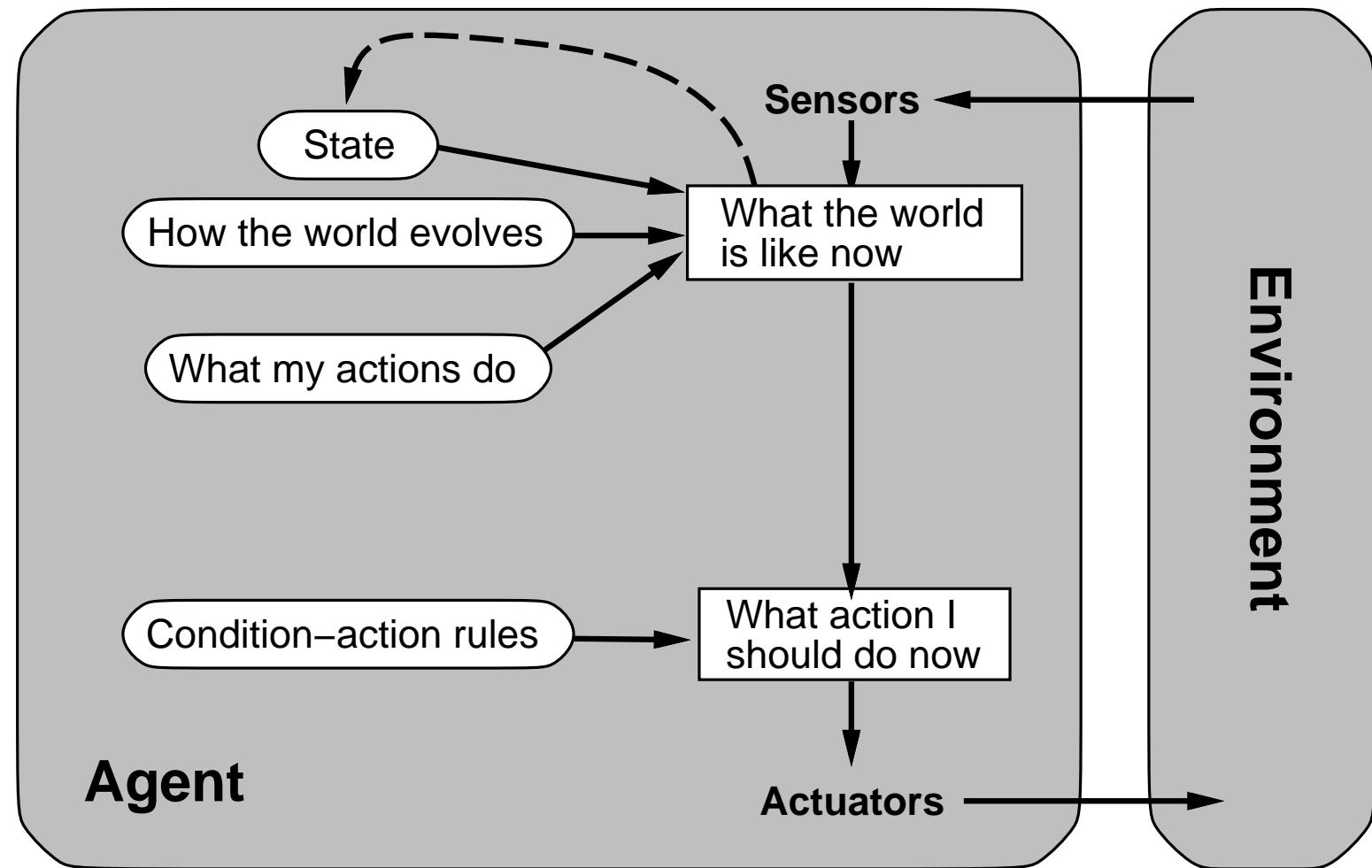
Example

```
function REFLEX-VACUUM-AGENT( [location,status]) returns an action
  if status = Dirty then return Suck
  else if location = A then return Right
  else if location = B then return Left
```

```
(setq joe (make-agent :name 'joe :body (make-agent-body)
                      :program (make-reflex-vacuum-agent-program))

(defun make-reflex-vacuum-agent-program ()
  #'(lambda (percept)
      (let ((location (first percept)) (status (second percept)))
        (cond ((eq status 'dirty) 'Suck)
              ((eq location 'A) 'Right)
              ((eq location 'B) 'Left))))
```

Reflex agents with state

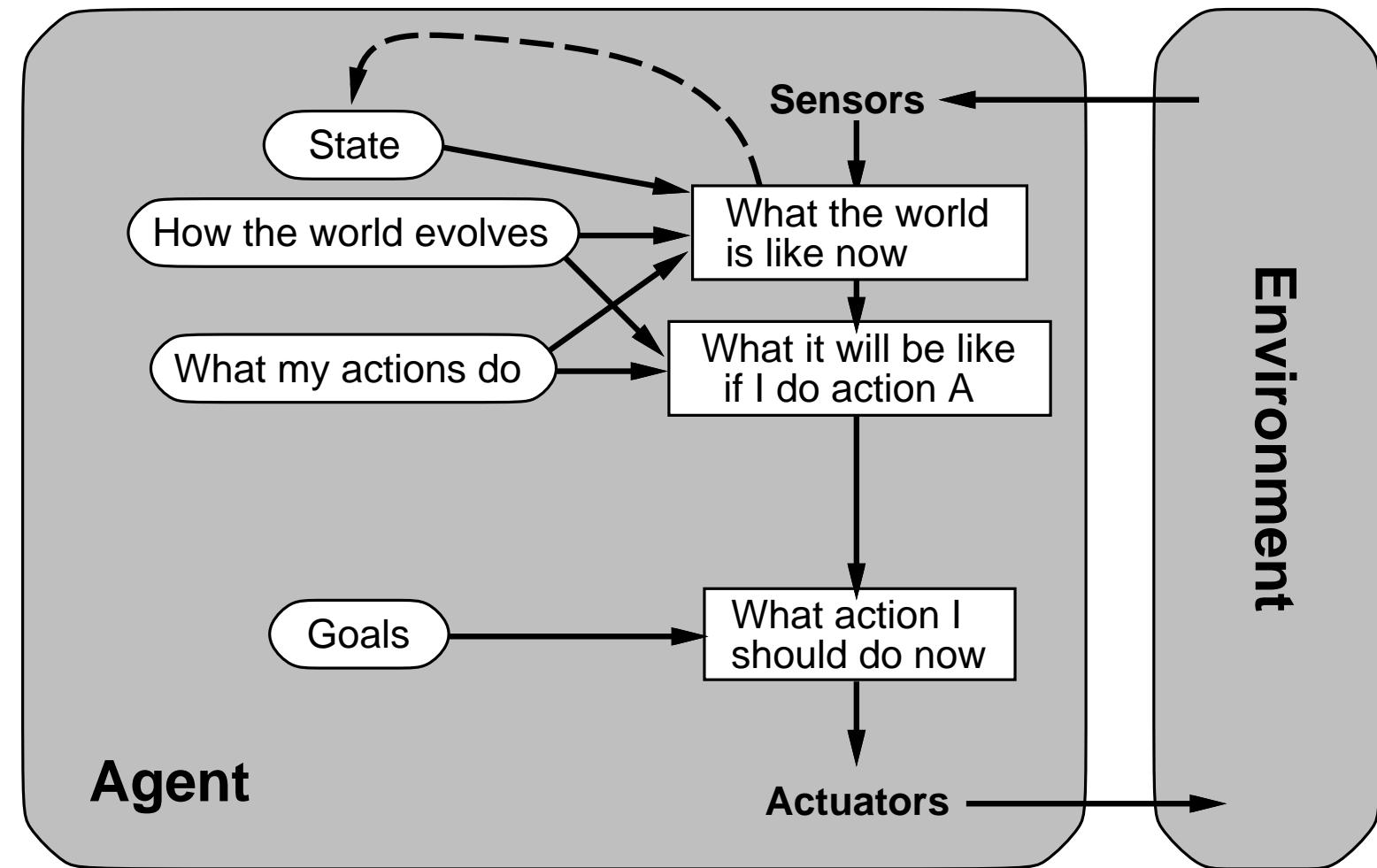


Example

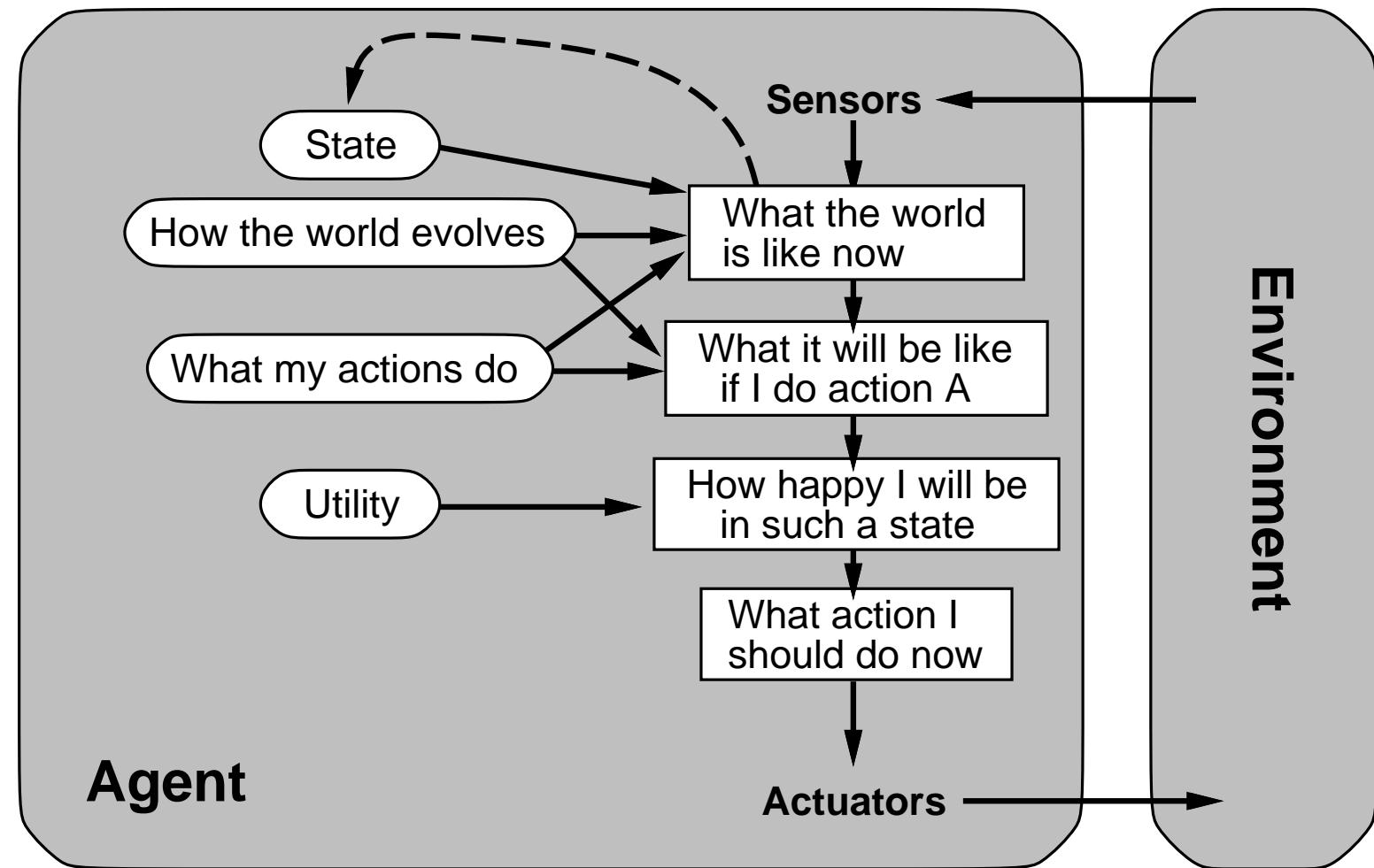
```
function REFLEX-VACUUM-AGENT( [location,status]) returns an action
static: last_A, last_B, numbers, initially ∞
    if status = Dirty then ...
```

```
(defun make-reflex-vacuum-agent-with-state-program ()
  (let ((last-A infinity) (last-B infinity))
    #'(lambda (percept)
        (let ((location (first percept)) (status (second percept)))
          (incf last-A) (incf last-B)
          (cond
            ((eq status 'dirty)
             (if (eq location 'A) (setq last-A 0) (setq last-B 0))
             'Suck)
            ((eq location 'A) (if (> last-B 3) 'Right 'NoOp))
            ((eq location 'B) (if (> last-A 3) 'Left 'NoOp)))))))
```

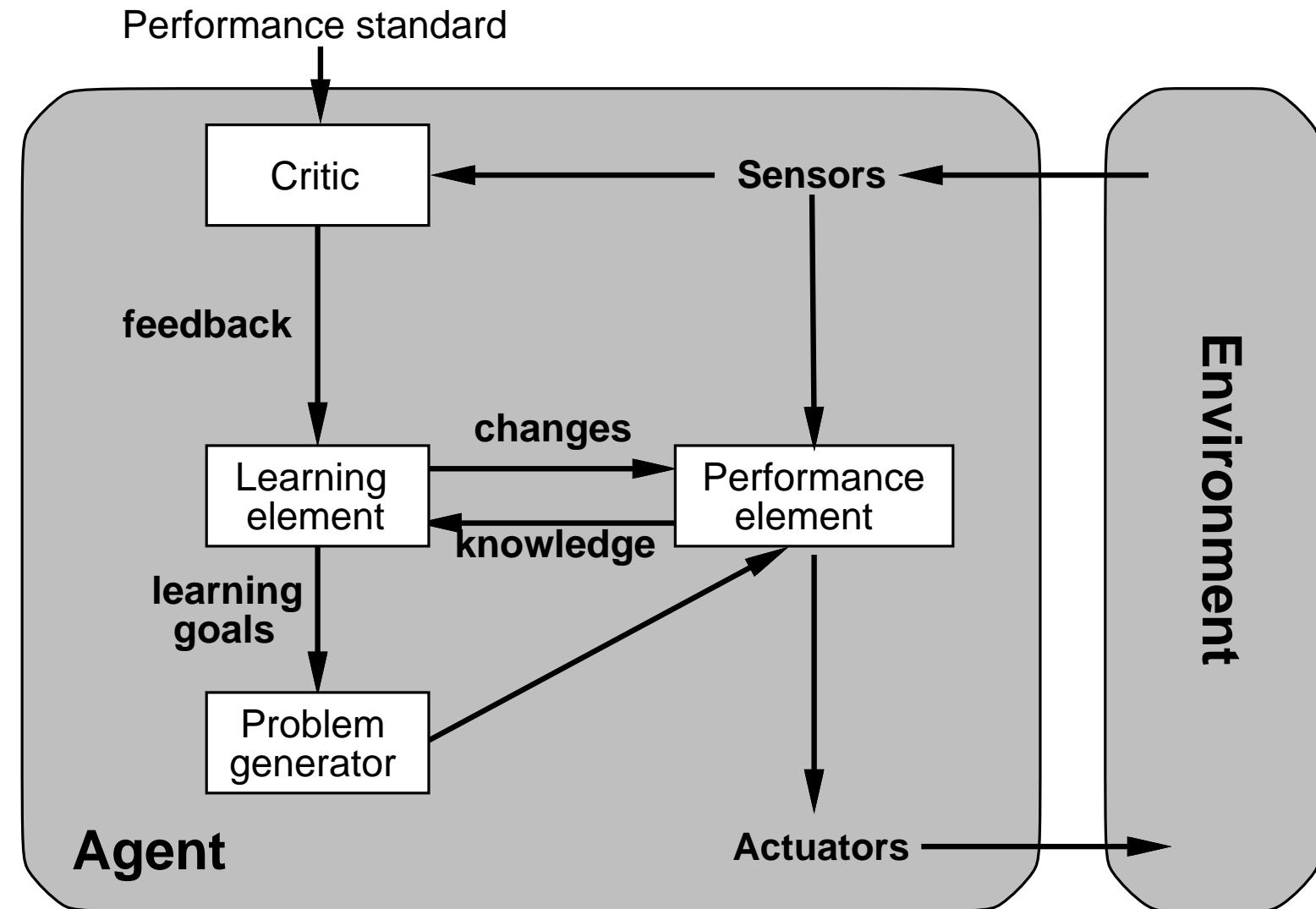
Goal-based agents



Utility-based agents



Learning agents



Summary

Agents interact with environments through **actuators** and **sensors**

The agent function describes what the agent does in all circumstances

The performance measure evaluates the environment sequence

A perfectly rational agent maximizes expected performance

Agent programs implement (some) agent functions

PEAS descriptions define task environments

Environments are categorized along several dimensions:

observable? deterministic? episodic? static? discrete? single-agent?

Several basic agent architectures exist:

reflex, reflex with state, goal-based, utility-based

Problem solving and search

AI

Problem Solving Agents

- When the correct action to take is not immediately obvious, an agent may need to plan ahead to consider a **sequence of actions** that form a ***path to a goal state***.
- Such agents is called a problem-solving agent, and the computational process it undertakes is called **search**.
- Problem Solving Phases (four-phase problem-solving process):
 - **Goal formulation:** Goals organize behaviour by limiting the objectives and hence the actions to be considered.
 - **Problem formulation:** The agent devises a description of the states and actions necessary to reach the goal.
 - **Search:** Before taking any action in the real world, the agent simulates sequences of actions in its model, searching until it finds a sequence of actions that reaches the goal.
 - Such a sequence is called a solution.
 - **Execution:** The agent can now execute the actions in the solution, one at a time.

Example: Romania

On holiday in Romania; currently in Arad.
Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

- Credits: AIMA by Peter N.

Example: Romania

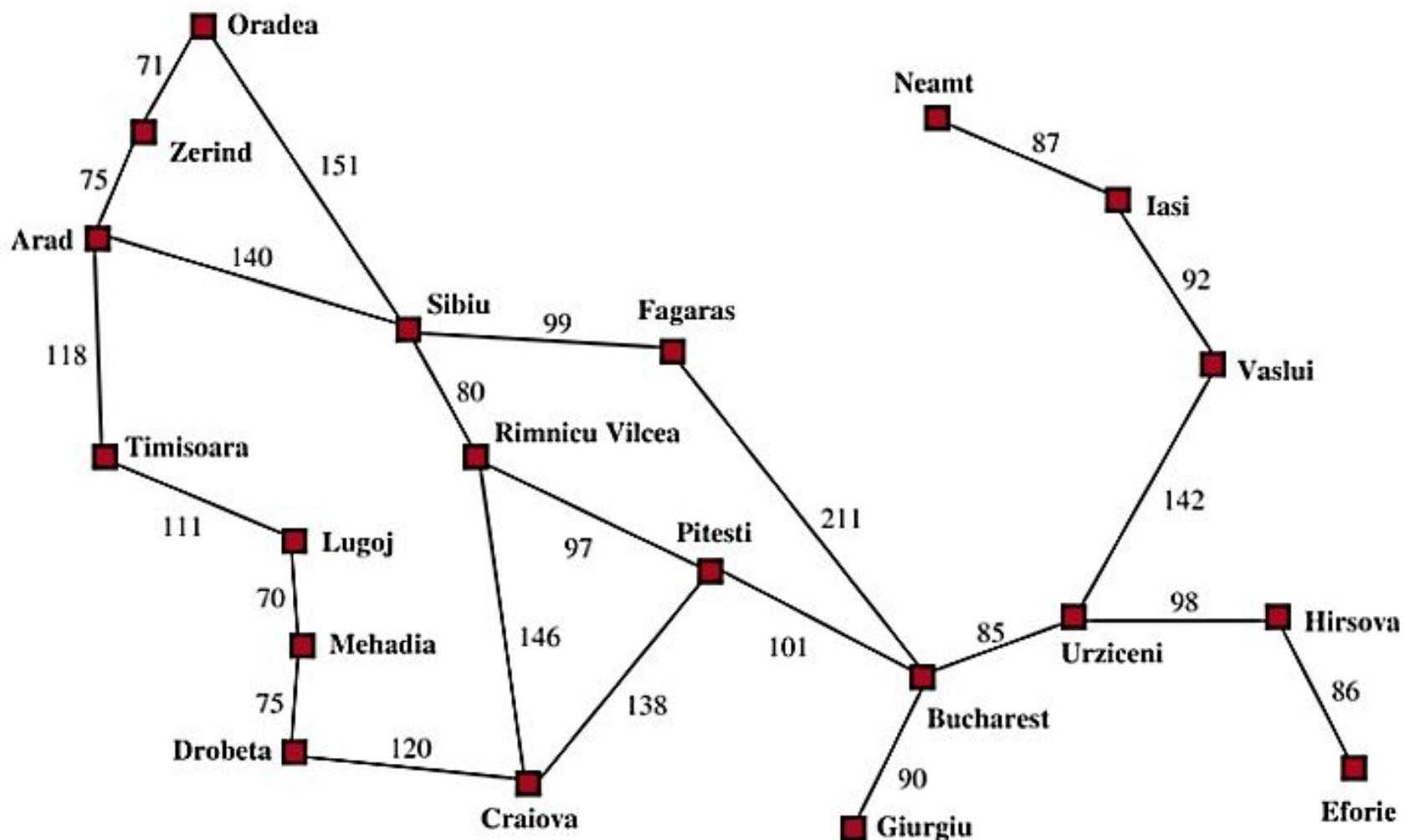


Figure 3.1 A simplified road map of part of Romania, with road distances in miles.

- Credits: AIMA by Peter N.

Search problems and solutions

- A search problem can be defined formally as follows:
 - A set of possible **states** that the environment can be in. We call this the **state space**.
 - The **initial state** that the agent starts in. For example: Arad.
 - A set of **one or more goal states**.
 - The **actions** available to the agent.
 - » ACTIONS (Arad) = {ToSibiu, ToTimisoara, ToZerind }
 - A **transition model**, which describes what each action does.
 - An **action cost function**, denoted by ACTION-COST(s, a, s'').

Search problems and solutions

- A sequence of actions forms a **path**, and a **solution** is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.
- The state space can be represented as a **graph** in which the vertices are states and the directed edges between them are actions.

Formulating problems

- Our formulation of the problem of getting to goal is a **model**—an abstract mathematical description.
- The process of removing detail from a representation is called **abstraction**.
- A good problem formulation has the right level of detail.

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

Problem types

Deterministic, fully observable \Rightarrow single-state problem

Agent knows exactly which state it will be in; solution is a sequence

Non-observable \Rightarrow conformant problem

Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable \Rightarrow contingency problem

percepts provide new information about current state

solution is a contingent plan or a policy

often interleave search, execution

Unknown state space \Rightarrow exploration problem ("online")

Example: vacuum world

Single-state, start in #5. Solution??

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}. Solution??

[*Right, Suck, Left, Suck*]

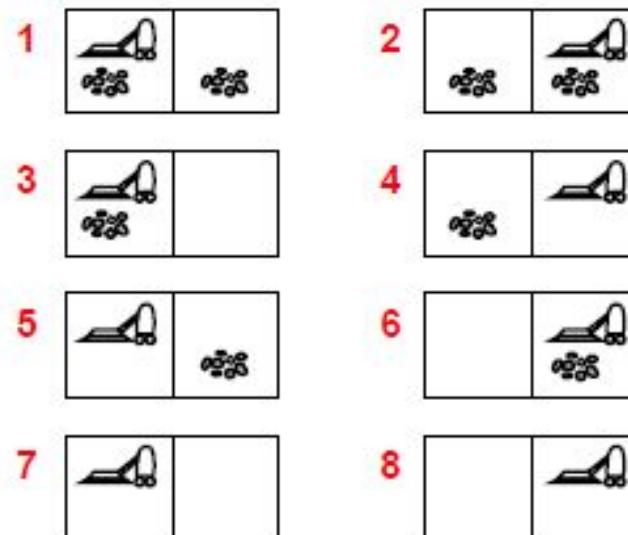
Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet

Local sensing: dirt, location only.

Solution??

[*Right, if dirt then Suck*]



Single-state problem formulation

A **problem** is defined by four items:

initial state e.g., "at Arad"

successor function $S(x)$ = set of action-state pairs

e.g., $S(Arad) = \{(Arad \rightarrow Zerind, Zerind), \dots\}$

goal test, can be

explicit, e.g., x = "at Bucharest"

implicit, e.g., $NoDirt(x)$

path cost (additive)

e.g., sum of distances, number of actions executed, etc.

$c(x, a, y)$ is the step cost, assumed to be ≥ 0

A **solution** is a sequence of actions

leading from the initial state to a goal state

Selecting a state space

Real world is absurdly complex

⇒ state space must be **abstracted** for problem solving

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions

e.g., "Arad → Zerind" represents a complex set
of possible routes, detours, rest stops, etc.

For guaranteed realizability, **any** real state "in Arad"

must get to **some** real state "in Zerind"

(Abstract) solution =

set of real paths that are solutions in the real world

Each abstract action should be "easier" than the original problem!

Example: Vacuum world state-space graph

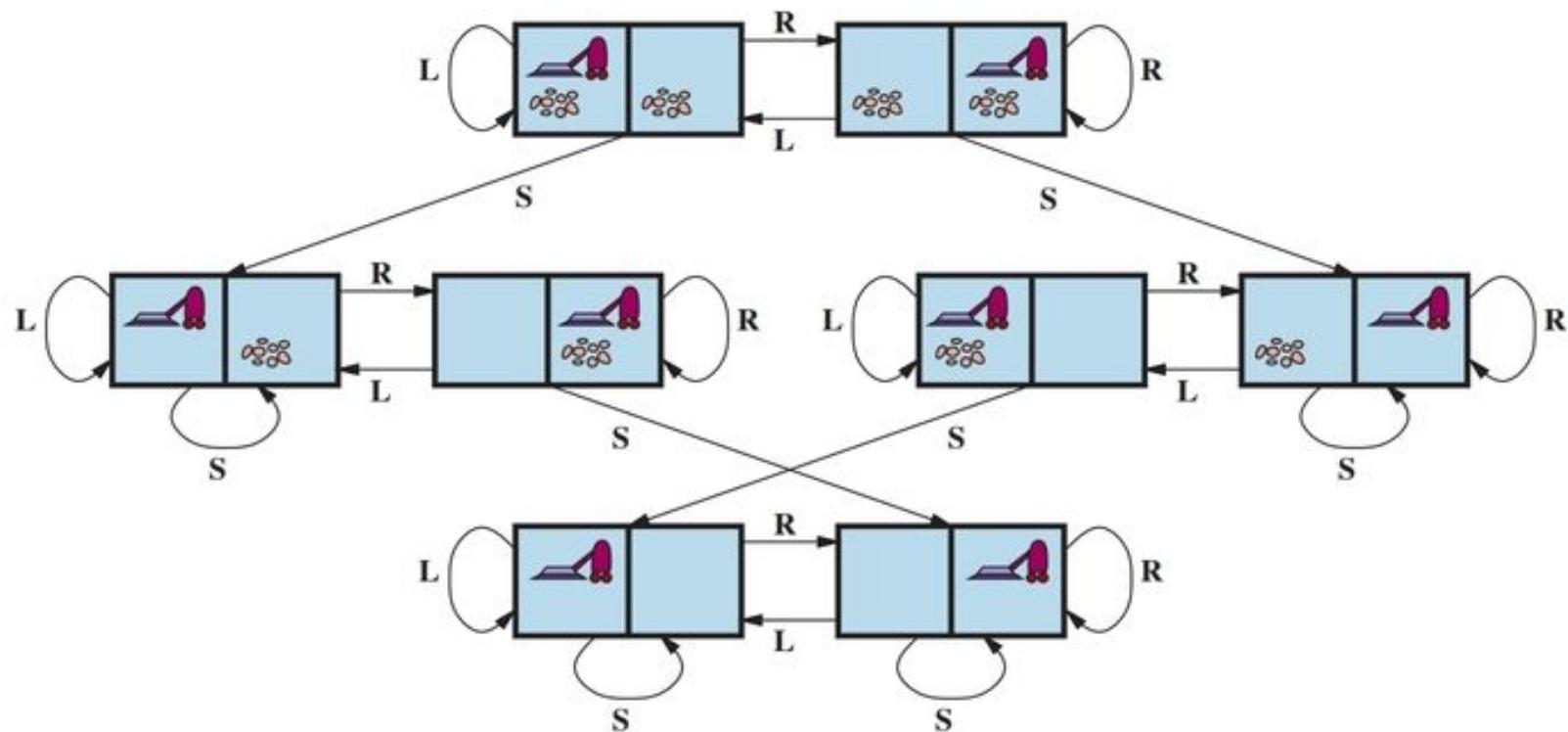
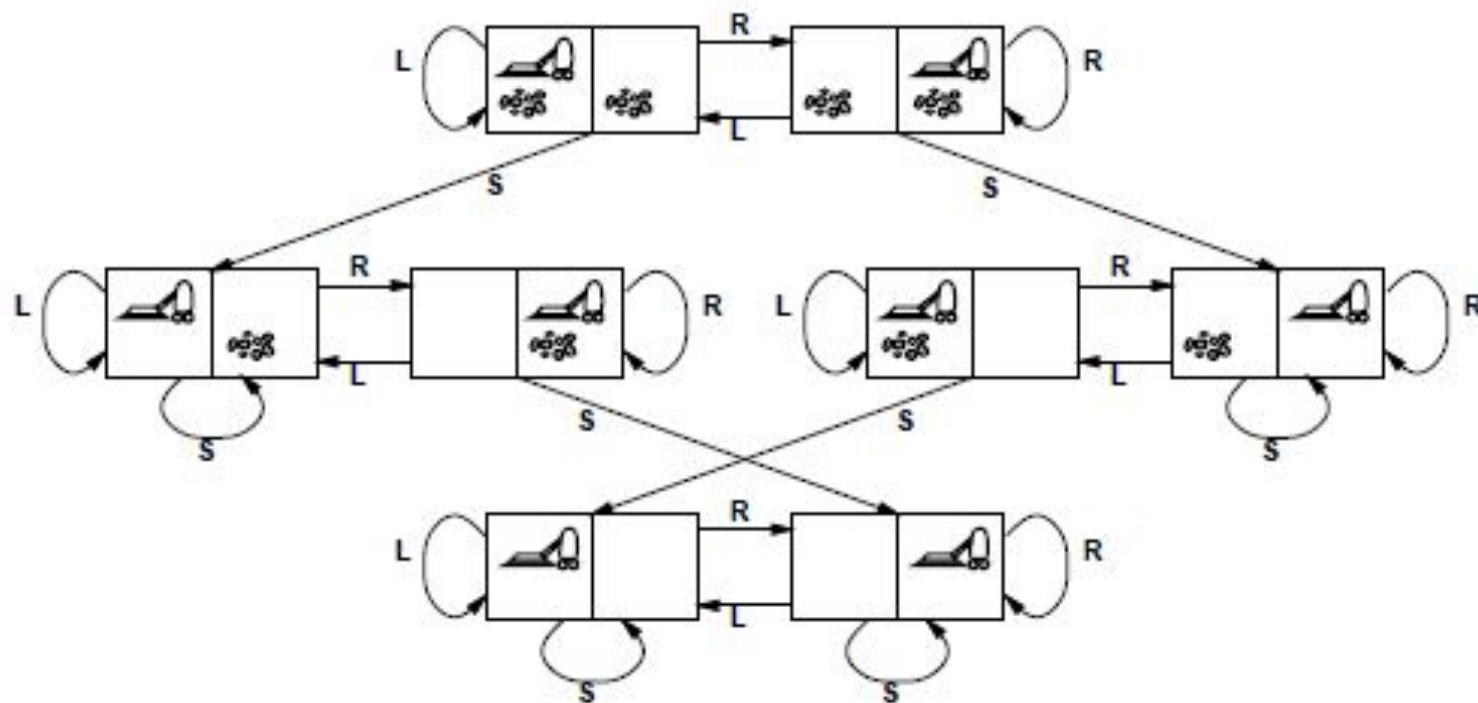


Figure 3.2 The state-space graph for the two-cell vacuum world. There are 8 states and three actions for each state: L = Left, R = Right, S = Suck.

Example: vacuum world state space graph



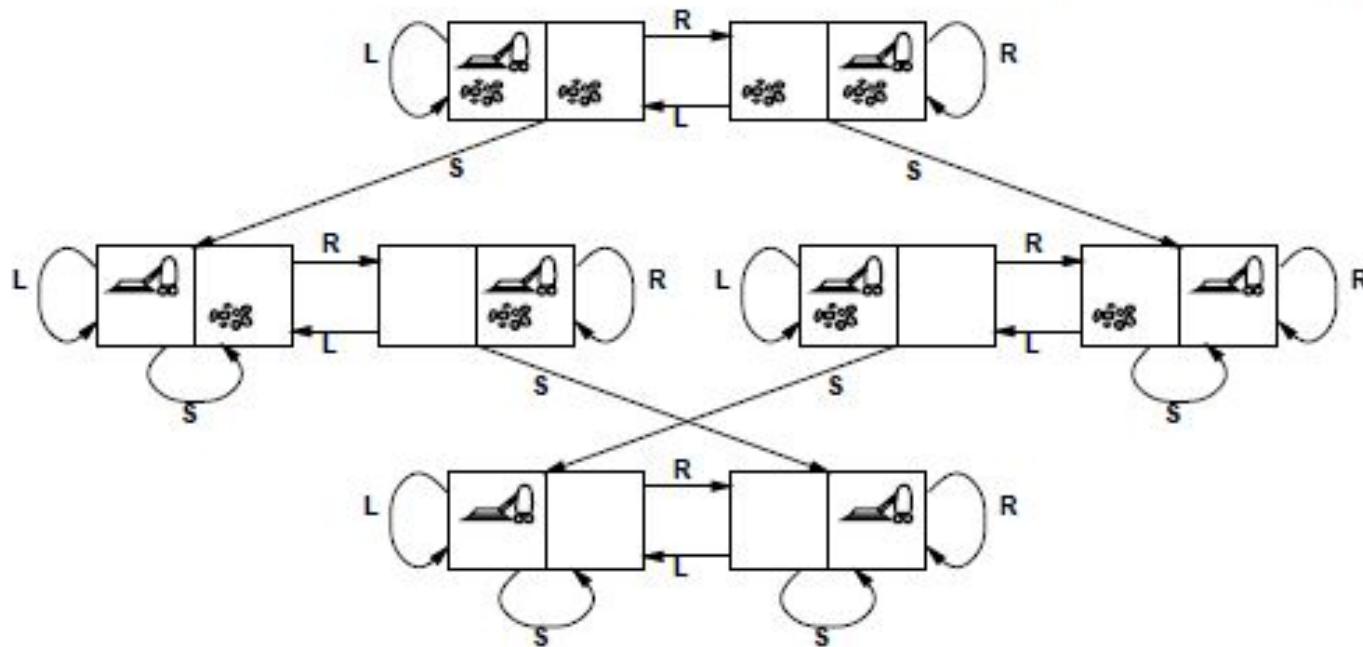
states??

actions??

goal test??

path cost??

Example: vacuum world state space graph



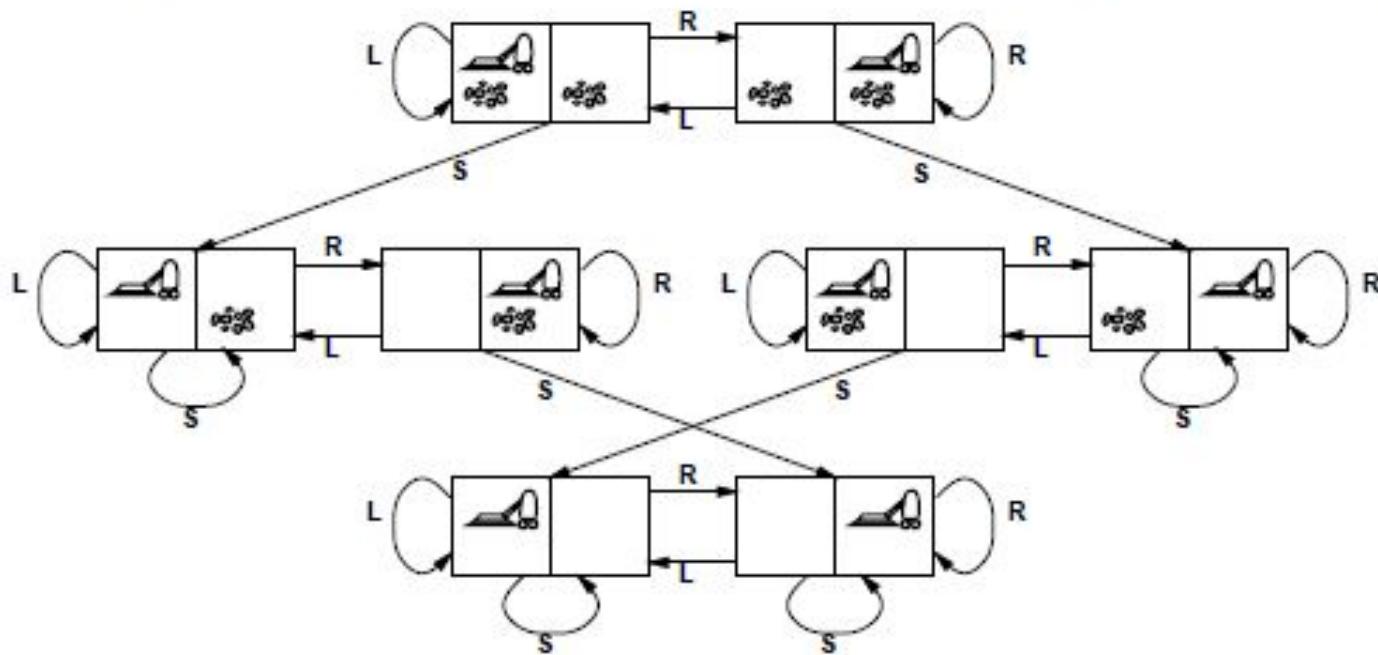
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??

goal test??

path cost??

Example: vacuum world state space graph



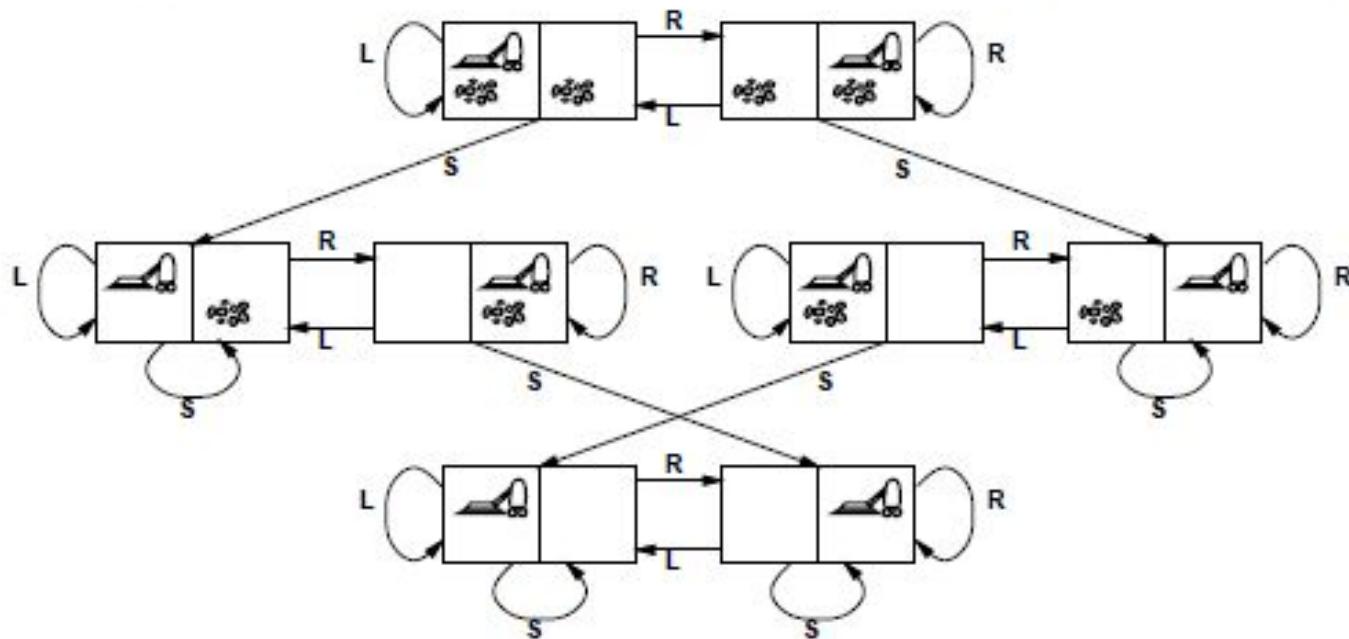
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left, Right, Suck, NoOp*

goal test??

path cost??

Example: vacuum world state space graph



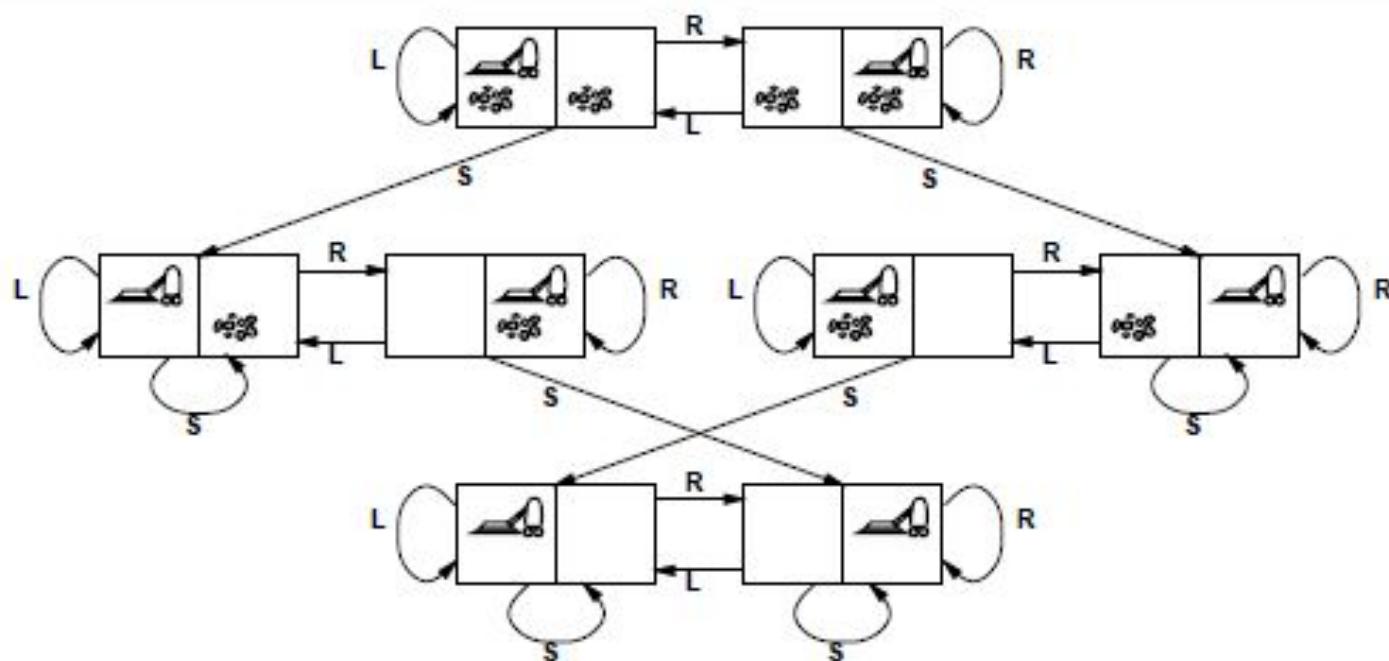
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

goal test??: no dirt

path cost??

Example: vacuum world state space graph



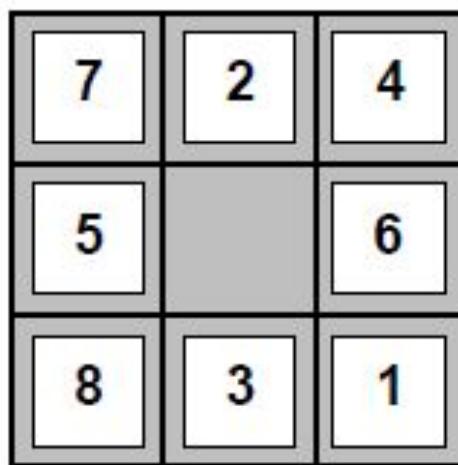
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

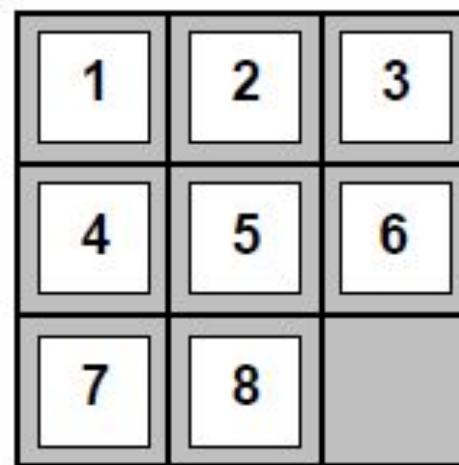
goal test??: no dirt

path cost??: 1 per action (0 for *NoOp*)

Example: The 8-puzzle



Start State



Goal State

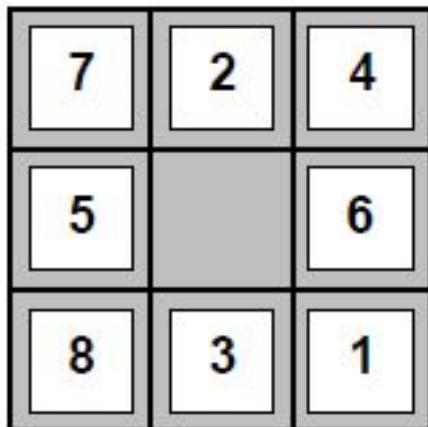
states??

actions??

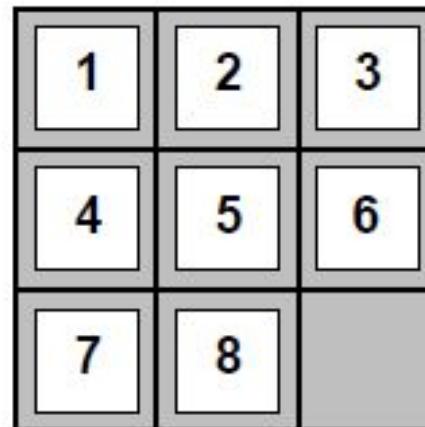
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

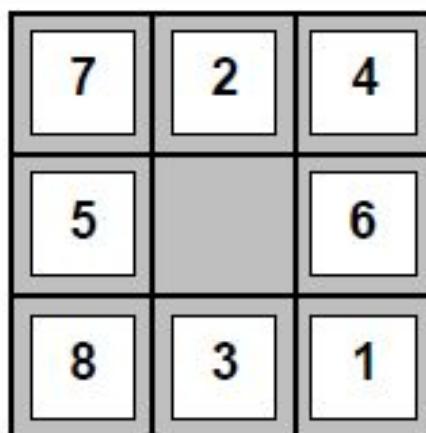
states??: integer locations of tiles (ignore intermediate positions)

actions??

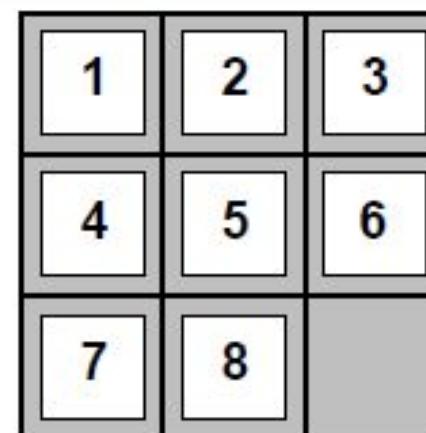
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

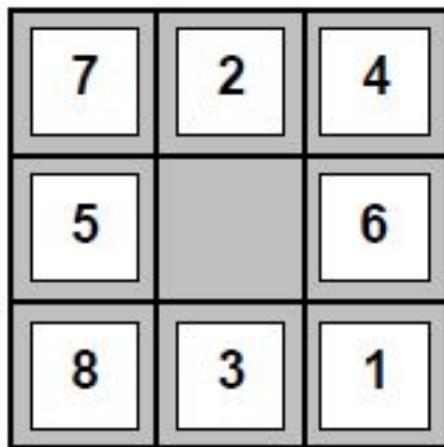
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

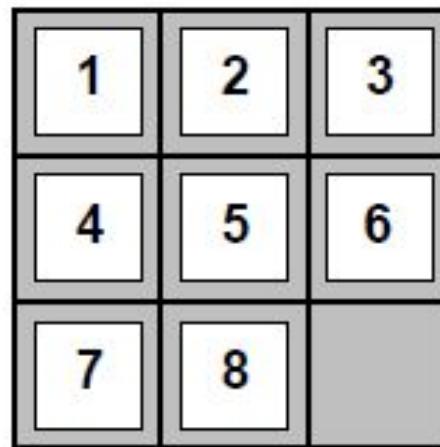
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

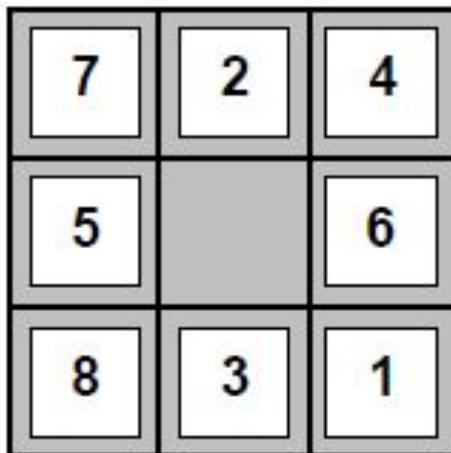
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

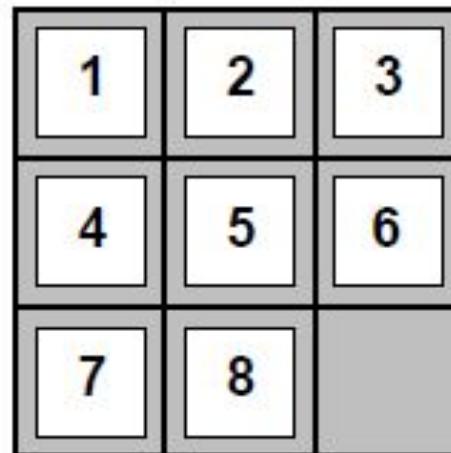
goal test??: = goal state (given)

path cost??

Example: The 8-puzzle



Start State



Goal State

states??: integer locations of tiles (ignore intermediate positions)

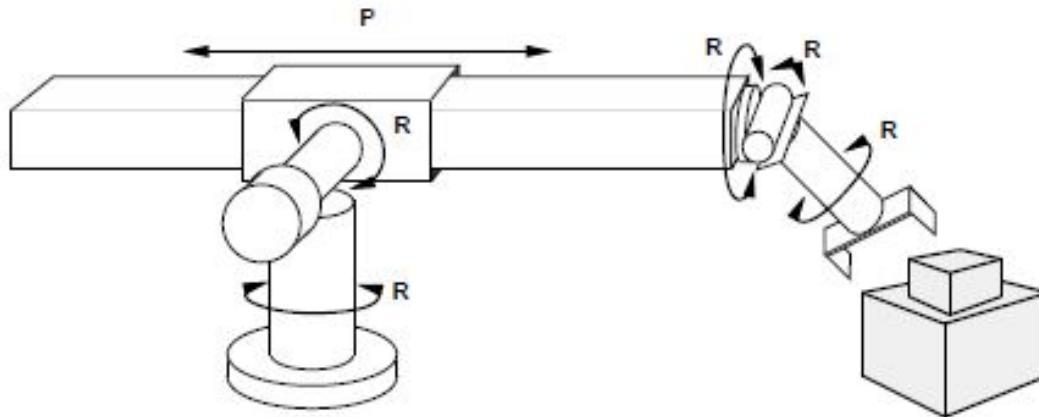
actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

[Note: optimal solution of n -Puzzle family is NP-hard]

Example: robotic assembly



states??: real-valued coordinates of robot joint angles
parts of the object to be assembled

actions??: continuous motions of robot joints

goal test??: complete assembly **with no robot included!**

path cost??: time to execute

PROBLEM SOLVING AND SEARCH

CHAPTER 3

Reminders

Assignment 0 due 5pm today

Assignment 1 posted, due 2/9

Section 105 will move to 9-10am starting next week

Outline

- ◊ Problem-solving agents
- ◊ Problem types
- ◊ Problem formulation
- ◊ Example problems
- ◊ Basic search algorithms

Problem-solving agents

Restricted form of general agent:

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

  state  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
  action  $\leftarrow$  RECOMMENDATION(seq, state)
  seq  $\leftarrow$  REMAINDER(seq, state)
  return action
```

Note: this is **offline** problem solving; solution executed “eyes closed.”

Online problem solving involves acting without complete knowledge.

Example: Romania

On holiday in Romania; currently in Arad.

Flight leaves tomorrow from Bucharest

Formulate goal:

be in Bucharest

Formulate problem:

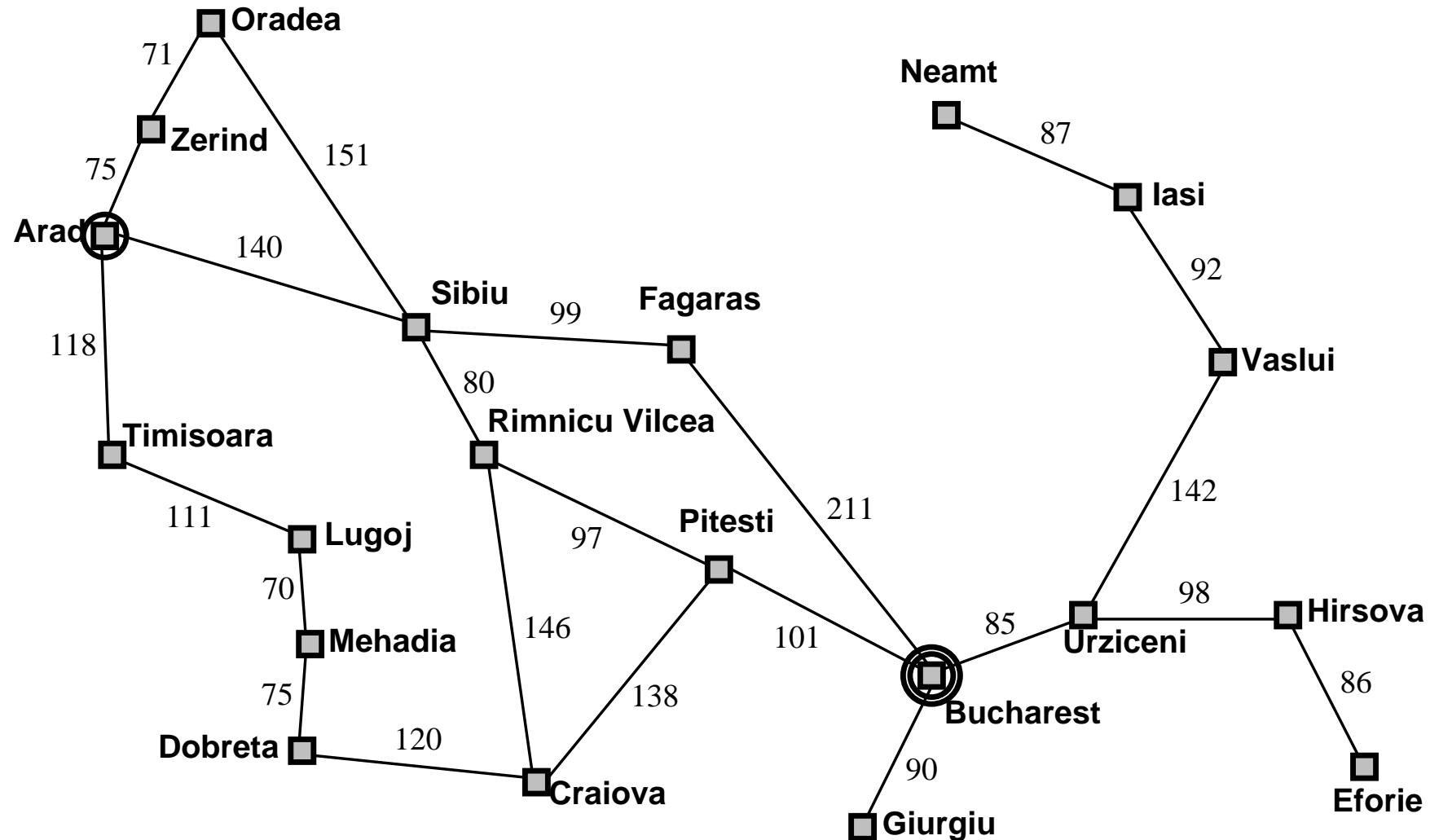
states: various cities

actions: drive between cities

Find solution:

sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Problem types

Deterministic, fully observable \Rightarrow single-state problem

Agent knows exactly which state it will be in; solution is a sequence

Non-observable \Rightarrow conformant problem

Agent may have no idea where it is; solution (if any) is a sequence

Nondeterministic and/or partially observable \Rightarrow contingency problem

percepts provide new information about current state

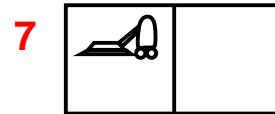
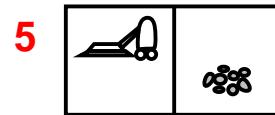
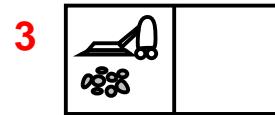
solution is a contingent plan or a policy

often interleave search, execution

Unknown state space \Rightarrow exploration problem (“online”)

Example: vacuum world

Single-state, start in #5. Solution??



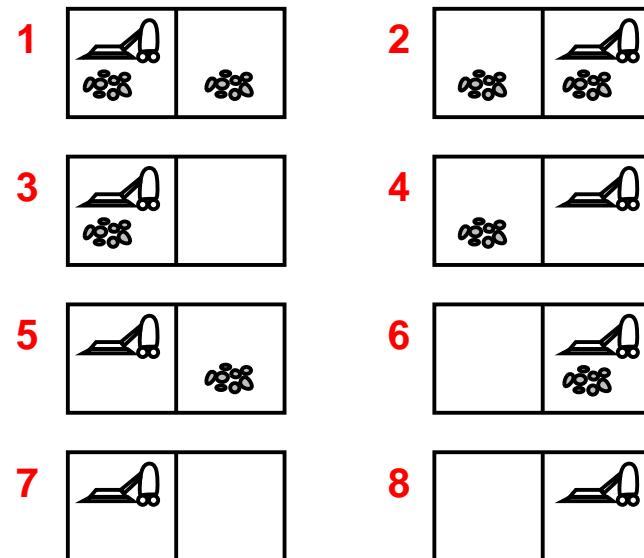
Example: vacuum world

Single-state, start in #5. [Solution??](#)

[*Right*, *Suck*]

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$

e.g., *Right* goes to $\{2, 4, 6, 8\}$. [Solution??](#)



Example: vacuum world

Single-state, start in #5. [Solution??](#)

[*Right, Suck*]

Conformant, start in $\{1, 2, 3, 4, 5, 6, 7, 8\}$

e.g., *Right* goes to $\{2, 4, 6, 8\}$. [Solution??](#)

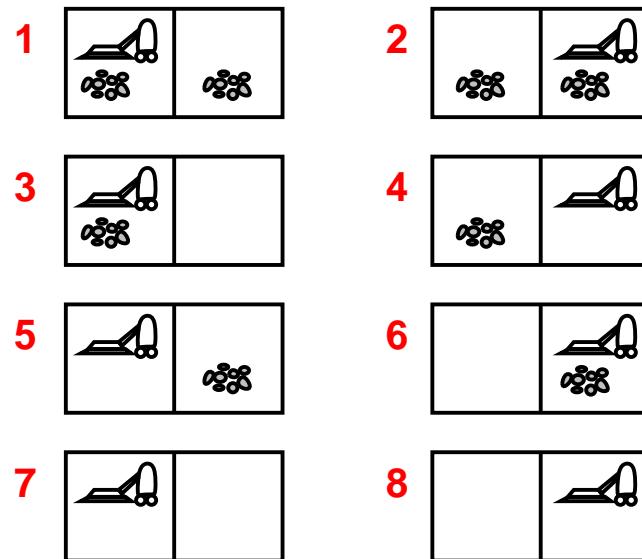
[*Right, Suck, Left, Suck*]

Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet

Local sensing: dirt, location only.

[Solution??](#)



Example: vacuum world

Single-state, start in #5. [Solution??](#)

[*Right, Suck*]

Conformant, start in {1, 2, 3, 4, 5, 6, 7, 8}

e.g., *Right* goes to {2, 4, 6, 8}. [Solution??](#)

[*Right, Suck, Left, Suck*]

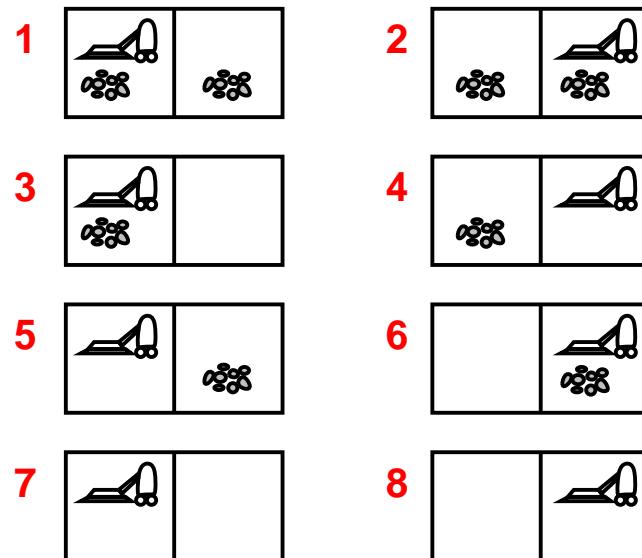
Contingency, start in #5

Murphy's Law: *Suck* can dirty a clean carpet

Local sensing: dirt, location only.

[Solution??](#)

[*Right, if dirt then Suck*]



Single-state problem formulation

A **problem** is defined by four items:

initial state e.g., “at Arad”

successor function $S(x)$ = set of action–state pairs

e.g., $S(Arad) = \{\langle Arad \rightarrow Zerind, Zerind \rangle, \dots\}$

goal test, can be

explicit, e.g., x = “at Bucharest”

implicit, e.g., $NoDirt(x)$

path cost (additive)

e.g., sum of distances, number of actions executed, etc.

$c(x, a, y)$ is the **step cost**, assumed to be ≥ 0

A **solution** is a sequence of actions

leading from the initial state to a goal state

Selecting a state space

Real world is absurdly complex

⇒ state space must be **abstracted** for problem solving

(Abstract) state = set of real states

(Abstract) action = complex combination of real actions

e.g., “Arad → Zerind” represents a complex set
of possible routes, detours, rest stops, etc.

For guaranteed realizability, **any** real state “in Arad”

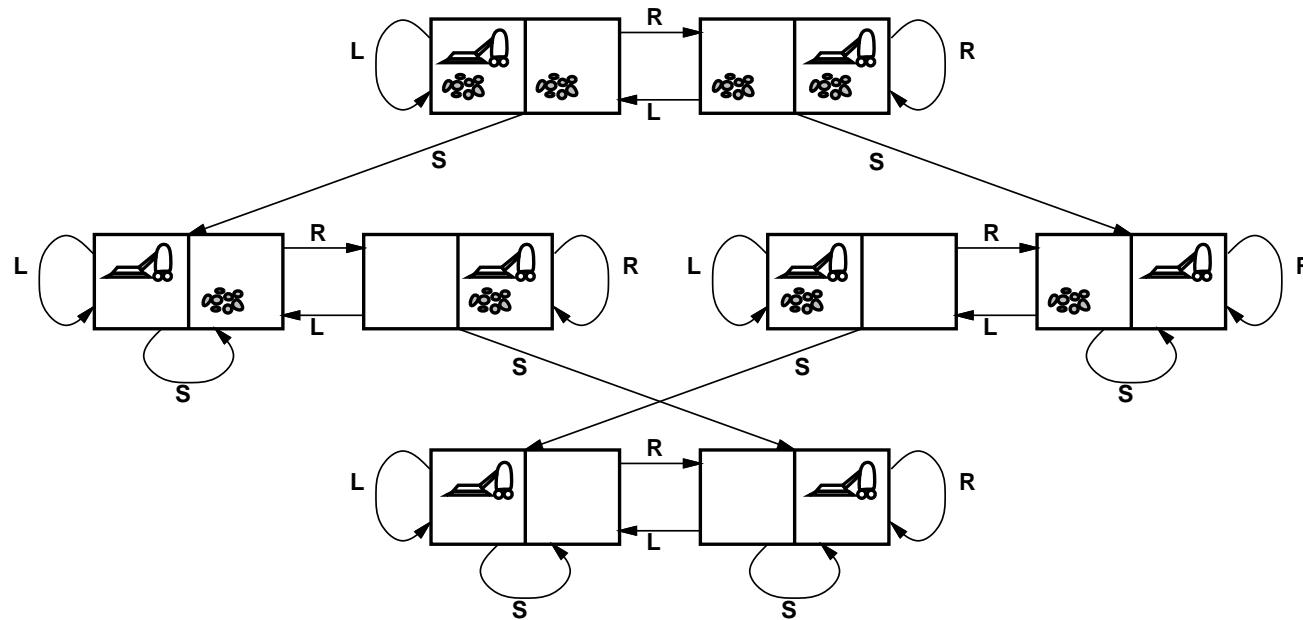
must get to some real state “in Zerind”

(Abstract) solution =

set of real paths that are solutions in the real world

Each abstract action should be “easier” than the original problem!

Example: vacuum world state space graph



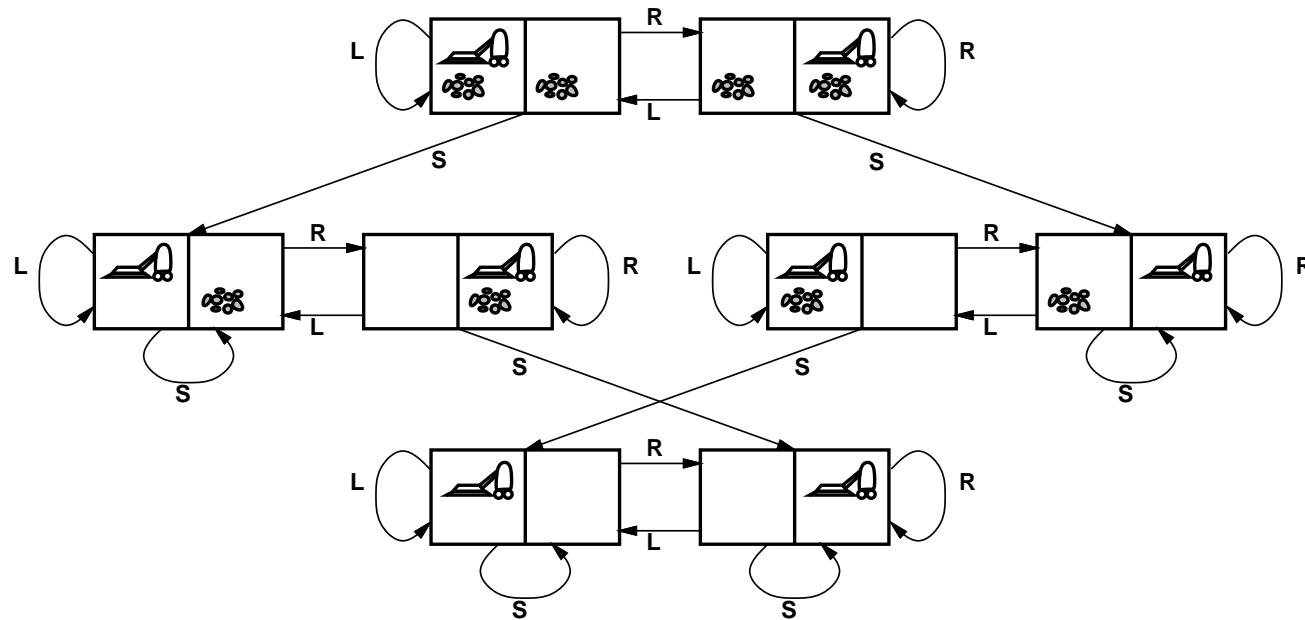
states??

actions??

goal test??

path cost??

Example: vacuum world state space graph



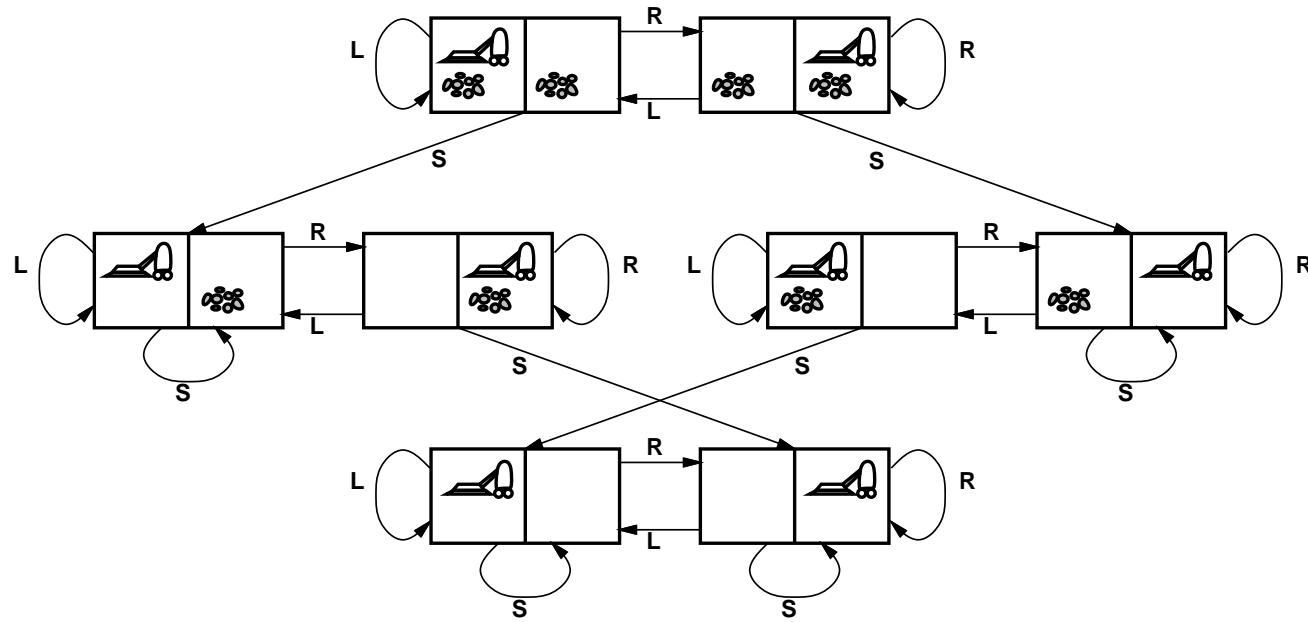
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??

goal test??

path cost??

Example: vacuum world state space graph



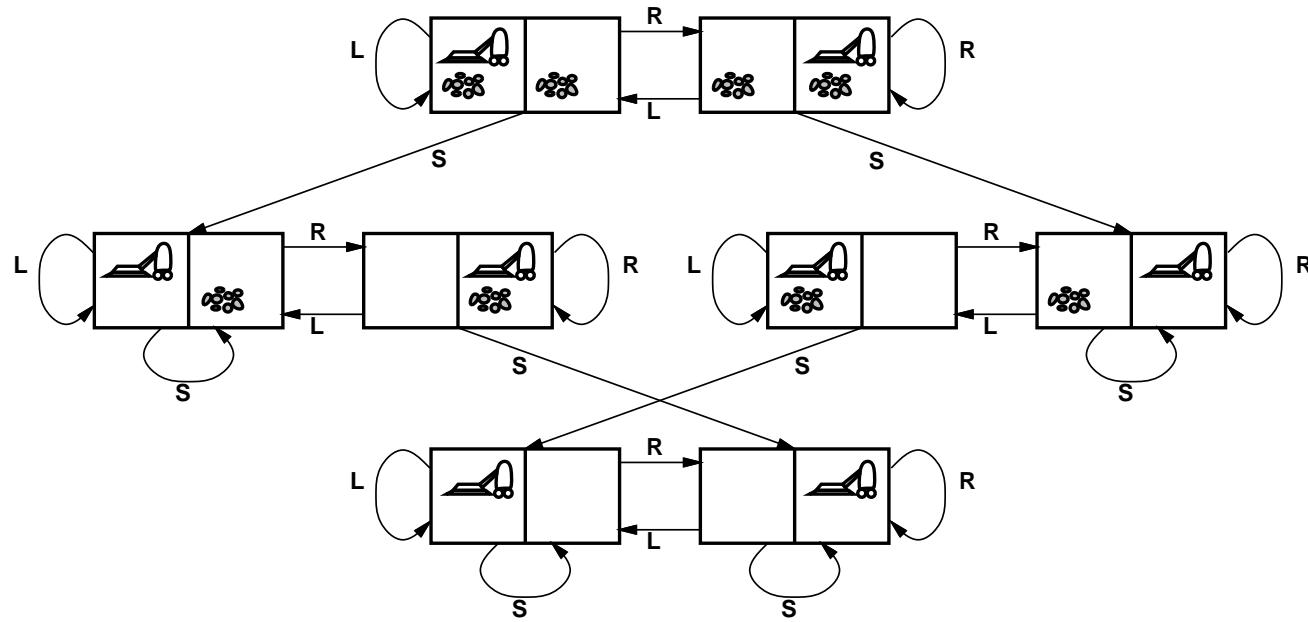
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

goal test??

path cost??

Example: vacuum world state space graph



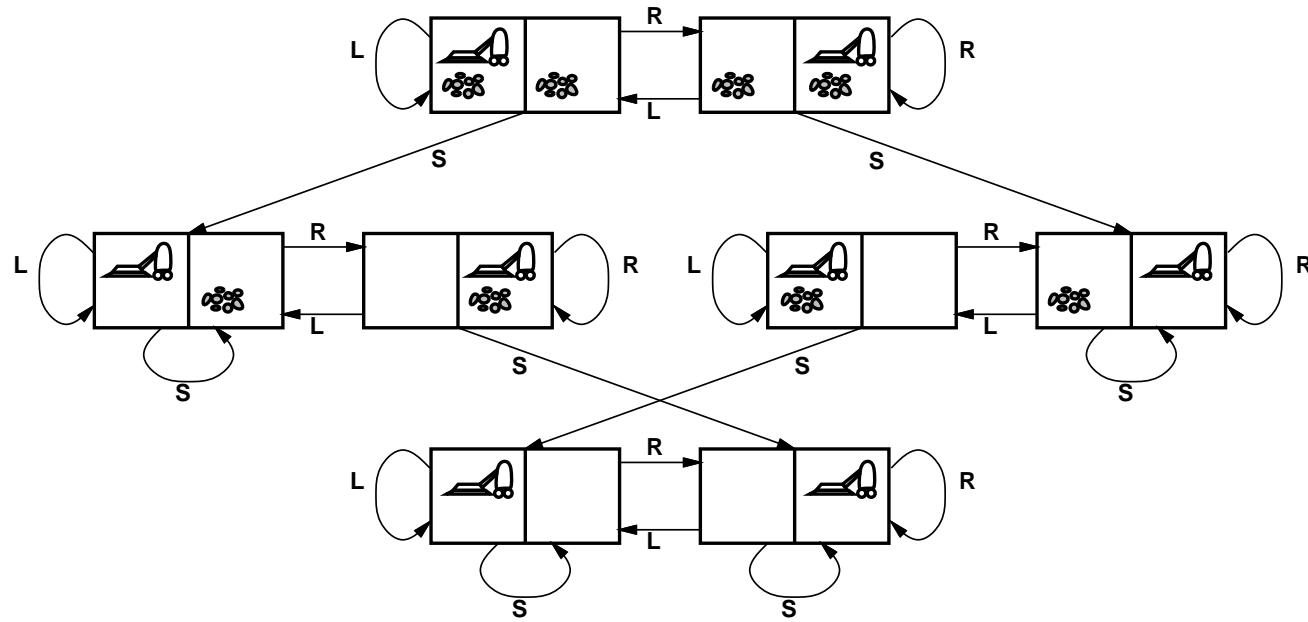
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

goal test??: no dirt

path cost??

Example: vacuum world state space graph



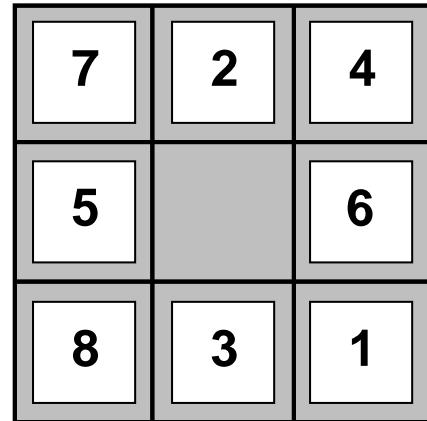
states??: integer dirt and robot locations (ignore dirt amounts etc.)

actions??: *Left*, *Right*, *Suck*, *NoOp*

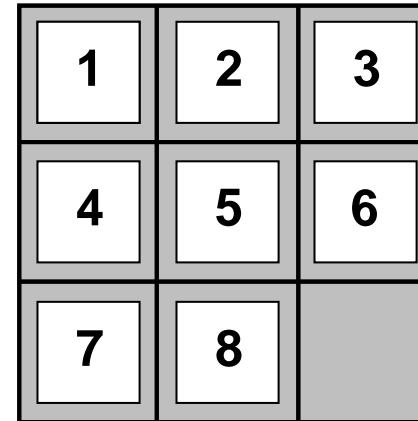
goal test??: no dirt

path cost??: 1 per action (0 for *NoOp*)

Example: The 8-puzzle



Start State



Goal State

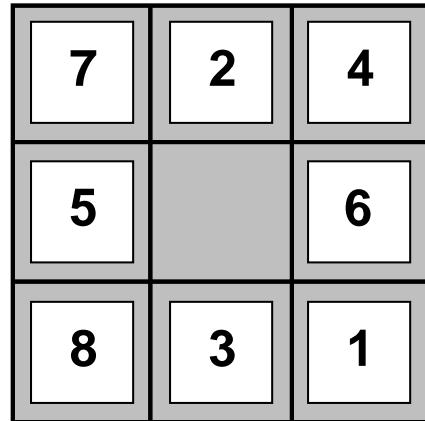
states??

actions??

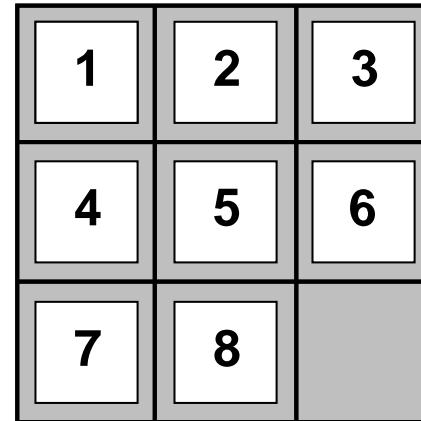
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

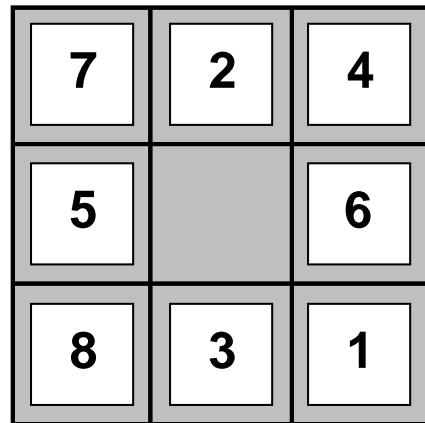
states??: integer locations of tiles (ignore intermediate positions)

actions??

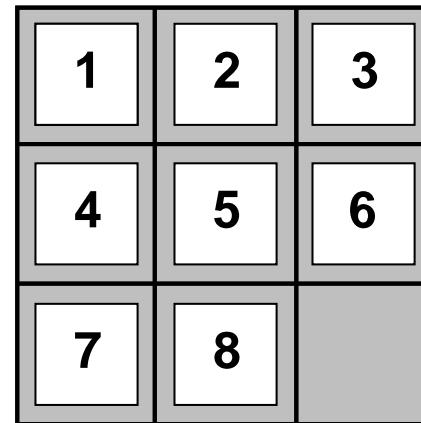
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

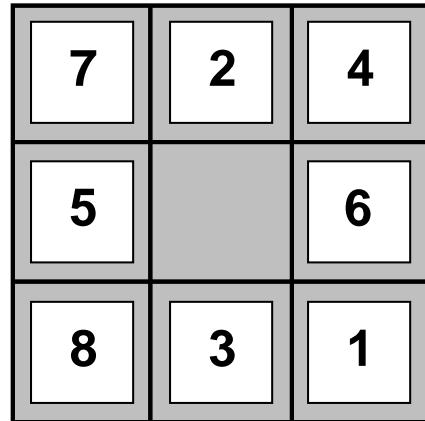
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

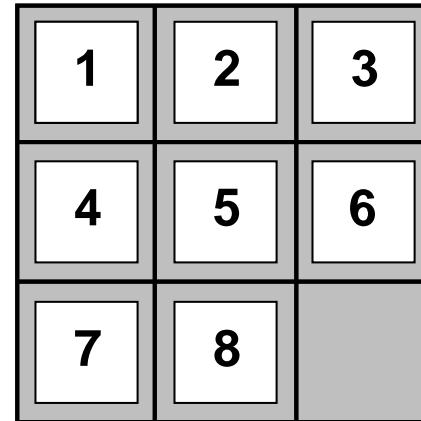
goal test??

path cost??

Example: The 8-puzzle



Start State



Goal State

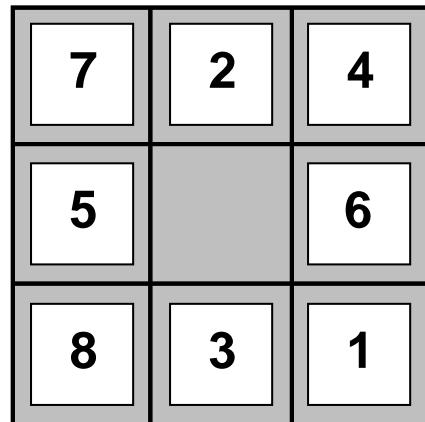
states??: integer locations of tiles (ignore intermediate positions)

actions??: move blank left, right, up, down (ignore unjamming etc.)

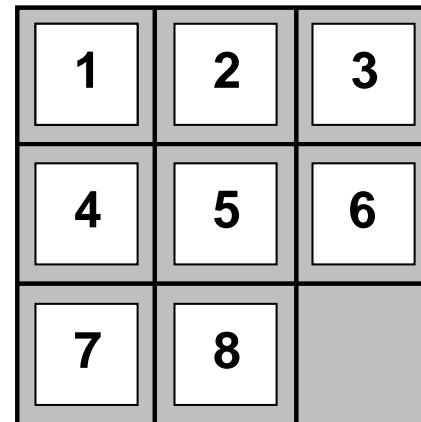
goal test??: = goal state (given)

path cost??

Example: The 8-puzzle



Start State



Goal State

states??: integer locations of tiles (ignore intermediate positions)

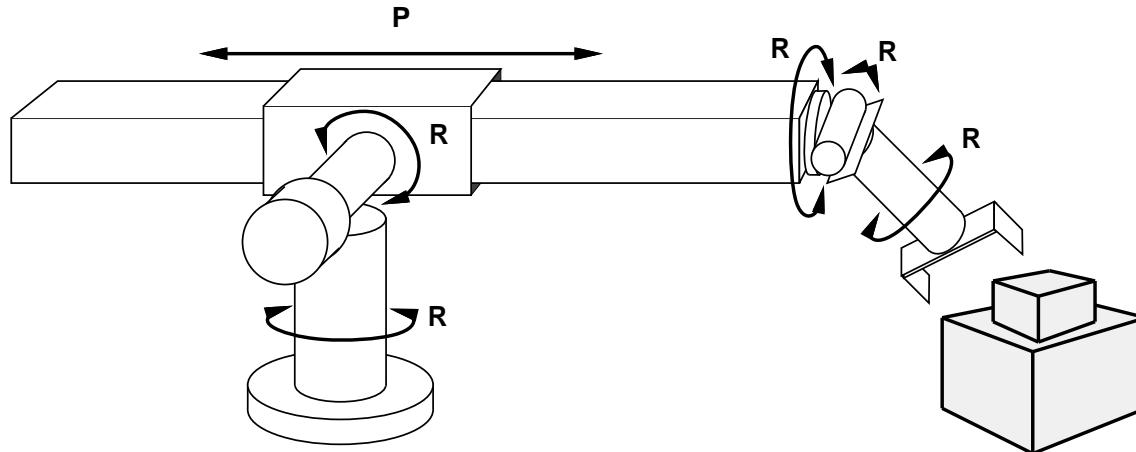
actions??: move blank left, right, up, down (ignore unjamming etc.)

goal test??: = goal state (given)

path cost??: 1 per move

[Note: optimal solution of n -Puzzle family is NP-hard]

Example: robotic assembly



states??: real-valued coordinates of robot joint angles
parts of the object to be assembled

actions??: continuous motions of robot joints

goal test??: complete assembly **with no robot included!**

path cost??: time to execute

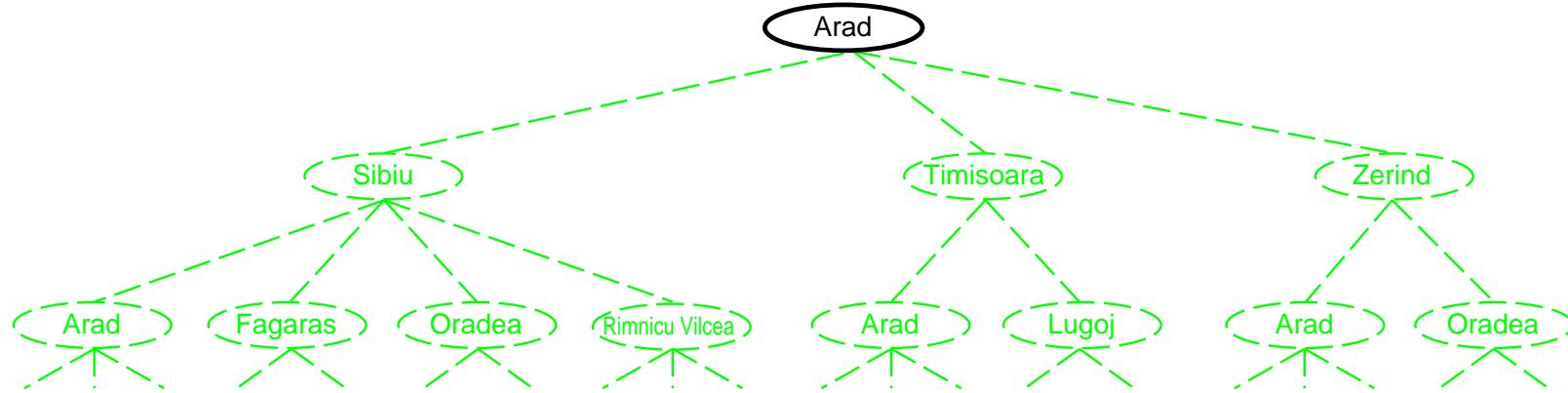
Tree search algorithms

Basic idea:

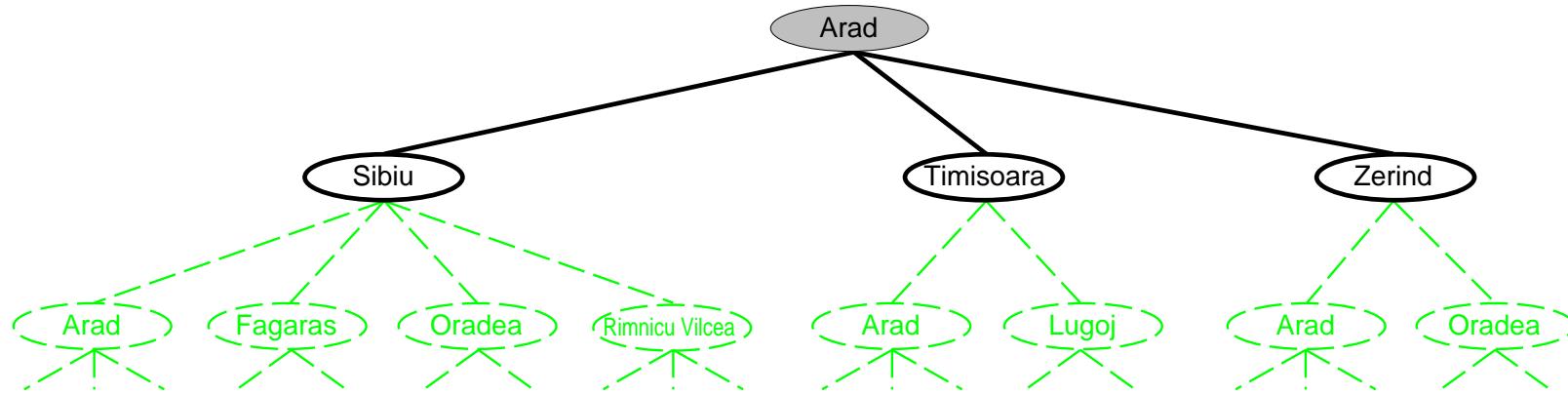
offline, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. **expanding** states)

```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

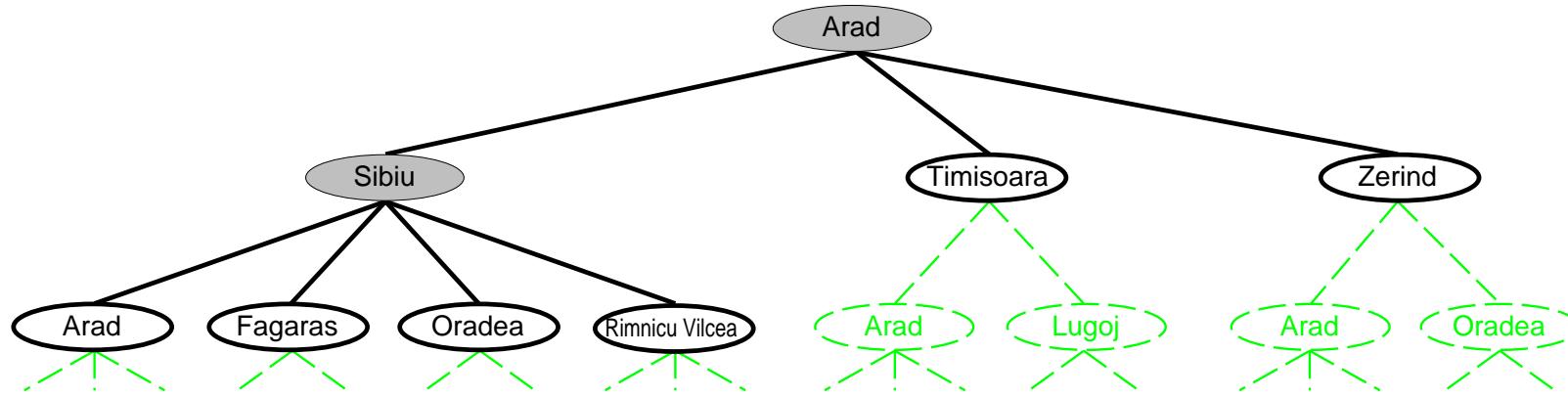
Tree search example



Tree search example



Tree search example

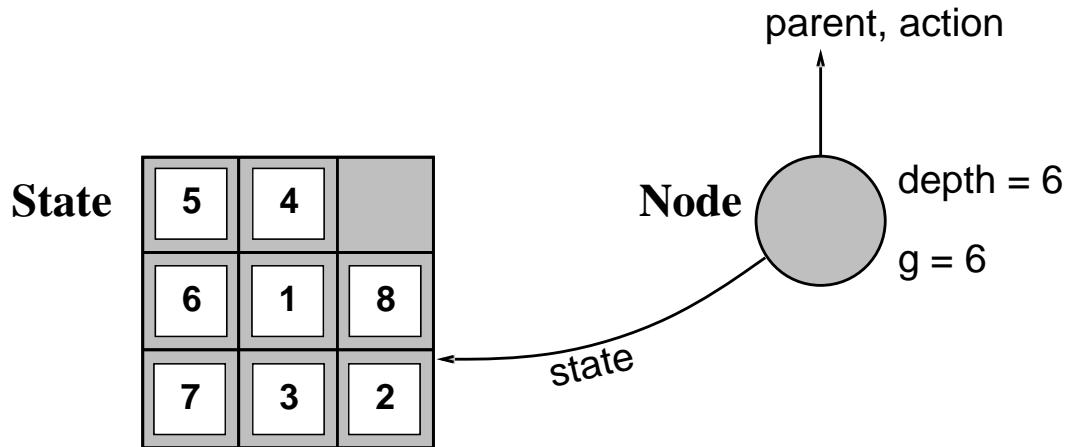


Implementation: states vs. nodes

A **state** is a (representation of) a physical configuration

A **node** is a data structure constituting part of a search tree
includes parent, children, depth, path cost $g(x)$

States do not have parents, children, depth, or path cost!



The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFn of the problem to create the corresponding states.

Implementation: general tree search

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE(node)) then return node
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND(node, problem) returns a set of nodes
  successors  $\leftarrow$  the empty set
  for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
    s  $\leftarrow$  a new NODE
    PARENT-NODE[s]  $\leftarrow$  node; ACTION[s]  $\leftarrow$  action; STATE[s]  $\leftarrow$  result
    PATH-COST[s]  $\leftarrow$  PATH-COST[node] + STEP-COST(node, action, s)
    DEPTH[s]  $\leftarrow$  DEPTH[node] + 1
    add s to successors
  return successors
```

Search strategies

A strategy is defined by picking the **order of node expansion**

Strategies are evaluated along the following dimensions:

completeness—does it always find a solution if one exists?

time complexity—number of nodes generated/expanded

space complexity—maximum number of nodes in memory

optimality—does it always find a least-cost solution?

Time and space complexity are measured in terms of

b —maximum branching factor of the search tree

d —depth of the least-cost solution

m —maximum depth of the state space (may be ∞)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition

Breadth-first search

Uniform-cost search

Depth-first search

Depth-limited search

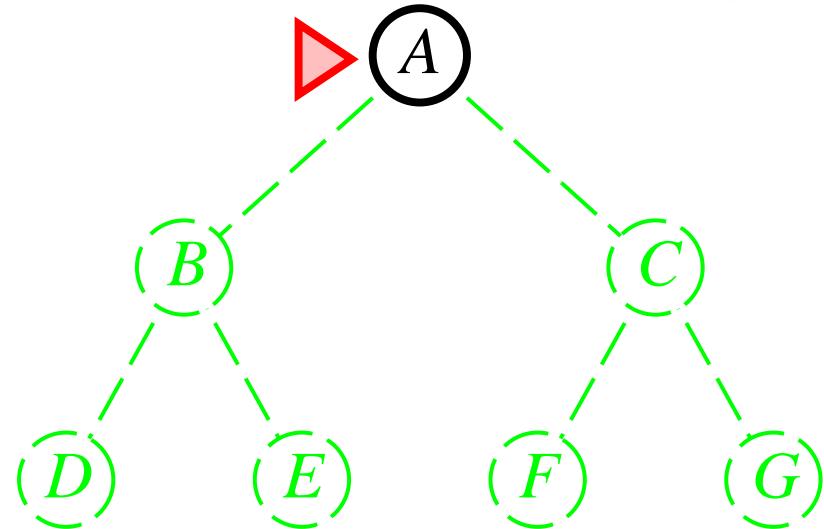
Iterative deepening search

Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

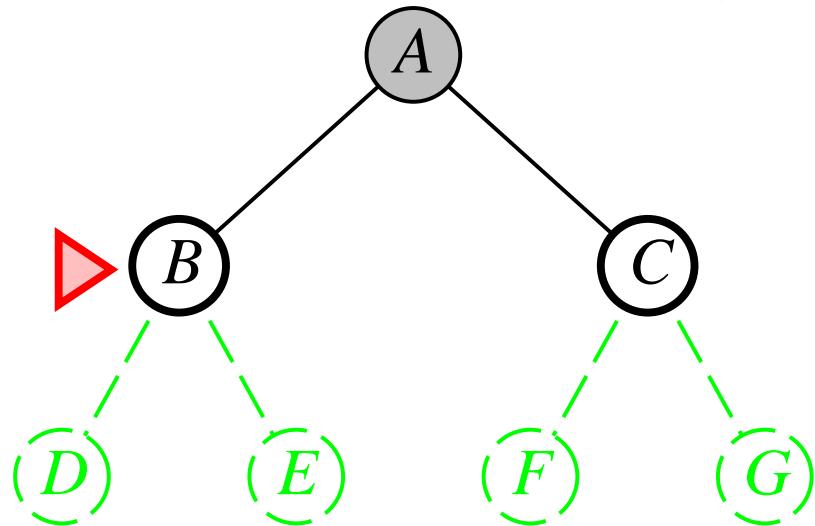


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

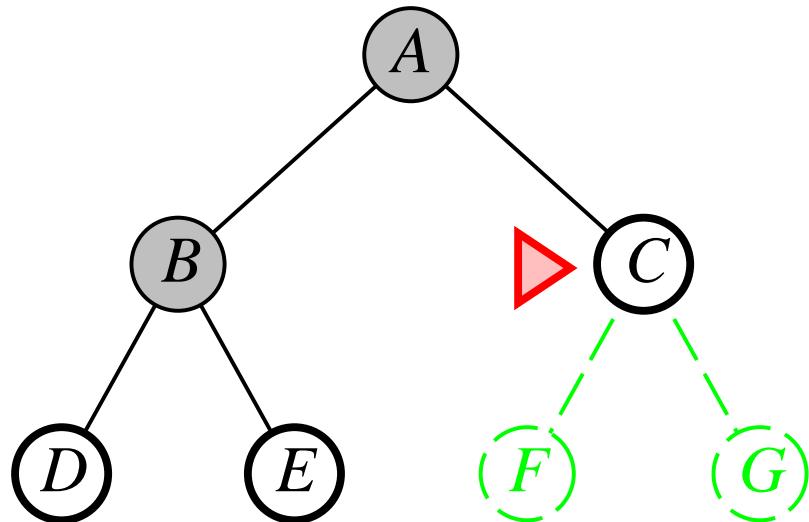


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end

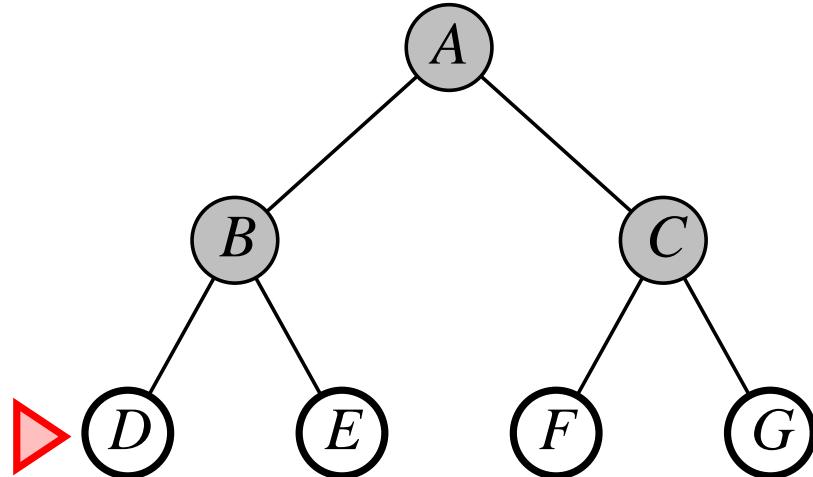


Breadth-first search

Expand shallowest unexpanded node

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



Properties of breadth-first search

Complete??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal??

Properties of breadth-first search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? Yes (if cost = 1 per step); not optimal in general

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

fringe = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

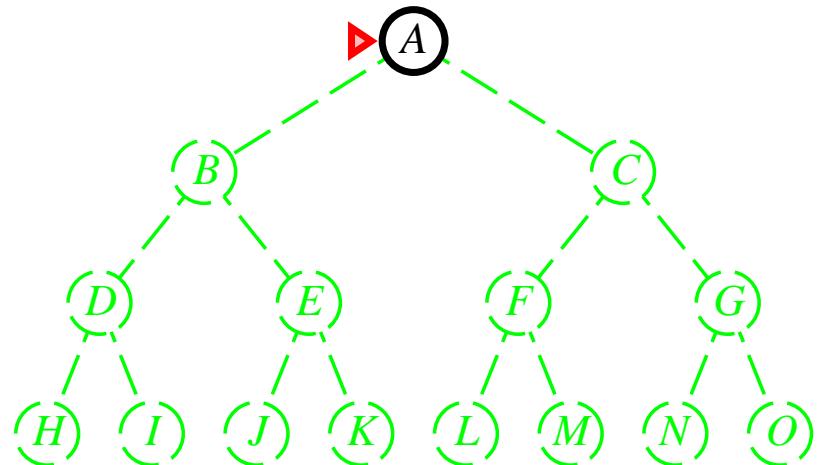
Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

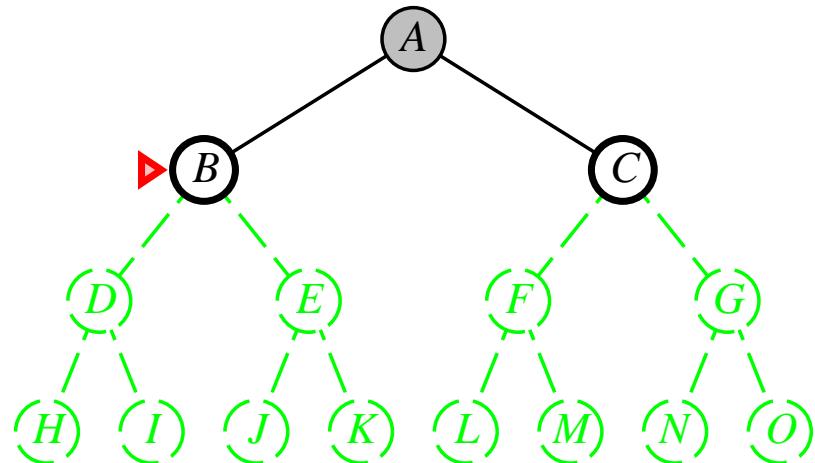


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

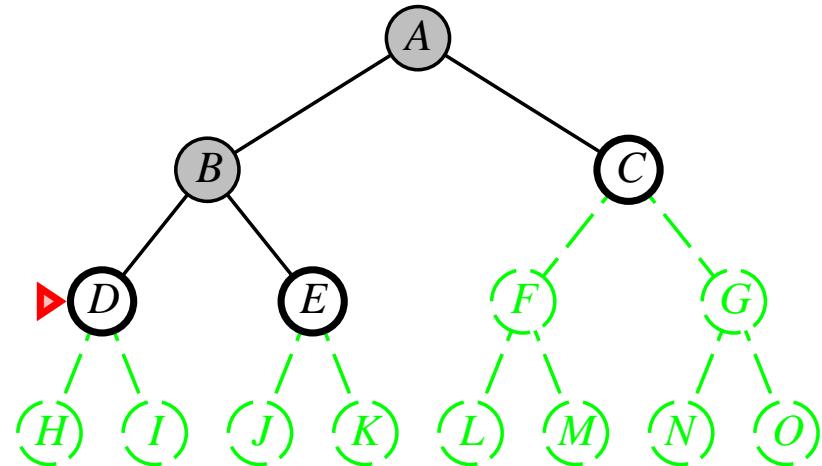


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

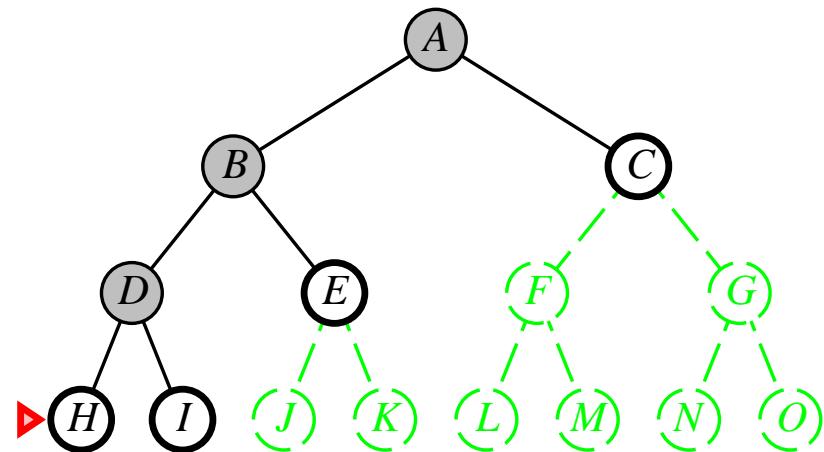


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

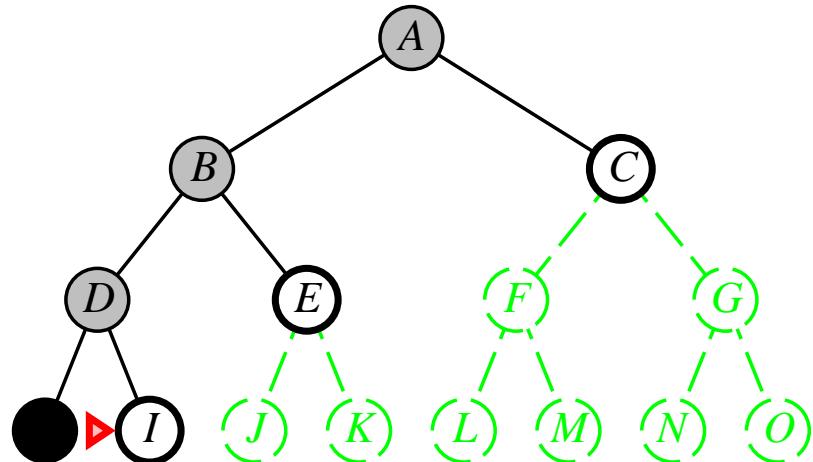


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

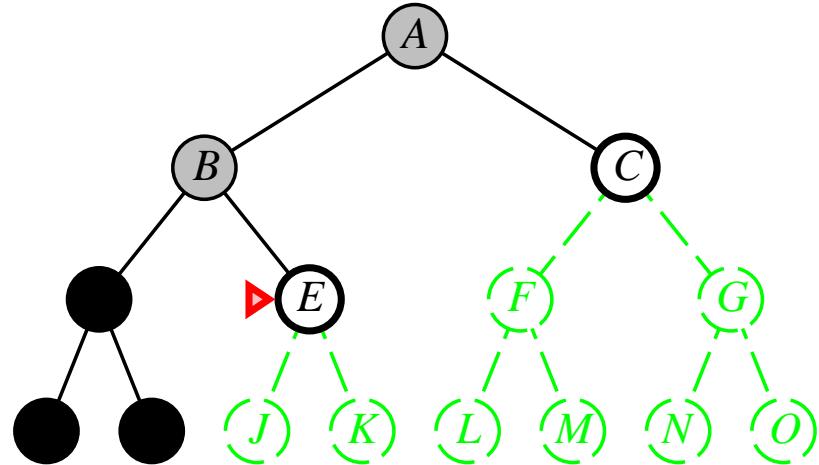


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

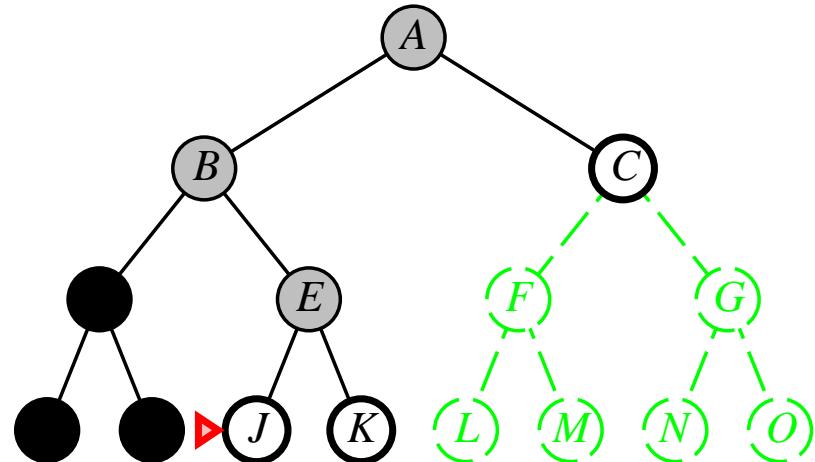


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

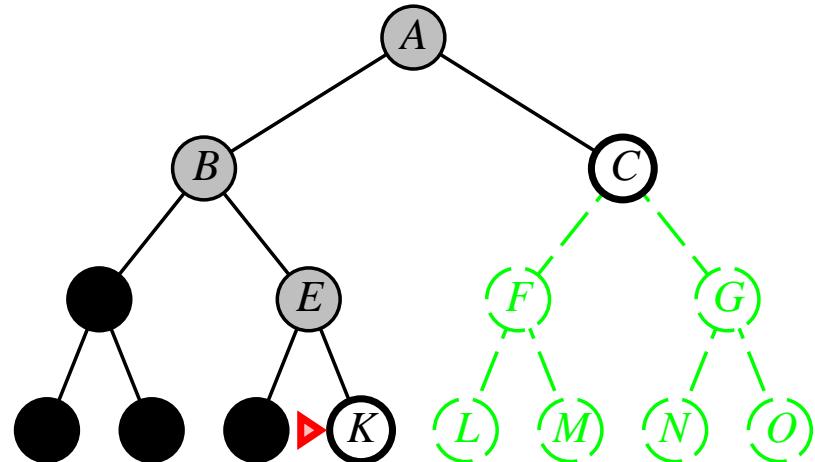


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

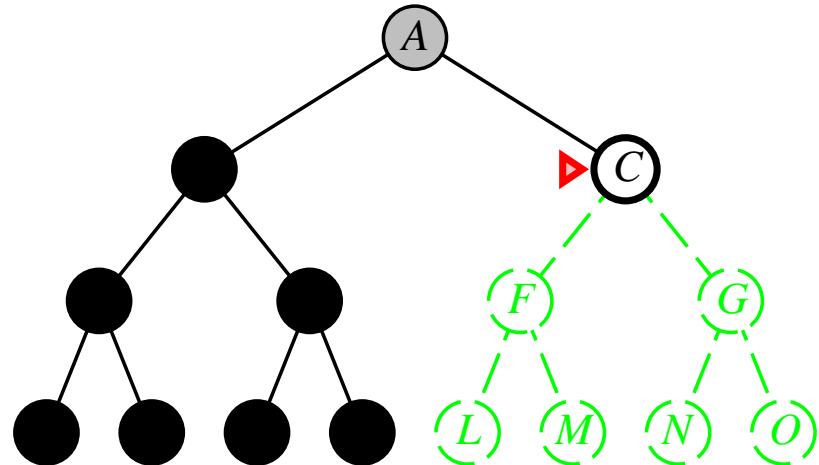


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

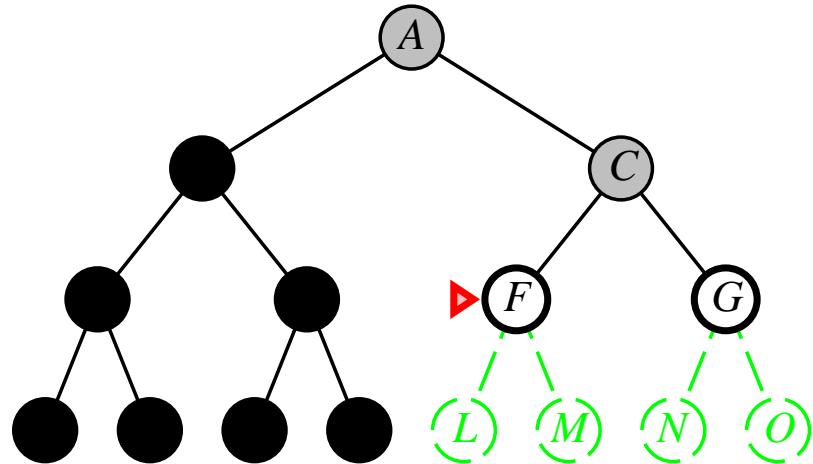


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

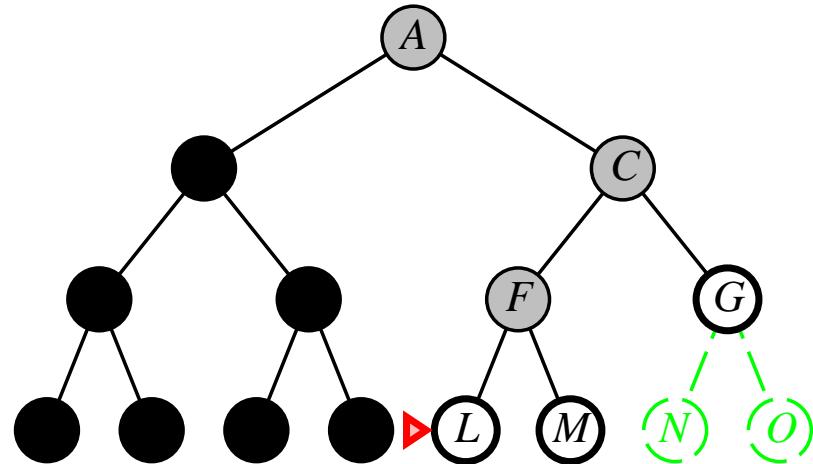


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

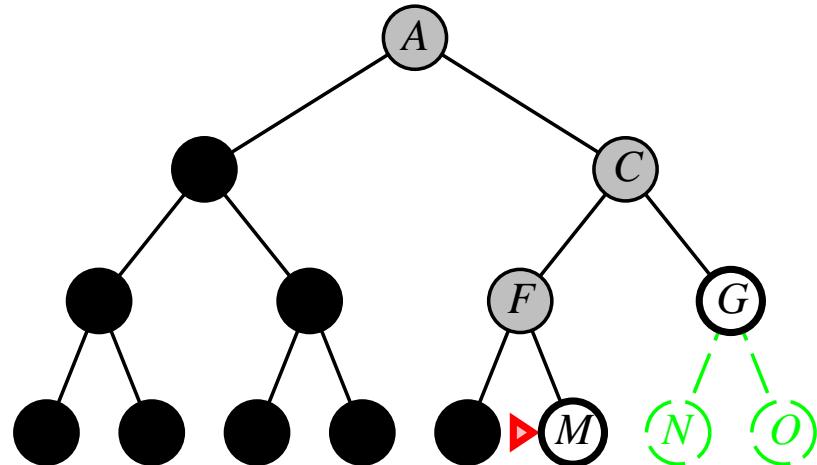


Depth-first search

Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front



Properties of depth-first search

Complete??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path
⇒ complete in finite spaces

Time??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first

Space??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal??

Properties of depth-first search

Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path
⇒ complete in finite spaces

Time?? $O(b^m)$: terrible if m is much larger than d
but if solutions are dense, may be much faster than breadth-first

Space?? $O(bm)$, i.e., linear space!

Optimal?? No

Depth-limited search

= depth-first search with depth limit l ,
i.e., nodes at depth l have no successors

Recursive implementation:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred?  $\leftarrow$  false
    if GOAL-TEST(problem, STATE[node]) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result  $\leftarrow$  RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred?  $\leftarrow$  true
        else if result  $\neq$  failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative deepening search

```
function ITERATIVE-DEEPENING-SEARCH( problem ) returns a solution
    inputs: problem, a problem
    for depth  $\leftarrow 0$  to  $\infty$  do
        result  $\leftarrow$  DEPTH-LIMITED-SEARCH( problem, depth )
        if result  $\neq$  cutoff then return result
    end
```

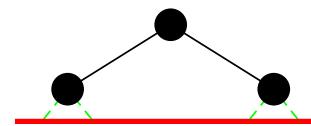
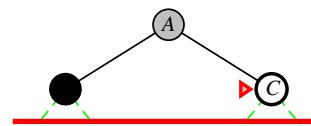
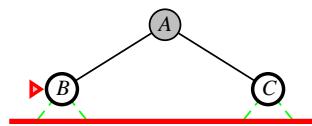
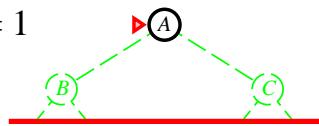
Iterative deepening search $l = 0$

Limit = 0



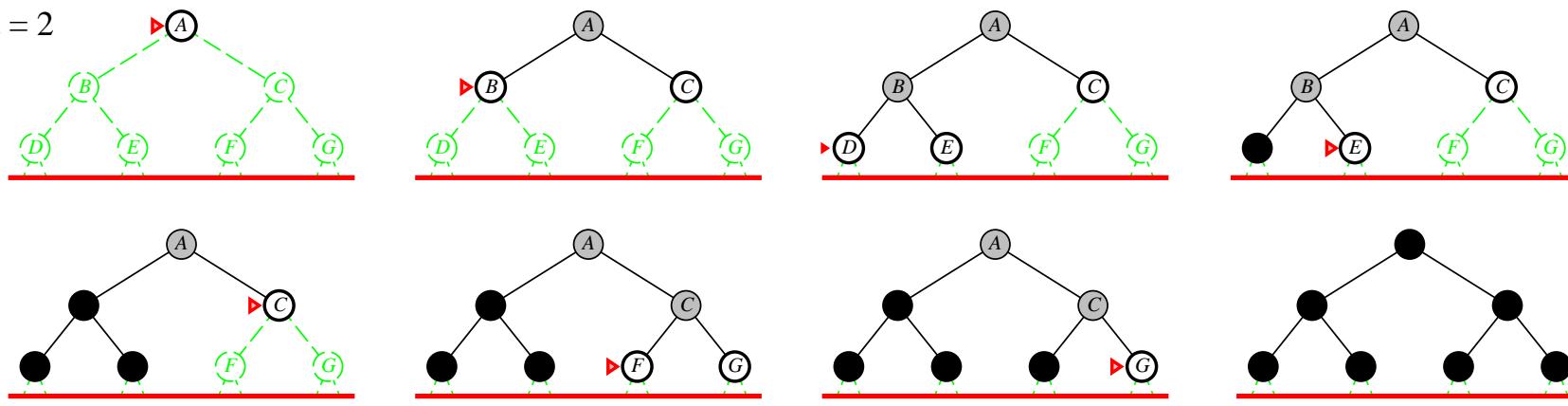
Iterative deepening search $l = 1$

Limit = 1

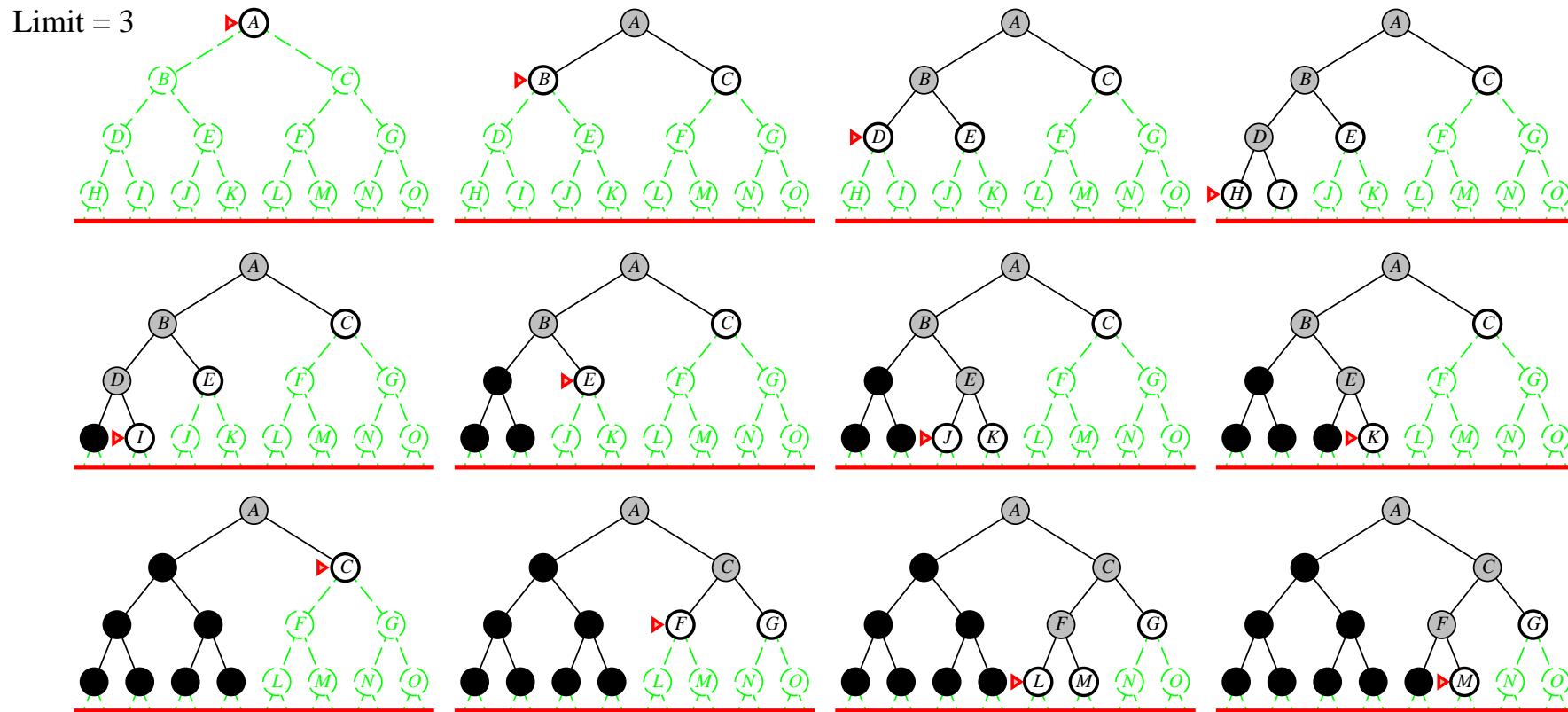


Iterative deepening search $l = 2$

Limit = 2



Iterative deepening search $l = 3$



Properties of iterative deepening search

Complete??

Properties of iterative deepening search

Complete?? Yes

Time??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal??

Properties of iterative deepening search

Complete?? Yes

Time?? $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? Yes, if step cost = 1

Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$

$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

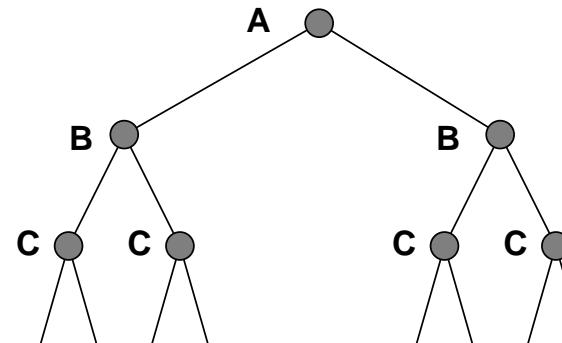
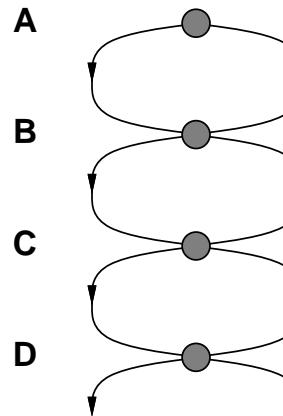
BFS can be modified to apply goal test when a node is **generated**

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes*	Yes*	No	Yes, if $l \geq d$	Yes
Time	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{\lceil C^*/\epsilon \rceil}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!



Graph search

```
function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed  $\leftarrow$  an empty set
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST(problem, STATE[node]) then return node
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
  end
```

Summary

Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

Variety of uninformed search strategies

Iterative deepening search uses only linear space
and not much more time than other uninformed algorithms

Graph search can be exponentially more efficient than tree search

INFORMED SEARCH ALGORITHMS

CHAPTER 4, SECTIONS 1–2

Outline

- ◊ Best-first search
- ◊ A* search
- ◊ Heuristics

Review: Tree search

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
  fringe  $\leftarrow$  INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if fringe is empty then return failure
    node  $\leftarrow$  REMOVE-FRONT(fringe)
    if GOAL-TEST[problem] applied to STATE(node) succeeds return node
    fringe  $\leftarrow$  INSERTALL(EXPAND(node, problem), fringe)
```

A strategy is defined by picking the **order of node expansion**

Best-first search

Idea: use an **evaluation function** for each node

- estimate of “desirability”

⇒ Expand most desirable unexpanded node

Implementation:

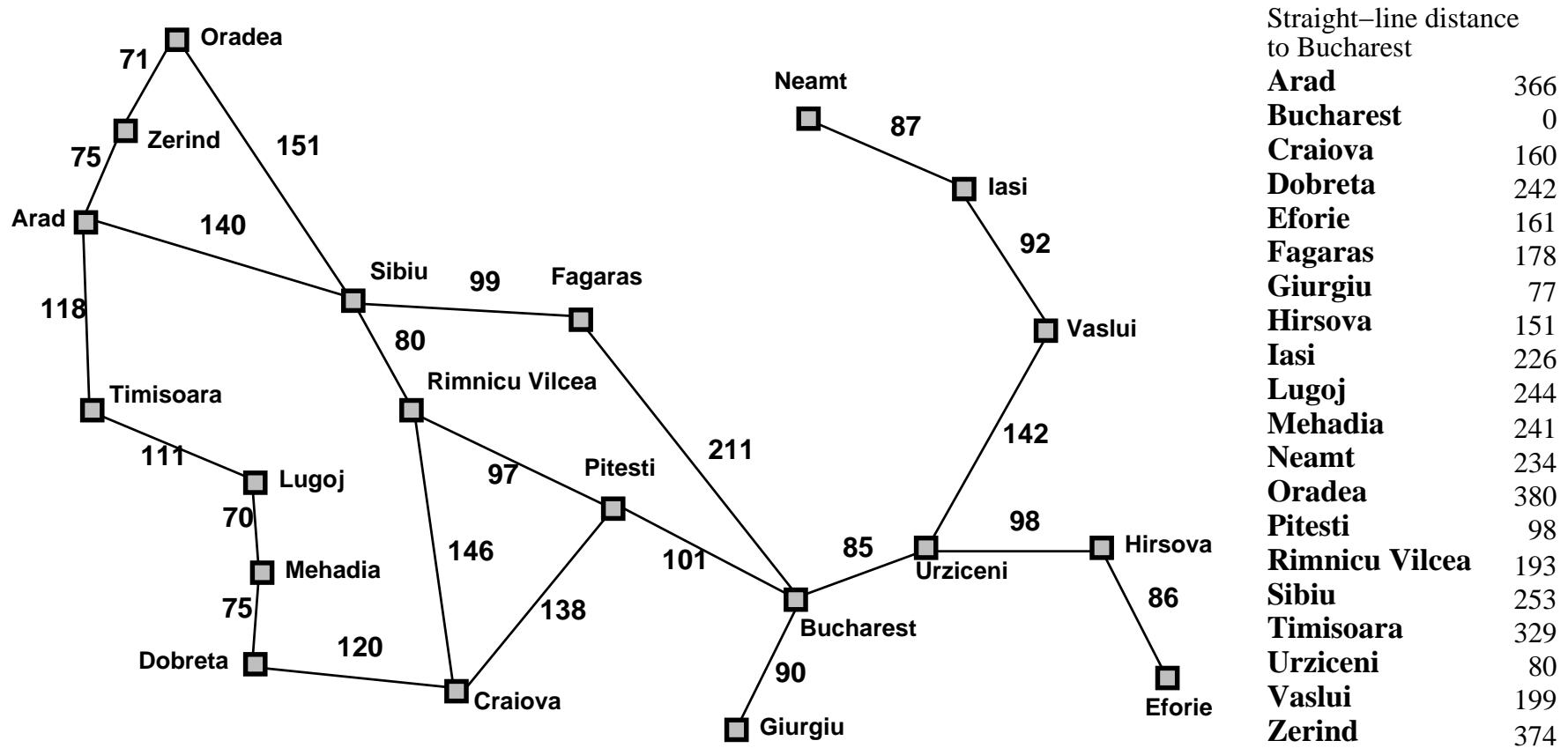
fringe is a queue sorted in decreasing order of desirability

Special cases:

greedy search

A* search

Romania with step costs in km



Greedy search

Evaluation function $h(n)$ (**heuristic**)
= estimate of cost from n to the closest goal

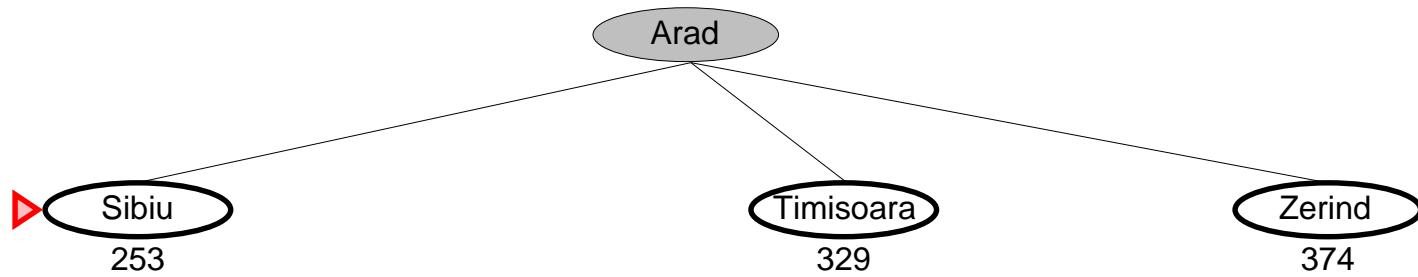
E.g., $h_{\text{SLD}}(n)$ = straight-line distance from n to Bucharest

Greedy search expands the node that **appears** to be closest to goal

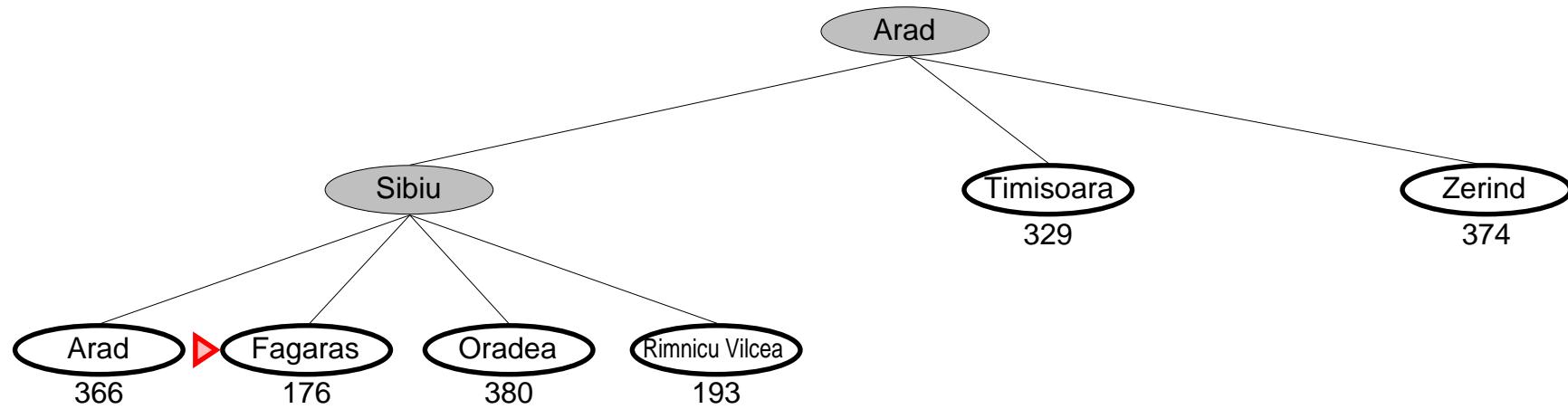
Greedy search example



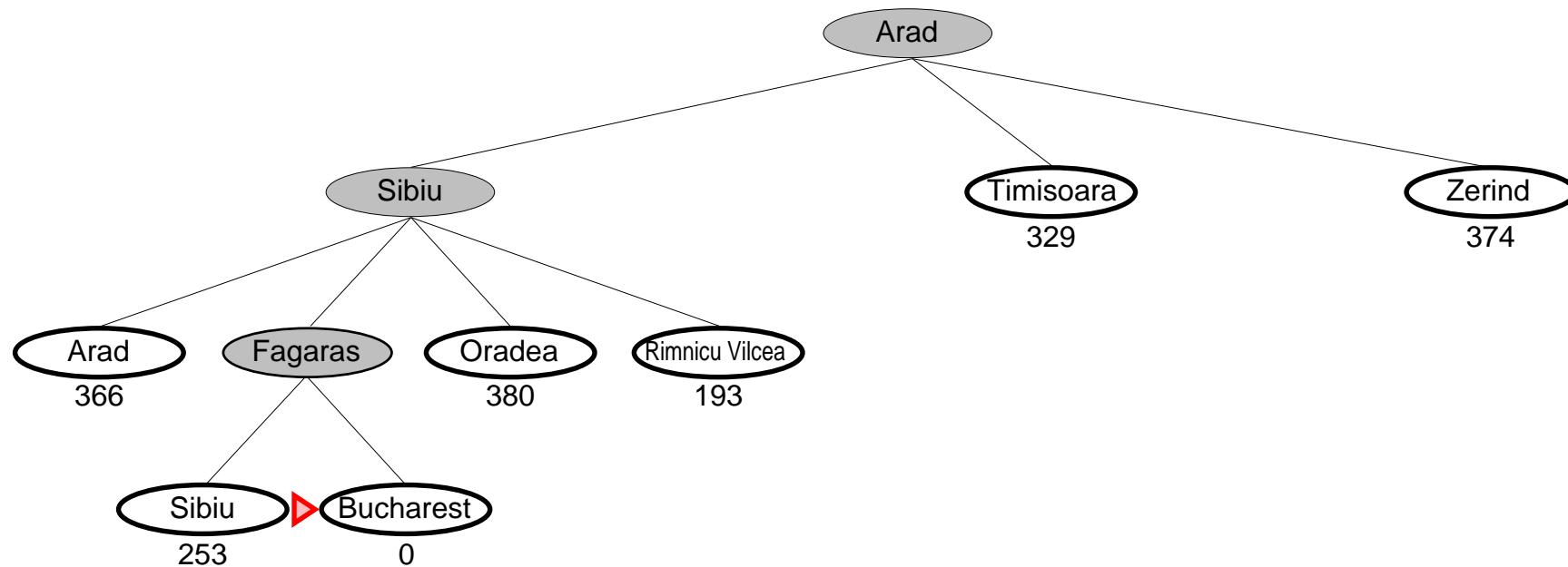
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g., with Oradea as goal,

lasi → Neamt → laси → Neamt →

Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi → Neamt → lași → Neamț →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi → Neamt → lași → Neamt →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$ —keeps all nodes in memory

Optimal??

Properties of greedy search

Complete?? No—can get stuck in loops, e.g.,

lasi → Neamt → lași → Neamț →

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$ —keeps all nodes in memory

Optimal?? No

A* search

Idea: avoid expanding paths that are already expensive

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

$g(n)$ = cost so far to reach n

$h(n)$ = estimated cost to goal from n

$f(n)$ = estimated total cost of path through n to goal

A* search uses an **admissible** heuristic

i.e., $h(n) \leq h^*(n)$ where $h^*(n)$ is the **true** cost from n .

(Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G .)

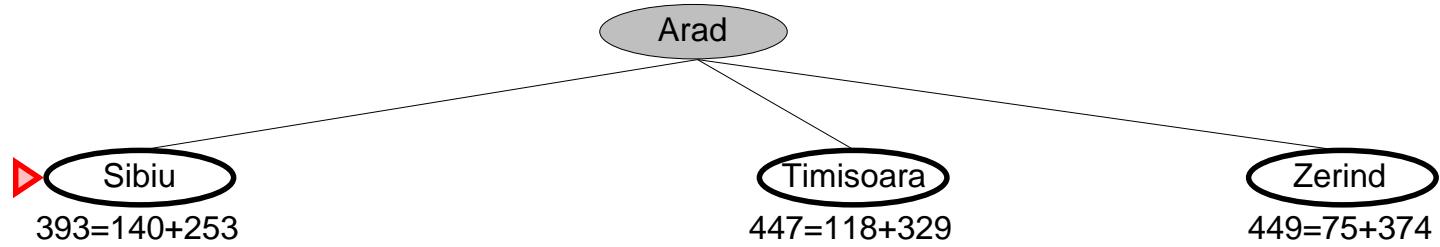
E.g., $h_{\text{SLD}}(n)$ never overestimates the actual road distance

Theorem: A* search is optimal

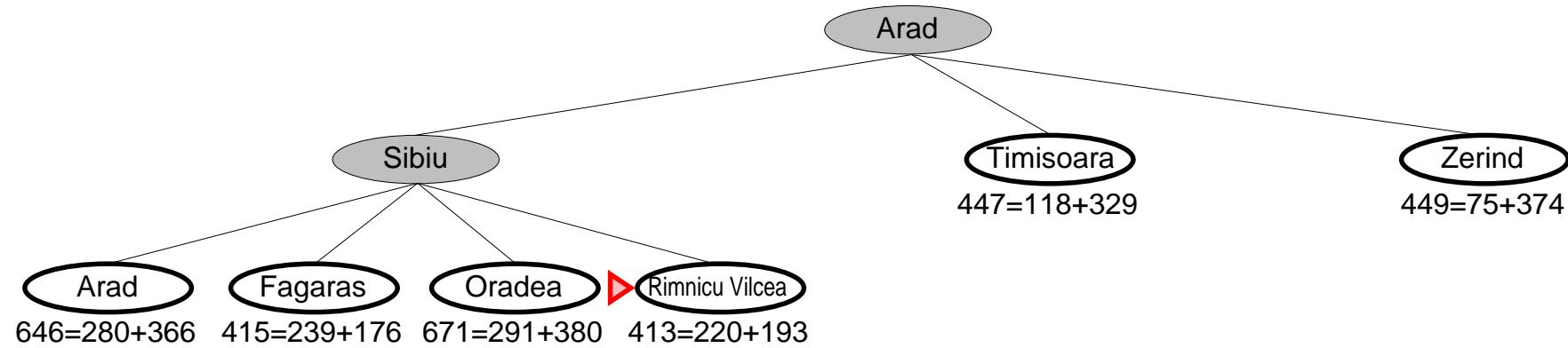
A* search example

► Arad
366=0+366

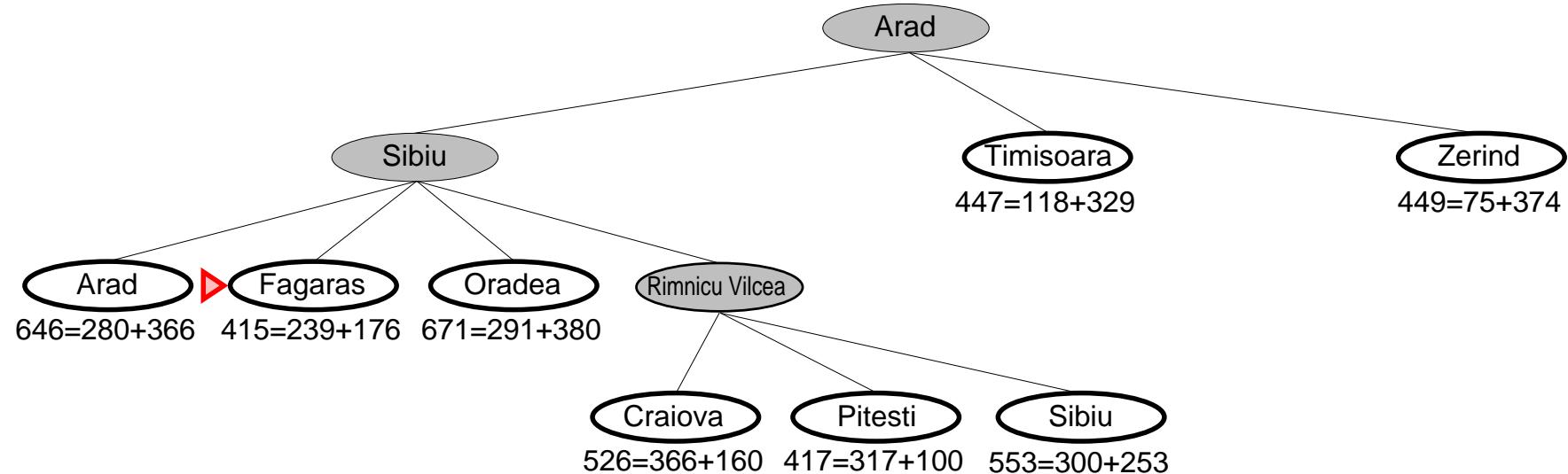
A* search example



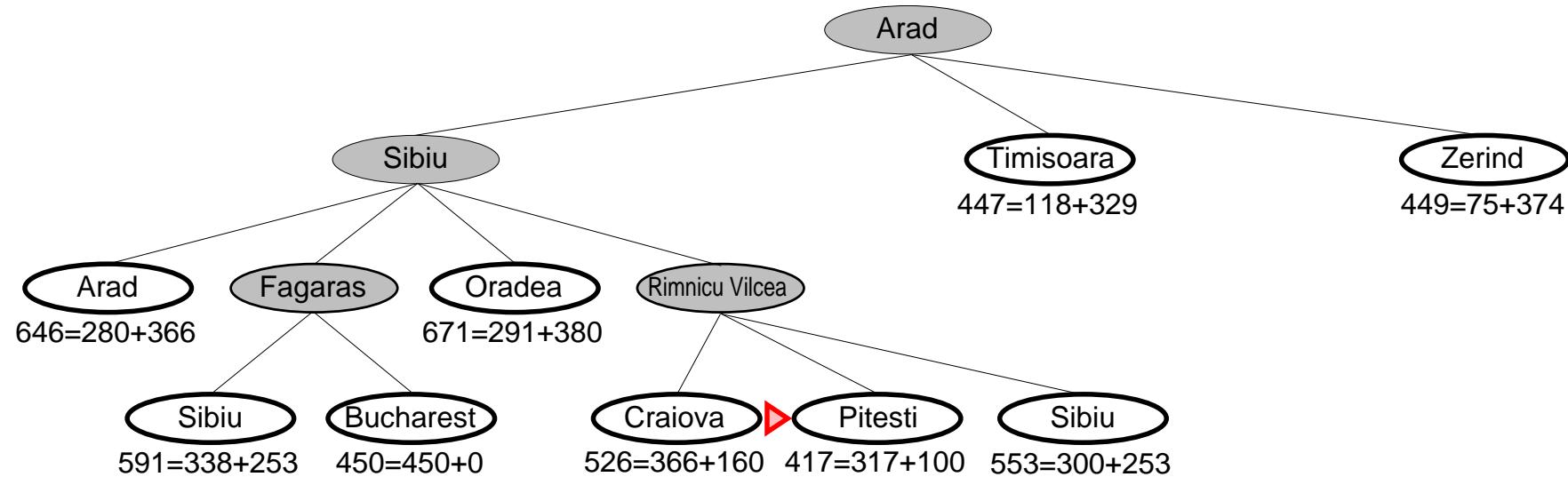
A* search example



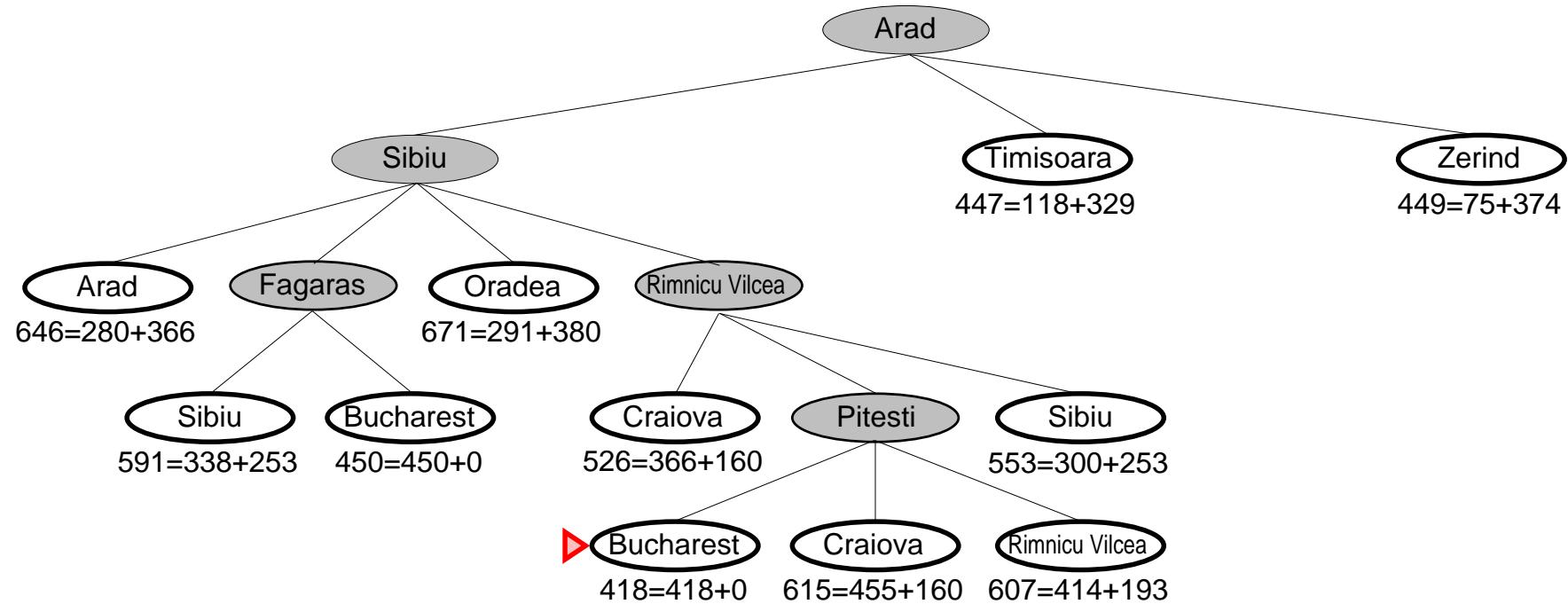
A* search example



A* search example

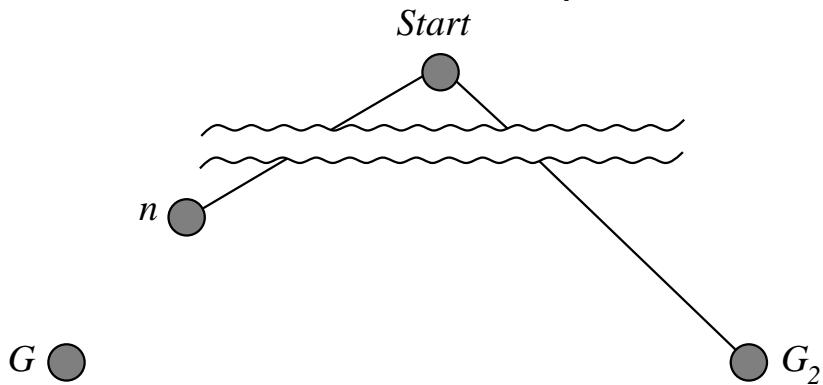


A* search example



Optimality of A* (standard proof)

Suppose some suboptimal goal G_2 has been generated and is in the queue.
Let n be an unexpanded node on a shortest path to an optimal goal G_1 .



$$\begin{aligned} f(G_2) &= g(G_2) && \text{since } h(G_2) = 0 \\ &> g(G_1) && \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) && \text{since } h \text{ is admissible} \end{aligned}$$

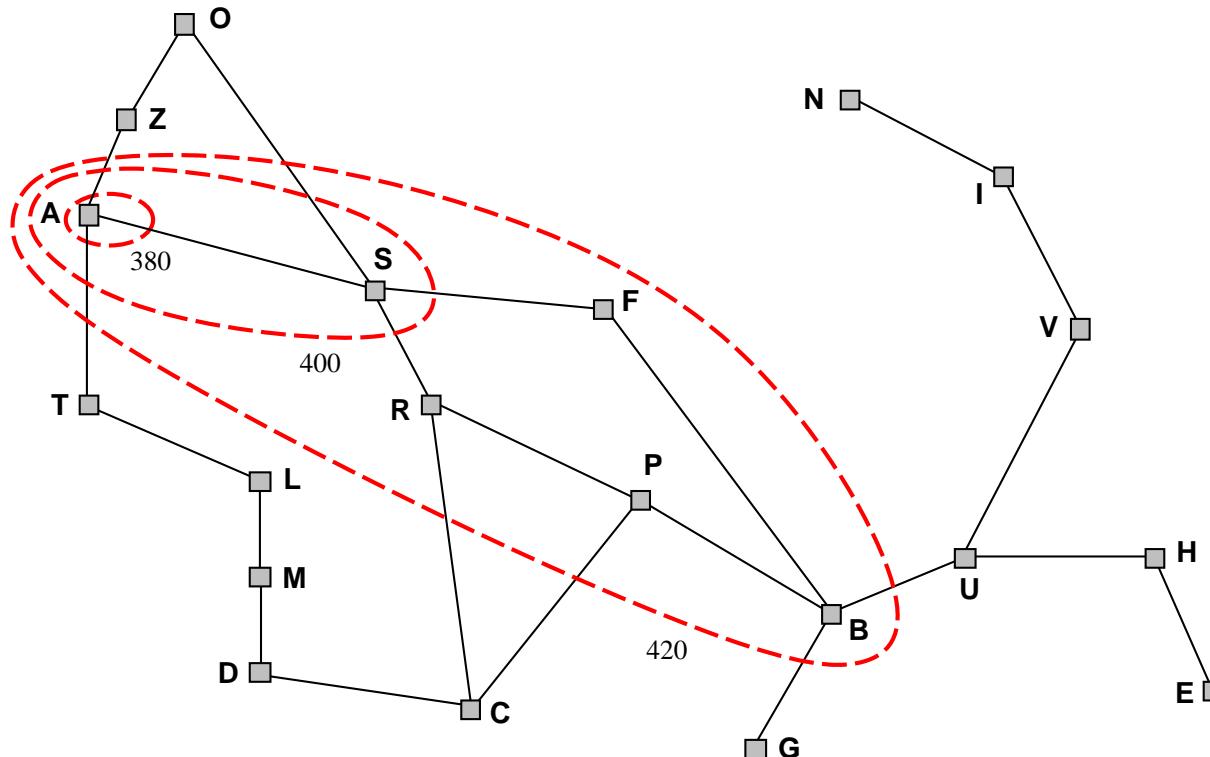
Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value*

Gradually adds “ f -contours” of nodes (cf. breadth-first adds layers)

Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A^*

Complete??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time??

Properties of \mathbf{A}^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space??

Properties of \mathbf{A}^*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Keeps all nodes in memory

Optimal??

Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

A* expands all nodes with $f(n) < C^*$

A* expands some nodes with $f(n) = C^*$

A* expands no nodes with $f(n) > C^*$

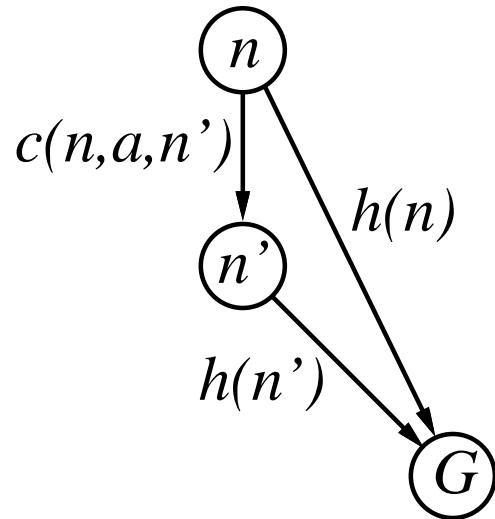
Proof of lemma: Consistency

A heuristic is **consistent** if

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

If h is consistent, we have

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



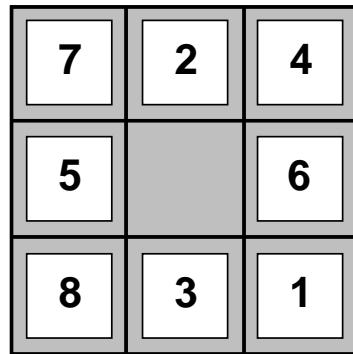
I.e., $f(n)$ is nondecreasing along any path.

Admissible heuristics

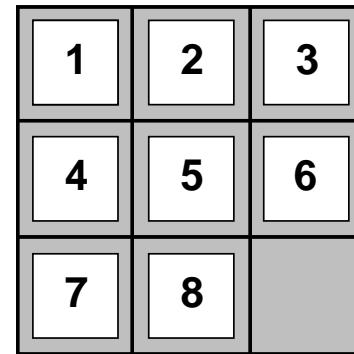
E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State



Goal State

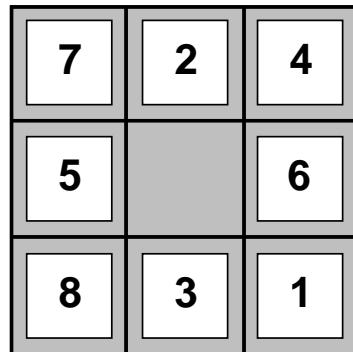
$$\begin{aligned} h_1(S) &= ?? \\ \underline{h_2(S)} &= ?? \end{aligned}$$

Admissible heuristics

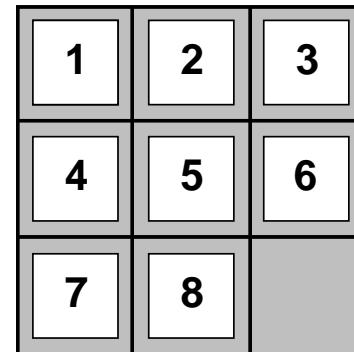
E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State



Goal State

$$h_1(S) = ?? \ 6$$

$$\underline{h_2(S) = ??} \ 4+0+3+3+1+0+2+1 = 14$$

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes

$A^*(h_1)$ = 539 nodes

$A^*(h_2)$ = 113 nodes

$d = 24$ IDS \approx 54,000,000,000 nodes

$A^*(h_1)$ = 39,135 nodes

$A^*(h_2)$ = 1,641 nodes

Given any admissible heuristics h_a, h_b ,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

Relaxed problems

Admissible heuristics can be derived from the **exact** solution cost of a **relaxed** version of the problem

If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then $h_1(n)$ gives the shortest solution

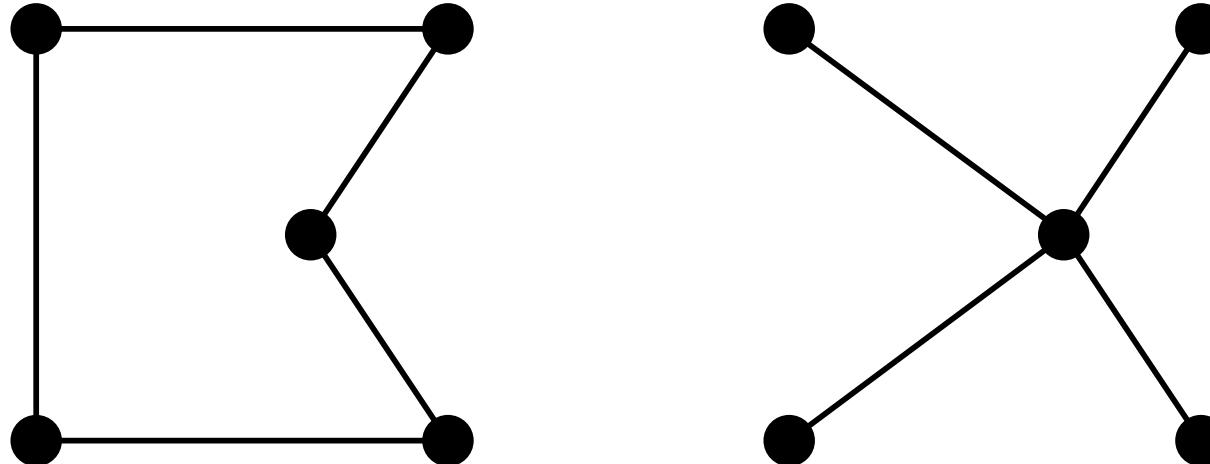
If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

Relaxed problems contd.

Well-known example: travelling salesperson problem (TSP)

Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $\mathcal{O}(n^2)$
and is a lower bound on the shortest (open) tour

Summary

Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest h

- incomplete and not always optimal

A* search expands lowest $g + h$

- complete and optimal
- also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

A* Search

- The most common informed search algorithm is A* search (pronounced “A-star search”), a best-first search that uses the evaluation function,

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

where $g(n)$ is the path cost from the initial state to node n , and $h(n)$ is the estimated cost of the shortest path from n to a goal state, so we have,

f(n) = estimated cost of the best path that continues from n to a goal.

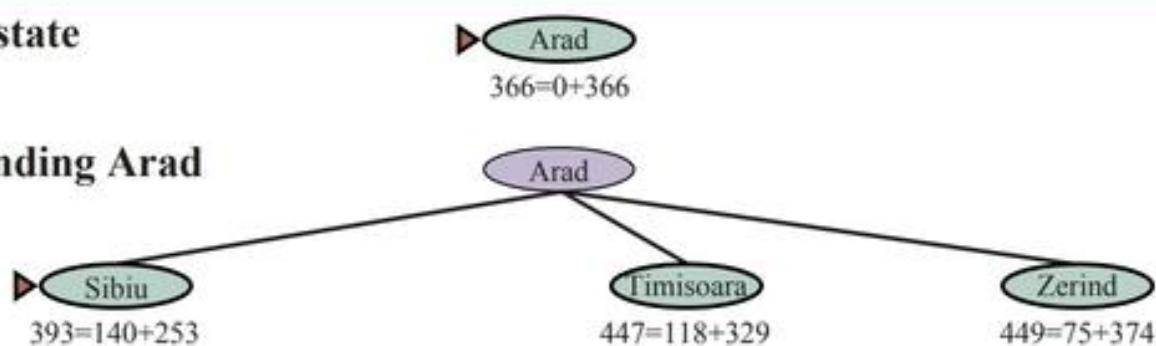
Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Figure 3.16 Values of h_{SLD} —straight-line distances to Bucharest.

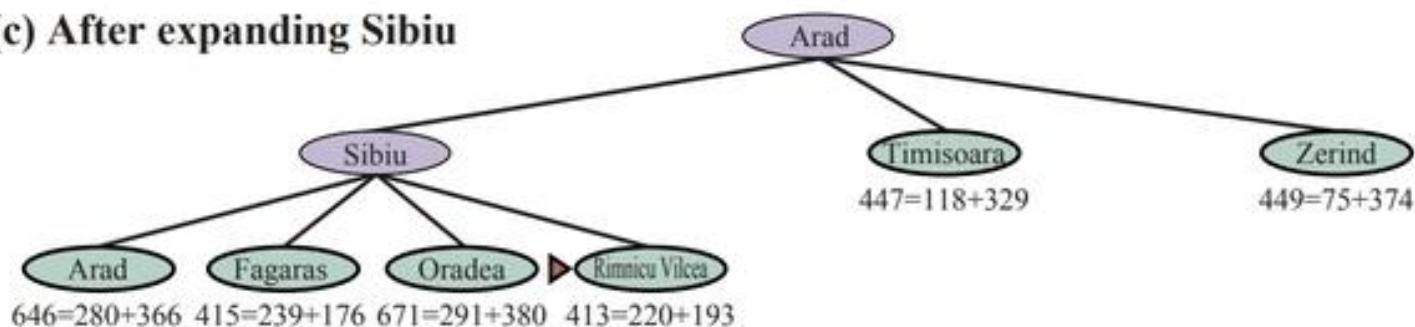
(a) The initial state



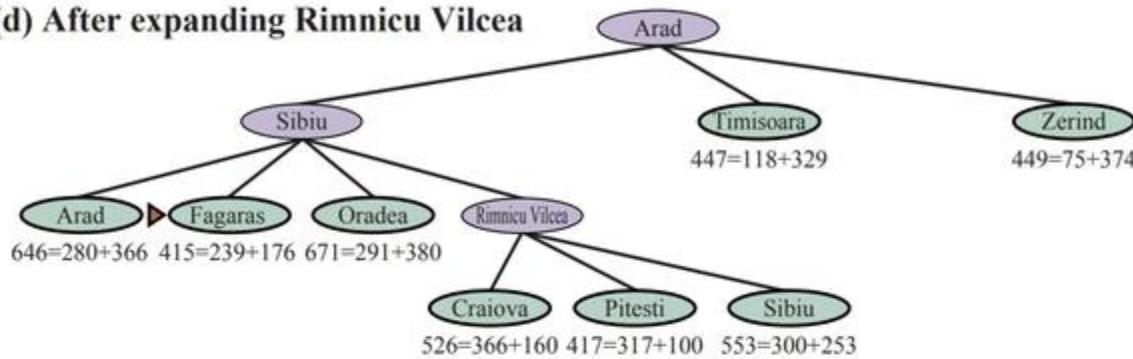
(b) After expanding Arad



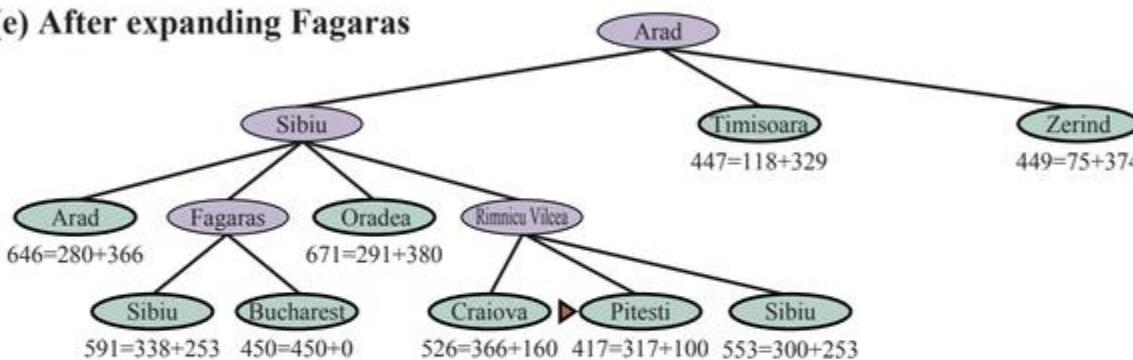
(c) After expanding Sibiu



(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti

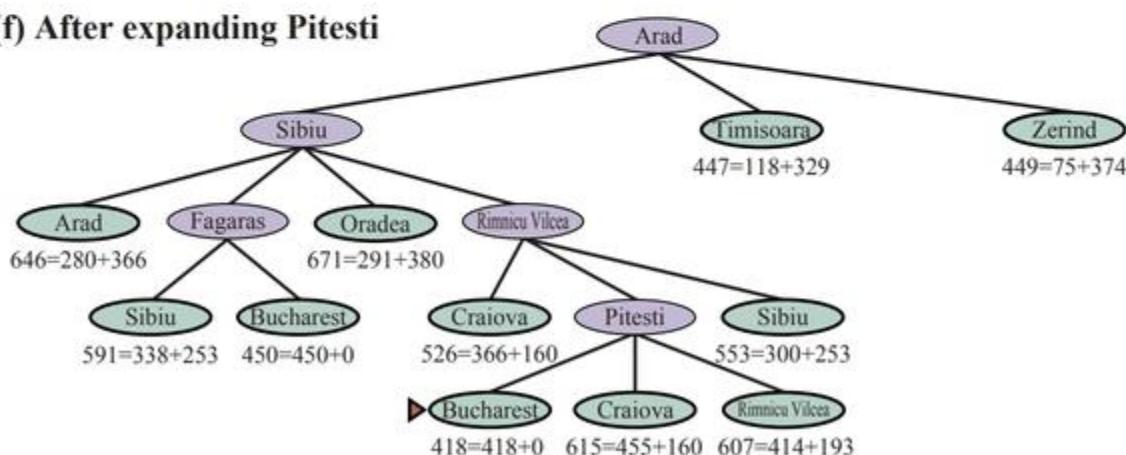


Figure 3.18 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 3.16.

A* search is complete.!!

- *Whether A* is cost-optimal depends on certain properties of the heuristic.*
- A key property is admissibility an admissible heuristic is one that never overestimates the cost to reach a goal. (An admissible heuristic is therefore optimistic.)
- With an admissible heuristic, A* is cost-optimal.
- Suppose the optimal path has cost C^* , but the algorithm returns a path with cost $C > C^*$. **Then there must be some node n which is on the optimal path and is unexpanded.**

- The notation $g^*(n)$ denote the cost of the optimal path from the start to n , and $h^*(n)$ denote the cost of the optimal path from n to the nearest goal, we have:

$$f(n) > C^* \text{ (otherwise } n \text{ would have been expanded)}$$

$$f(n) = g(n) + h(n) \text{ (by definition)}$$

$$f(n) = g^*(n) + h(n) \text{ (because } n \text{ is on an optimal path)}$$

$$f(n) \leq g^*(n) + h^*(n) \text{ (because of admissibility, } h(n) \leq h^*(n))$$

$$f(n) \leq C^* \text{ (by definition, } C^* = g^*(n) + h^*(n))$$

- a suboptimal path must be wrong—it must be that A* returns only cost-optimal paths.

- A heuristic $i(n)$ is consistent if, for every node n and every successor n' of n generated by an action a , we have:

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

- This is a form of the triangle inequality, which stipulates that a side of a triangle cannot be longer than the sum of the other two sides.

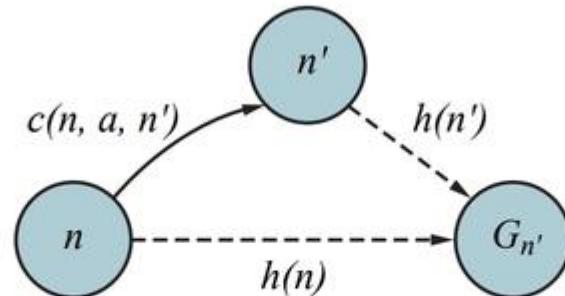


Figure 3.19 Triangle inequality: If the heuristic h is **consistent**, then the single number $h(n)$ will be less than the sum of the cost $c(n, a, n')$ of the action from n to n' plus the heuristic estimate $h(n')$.

Search contours

- A useful way to visualize a search is to draw contours in the state space, just like the contours in a topographic map.

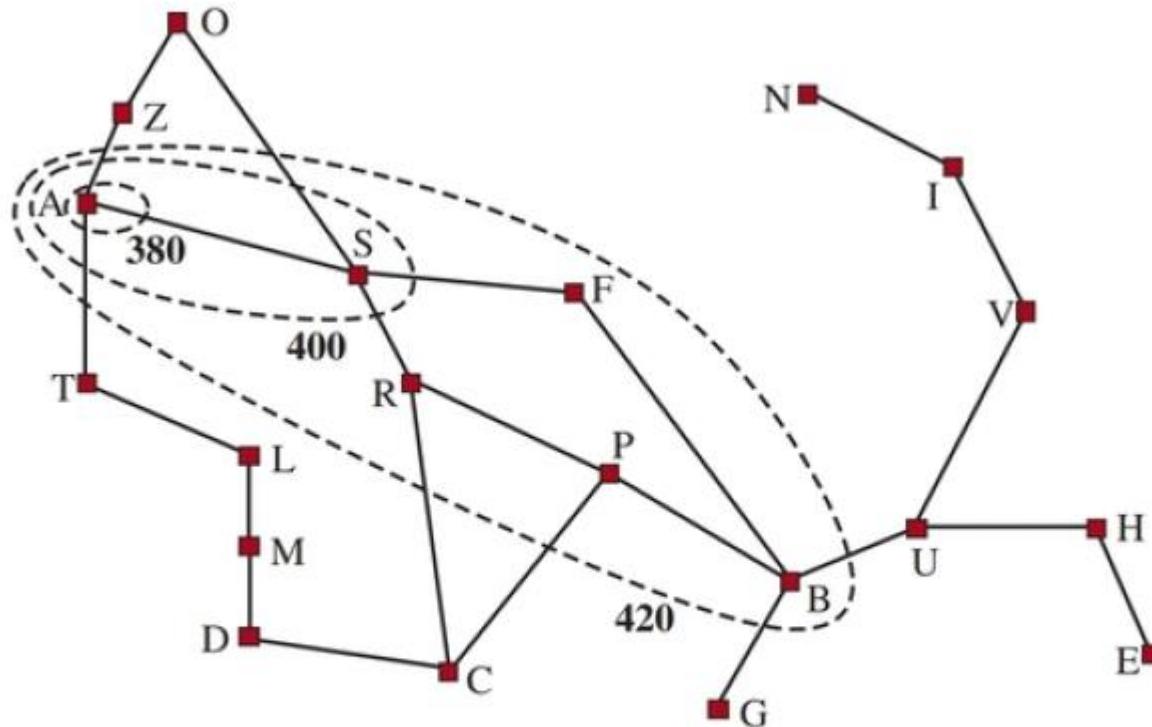


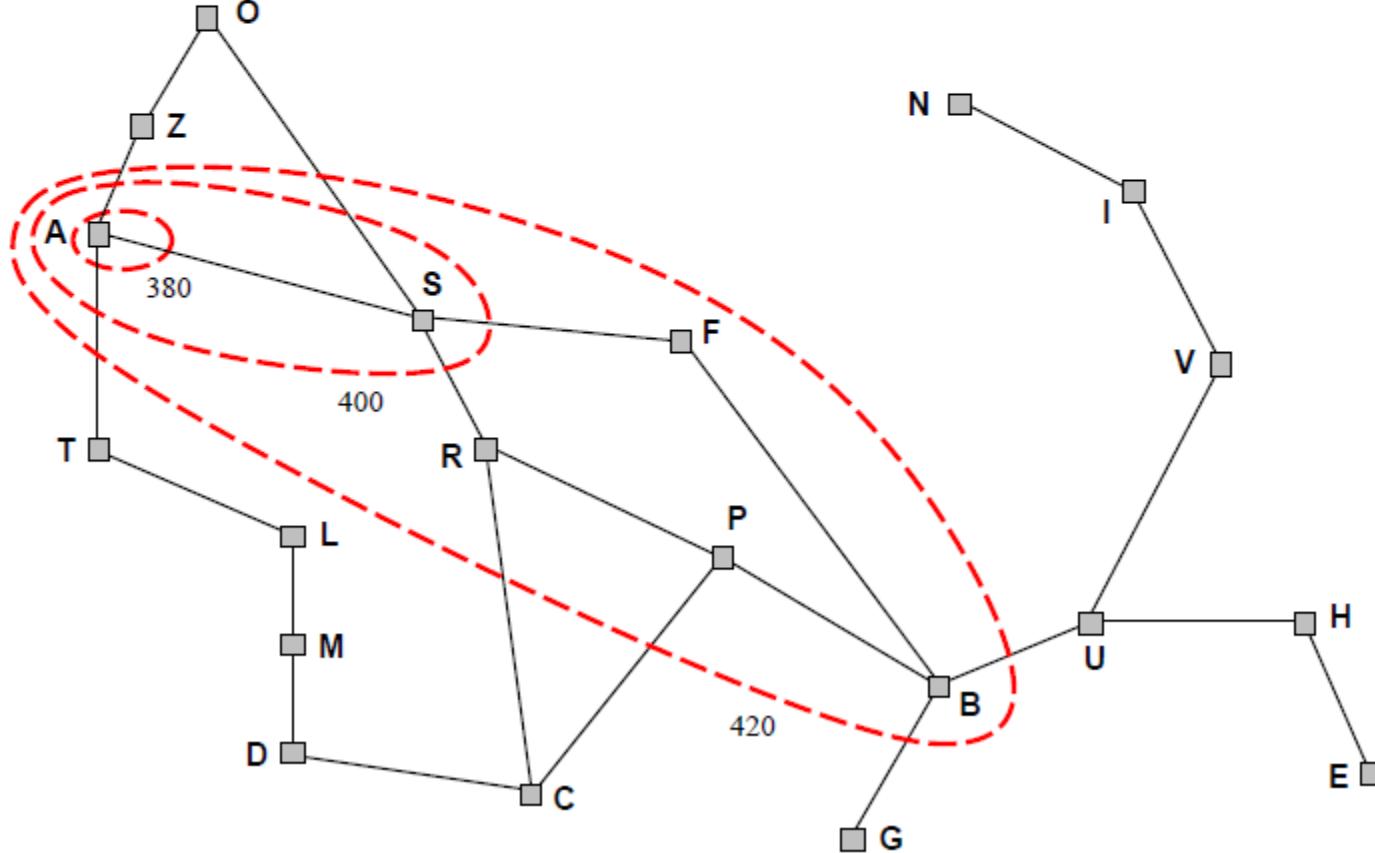
Figure 3.20 Map of Romania showing contours at $f = 380$, $f = 400$, and $f = 420$, with Arad as the start state. Nodes inside a given contour have $f = g + h$ costs less than or equal to the contour value.

Optimality of A* (more useful)

Lemma: A* expands nodes in order of increasing f value*

Gradually adds “ f -contours” of nodes (cf. breadth-first adds layers)

Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in $h \times$ length of soln.]

Space?? Keeps all nodes in memory

Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

A* expands all nodes with $f(n) < C^*$

A* expands some nodes with $f(n) = C^*$

A* expands no nodes with $f(n) > C^*$

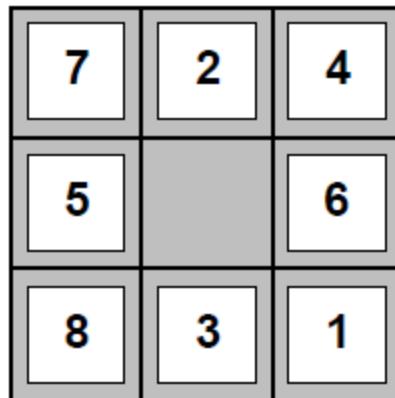
Admissible heuristics

E.g., for the 8-puzzle:

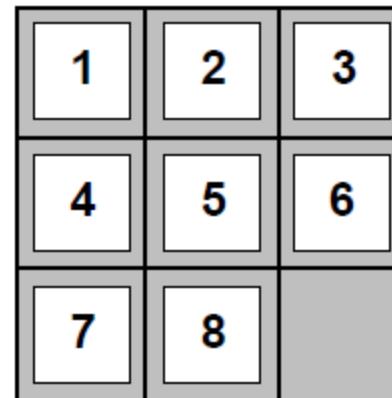
$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State



Goal State

$$h_1(S) = ??$$

$$h_2(S) = ??$$

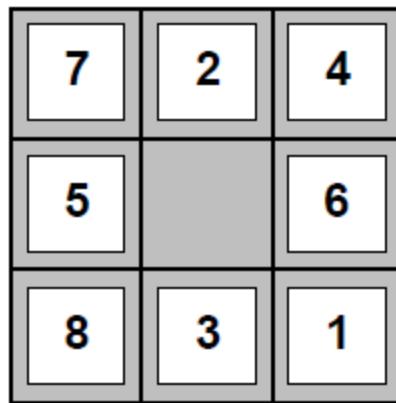
Admissible heuristics

E.g., for the 8-puzzle:

$h_1(n)$ = number of misplaced tiles

$h_2(n)$ = total Manhattan distance

(i.e., no. of squares from desired location of each tile)



Start State



Goal State

$$h_1(S) = ?? \quad 6$$

$$h_2(S) = ?? \quad 4+0+3+3+1+0+2+1 = 14$$

Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes

$A^*(h_1)$ = 539 nodes

$A^*(h_2)$ = 113 nodes

$d = 24$ IDS \approx 54,000,000,000 nodes

$A^*(h_1)$ = 39,135 nodes

$A^*(h_2)$ = 1,641 nodes

Given any admissible heuristics h_a , h_b ,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a , h_b

Relaxed problems

Admissible heuristics can be derived from the **exact** solution cost of a **relaxed** version of the problem

If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then $h_1(n)$ gives the shortest solution

If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution

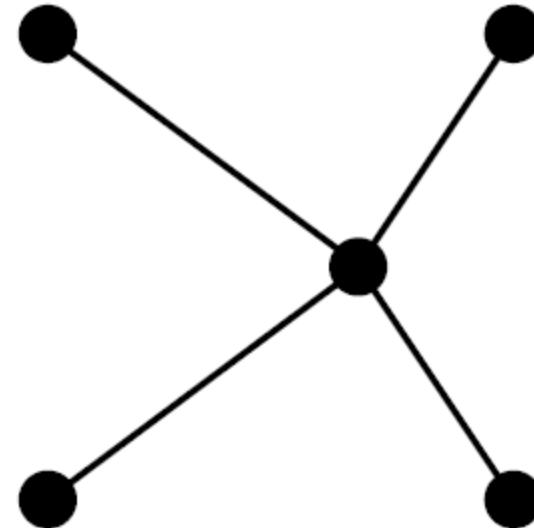
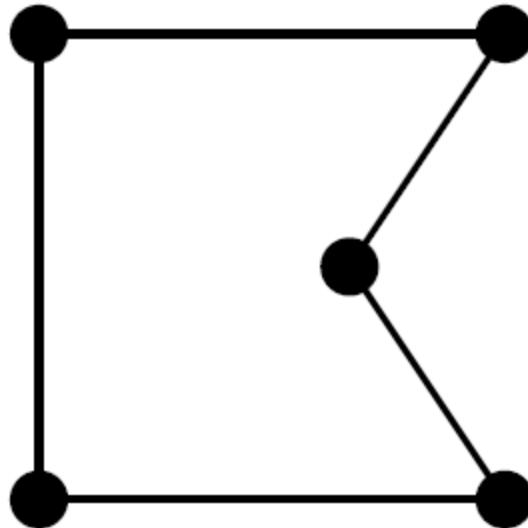
Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

- A problem with fewer restrictions on the actions is called a relaxed problem.
- The state-space graph of the relaxed problem is a supergraph of the original state space because the removal of restrictions creates added edges in the graph.
- *Hence, the cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem.*

Relaxed problems contd.

Well-known example: travelling salesperson problem (TSP)

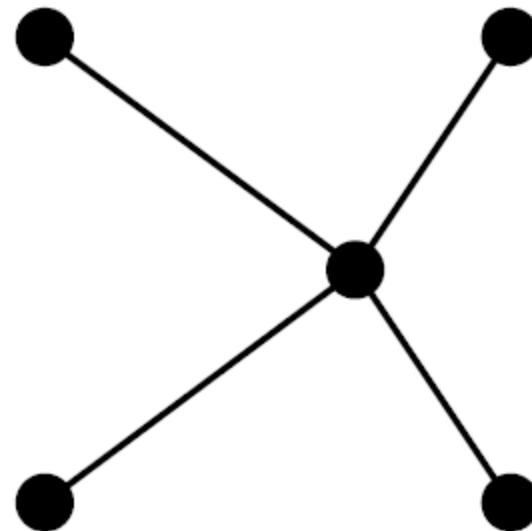
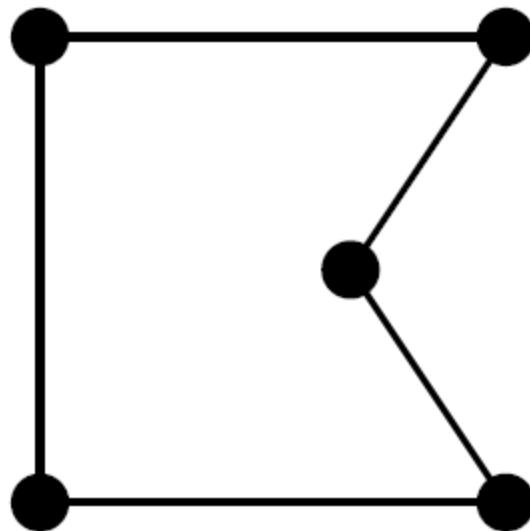
Find the shortest tour visiting all cities exactly once



Relaxed problems contd.

Well-known example: travelling salesperson problem (TSP)

Find the shortest tour visiting all cities exactly once



Minimum spanning tree can be computed in $O(n^2)$
and is a lower bound on the shortest (open) tour

Summary

Heuristic functions estimate costs of shortest paths

Good heuristics can dramatically reduce search cost

Greedy best-first search expands lowest h

- incomplete and not always optimal

A^* search expands lowest $g + h$

- complete and optimal
- also optimally efficient (up to tie-breaks, for forward search)

Admissible heuristics can be derived from exact solution of relaxed problems

Local Search Algorithms

Local Search Algorithms

- Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached.
- That means they are not systematic—they might never explore a portion of the search space where a solution actually resides.
- However, they have two key advantages:
 - (1) they use very little memory; and
 - (2) they can often find reasonable solutions in large or infinite state spaces for which systematic algorithms are unsuitable.
- Local search algorithms can also solve **optimization problems**, in which the aim is to find the best state according to an **objective function**.

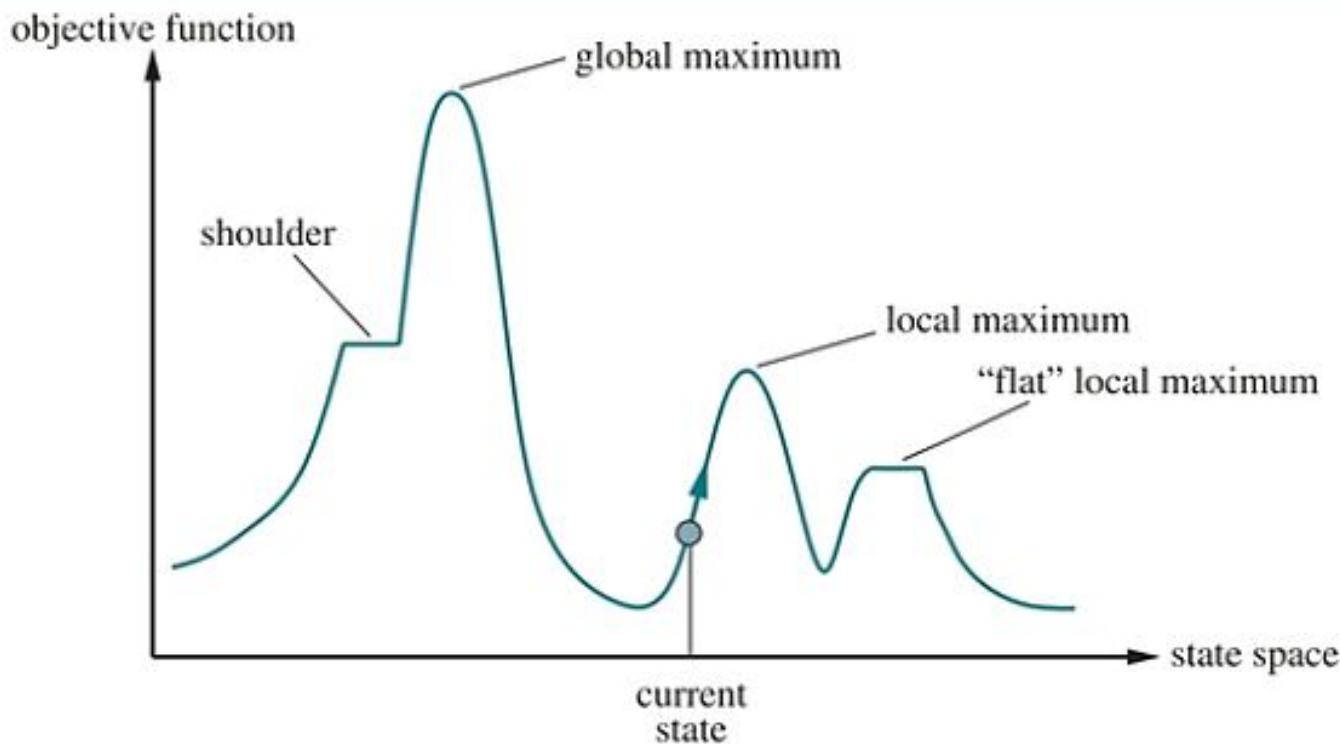


Figure 4.1 A one-dimensional state-space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum.

- Each point (state) in the landscape has an “elevation,” defined by the value of the objective function.
- If elevation corresponds to an objective function then the aim is to find the highest peak—a ***global maximum***—and we call the process **hill climbing**.
- If elevation corresponds to cost, then the aim is to find the lowest valley—a ***global minimum***—and we call it **gradient descent**.

Hill-climbing search

- It keeps track of one current state and on each iteration moves to the neighbouring state with highest value
 - that is, it heads in the direction that provides the steepest ascent.
- It terminates when it reaches a “peak” where no neighbour has a higher value.
- Hill climbing does not look ahead beyond the immediate neighbours of the current state.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum  
    current  $\leftarrow$  problem.INITIAL  
    while true do  
        neighbor  $\leftarrow$  a highest-valued successor state of current  
        if VALUE(neighbor)  $\leq$  VALUE(current) then return current  
        current  $\leftarrow$  neighbor
```

Figure 4.2 The hill-climbing search algorithm, which is the most basic local search technique. At each step the current node is replaced by the best neighbor.

Hill Climbing: Example (Minimizing h)

start

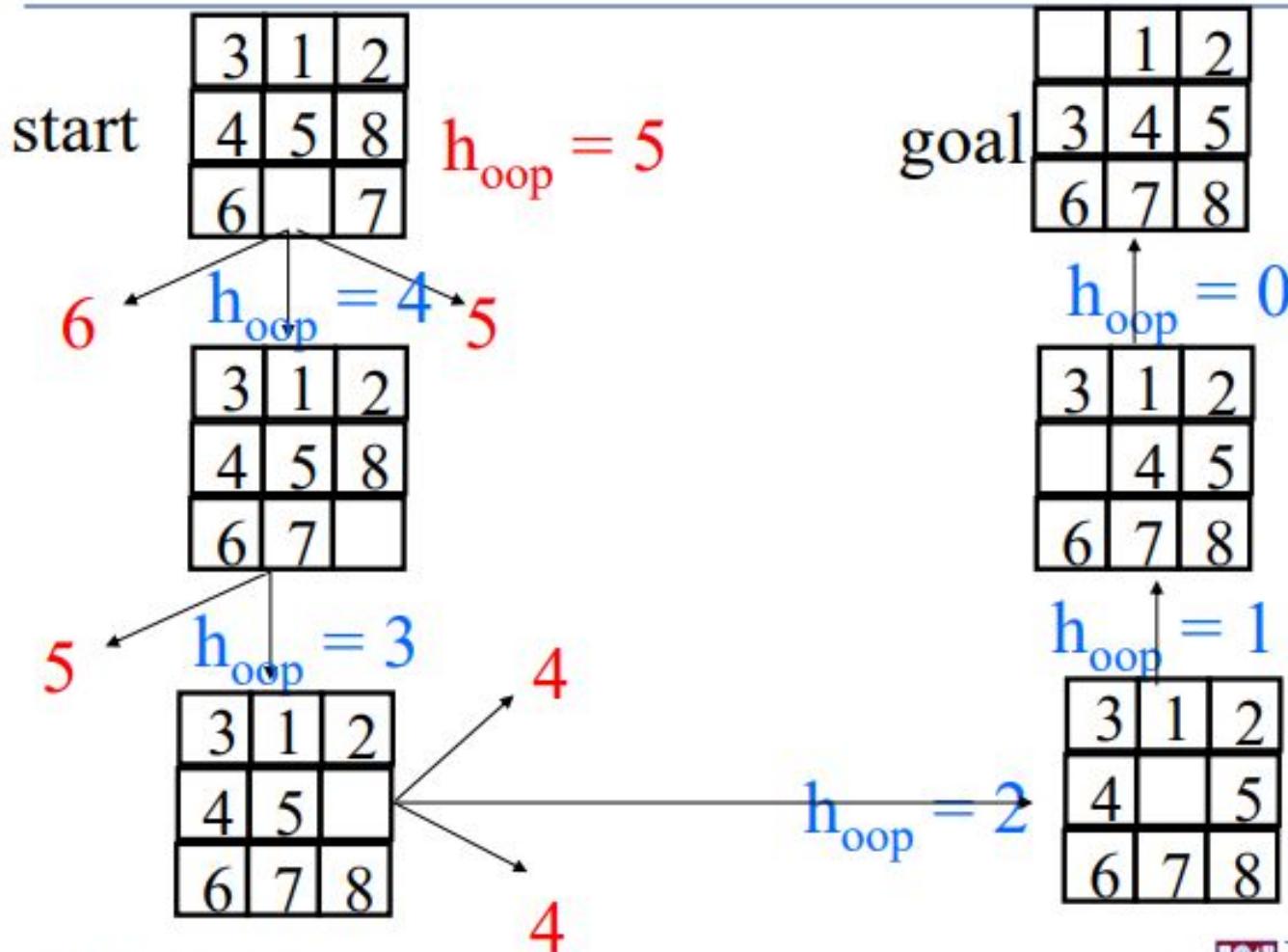
3	1	2
4	5	8
6		7

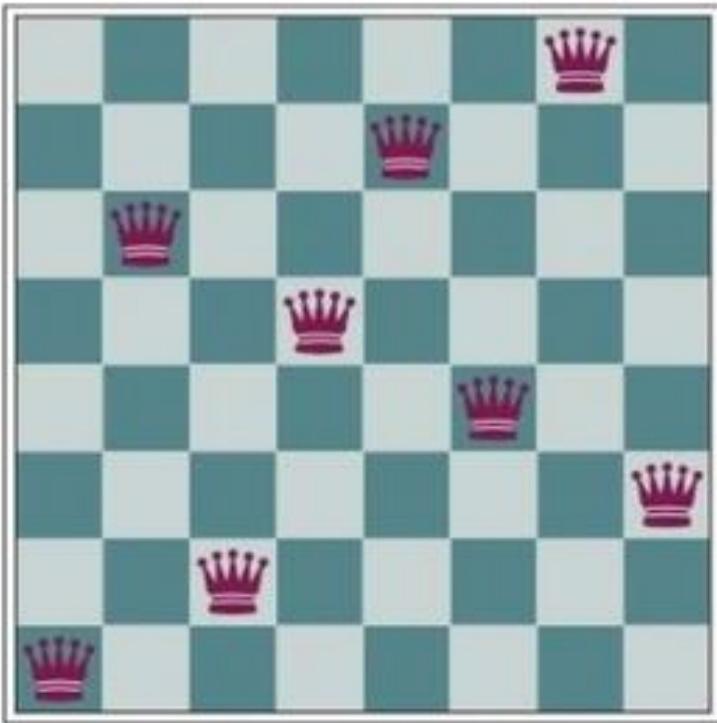
$h_{oop} = 5$

goal

	1	2
3	4	5
6	7	8

Hill Climbing: Example (Minimizing h)





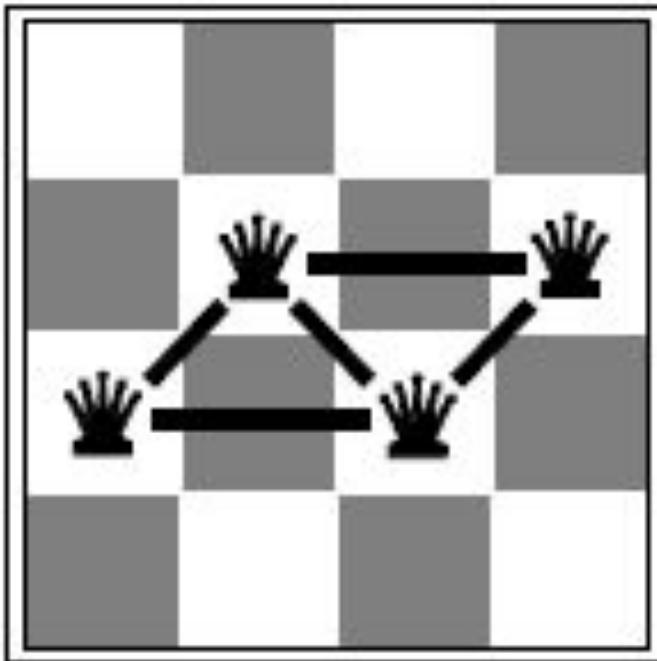
(a)

18	12	14	13	13	12	14	14
14	16	13	15	12	14	12	16
14	12	18	13	15	12	14	14
15	14	14	14	13	16	13	16
14	14	17	15	14	14	16	16
17	14	16	18	15	15	15	14
18	14	14	15	15	14	14	16
14	14	13	17	12	14	12	18

(b)

Figure 4.3 (a) The 8-queens problem: place 8 queens on a chess board so that no queen attacks another. (A queen attacks any piece in the same row, column, or diagonal.) This position is almost a solution, except for the two queens in the fourth and seventh columns that attack each other along the diagonal. (b) An 8-queens state with heuristic cost estimate $h=17$. The board shows the value of h for each possible successor obtained by moving a queen within its column. There are 8 moves that are tied for best, with $h=12$. The hill-climbing algorithm will pick one of these.

Reduce the number of conflicts



h = 5

- Hill climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next.
- Hill climbing can get stuck for any of the following reasons:
 - Local maxima: A local maximum is a peak that is higher than each of its neighboring states but lower than the global maximum.
 - Ridges: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
 - Plateaus: A plateau is a flat area of the state-space landscape

Types of Hill Climbing Algorithms

- **Stochastic hill climbing:** chooses at random from among the uphill moves; the probability of selection can vary with the steepness of the uphill move.
- **First-choice hill climbing:** implements stochastic hill climbing by generating successors randomly until one is generated that is better than the current state.
- **Random-restart hill climbing:** It conducts a series of hill-climbing searches from randomly generated initial states, until a goal is found.

- A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is always vulnerable to getting stuck in a local maximum.
- In contrast, a purely random walk that moves to a successor state without concern for the value will eventually stumble upon the global maximum, but will be extremely inefficient.
- Combination of hill climbing with a random walk in a way that yields both efficiency and completeness.

Simulated annealing

- In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state.
- Simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).
- Instead of picking the best move, however, it picks a random move. If the move improves the situation, it is always accepted.
- Otherwise, the algorithm accepts the move with some probability less than 1.

- Basic idea:
 - Allow “bad” moves occasionally, depending on “temperature”
 - High temperature => more bad moves allowed, shake the system out of its local minimum
 - Gradually reduce temperature according to some schedule
 - Sounds pretty flaky, doesn’t it?

Simulated annealing algorithm

```
function SIMULATED-ANNEALING(problem,schedule) returns a state
    current ← problem.initial-state
    for t = 1 to ∞ do
        T ← schedule(t)
        if T = 0 then return current
        next ← a randomly selected successor of current
        ΔE ← next.value – current.value
        if ΔE > 0 then current ← next
        else current ← next only with probability  $e^{\Delta E/T}$ 
```

Simulated Annealing

- The probability decreases exponentially with the “badness” b of the move—the amount ΔE by which the evaluation is worsened.
- The probability also decreases as the “temperature” T goes down: “bad” moves are more likely to be allowed at the start when T is high, and they become more unlikely as T decreases.
- If the schedule lowers T to 0 slowly enough, then a property of the Boltzmann distribution,
$$e^{\Delta E/T}$$
- Is that all the probability is concentrated on the global maxima, which the algorithm will find with probability approaching 1.

Simulated Annealing

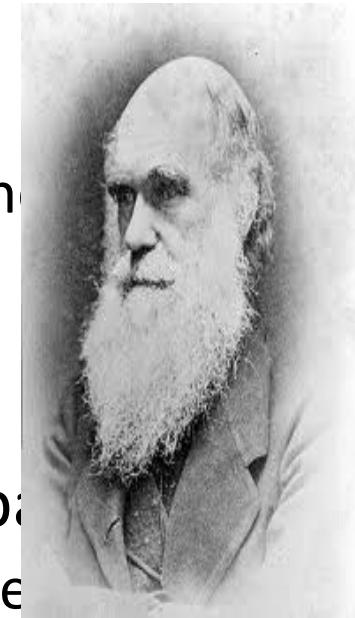
- Is this convergence an interesting guarantee?
- Sounds like magic, but reality is reality:
 - The more downhill steps you need to escape a local optimum, the less likely you are to ever make them all in a row
 - “Slowly enough” may mean exponentially slowly
 - Random restart hillclimbing also converges to optimal state...
- Simulated annealing and its relatives are a key workhorse in VLSI layout and other optimal configuration problems

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  current  $\leftarrow$  problem.INITIAL
  for t = 1 to  $\infty$  do
    T  $\leftarrow$  schedule(t)
    if T = 0 then return current
    next  $\leftarrow$  a randomly selected successor of current
     $\Delta E \leftarrow \text{VALUE}(\textit{current}) - \text{VALUE}(\textit{next})$ 
    if  $\Delta E > 0$  then current  $\leftarrow$  next
    else current  $\leftarrow$  next only with probability  $e^{-\Delta E/T}$ 
```

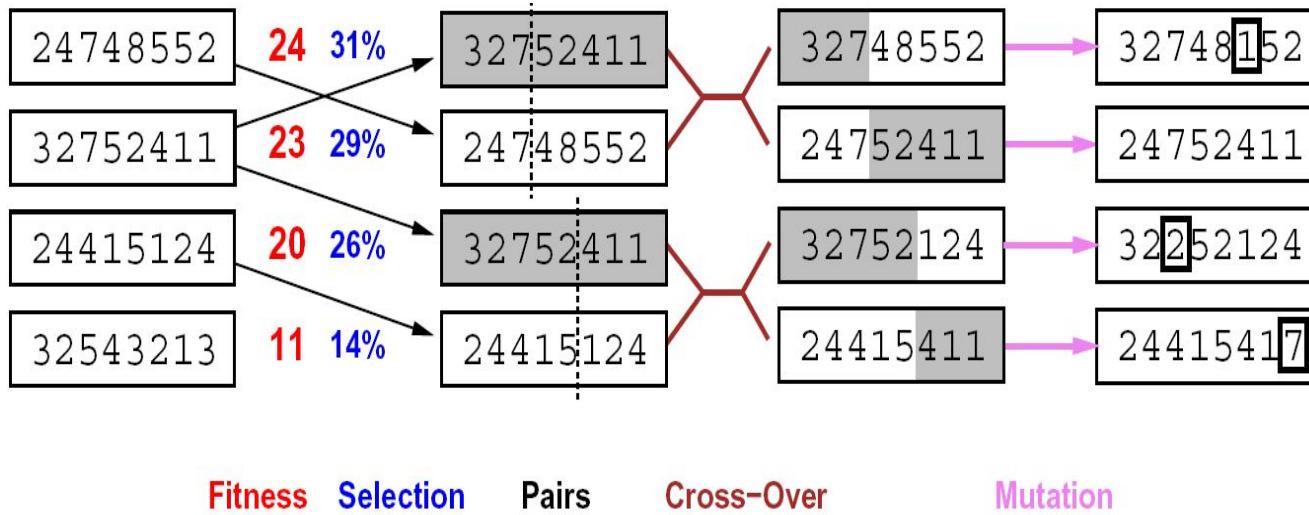
Figure 4.5 The simulated annealing algorithm, a version of stochastic hill climbing where some downhill moves are allowed. The *schedule* input determines the value of the “temperature” *T* as a function of time.

Local beam search

- Basic idea:
 - K copies of a local search algorithm, initialized randomly
 - For each iteration
 - Generate ALL successors from K current states
 - Choose best K of these to be the new current states
- Why is this different from K local searches in parallel?
 - The searches **communicate**! “Come over here, there’s greener!”
- What other well-known algorithm does this remind you of?
 - Evolution!

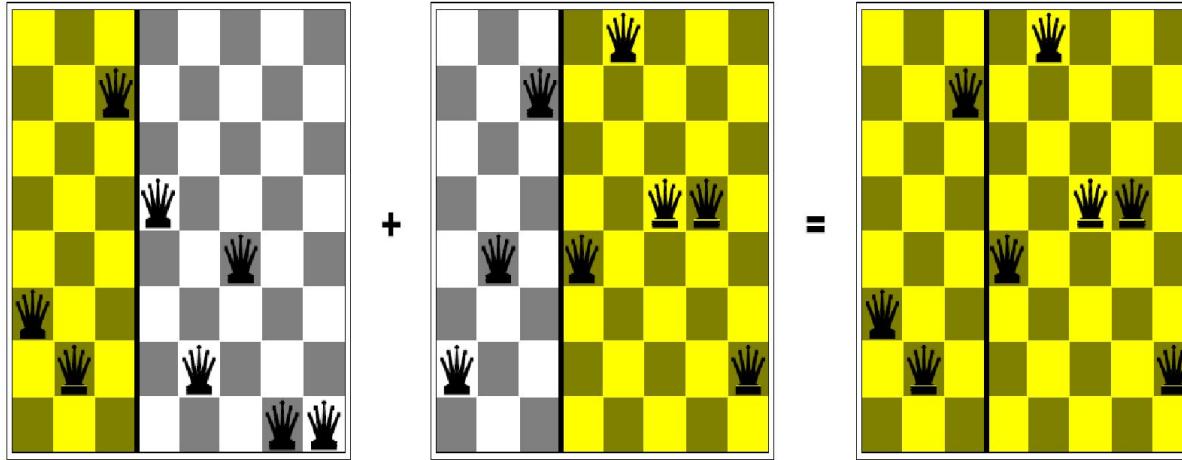


Genetic algorithms



- Genetic algorithms use a natural selection metaphor
 - Resample K individuals at each step (selection) weighted by fitness function
 - Combine by pairwise crossover operators, plus mutation to give variety

Example: N-Queens



- Does crossover make sense here?
- What would mutation be?
- What would a good fitness function be?

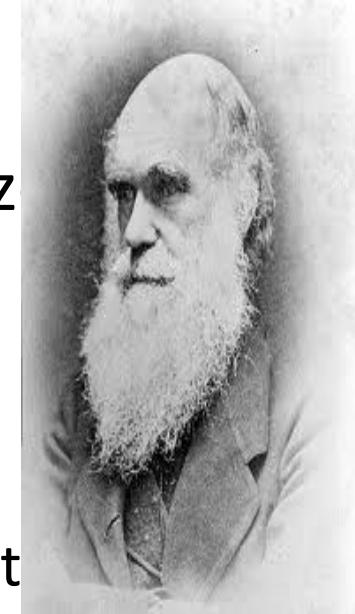
Summary

- Many configuration and optimization problems can be formulated as local search
- General families of algorithms:
 - Hill-climbing, continuous optimization
 - Simulated annealing (and other stochastic methods)
 - Local beam search: multiple interaction searches
 - Genetic algorithms: break and recombine states

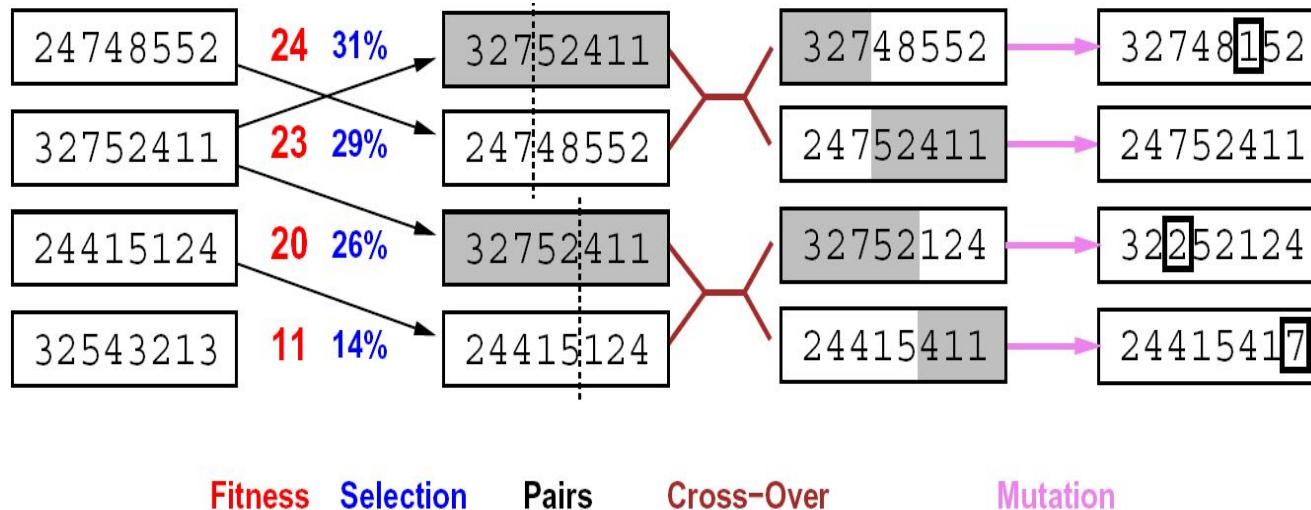
Many machine learning algorithms are local searches

Local beam search

- Basic idea:
 - K copies of a local search algorithm, initialized randomly
 - For each iteration
 - Generate ALL successors from K current states
 - Choose **best K** of these to be the new current state
- Or, K chosen randomly with a bias towards good ones
- Why is this different from K local searches in parallel?
 - The searches **communicate**! “Come over here, the grass is greener!”
- What other well-known algorithm does this

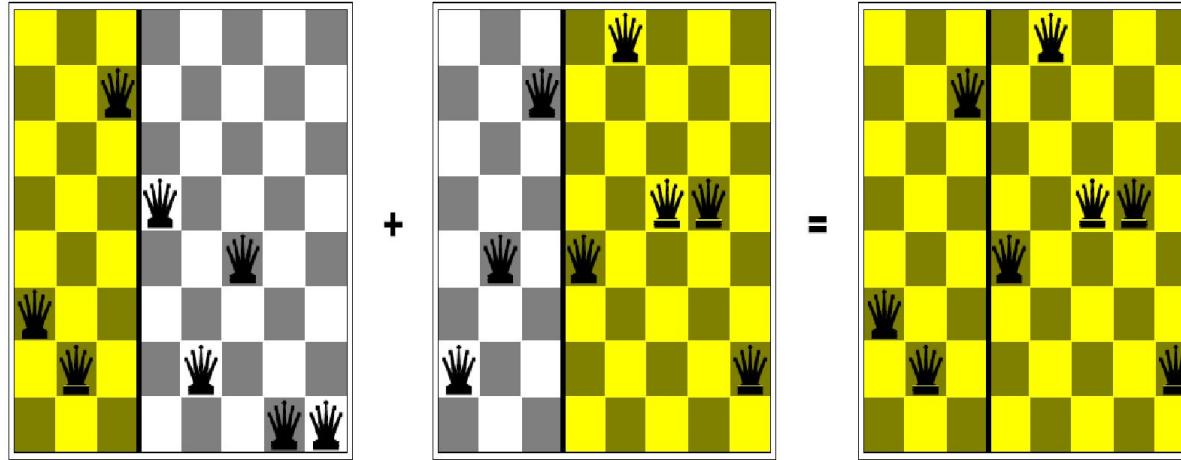


Genetic algorithms



- Genetic algorithms use a natural selection metaphor
 - Resample K individuals at each step (selection) weighted by fitness function
 - Combine by pairwise crossover operators, plus mutation to give variety

Example: N-Queens



- Does crossover make sense here?
- What would mutation be?
- What would a good fitness function be?

Summary

- Many configuration and optimization problems can be formulated as local search
- General families of algorithms:
 - Hill-climbing, continuous optimization
 - Simulated annealing (and other stochastic methods)
 - Local beam search: multiple interaction searches
 - Genetic algorithms: break and recombine states

Genetic Algorithms

Credits:

Richard Frankel

Stanford University

And

Anas S. To'meh

Outline

- Motivations/applicable situations
- What are genetic algorithms?
- Example
- Pros and cons

Motivation

- Searching some search spaces with traditional search methods would be intractable. This is often true when states/candidate solutions have a large number of successors.
 - Example: Designing the surface of an aircraft.

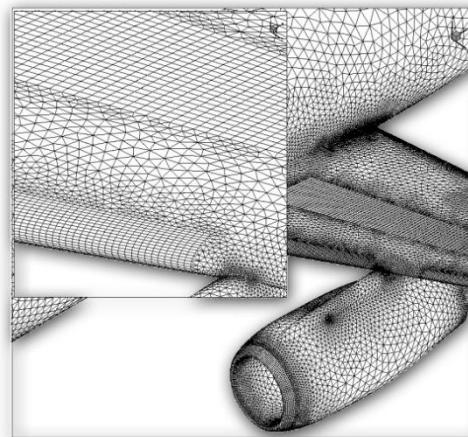
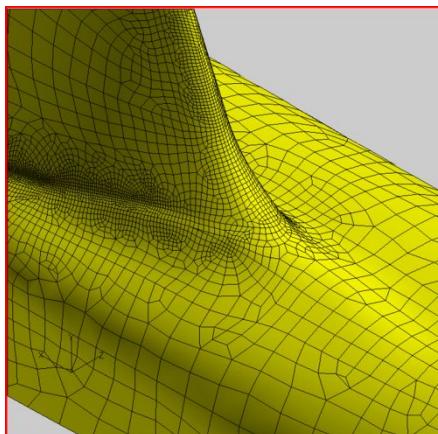


Image source: <http://www.centaursoft.com/cms/index.php?section=25>

Applicable situations

- Often used for optimization (scheduling, design, etc.) problems, though can be used for many other things as well, as we'll see a bit later.
 - Good problem for GA: Scheduling air traffic
 - Bad problems for GA: Finding large primes (why?)

Applicable situations

- Genetic algorithms work best when the “fitness landscape” is continuous (in some dimensions). This is also true of standard search, e.g. A*.
 - Intuitively, this just means that we can find a heuristic that gives a rough idea of how close a candidate is to being a solution.

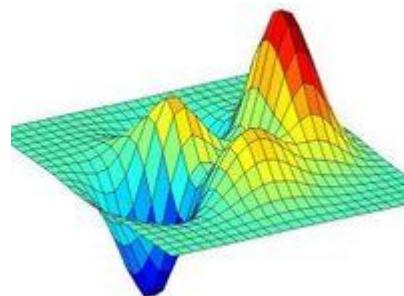


Image source: scholarpedia.org

So what *is* a genetic algorithm?

- Genetic algorithms are a randomized heuristic search strategy.
- Basic idea: Simulate natural selection, where the population is composed of *candidate solutions*.
- Focus is on evolving a population from which strong and diverse candidates can emerge via mutation and crossover (mating).

Basic algorithm

- Create an initial population, either random or “blank”.
- While the best candidate so far is not a solution:
 - Create new population using successor functions.
 - Evaluate the fitness of each candidate in the population.
- Return the best candidate found.

Simple example – alternating string

- Let's try to evolve a length 4 alternating string
- Initial population: $C1=1000$, $C2=0011$
- We roll the dice and end up creating $C1' = \text{cross } (C1, C2) = 1011$ and $C2' = \text{cross } (C1, C1) = 1000$.
- We mutate $C1'$ and the fourth bit flips, giving 1010. We mutate $C2'$ and get 1001.
- We run our solution test on each. If $C1'$ is a solution, so we return it and are done.

Basic components

- Candidate representation
 - Important to choose this well. More work here means less work on the successor functions.
- Successor function(s)
 - Mutation, crossover
- Fitness function
- Solution test
- Some parameters
 - Population size
 - Generation limit

Candidate representation

- We want to encode candidates in a way that makes mutation and crossover easy.
- The typical candidate representation is a binary string. This string can be thought of as the genetic code of a candidate – thus the term “genetic algorithm”!
 - Other representations are possible, but they make crossover and mutation harder.

Candidate representation example

- Let's say we want to represent a rule for classifying bikes as mountain bikes or hybrid, based on these attributes*:
 - Make (Bridgestone, Cannondale, Nishiki, or Gary Fisher)
 - Tire type (knobby, treads)
 - Handlebar type (straight, curved)
 - Water bottle holder (*Boolean*)
- We can encode a rule as a binary string, where each bit represents whether a value is accepted.

Make	Tires	Handlebars	Water bottle
B C N G	K T	S C	Y N

*Bikes scheme used with permission from Mark Maloof.

Candidate representation example

- The candidate will be a bit string of length 10, because we have 10 possible attribute values.
- Let's say we want a rule that will match any bike that is made by Bridgestone or Cannondale, has treaded tires, and has straight handlebars. This rule could be represented as 1100011011:

Make	Tires		Handlebars		Water bottle				
1 B	1 C	0 N	0 G	0 K	1 T	1 S	0 C	1 Y	1 N

Successor functions

- Mutation – Given a candidate, return a slightly different candidate.
- Crossover – Given two candidates, produce one that has elements of each.
- We don't always generate a successor for each candidate. Rather, we generate a successor *population* based on the candidates in the current population, weighted by fitness.

Successor functions

- If your candidate representation is just a binary string, then these are easy:
 - Mutate(c): Copy c as c' . For each bit b in c' , flip b with probability p . Return c' .
 - Cross (c_1, c_2): Create a candidate c such that $c[i] = c_1[i]$ if $i \% 2 = 0$, $c[i] = c_2[i]$ otherwise. Return c .
 - Alternatively, any other scheme such that c gets roughly equal information from c_1 and c_2 .

Fitness function

- The fitness function is analogous to a heuristic that estimates how close a candidate is to being a solution.
- In general, the fitness function should be consistent for better performance.
- However, even if it is, there are no guarantees. This is a probabilistic algorithm!

Solution test

- Given a candidate, return whether the candidate is a solution.
- Often just answers the question “does the candidate satisfy some set of constraints?”
- Optional! Sometimes you just want to do the best you can in a given number of generations.

New population generation

- How do we come up with a new population?
 - Given a population P , generate P' by performing crossover $|P|$ times, each time selecting candidates with probability proportional to their fitness.
 - Get P'' by mutating each candidate in P' .
 - Return P'' .

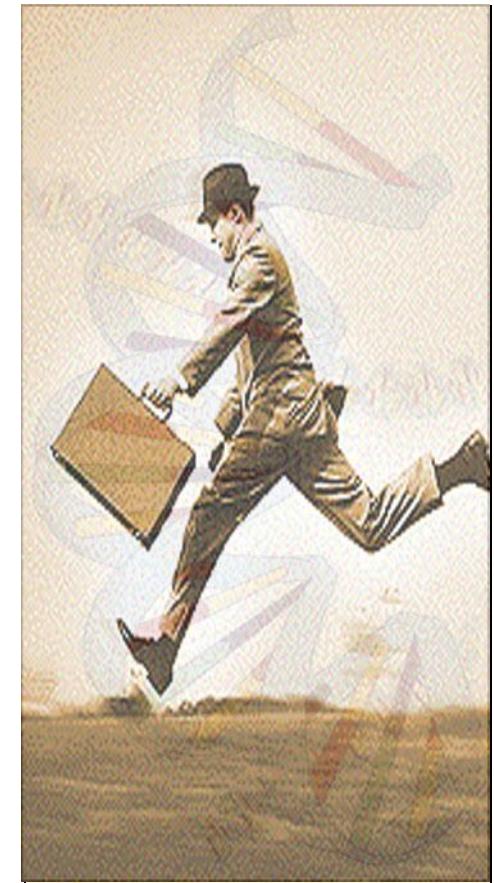
Basic algorithm (recap)

- Create an initial population, either random or “blank”.
- While the best candidate so far is not a solution:
 - Create new population using successor functions.
 - Evaluate the fitness of each candidate in the population.
- Return the best candidate found.

Knapsack Problem

Problem Description:

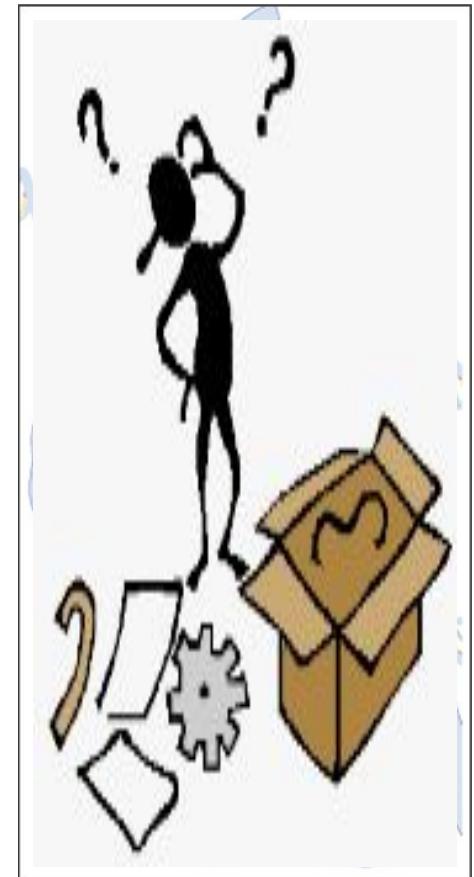
- You are going on a picnic.
- And have a number of items that you could take along.
- Each item has a weight and a benefit or value.
- You can take one of each item at most.
- There is a capacity limit on the weight you can carry.
- You should carry items with max. values.



Knapsack Problem

Example:

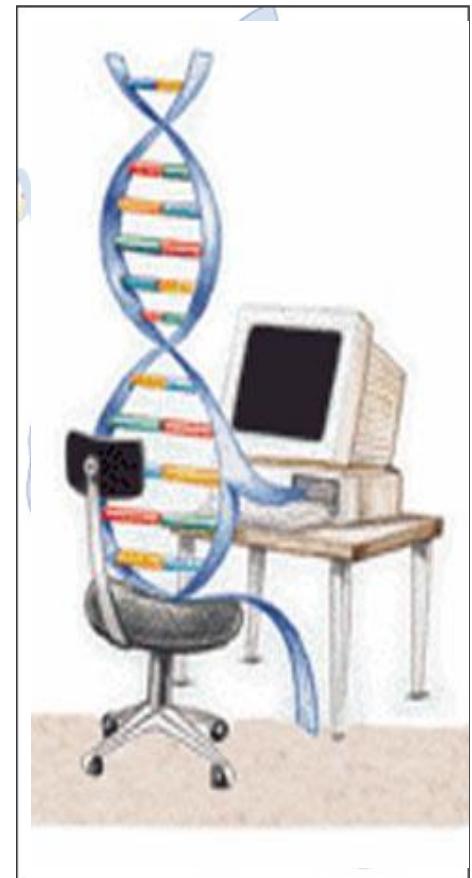
- Item: 1 2 3 4 5 6 7
- Benefit: 5 8 3 2 7 9 4
- Weight: 7 8 4 10 4 6 4
- Knapsack holds a maximum of 22 pounds
- Fill it to get the maximum benefit



Genetic Algorithm

Outline of the Basic Genetic Algorithm

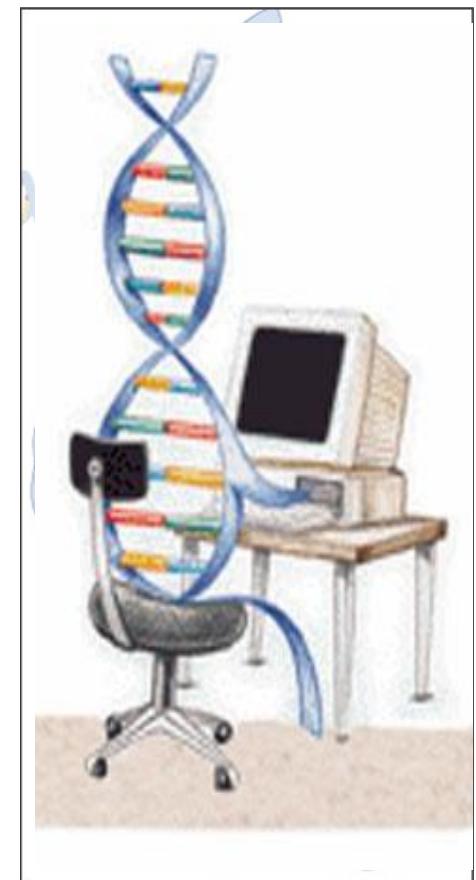
1. **[Start]**
 - ✓ Encoding: represent the individual.
 - ✓ Generate random population of n chromosomes (suitable solutions for the problem).
2. **[Fitness]** Evaluate the fitness of each chromosome.
3. **[New population]** repeating following steps until the new population is complete.
4. **[Selection]** Select the best two parents.
5. **[Crossover]** cross over the parents to form a new offspring (children).



Genetic Algorithm

Outline of the Basic Genetic Algorithm Cont.

6. **[Mutation]** With a mutation probability.
7. **[Accepting]** Place new offspring in a new population.
8. **[Replace]** Use new generated population for a further run of algorithm.
9. **[Test]** If the end condition is satisfied, then **stop**.
10. **[Loop]** Go to step 2 .



Basic Steps

Start

- Encoding: 0 = not exist, 1 = exist in the Knapsack

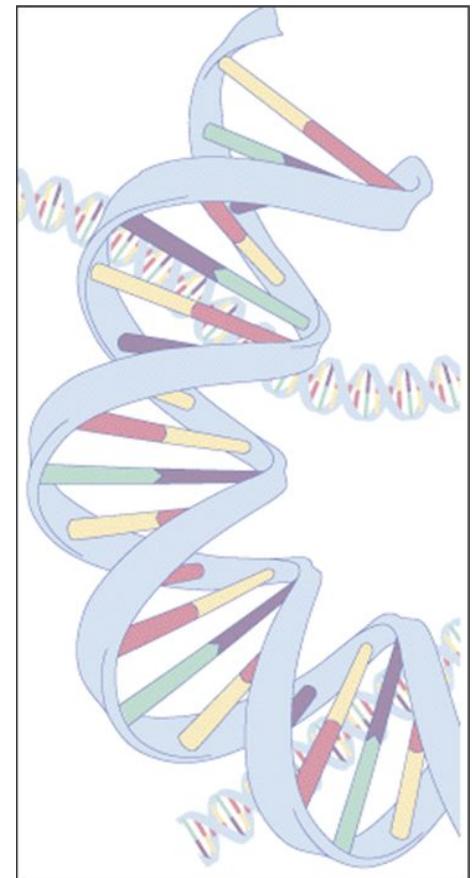
Chromosome: 1010110

Item.	1	2	3	4	5	6	7
Chro	1	0	1	0	1	1	0
Exist?	y	n	y	n	y	y	n

=> Items taken: 1, 3 , 5, 6.

- Generate random population of n chromosomes:

- 0101010
- 1100100
- 0100011



Basic Steps Cont.

Fitness & Selection

a) 0101010: Benefit= 19, Weight= 24

Item	1	2	3	4	5	6	7
Chro	0	1	0	1	0	1	0
Benefit	5	8	3	2	7	9	4
Weight	7	8	4	10	4	6	4

b) 1100100: Benefit= 20, Weight= 19.

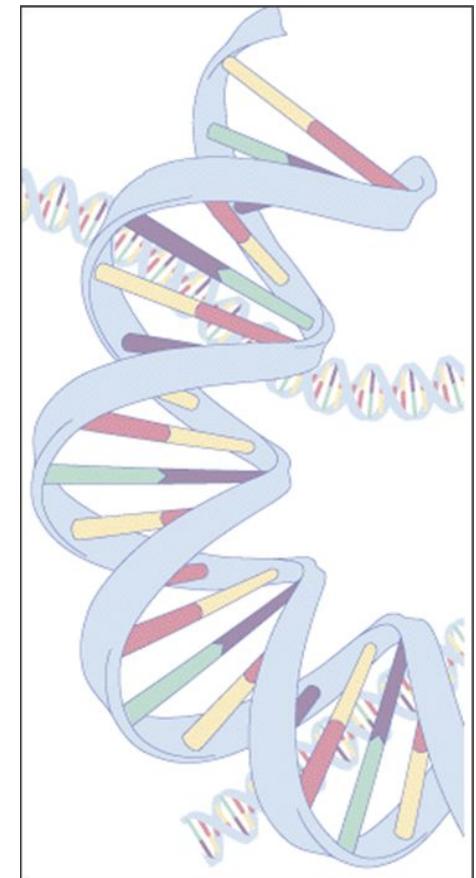
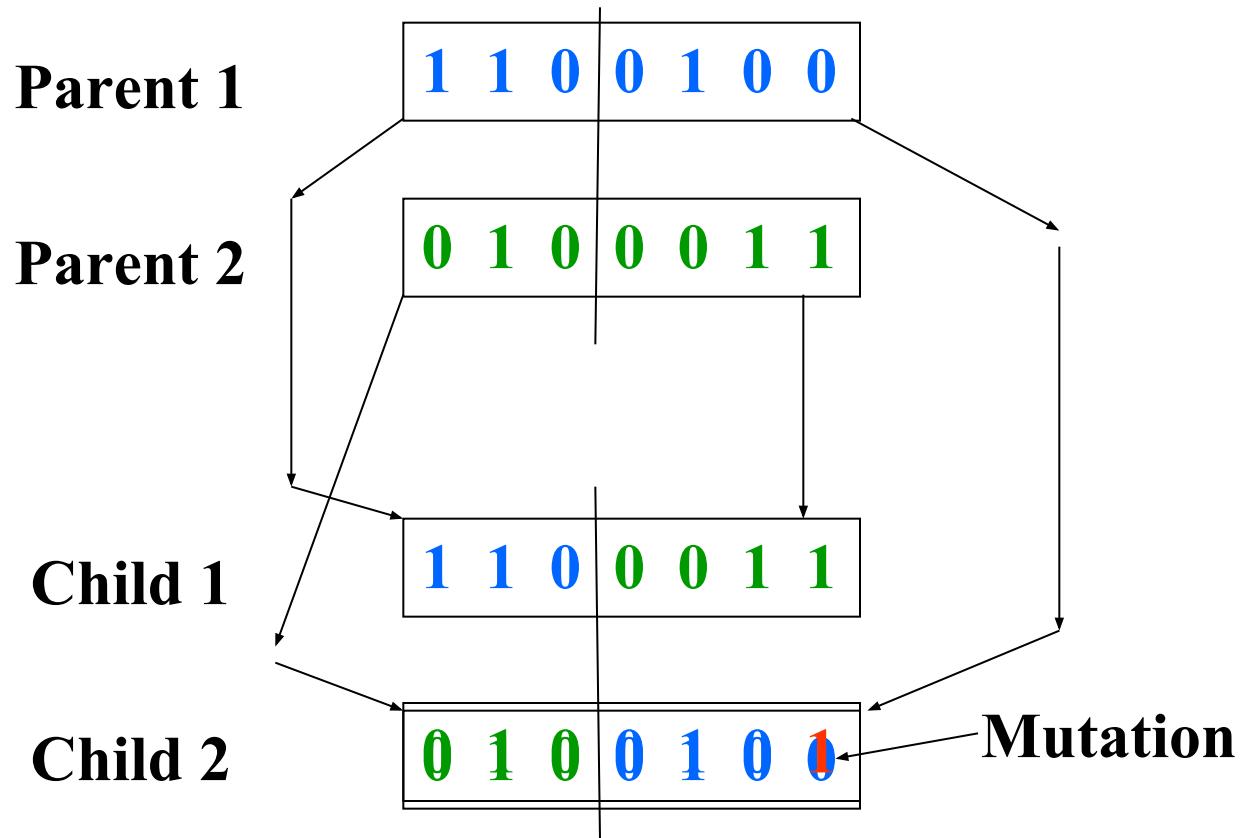
c) 0100011: Benefit= 21, Weight= 18.

=> We select Chromosomes b & c.



Basic Steps Cont.

Crossover & Mutation



Basic Steps Cont.

Accepting, Replacing & Testing

- ✓ Place new offspring in a new population.
- ✓ Use new generated population for a further run of algorithm.
- ✓ If the end condition is satisfied, then **stop**.
End conditions:
 - Number of populations.
 - Improvement of the best solution.
- ✓ Else, return to step 2 **[Fitness]**.



Genetic Algorithm

Conclusion

- GA is nondeterministic – two runs may end with different results
- There's no indication whether best individual is optimal



Pros and Cons

- Pros
 - Faster (and lower memory requirements) than searching a very large search space.
 - Easy, in that if your candidate representation and fitness function are correct, a solution can be found without any explicit analytical work.
- Cons
 - Randomized – not optimal or even complete.
 - Can get stuck on local maxima, though crossover can help mitigate this.
 - It can be hard to work out how best to represent a candidate as a bit string (or otherwise).

Constraint Satisfaction Problem

Constraint Satisfaction Problem (CSP)

- A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or CSP.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

Definition of CSP

- Constraint satisfaction problem consists of three components, X, D, and C:
 - X is a set of variables, {X₁,...,X_n}.
 - D is a set of domains, {D₁,...,D_n}, one for each variable.
 - C is a set of constraints that specify allowable combinations of values.
- A domain, D, consists of a set of allowable values, {v₁,..., v_n}, for variable X;
 - For example, a Boolean variable would have the domain {true, false}.
- Each constraint C_j consists of a pair (scope, rel),
 - where scope is a tuple of variables that participate in the constraint
 - rel is a relation that defines the values that those variables can take on.

- A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation.
 - For example, if X_1 and X_2 , both have the domain $\{1,2,3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as,
 $((X_1, X_2), \{(3,1), (3,2), (2,1)\})$ or as $((X_1, X_2), X_1 > X_2)$.

- CSPs deal with assignments of values to variables, $\{X_i = v_i, X_j = v_j, \dots\}$.
- An assignment that does not violate any constraints is called a **consistent or legal assignment**.
- A **complete assignment** is one in which every variable is assigned a value, and a **solution** to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that leaves some variables unassigned, and a partial solution is a partial assignment that is **consistent**.

Example problem: Map coloring

- A map of Australia showing each of its states and territories.
- We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.



Example problem: Map coloring

- We define the variables to be the regions:

$$X = \{WA, NT, Q, NSW, V, SA, T\}.$$

- The domain of every variable is the set

$$D_i = \{\text{red, green, blue}\}.$$

- The constraints require neighboring regions to have distinct colors.

- Since there are nine places where regions border, there are nine constraints:

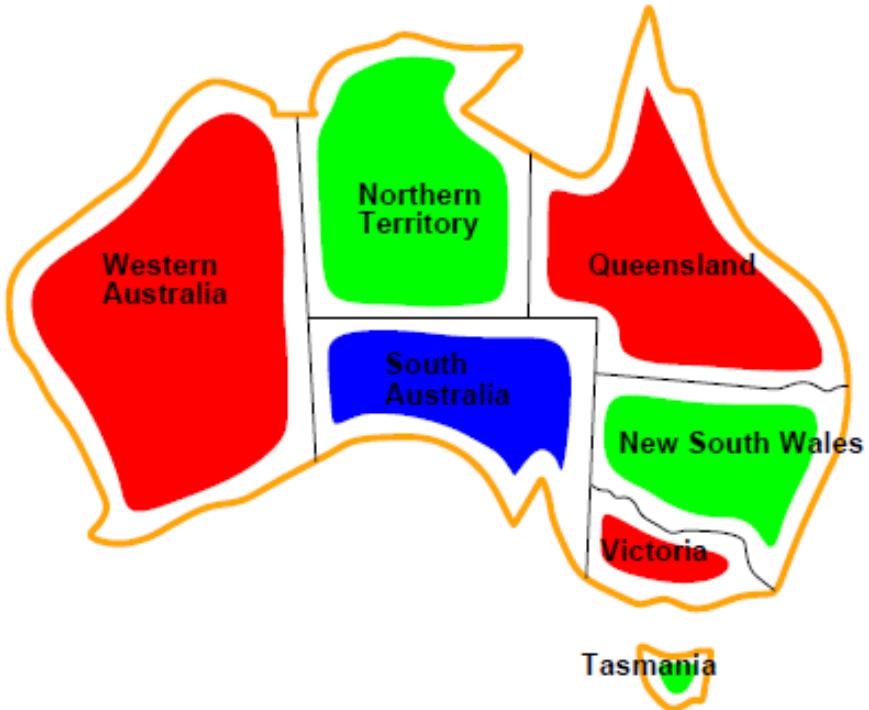
$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Example problem: Map coloring

- Here we are using abbreviations; $SA \neq WA$ is a shortcut for $((SA, WA), SA \neq WA)$,
- Where $SA \neq WA$ can be fully enumerated in turn as
 $\{(red, green), (red, blue), (green, red), (green, blue), (blue, red), (blue, green)\}$.
- Solutions to this problem?

Example problem: Map coloring

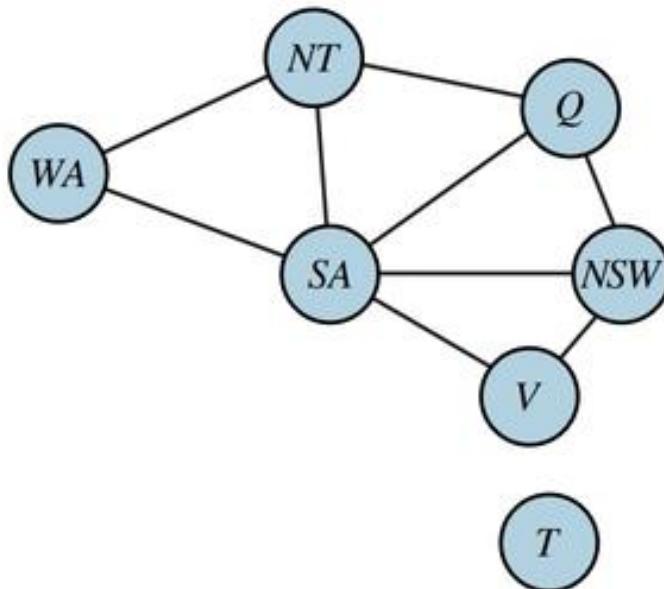
- There are many possible solutions to this problem, such as
 $\{\text{WA} = \text{red}, \text{NT} = \text{green}, \text{Q} = \text{red}, \text{NSW} = \text{green}, \text{V} = \text{red}, \text{SA} = \text{blue}, \text{T} = \text{red}\}$.



Solutions are assignments satisfying all constraints, e.g.,
 $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green}\}$

Example problem: Map coloring

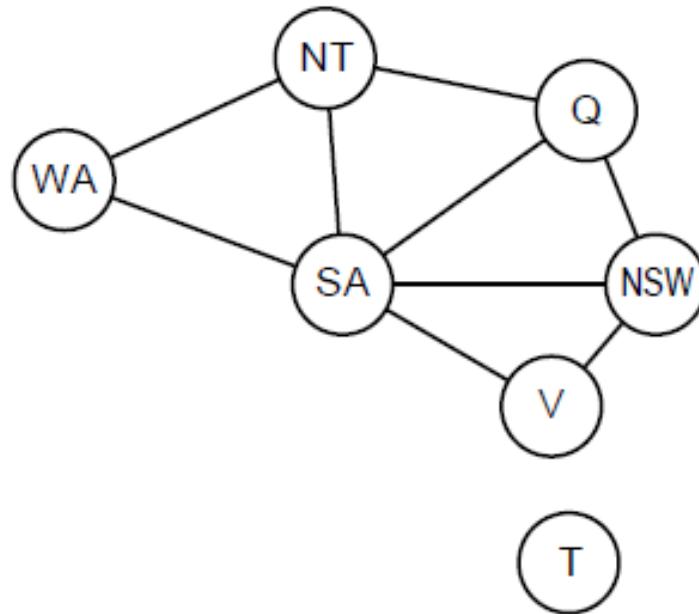
- It can be helpful to visualize a CSP as a constraint graph, as shown in Figure.
- The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.



Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Types of CSP

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ complete assignments

- ◊ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- ◊ e.g., job scheduling, variables are start/end days for each job
- ◊ need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- ◊ linear constraints solvable, nonlinear undecidable

Continuous variables

- ◊ e.g., start/end times for Hubble Telescope observations
- ◊ linear constraints solvable in poly time by LP methods

Constraint Satisfaction Problem

Constraint Satisfaction Problem (CSP)

- A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or CSP.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

Definition of CSP

- Constraint satisfaction problem consists of three components, X, D, and C:
 - X is a set of variables, $\{X_1, \dots, X_n\}$.
 - D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.
 - C is a set of constraints that specify allowable combinations of values.
- A domain, D, consists of a set of allowable values, $\{v_1, \dots, v_n\}$, for variable X;
 - For example, a Boolean variable would have the domain {true, false}.
- Each constraint C_j consists of a pair (scope, rel),
 - where scope is a tuple of variables that participate in the constraint
 - rel is a relation that defines the values that those variables can take on.

- A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation.
 - For example, if X_1 and X_2 both have the domain $\{1,2,3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as,
 $((X_1, X_2), \{(3,1), (3,2), (2,1)\})$ or as $((X_1, X_2), X_1 > X_2)$.

- CSPs deal with assignments of values to variables, $\{X_i = v_i, X_j = v_j, \dots\}$.
- An assignment that does not violate any constraints is called a **consistent or legal assignment**.
- A **complete assignment** is one in which every variable is assigned a value, and a **solution** to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that leaves some variables unassigned, and a partial solution is a partial assignment that is **consistent**.

Example problem: Map coloring

- A map of Australia showing each of its states and territories.
- We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.



Example problem: Map coloring

- We define the variables to be the regions:

$$X = \{WA, NT, Q, NSW, V, SA, T\}.$$

- The domain of every variable is the set

$$D_i = \{\text{red, green, blue}\}.$$

- The constraints require neighboring regions to have distinct colors.
- Since there are nine places where regions border, there are nine constraints:

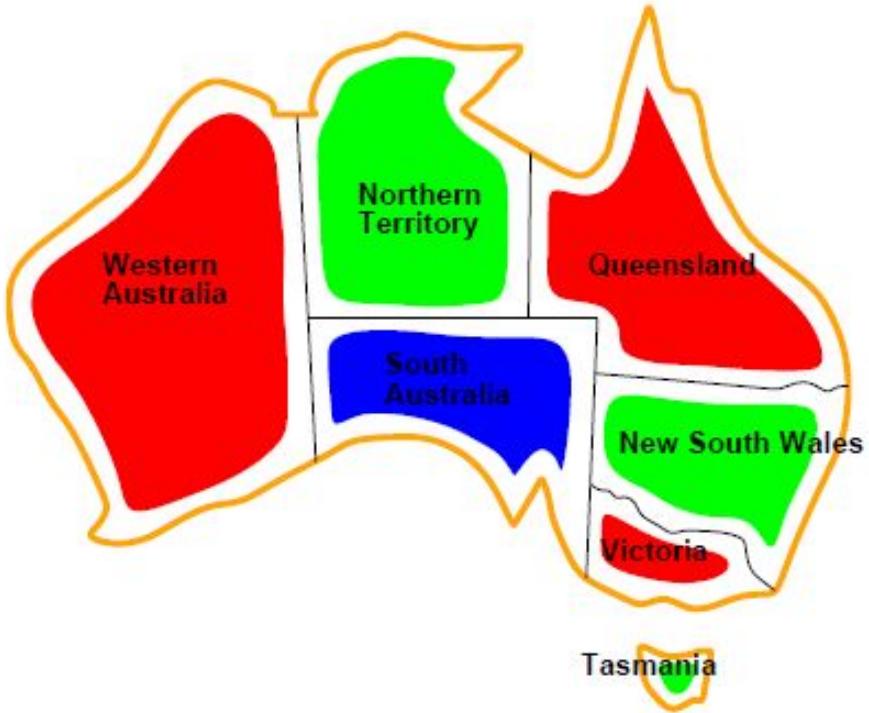
$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Example problem: Map coloring

- Here we are using abbreviations; $SA \neq WA$ is a shortcut for $((SA, WA), SA \neq WA)$,
- Where $SA \neq WA$ can be fully enumerated in turn as
 $\{(red, green), (red, blue), (green, red), (green, blue),$
 $(blue, red), (blue, green)\}$.
- Solutions to this problem?

Example problem: Map coloring

- There are many possible solutions to this problem, such as
 $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{red}\}.$

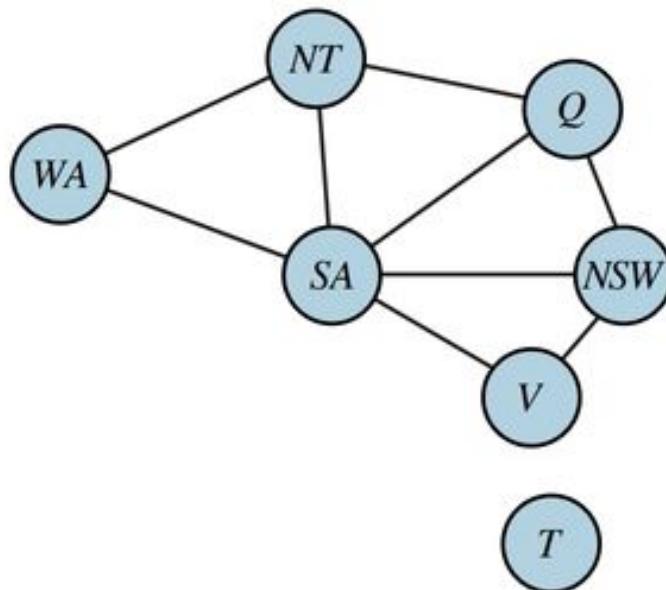


Solutions are assignments satisfying all constraints, e.g.,

$$\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green}\}$$

Example problem: Map coloring

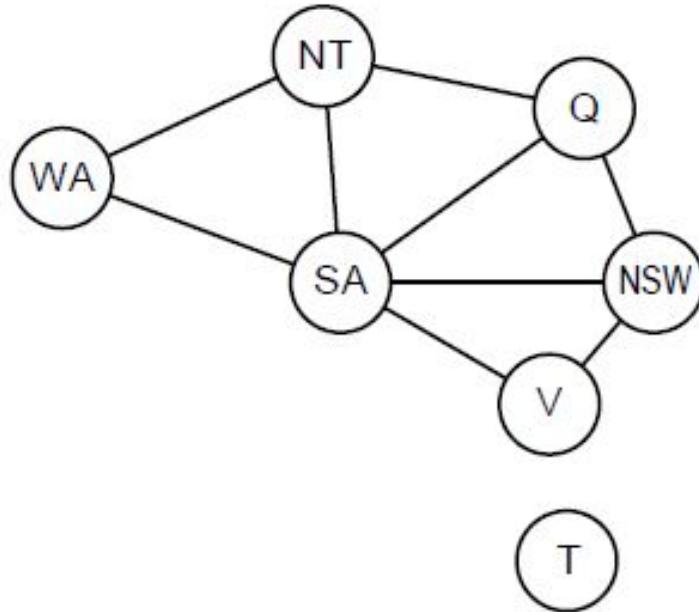
- It can be helpful to visualize a CSP as a constraint graph, as shown in Figure.
- The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.



Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Types of CSP

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ complete assignments

- ◊ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- ◊ e.g., job scheduling, variables are start/end days for each job
- ◊ need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- ◊ linear constraints solvable, nonlinear undecidable

Continuous variables

- ◊ e.g., start/end times for Hubble Telescope observations
- ◊ linear constraints solvable in poly time by LP methods

Varieties of constraints

Unary constraints involve a single variable,

e.g., $SA \neq green$

Binary constraints involve pairs of variables,

e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,

e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., red is better than $green$

often representable by a cost for each variable assignment

→ constrained optimization problems

Cryptarithmetic Problem

- Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols.
- In **cryptarithmetic problem**, the digits (0-9) get substituted by some possible alphabets or symbols.
- The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

Rules and constraints

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., $2+2=4$, nothing else.
- Digits should be from **0-9** only.
- The problem can be solved from both sides, i.e., **lefthand side (L.H.S)**, or **righthand side (R.H.S)**

Example-1

BASE

+BALL

GAMES

Example-1

$$\begin{array}{r} \text{B A S E} \\ + \text{B A L L} \\ \hline \text{G A M E S} \end{array}$$

Variables: {B,A,S,E,L,G,M}
Domain:{0,1,2,3,4,5,6,7,8,9}
Constraint: AllDiffz(B,A,S,E,L,G,M)

Example-1

BASE
+ **BALL** -

GAMES

B	5
B	5
G A	10
G	1
A	0

Example-1

B A S E
+ B A L L -
G A M E S

B	6
B	6
G A	12
G	1
A	2

Example-1

BASE

+BALL

GAMES

Cr	0	Cr	0
B	6	A	2
B	6	A	2
GA	12	M	4
G	1		
A	2		

Example-1

BASE
+ **BALL** -
GAMES

Cr	0	Cr	0	Cr	
B	6	A	2	S	3
B	6	A	2	L	5
GA	12	M	4	E	8
G	1				
A	2				

Example-1

BASE
+ **BALL** -
GAMES

Cr	0	Cr	0	C r		Cr	
B	6	A	2	S	3	E	8
B	6	A	2	L	5	L	5
GA	12	M	4	E	8	S	
G	1						
A	2						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	0	Cr		Cr	
B	7	A		S		E	
B	7	A		L		L	
GA	14	M		E		S	
G	1						
A	4						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	0	Cr		Cr	
B	7	A	4	S		E	
B	7	A	4	L		L	
GA	14	M	8	E		S	
G	1						
A	4						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	1	Cr		Cr	
B	7	A	4	S	8	E	
B	7	A	4	L	5	L	
GA	14	M	9	E	3	S	
G	1						
A	4						

Example-1 Solution

BASE

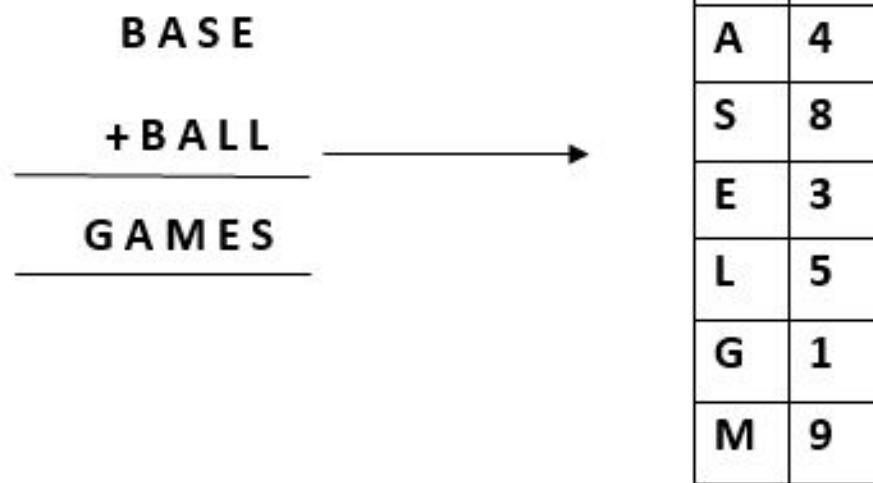
+BALL

GAMES

Cr	0	Cr	1	Cr	0	Cr	0
B	7	A	4	S	8	E	3
B	7	A	4	L	5	L	5
GA	14	M	9	E	3	S	8
G	1						
A	4						

BASE
+ **BALL**

GAMES



B	7
A	4
S	8
E	3
L	5
G	1
M	9

S E N D

+ M O R E

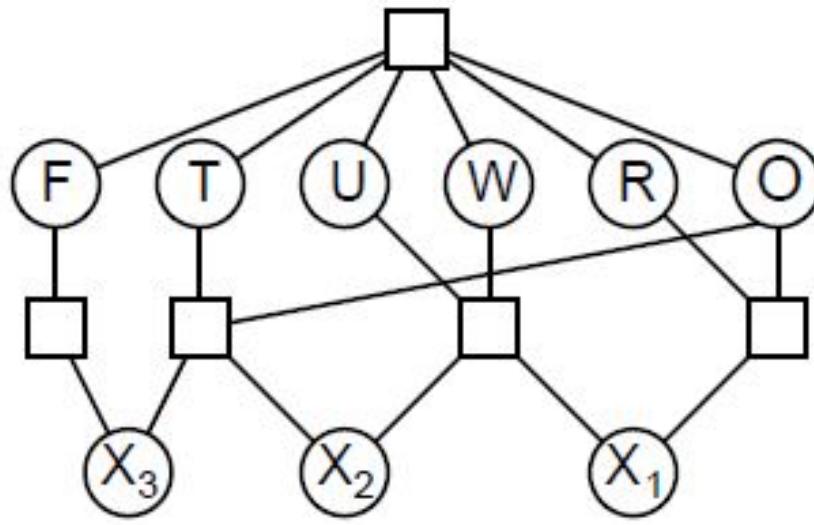
M O N E Y

S E N D
+ M O R E
—
M O N E Y
—

Cr		Cr		Cr		Cr	
S		E		N		D	
M		O		R		E	
MO		N		E		Y	
M							
O							

Example: Cryptarithmetic

$$\begin{array}{r} \text{T} \ \text{W} \ \text{O} \\ + \ \text{T} \ \text{W} \ \text{O} \\ \hline \text{F} \ \text{O} \ \text{U} \ \text{R} \end{array}$$



Variables: $F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.

$$\begin{array}{r}
 \text{T} \quad \text{W} \quad \text{O} \\
 + \quad \text{T} \quad \text{W} \quad \text{O} \\
 \hline
 \text{F} \quad \text{O} \quad \text{U} \quad \text{R}
 \end{array}$$

<i>Cr</i>	0	<i>Cr</i>	0	<i>Cr</i>	0
T	7	W	3	O	4
T	7	W	3	O	4
FO	14	U	6	R	8
F	1				
O	4				

Real-world CSPs

Assignment problems

e.g., who teaches what class

Timetabling problems

e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables

Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

- ◊ **Initial state:** the empty assignment, {}
- ◊ **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment.
⇒ fail if no legal assignments (not fixable!)
- ◊ **Goal test:** the current assignment is complete

- 1) This is the same for all CSPs! 😊
- 2) Every solution appears at depth n with n variables
⇒ use depth-first search
- 3) Path is irrelevant, so can also use complete-state formulation
- 4) $b = (n - \ell)d$ at depth ℓ , hence $n!d^n$ leaves!!!! 😞

Genetic Algorithm

Example

Maximizing a Function

- Consider the problem of maximizing the function, $f(x) = x^2$
- where x is permitted to vary between 0 to 31.

- Step 1: For using genetic algorithms approach, one must first code the decision variable ‘x’ into a finite length string.
- Using a five bit (binary integer) unsigned integer, numbers between 0(00000) and 31(11111) can be obtained.
- The objective function here is $f(x) = x^2$, which is to be maximized.
- A single generation of a genetic algorithm is performed here with encoding, selection, crossover and mutation.

Select initial population. Psize=4

String No.	Initial popu- lation (randomly selected)
1	01100
2	11001
3	00101
4	10011

- Step 2: Obtain the decoded x values for the initial population generated. Consider string 1, Thus for all the four strings the decoded values are obtained.
- Step 3: Calculate the fitness or objective function. This is obtained by simply squaring the ‘x’ value, since the given function is $f(x) = x^2$.
- When, $x = 12$, the fitness value is, $f(x) = 144$ for $x = 25$, $f(x) = 625$ and so on, until the entire population is computed

String No.	Initial popu- lation (randomly selected)	x value	Fitness value $f(x) = x^2$
1	01100	12	144
2	11001	25	625
3	00101	5	25
4	10011	19	361
Sum			1155
average			288.75
maximum			625

- Step 4: Compute the probability of selection,

$$\text{Prob}(x_i) = \frac{f(x_i)}{\sum_{i=1}^n f(x_i)}$$

where

n = no of populations

f(x)=fitness value corresponding to a particular individual in the population

$\Sigma f(x)$ - Summation of all the fitness value of the entire population.

For e.g: Considering string 1, Fitness $f(x) = 144$ $\Sigma f(x) = 1155$

The probability that string 1 occurs is given by, $P_1 = 144/1155 = 0.1247$

String No.	Initial popu- lation (randomly selected)	x value	Fitness value $f(x) = x^2$	Prob _i	Percentage probability
1	01100	12	144	0.1247	12.47%
2	11001	25	625	0.5411	54.11%
3	00101	5	25	0.0216	2.16%
4	10011	19	361	0.3126	31.26%
Sum			1155	1.0000	100%
average			288.75	0.2500	25%
maximum			625	0.5411	54.11%

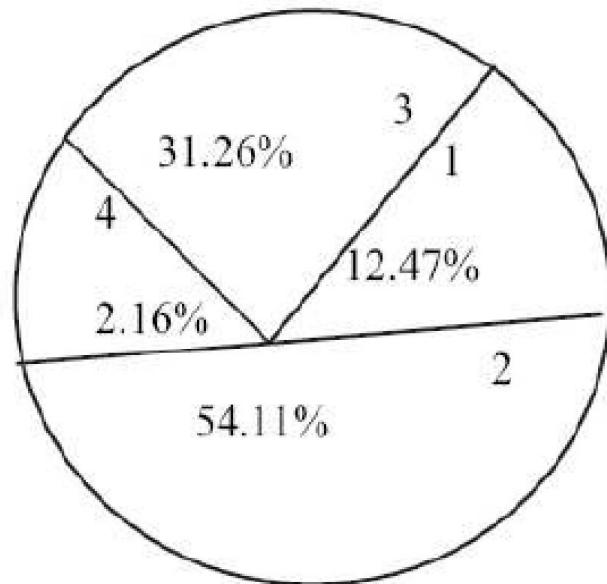
- Step 5: The next step is to calculate the expected count, which is calculated as,

$$\text{Expected count} = \frac{f(x_i)}{(\sum_{i=1}^n f(x_i))/n}$$

The expected count gives an idea of which population can be selected for further processing in the mating pool.

String No.	Initial popu- lation (randomly selected)	x value	Fitness value $f(x) = x^2$	Prob _i	Percentage probability	Expected count
1	01100	12	144	0.1247	12.47%	0.4987
2	11001	25	625	0.5411	54.11%	2.1645
3	00101	5	25	0.0216	2.16%	0.0866
4	10011	19	361	0.3126	31.26%	1.2502
Sum			1155	1.0000	100%	4.0000
average			288.75	0.2500	25%	1.0000
maximum			625	0.5411	54.11%	2.1645

- Step 6: Now the actual count is to be obtained to select the individuals, which would participate in the crossover cycle using Roulette wheel selection:



actual count

- String 1 occupies 12.47%, so there is a chance for it to occur at least once. Hence its actual count may be 1.
- With string 2 occupying 54.11% of the Roulette wheel, it has a fair chance of being selected twice. Thus its actual count can be considered as 2.
- On the other hand, string 3 has the least probability percentage of 2.16%, so their occurrence for next cycle is very poor. As a result, its actual count is 0.
- String 4 with 31.26% has at least one chance for occurring while Roulette wheel is spun, thus its actual count is 1.

String No.	Initial population (randomly selected)	x value	Fitness value $f(x) = x^2$	Prob _i	Percentage probability	Expected count	Actual count
1	01100	12	144	0.1247	12.47%	0.4987	1
2	11001	25	625	0.5411	54.11%	2.1645	2
3	00101	5	25	0.0216	2.16%	0.0866	0
4	10011	19	361	0.3126	31.26%	1.2502	1
Sum			1155	1.0000	100%	4.0000	4
average			288.75	0.2500	25%	1.0000	1
maximum			625	0.5411	54.11%	2.1645	2

- **Step 7:** Now, writing the mating pool based upon the actual count as shown in Table.
 - The actual count of string no 1 is 1, hence it occurs once in the mating pool.
 - The actual count of string no 2 is 2, hence it occurs twice in the mating pool.
 - Since the actual count of string no 3 is 0, it does not occur in the mating pool.
 - Similarly, the actual count of string no 4 being 1, it occurs once in the mating pool.
 - Based on this, the mating pool is formed.
-

String No.	Mating pool
1	0 1 1 0 0
2	1 1 0 0 1
2	1 1 0 0 1
4	1 0 0 1 1

- **Step 8:** Crossover operation is performed to produce new offspring (children).
- The crossover point is specified and based on the crossover point, single point crossover is performed and new offspring is produced.
 - Parent 1 0 1 1 0 0
 - Parent 2 1 1 0 0 1
 - The offspring is produced as,
 - Offspring 1 0 1 1 0 1
 - Offspring 2 1 1 0 0 0

Note: The crossover probability was assumed to 1.0

String No.	Mating pool	Crossover point	Offspring after crossover	x value	Fitness value $f(x) = x^2$
1	0 1 1 0 0	4	0 1 1 0 1	13	169
2	1 1 0 0 1	4	1 1 0 0 0	24	576
2	1 1 0 0 1	2	1 1 0 1 1	27	729
4	1 0 0 1 1	2	1 0 0 0 1	17	289
Sum					1763
average					440.75
maximum					729

- **Step 9:** After crossover operations, new off springs are produced and ‘x’ values are decodes and fitness is calculated.
- **Step 10:** In this step, mutation operation is performed to produce new off springs after crossover operation.
 - In mutation-flipping operation is performed and new off springs are produced.

String No.	Offspring after crossover	Mutation chromosomes for flipping	Offspring after Mutation	X value	Fitness value $F(x) = x^2$
1	0 1 1 0 1	1 0 0 0 0	1 1 1 0 1	29	841
2	1 1 0 0 0	0 0 0 0 0	1 1 0 0 0	24	576
2	1 1 0 1 1	0 0 0 0 0	1 1 0 1 1	27	729
4	1 0 0 0 1	0 0 1 0 0	1 0 1 0 0	20	400
Sum					2546
average					636.5
maximum					841

Note: The mutation probability was assumed to 0.001

- Once the off springs are obtained after mutation, they are decoded to x value and find fitness values are computed.
-
- This completes one generation.
- The mutation is performed on a bit-bit by basis .

Backtracking

CSP

- Sometimes we can finish the constraint propagation process and still have variables with multiple possible values.
- In that case we have to search for a **solution**.
- For a CSP with n variables of domain size d we would end up with a search tree where all the complete assignments (and thus all the solutions) are leaf nodes at depth n . But notice that the branching factor at the top level would be nd because any of d values can be assigned to any of n variables.
- At the next level, the branching factor is $(n - 1)d$, and so on for n levels.
- So the tree has $n!.d^n$ leaves, even though there are only d^n possible complete assignments!

Backtracking search

Variable assignments are commutative, i.e.,

$[WA = \text{red} \text{ then } NT = \text{green}]$ same as $[NT = \text{green} \text{ then } WA = \text{red}]$

Only need to consider assignments to a single variable at each node

$\Rightarrow b = d$ and there are d^n leaves

Depth-first search for CSPs with single-variable assignments
is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

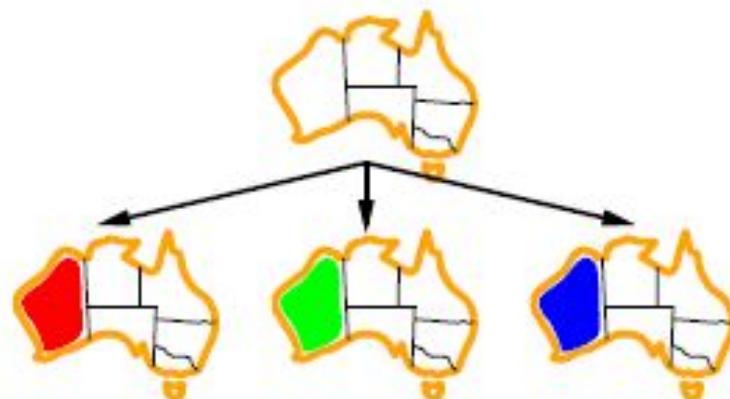
Backtracking search

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
    return RECURSIVE-BACKTRACKING({ }, csp)
function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
            add  $\{var = value\}$  to assignment
            result  $\leftarrow$  RECURSIVE-BACKTRACKING(assignment, csp)
            if result  $\neq$  failure then return result
            remove  $\{var = value\}$  from assignment
    return failure
```

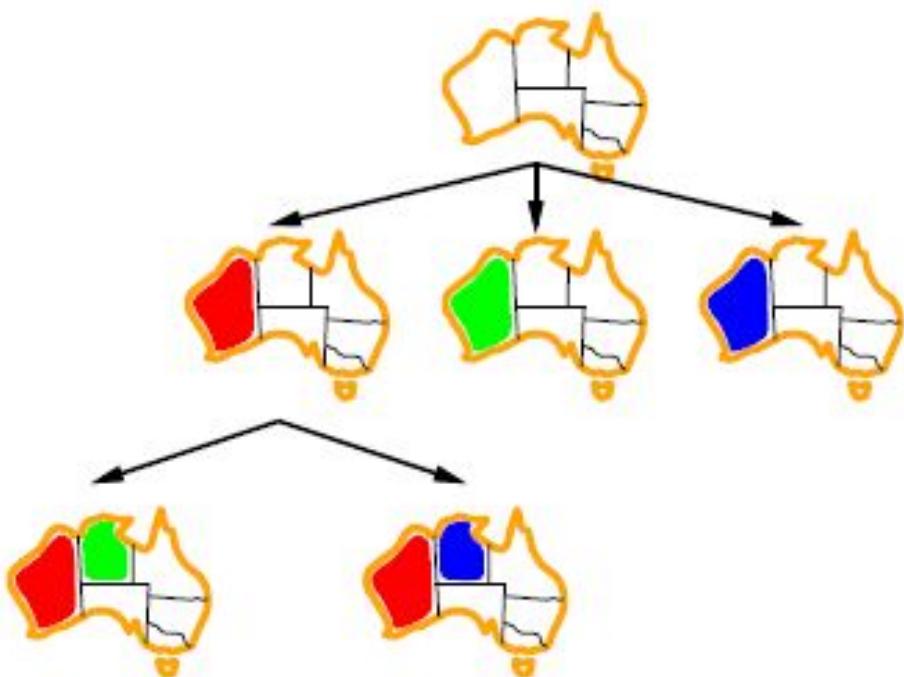
Backtracking example



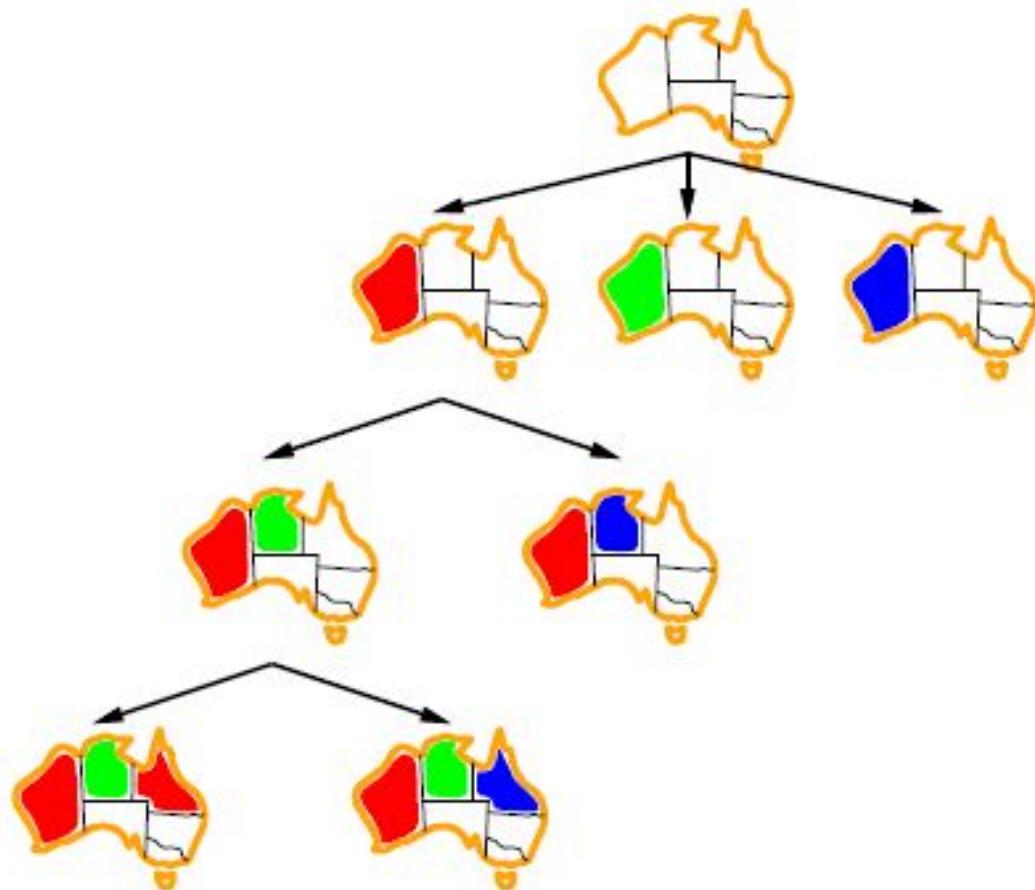
Backtracking example



Backtracking example



Backtracking example



Improving backtracking efficiency

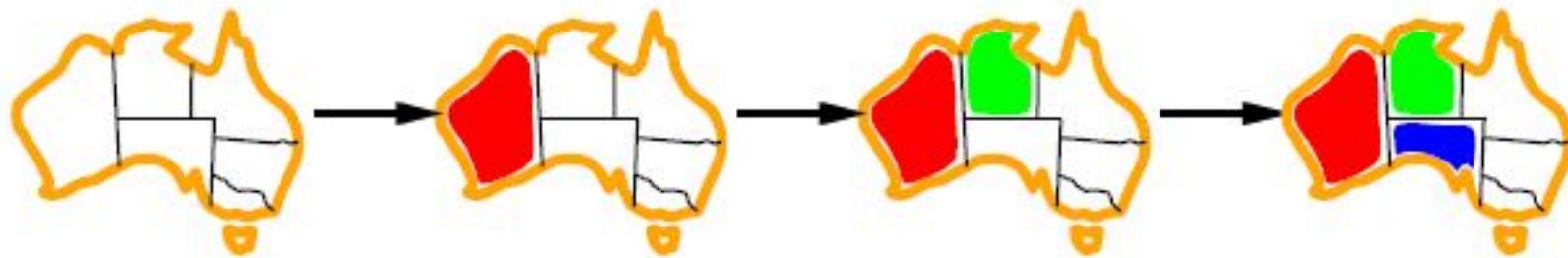
General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?

Minimum remaining values

Minimum remaining values (MRV):

choose the variable with the fewest legal values

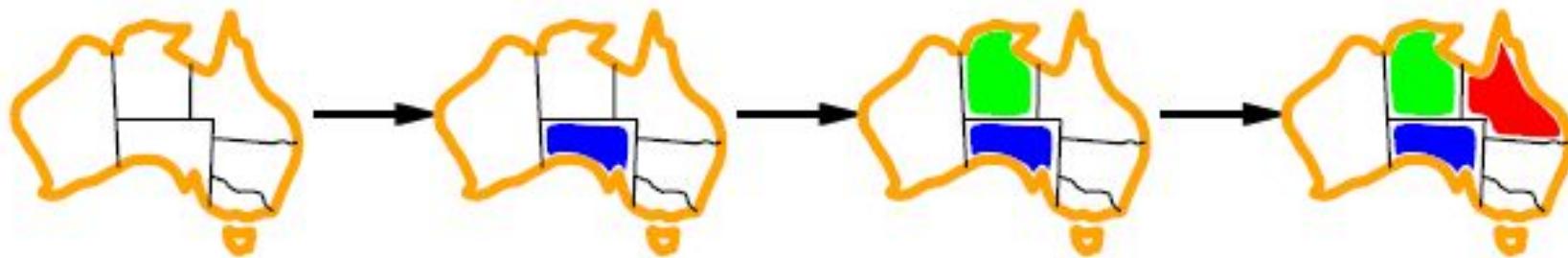


Degree heuristic

Tie-breaker among MRV variables

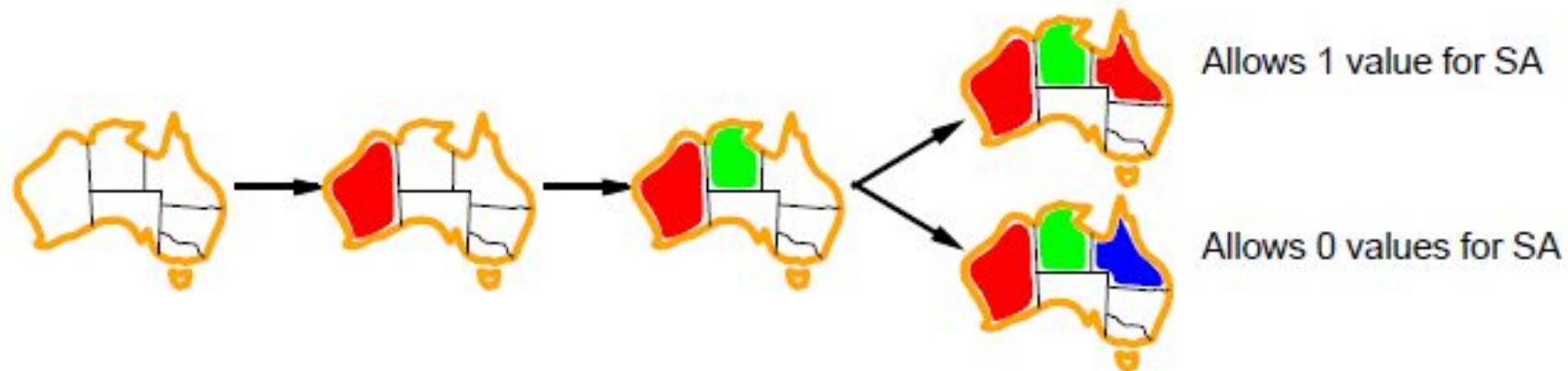
Degree heuristic:

choose the variable with the most constraints on remaining variables



Least constraining value

Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables



Combining these heuristics makes 1000 queens feasible

Forward checking

Idea: Keep track of remaining legal values for unassigned variables

Terminate search when any variable has no legal values



WA

NT

Q

NSW

V

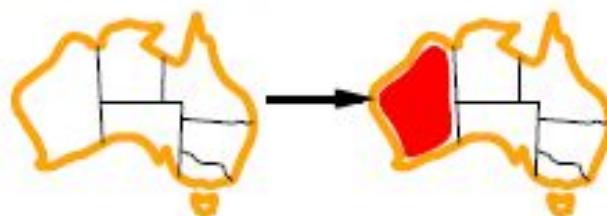
SA

T



Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



WA	NT	Q	NSW	V	SA	T
Red	Green	Blue	Red	Green	Blue	Red
Red		Green	Blue	Red	Green	Blue

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values



WA

NT

Q

NSW

V

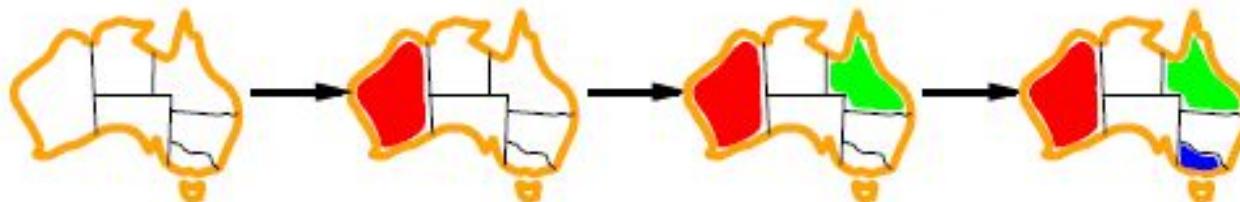
SA

T

[Red]	[Green]	[Blue]	[Red]	[Green]	[Blue]	[Red]	[Green]	[Blue]
[Red]		[Green]	[Blue]	[Red]	[Green]	[Blue]	[Red]	[Green]
[Red]		[Blue]	[Green]	[Red]	[Blue]		[Blue]	[Red]

Forward checking

Idea: Keep track of remaining legal values for unassigned variables
Terminate search when any variable has no legal values

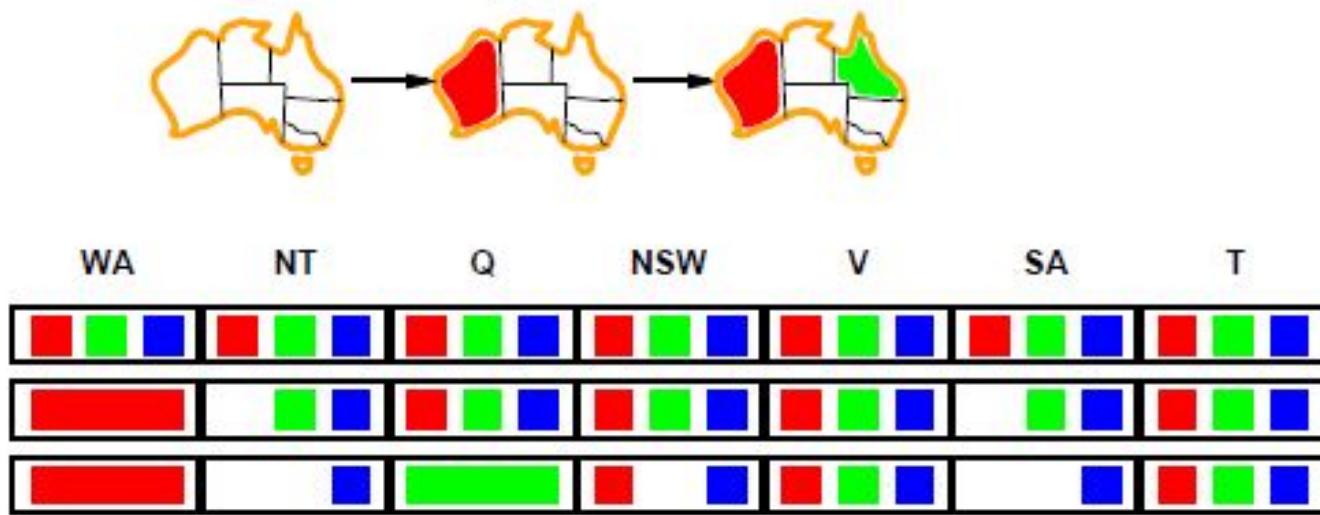


WA NT Q NSW V SA T

█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue	█ Red	█ Green	█ Blue
█ Red		█ Green	█ Blue	█ Red	█ Green	█ Blue	█ Green	█ Blue
█ Red		█ Blue	█ Green	█ Red	█ Blue	█ Red	█ Green	█ Blue
█ Red		█ Blue	█ Green	█ Red		█ Blue	█ Red	█ Blue

Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:



NT and *SA* cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Arc-consistency

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

Example

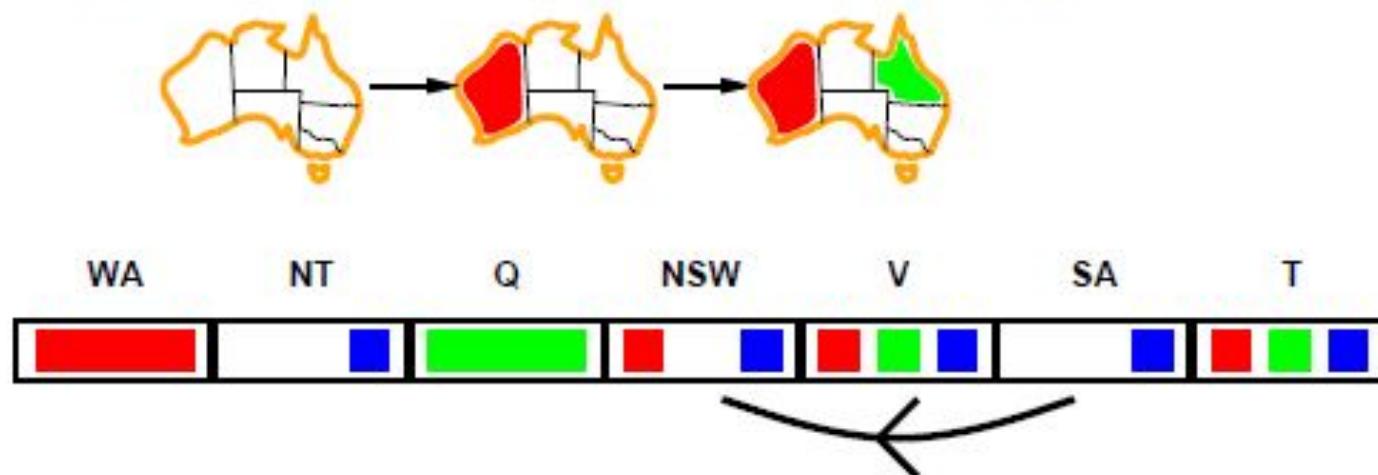
- Consider the constraint $Y = X^2$
 - where the domain of both X and Y is the set of decimal digits.
- We can write this constraint explicitly as
 $((X,Y), \{(0,0), (1,1), (2,4), (3,9)\})$
- To make X arc-consistent with respect to Y ,
 - we reduce X 's domain to $\{0, 1, 2, 3\}$.
- If we also make Y arc-consistent with respect to X ,
 - then Y 's domain becomes $\{0, 1, 4, 9\}$, and the whole CSP is arc-consistent.

Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for every value x of X there is some allowed y

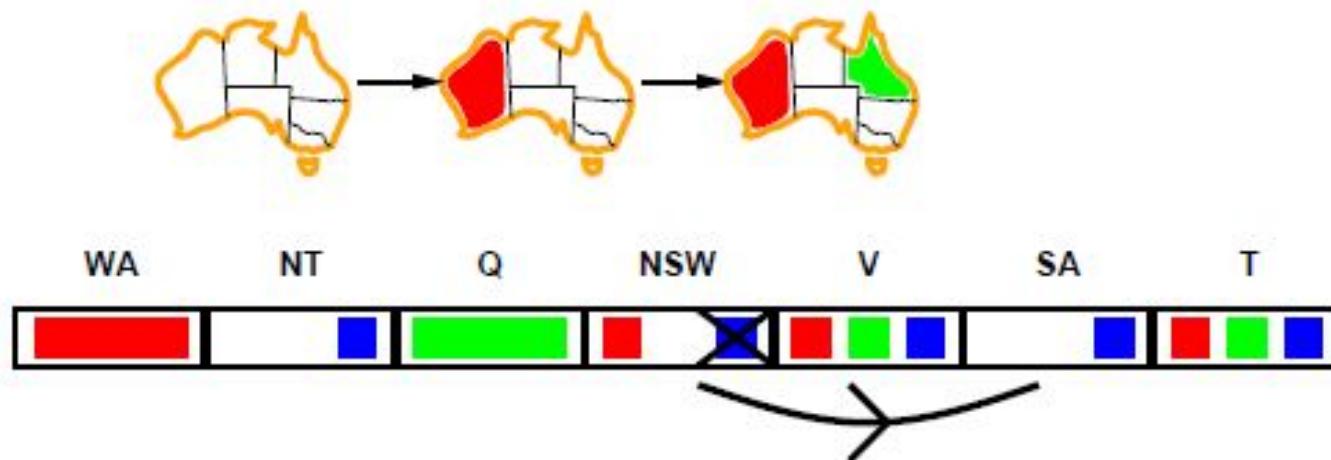


Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for every value x of X there is some allowed y

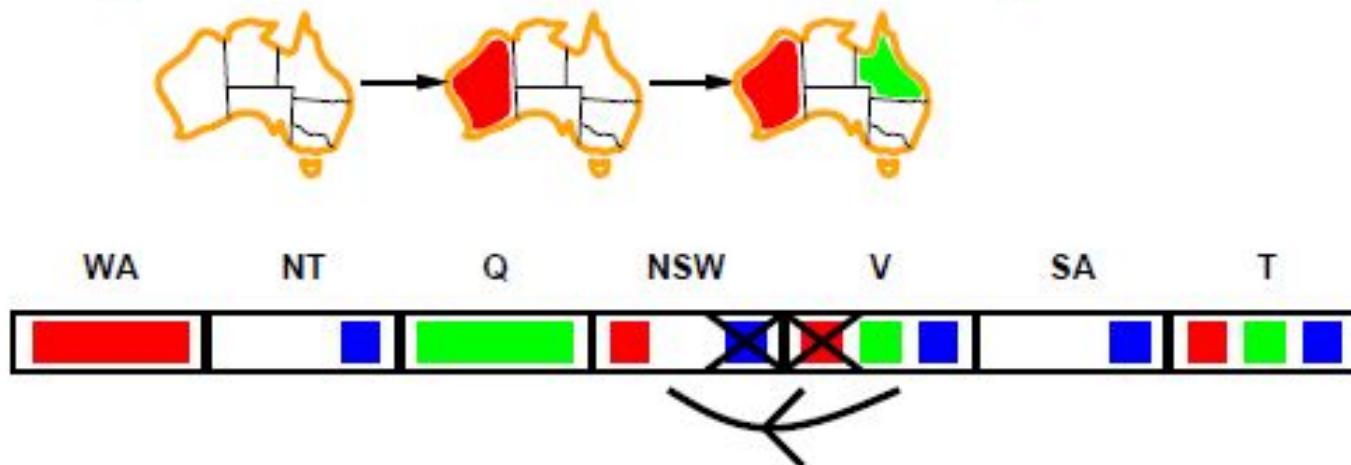


Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for every value x of X there is some allowed y



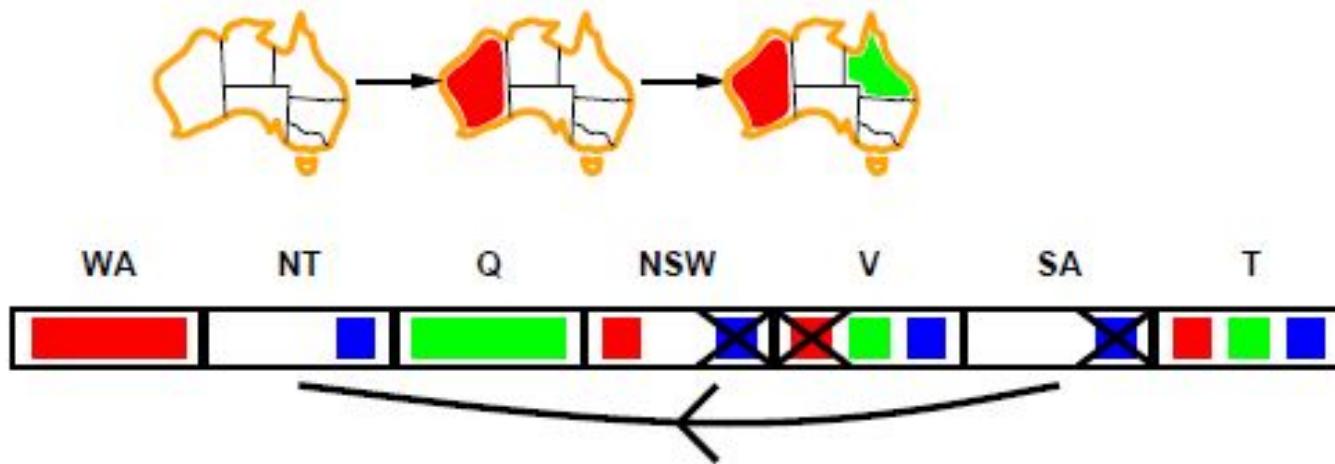
If X loses a value, neighbors of X need to be rechecked

Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for every value x of X there is some allowed y



If X loses a value, neighbors of X need to be rechecked

Arc consistency detects failure earlier than forward checking

Can be run as a preprocessor or after each assignment

Arc consistency algorithm

```
function AC-3( csp) returns the CSP, possibly with reduced domains
  inputs: csp, a binary CSP with variables  $\{X_1, X_2, \dots, X_n\}$ 
  local variables: queue, a queue of arcs, initially all the arcs in csp
  while queue is not empty do
     $(X_i, X_j) \leftarrow \text{REMOVE-FIRST}(\textit{queue})$ 
    if REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) then
      for each  $X_k$  in NEIGHBORS[ $X_i$ ] do
        add  $(X_k, X_i)$  to queue



---


function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff succeeds
  removed  $\leftarrow \text{false}$ 
  for each  $x$  in DOMAIN[ $X_i$ ] do
    if no value  $y$  in DOMAIN[ $X_j$ ] allows  $(x,y)$  to satisfy the constraint  $X_i \leftrightarrow X_j$ 
      then delete  $x$  from DOMAIN[ $X_i$ ]; removed  $\leftarrow \text{true}$ 
  return removed
```

$O(n^2d^3)$, can be reduced to $O(n^2d^2)$ (but detecting all is NP-hard)

AC-3 algorithm

We can represent the AC-3 algorithm in 3 steps:

1. Get all the constraints and turn each one into two arcs. For example:
 $A > B$ becomes $A > B$ and $B < A$.
2. Add all the arcs to a queue.
3. Repeat until the queue is empty:
 - 3.1. Take the first arc (x, y) , off the queue (**dequeue**).
 - 3.2. For **every** value in the x domain, there must be some value of the y domain.
 - 3.3. Make x arc consistent with y . To do so, remove values from x domain for which there is no possible corresponding value for y domain.
 - 3.4. If the x domain has changed, add all arcs of the form (k, x) to the queue (**enqueue**). Here k is another variable different from y that has a relation to x .

Example

- Let's take this example, where we have three variables A , B , and C and the constraints: $A > B$ and $B = C$.
- $A=\{1,2,3\}$
- $B=\{1,2,3\}$
- $C=\{1,2,3\}$

- **Step 1: Generate All Arcs**

A>B

B<A

B=C

C=B

- Step 2: Create the Queue

Queue:

A>B
B<A
B=C
C=B

Arcs

A>B
B<A
B=C
C=B

- Step 3: Iterate Over the Queue

-

Queue:

A>B
B<A
B=C
C=B

A={1,2,3}
B={1,2,3}
C={1,2,3}

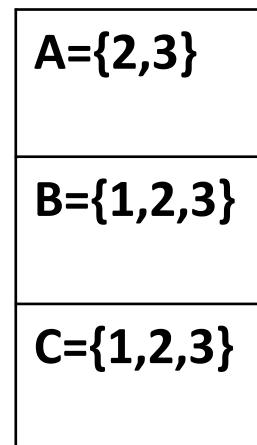
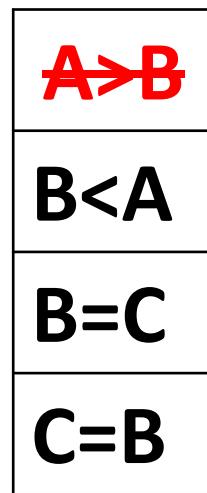
A>B
B<A
B=C
C=B

Arcs

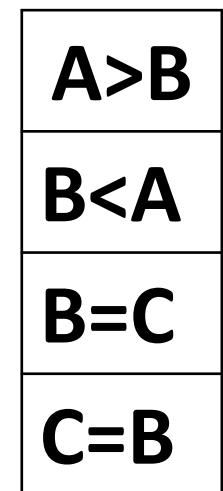
- Step 3: Iterate Over the Queue

-

Queue:



Arcs



- Step 3: Iterate Over the Queue

-

Queue:

A>B
B<A
B=C
C=B

A={2,3}
B={1,2,3}
C={1,2,3}

A>B
B<A
B=C
C=B

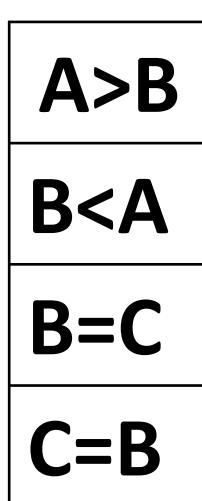
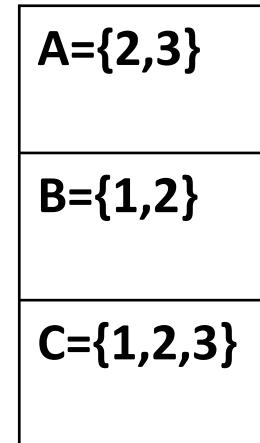
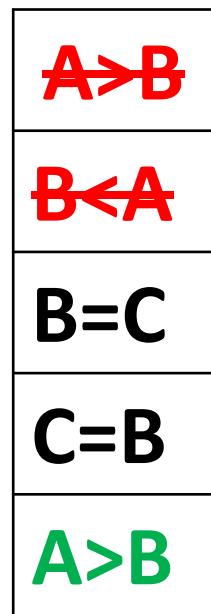
Arcs

- Step 3: Iterate Over the Queue

-

Arcs

Queue:

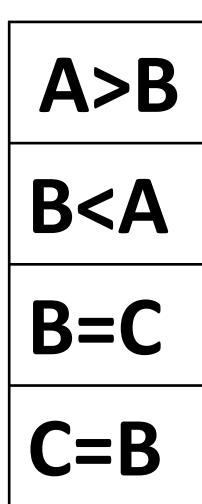
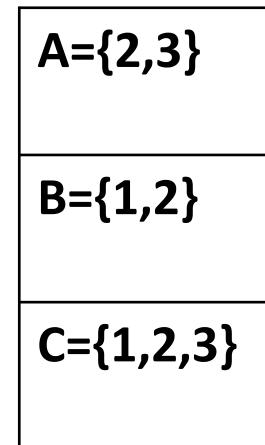
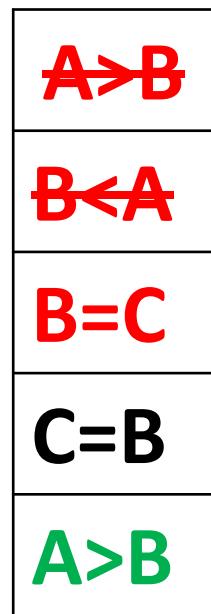


- Step 3: Iterate Over the Queue

-

Arcs

Queue:

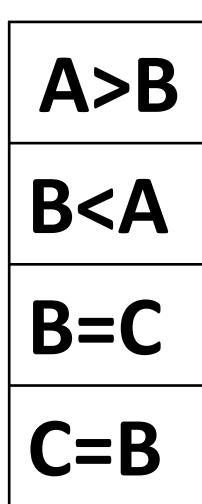
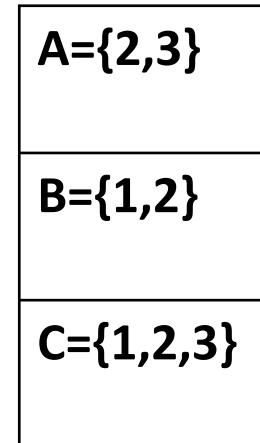
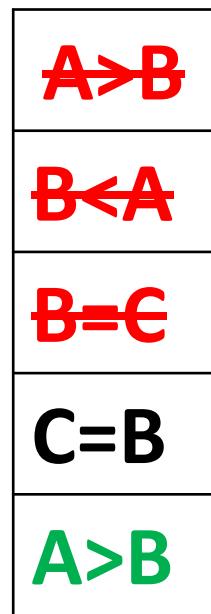


- Step 3: Iterate Over the Queue

-

Arcs

Queue:

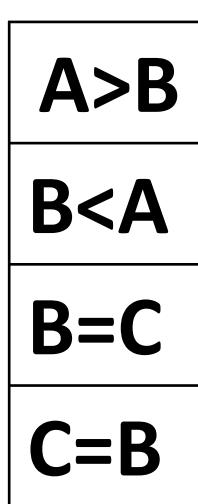
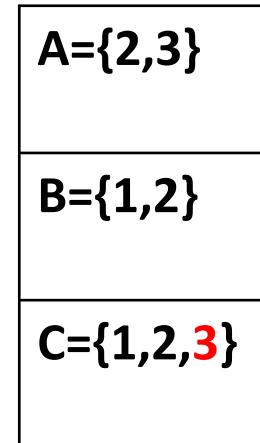
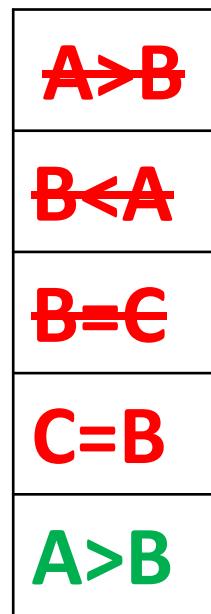


- Step 3: Iterate Over the Queue

-

Arcs

Queue:

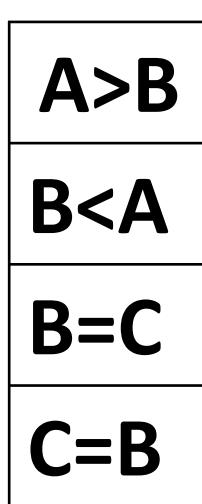
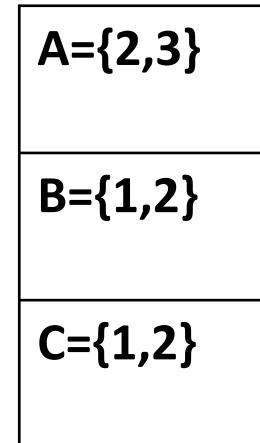
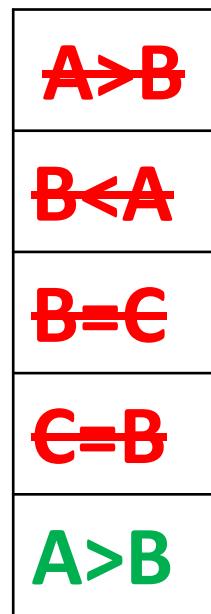


- Step 3: Iterate Over the Queue

-

Arcs

Queue:



- Step 3: Iterate Over the Queue

-

Queue:

A>B
B<A
B=C
C=B
A>B
B=C

A={2,3}

B={1,2}

C={1,2}

Arcs

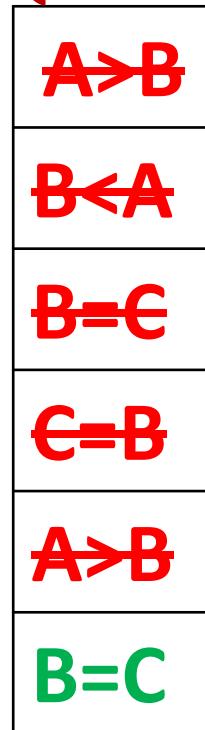
A>B
B<A
B=C
C=B

- Step 3: Iterate Over the Queue

-

Arcs

Queue:



A={2,3}

B={1,2}

C={1,2}

A>B

B<A

B=C

C=B

- Step 3: Iterate Over the Queue

-

Arcs

Queue:

A>B
B<A
B=C
C=B
A>B
B=C

A={2,3}

B={1,2}

C={1,2}

A>B
B<A
B=C
C=B

- Step 4: Stop if no more arc in the queue
-

Queue:

A>B
B<A
B=C
C=B
A>B
B=C

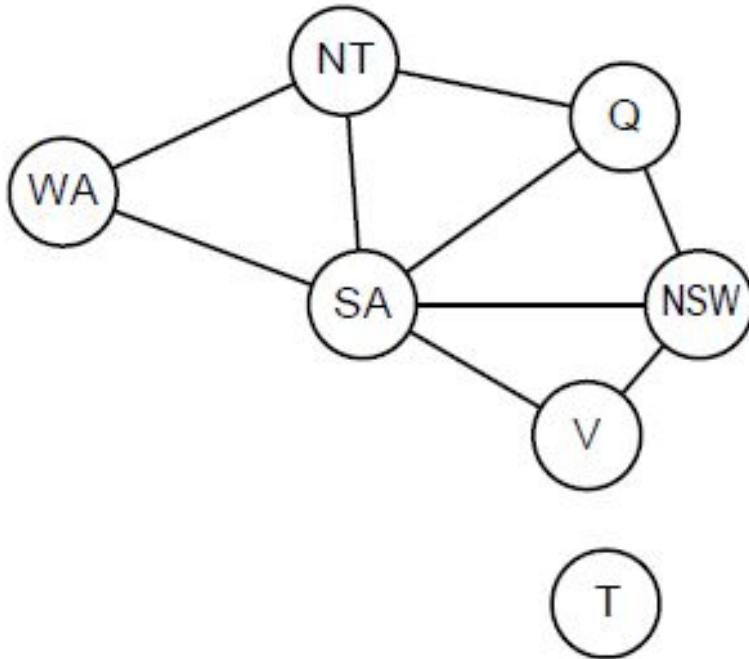
Arcs
A>B
B<A
B=C
C=B

A={2,3}

B={1,2}

C={1,2}

Problem structure



Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph

Problem structure contd.

Suppose each subproblem has c variables out of n total

Worst-case solution cost is $n/c \cdot d^c$, linear in n

E.g., $n = 80$, $d = 2$, $c = 20$

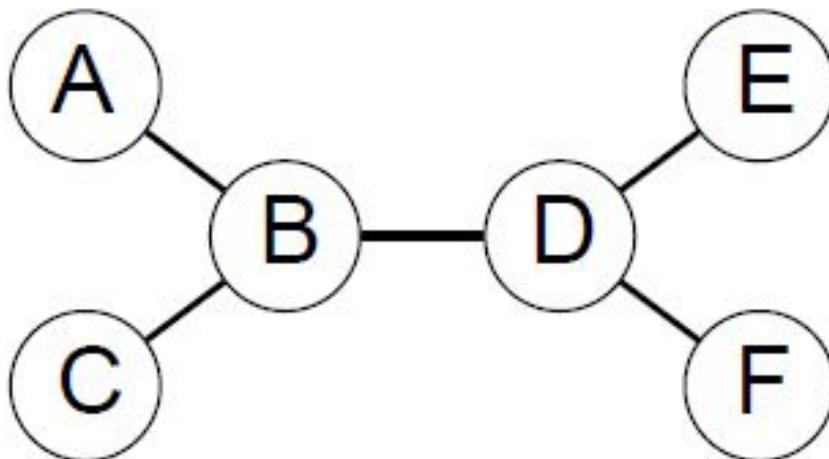
$2^{80} = 4$ billion years at 10 million nodes/sec

$4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec

- A Constraint graph is a tree when any two variables are connected by only one path.
- We will show that any tree-structure CSP can be solved in time linear in the number of variables.
- The key is a new notion of consistency, called directional arc consistency or DAC.
- A CSP is defined to be directional arc-consistent under an ordering of variables X_1, X_2, \dots, X_n , if and only if every X_i is arc consistent with each X_j for $j > i$.

- To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree.
- Such an ordering is called a **topological sort**.

Tree-structured CSPs



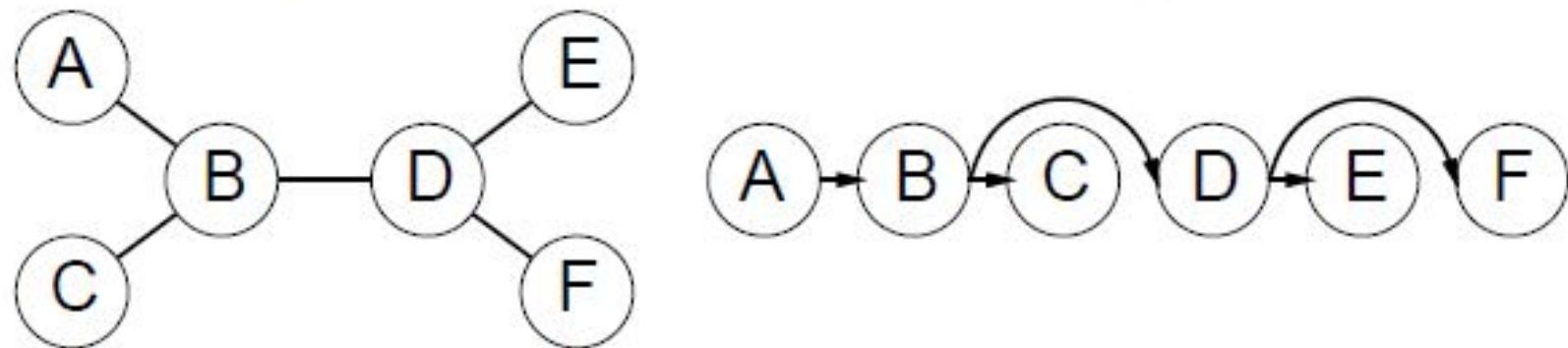
Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time

Compare to general CSPs, where worst-case time is $O(d^n)$

This property also applies to logical and probabilistic reasoning:
an important example of the relation between syntactic restrictions
and the complexity of reasoning.

Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node's parent precedes it in the ordering

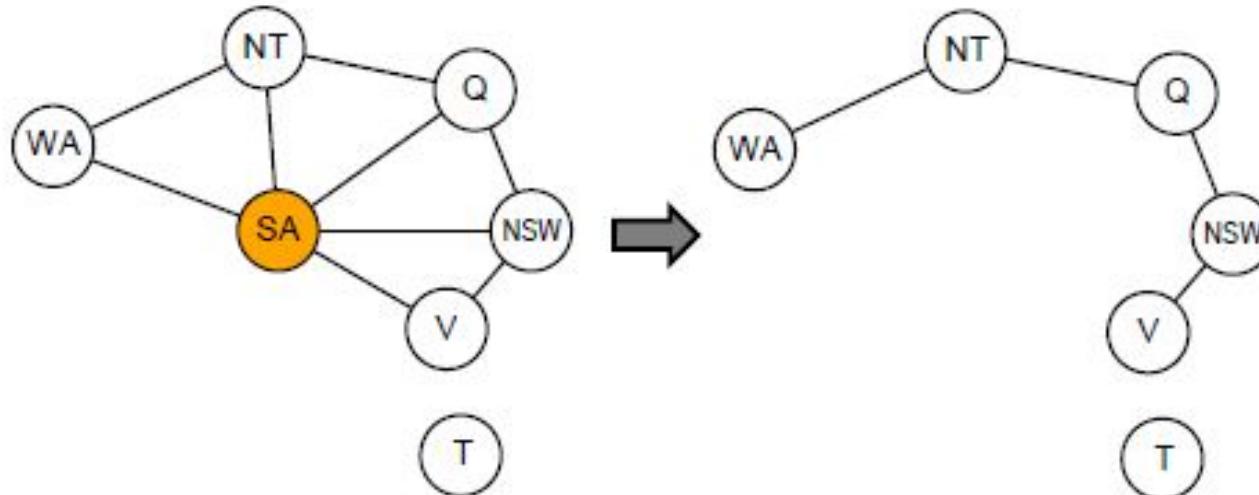


2. For j from n down to 2, apply $\text{REMOVEINCONSISTENT}(\text{Parent}(X_j), X_j)$
3. For j from 1 to n , assign X_j consistently with $\text{Parent}(X_j)$

- Once we have a **directed arc-consistent graph**, we can just march down the list of variables and choose any remaining value.
- Since each edge from a parent to its child is **arc-consistent**, we know that for any value we choose for the parent, there will be a valid value left to choose for the child.
- That means we won't have to backtrack; we can move linearly through the variables.

Nearly tree-structured CSPs

Conditioning: instantiate a variable, prune its neighbors' domains



Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size c \Rightarrow runtime $O(d^c \cdot (n - c)d^2)$, very fast for small c

Cycle Cutset Algorithm

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S . is called a cycle cutset.
2. For each possible assignment to the variables in S that satisfies all constraints on S ,
 - (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S , and
 - (b) if the remaining CSP has a solution, return it together with the assignment for S

Iterative algorithms for CSPs

Hill-climbing, simulated annealing typically work with
“complete” states, i.e., all variables assigned

To apply to CSPs:

- allow states with unsatisfied constraints
- operators **reassign** variable values

Variable selection: randomly select any conflicted variable

Value selection by **min-conflicts** heuristic:

- choose value that violates the fewest constraints
- i.e., hillclimb with $h(n)$ = total number of violated constraints

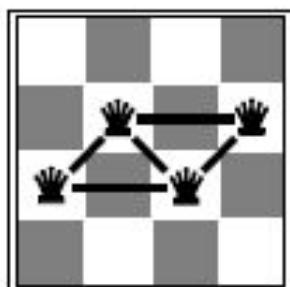
Example: 4-Queens

States: 4 queens in 4 columns ($4^4 = 256$ states)

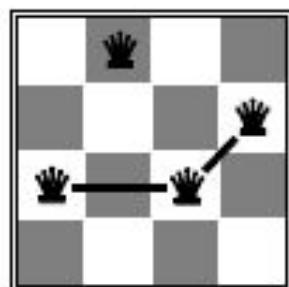
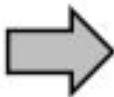
Operators: move queen in column

Goal test: no attacks

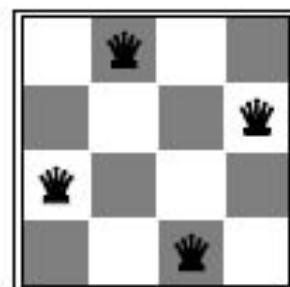
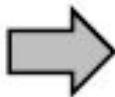
Evaluation: $h(n)$ = number of attacks



$h = 5$



$h = 2$



$h = 0$

Summary

CSPs are a special kind of problem:

states defined by values of a fixed set of variables
goal test defined by constraints on variable values

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice

Constraint Satisfaction Problem

Constraint Satisfaction Problem (CSP)

- A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a **constraint satisfaction problem**, or CSP.
- The main idea is to eliminate large portions of the search space all at once by identifying variable/value combinations that violate the constraints.

Definition of CSP

- Constraint satisfaction problem consists of three components, X, D, and C:
 - X is a set of variables, $\{X_1, \dots, X_n\}$.
 - D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.
 - C is a set of constraints that specify allowable combinations of values.
- A domain, D, consists of a set of allowable values, $\{v_1, \dots, v_n\}$, for variable X;
 - For example, a Boolean variable would have the domain {true, false}.
- Each constraint C_j consists of a pair (scope, rel),
 - where scope is a tuple of variables that participate in the constraint
 - rel is a relation that defines the values that those variables can take on.

- A relation can be represented as an explicit set of all tuples of values that satisfy the constraint, or as a function that can compute whether a tuple is a member of the relation.
 - For example, if X_1 and X_2 both have the domain $\{1,2,3\}$, then the constraint saying that X_1 ; must be greater than X_2 can be written as,
 $((X_1, X_2), \{(3,1), (3,2), (2,1)\})$ or as $((X_1, X_2), X_1 > X_2)$.

- CSPs deal with assignments of values to variables, $\{X_i = v_i, X_j = v_j, \dots\}$.
- An assignment that does not violate any constraints is called a **consistent or legal assignment**.
- A **complete assignment** is one in which every variable is assigned a value, and a **solution** to a CSP is a consistent, complete assignment.
- A **partial assignment** is one that leaves some variables unassigned, and a partial solution is a partial assignment that is **consistent**.

Example problem: Map coloring

- A map of Australia showing each of its states and territories.
- We are given the task of coloring each region either red, green, or blue in such a way that no two neighboring regions have the same color.



Example problem: Map coloring

- We define the variables to be the regions:

$$X = \{WA, NT, Q, NSW, V, SA, T\}.$$

- The domain of every variable is the set

$$D_i = \{\text{red, green, blue}\}.$$

- The constraints require neighboring regions to have distinct colors.
- Since there are nine places where regions border, there are nine constraints:

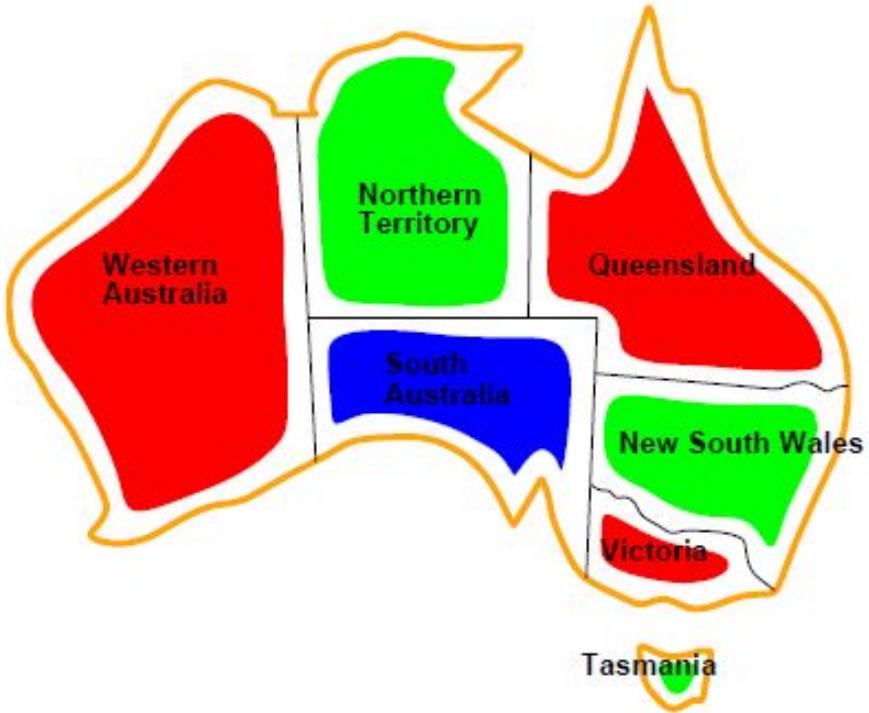
$$C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, \\ WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}.$$

Example problem: Map coloring

- Here we are using abbreviations; $SA \neq WA$ is a shortcut for $((SA, WA), SA \neq WA)$,
- Where $SA \neq WA$ can be fully enumerated in turn as
 $\{(red, green), (red, blue), (green, red), (green, blue),$
 $(blue, red), (blue, green)\}$.
- Solutions to this problem?

Example problem: Map coloring

- There are many possible solutions to this problem, such as
 $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{red}\}.$

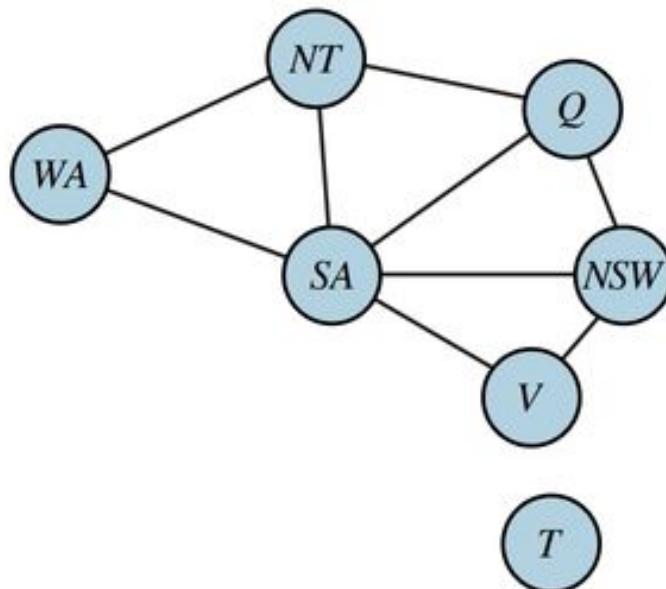


Solutions are assignments satisfying all constraints, e.g.,

$$\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green}\}$$

Example problem: Map coloring

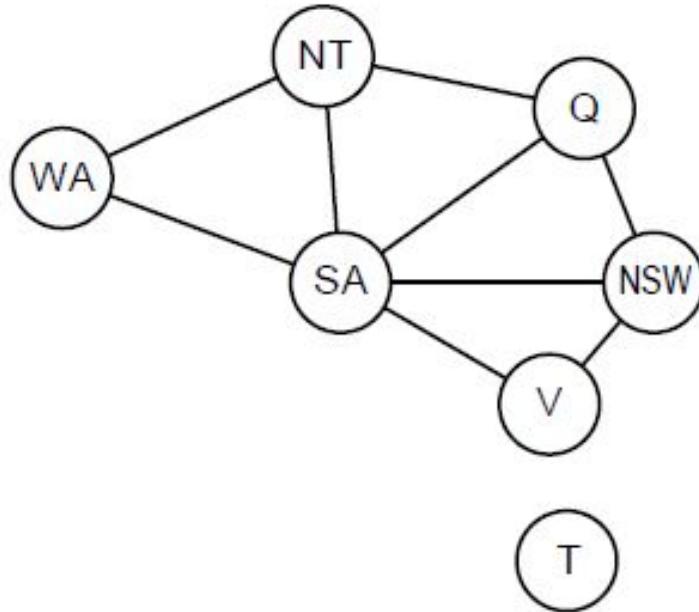
- It can be helpful to visualize a CSP as a constraint graph, as shown in Figure.
- The nodes of the graph correspond to variables of the problem, and an edge connects any two variables that participate in a constraint.



Constraint graph

Binary CSP: each constraint relates at most two variables

Constraint graph: nodes are variables, arcs show constraints



General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!

Types of CSP

Discrete variables

finite domains; size $d \Rightarrow O(d^n)$ complete assignments

- ◊ e.g., Boolean CSPs, incl. Boolean satisfiability (NP-complete)

infinite domains (integers, strings, etc.)

- ◊ e.g., job scheduling, variables are start/end days for each job
- ◊ need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$
- ◊ linear constraints solvable, nonlinear undecidable

Continuous variables

- ◊ e.g., start/end times for Hubble Telescope observations
- ◊ linear constraints solvable in poly time by LP methods

Varieties of constraints

Unary constraints involve a single variable,

e.g., $SA \neq green$

Binary constraints involve pairs of variables,

e.g., $SA \neq WA$

Higher-order constraints involve 3 or more variables,

e.g., cryptarithmetic column constraints

Preferences (soft constraints), e.g., red is better than $green$

often representable by a cost for each variable assignment

→ constrained optimization problems

Cryptarithmetic Problem

- Cryptarithmetic Problem is a type of constraint satisfaction problem where the game is about digits and its unique replacement either with alphabets or other symbols.
- In **cryptarithmetic problem**, the digits (0-9) get substituted by some possible alphabets or symbols.
- The task in cryptarithmetic problem is to substitute each digit with an alphabet to get the result arithmetically correct.

Rules and constraints

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules, i.e., $2+2=4$, nothing else.
- Digits should be from **0-9** only.
- The problem can be solved from both sides, i.e., **lefthand side (L.H.S)**, or **righthand side (R.H.S)**

Example-1

BASE

+BALL

GAMES

Example-1

$$\begin{array}{r} \text{B A S E} \\ + \text{B A L L} \\ \hline \text{G A M E S} \end{array}$$

Variables: {B,A,S,E,L,G,M}
Domain:{0,1,2,3,4,5,6,7,8,9}
Constraint: AllDiffz(B,A,S,E,L,G,M)

Example-1

BASE
+ **BALL** -

GAMES

B	5
B	5
G A	10
G	1
A	0

Example-1

B A S E
+ B A L L -
G A M E S

B	6
B	6
G A	12
G	1
A	2

Example-1

BASE

+BALL

GAMES

Cr	0	Cr	0
B	6	A	2
B	6	A	2
GA	12	M	4
G	1		
A	2		

Example-1

BASE
+ **BALL** -
GAMES

Cr	0	Cr	0	Cr	
B	6	A	2	S	3
B	6	A	2	L	5
GA	12	M	4	E	8
G	1				
A	2				

Example-1

BASE
+ **BALL** -
GAMES

Cr	0	Cr	0	C r		Cr	
B	6	A	2	S	3	E	8
B	6	A	2	L	5	L	5
GA	12	M	4	E	8	S	
G	1						
A	2						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	0	Cr		Cr	
B	7	A		S		E	
B	7	A		L		L	
GA	14	M		E		S	
G	1						
A	4						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	0	Cr		Cr	
B	7	A	4	S		E	
B	7	A	4	L		L	
GA	14	M	8	E		S	
G	1						
A	4						

Example-1 Solution

BASE

+BALL

GAMES

Cr	0	Cr	1	Cr		Cr	
B	7	A	4	S	8	E	
B	7	A	4	L	5	L	
GA	14	M	9	E	3	S	
G	1						
A	4						

Example-1 Solution

BASE

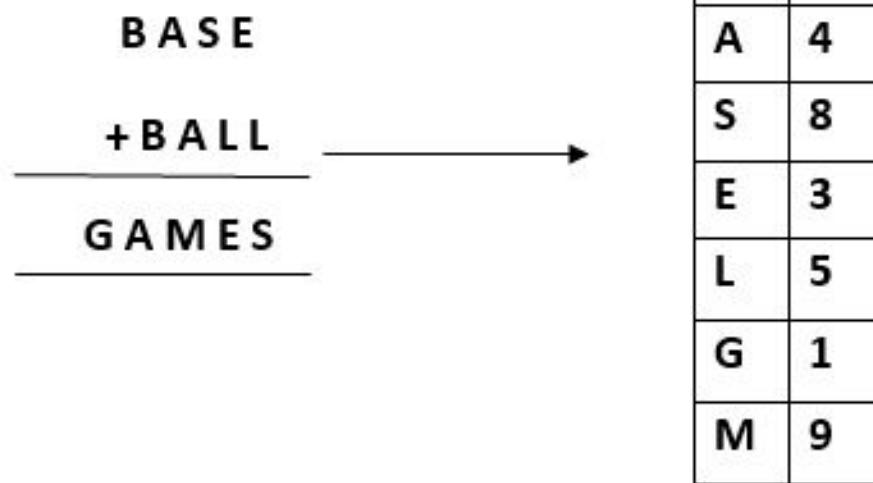
+BALL

GAMES

Cr	0	Cr	1	Cr	0	Cr	0
B	7	A	4	S	8	E	3
B	7	A	4	L	5	L	5
GA	14	M	9	E	3	S	8
G	1						
A	4						

BASE
+ **BALL**

GAMES



B	7
A	4
S	8
E	3
L	5
G	1
M	9

S E N D

+ M O R E

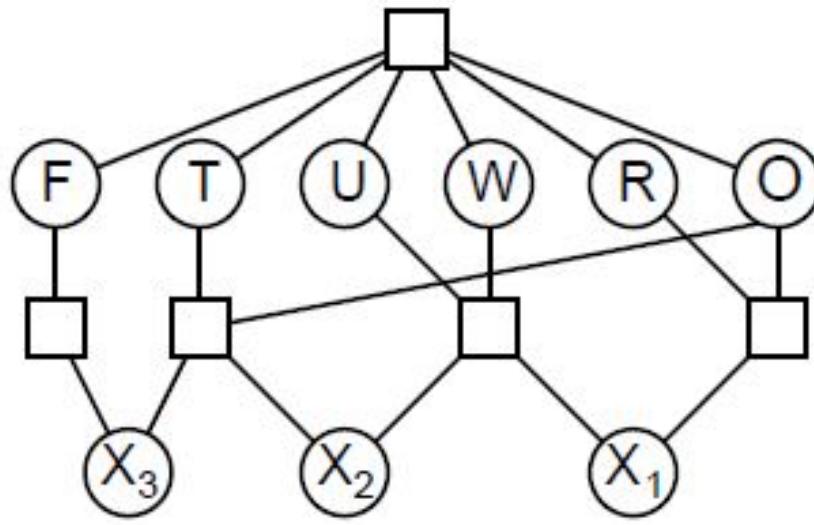
M O N E Y

S E N D
+ M O R E
—
M O N E Y
—

Cr		Cr		Cr		Cr	
S		E		N		D	
M		O		R		E	
MO		N		E		Y	
M							
O							

Example: Cryptarithmetic

$$\begin{array}{r} \text{T} \ \text{W} \ \text{O} \\ + \ \text{T} \ \text{W} \ \text{O} \\ \hline \text{F} \ \text{O} \ \text{U} \ \text{R} \end{array}$$



Variables: $F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3$

Domains: $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Constraints

$\text{alldiff}(F, T, U, W, R, O)$

$O + O = R + 10 \cdot X_1$, etc.

$$\begin{array}{r}
 \text{T} \quad \text{W} \quad \text{O} \\
 + \quad \text{T} \quad \text{W} \quad \text{O} \\
 \hline
 \text{F} \quad \text{O} \quad \text{U} \quad \text{R}
 \end{array}$$

<i>Cr</i>	0	<i>Cr</i>	0	<i>Cr</i>	0
T	7	W	3	O	4
T	7	W	3	O	4
FO	14	U	6	R	8
F	1				
O	4				

Real-world CSPs

Assignment problems

e.g., who teaches what class

Timetabling problems

e.g., which class is offered when and where?

Hardware configuration

Spreadsheets

Transportation scheduling

Factory scheduling

Floorplanning

Notice that many real-world problems involve real-valued variables

Standard search formulation (incremental)

Let's start with the straightforward, dumb approach, then fix it

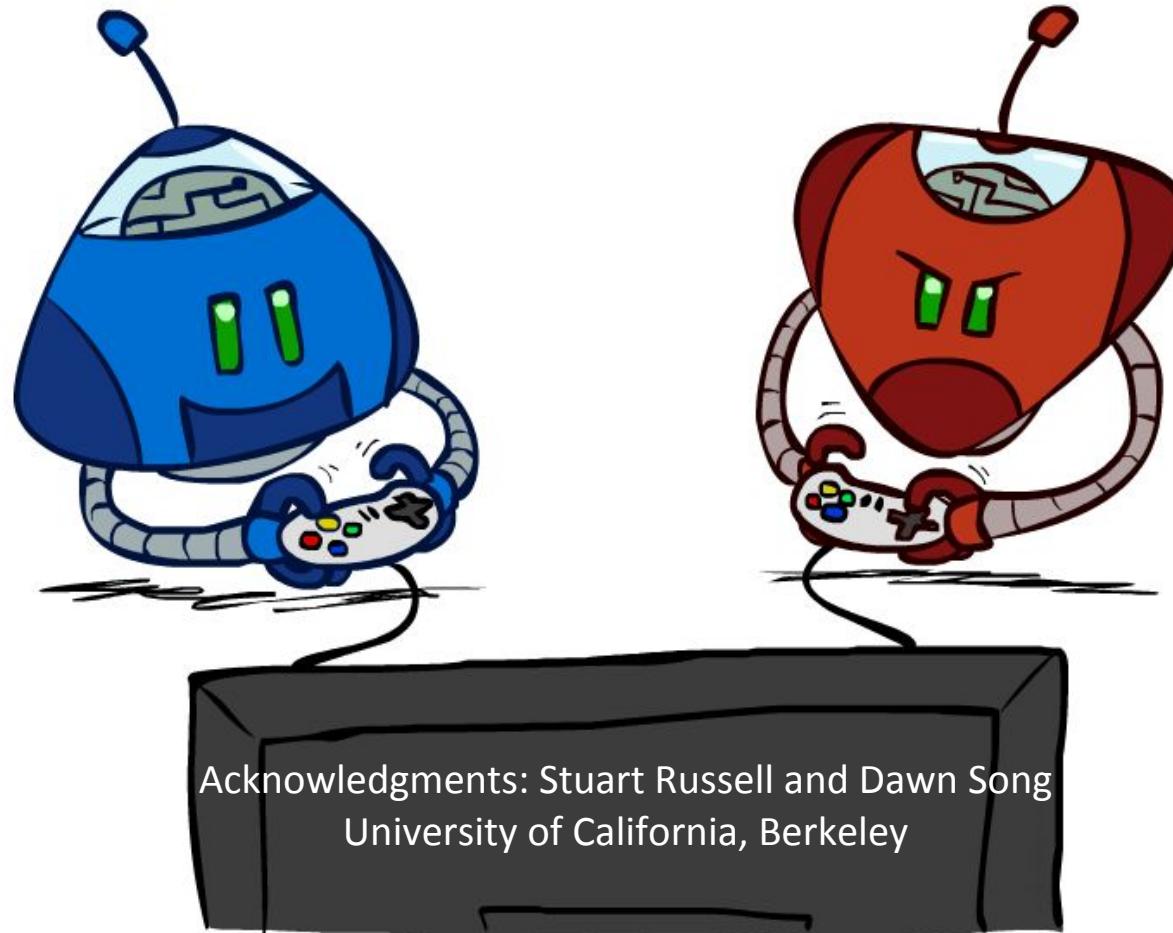
States are defined by the values assigned so far

- ◊ **Initial state:** the empty assignment, {}
- ◊ **Successor function:** assign a value to an unassigned variable that does not conflict with current assignment.
⇒ fail if no legal assignments (not fixable!)
- ◊ **Goal test:** the current assignment is complete

- 1) This is the same for all CSPs! 😊
- 2) Every solution appears at depth n with n variables
⇒ use depth-first search
- 3) Path is irrelevant, so can also use complete-state formulation
- 4) $b = (n - \ell)d$ at depth ℓ , hence $n!d^n$ leaves!!!! 😞

Artificial Intelligence

Game Playing



Acknowledgments: Stuart Russell and Dawn Song
University of California, Berkeley

A brief history

- **Checkers:**

- 1950: First computer player.
- 1959: Samuel's self-taught program.
- 1994: First computer world champion: Chinook defeats Tinsley
- 2007: Checkers solved! Endgame database of 39 trillion states

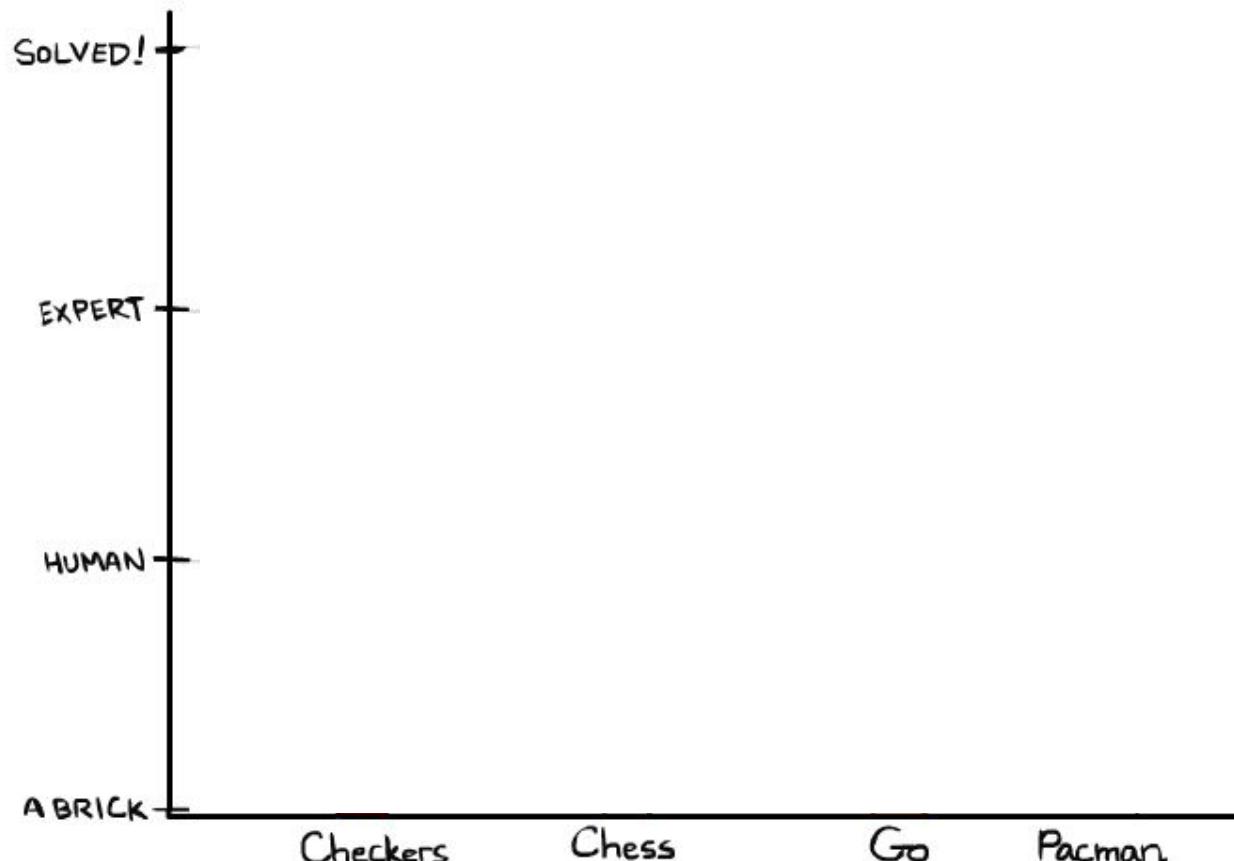
- **Chess:**

- 1945-1960: Zuse, Wiener, Shannon, Turing, Newell & Simon, McCarthy.
- 1960s onward: gradual improvement under "standard model"
- 1997: Deep Blue defeats human champion Garry Kasparov
- 2022: Stockfish rating 3541 (vs 2882 for Magnus Carlsen 2015).

- **Go:**

- 1968: Zobrist's program plays legal Go, barely ($b>300!$)
- 1968-2005: various ad hoc approaches tried, novice level
- 2005-2014: Monte Carlo tree search -> strong amateur
- 2016-2017: AlphaGo defeats human world champions

- **Pacman**



Types of Games

- Game = task environment with > 1 agent
- Axes:
 - Deterministic or stochastic?
 - Perfect information (fully observable)?
 - Two, three, or more players?
 - Teams or individuals?
 - Turn-taking or simultaneous?
 - Zero sum (good for one and bad for the other)?
- Want algorithms for calculating a ***contingent plan*** (a.k.a. **strategy** or **policy**) which recommends a move for every possible eventuality



Games vs. search problems

"Unpredictable" opponent \Rightarrow solution is a strategy specifying a move for every possible opponent reply

Time limits \Rightarrow unlikely to find goal, must approximate

Plan of attack:

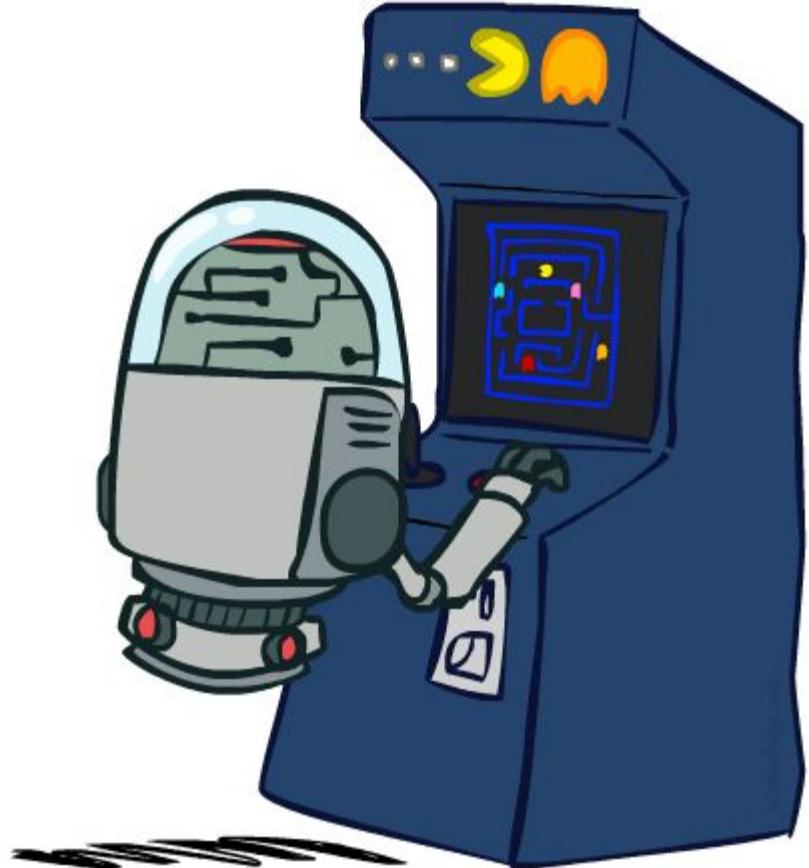
- Computer considers possible lines of play (Babbage, 1846)
- Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First chess program (Turing, 1951)
- Machine learning to improve evaluation accuracy (Samuel, 1952–57)
- Pruning to allow deeper search (McCarthy, 1956)

Types of games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

“Standard” Games

- Standard games are deterministic, observable, two-player, turn-taking, zero-sum
- Game formulation:
 - Initial state: s_0
 - Players: **Player(s)** indicates whose move it is
 - Actions: **Actions(s)** for player on move
 - Transition model: **Result(s,a)**
 - Terminal test: **Terminal-Test(s)**
 - Terminal values: **Utility(s,p)** for player p
 - Or just **Utility(s)** for player making the decision at root



Zero-Sum Games



- Zero-Sum Games
 - Agents have **opposite** utilities
 - Pure competition:
 - One **maximizes**, the other **minimizes**

- General-Sum Games
 - Agents have **independent** utilities
 - Cooperation, indifference, competition, shifting alliances, and more are all possible
- Team Games
 - Common payoff for all team members

Two-player zero-sum games

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

Two-player zero-sum games

- We will call our two players MAX and MIN.
- MAX moves first, and then the players take turns moving until the game is over.
- At the end of the game, points are awarded to the winning player and penalties are given to the loser.

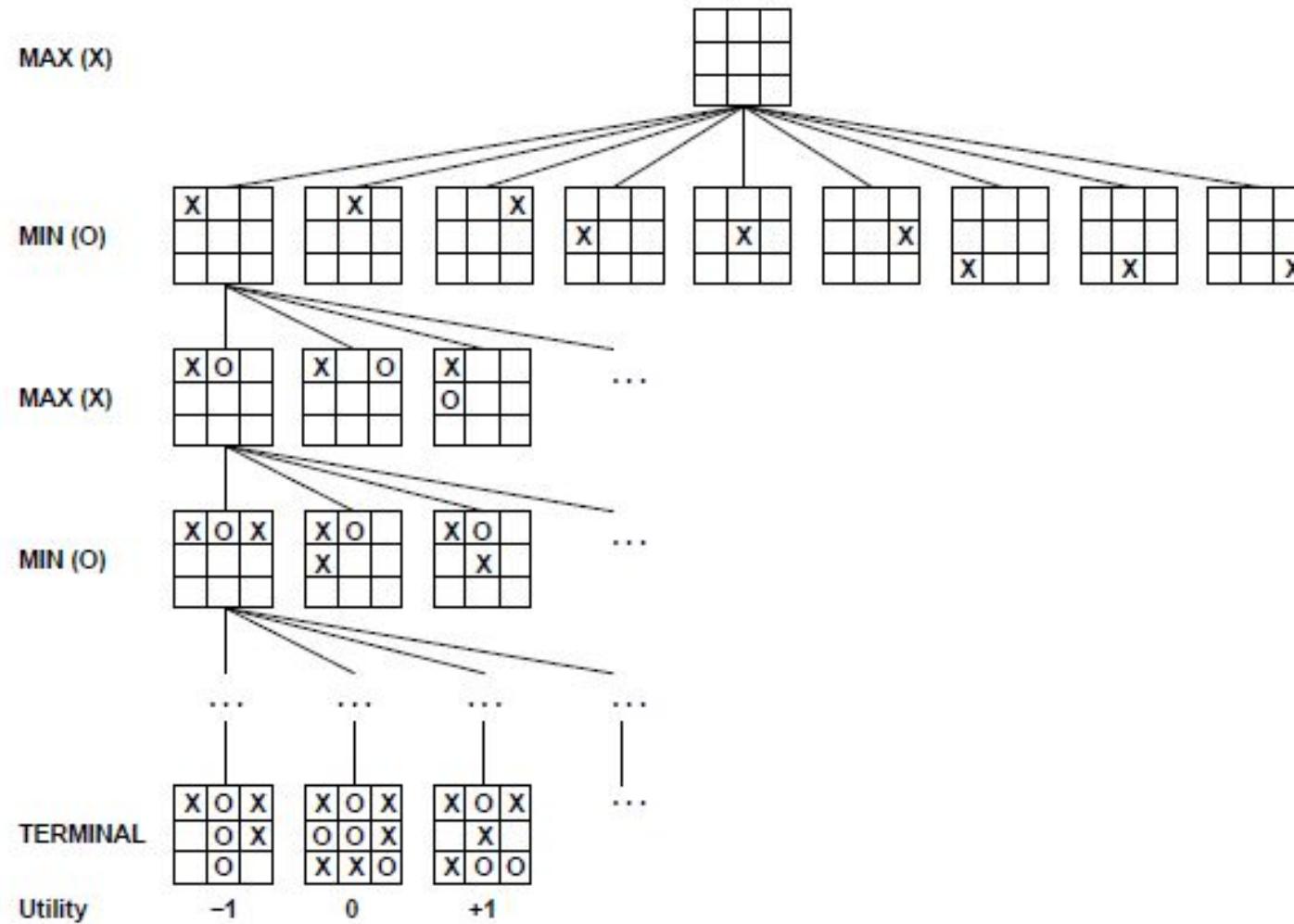
A game can be formally defined with the following elements:

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

Game Tree

- We define the complete game tree as a search tree that follows every sequence of moves all the way to a terminal state.
- The game tree may be infinite if the state space itself is unbounded or if the rules of the game allow for infinitely repeating positions.

Game tree (2-player, deterministic, turns)



Optimal Decisions in Games

- MAX wants to find a sequence of actions leading to a win, but MIN has something to say about it.
- This means that MAX's strategy must be a conditional plan—a contingent strategy specifying a response to each of MIN's possible moves.
- In fact, for such games, desirable outcome must be guaranteed no matter what the “other side” does.

- Given a game tree, the optimal strategy can be determined by working out the minimax value of each state in the tree, which we write as $\text{MINIMAX}(s)$.
- The minimax value is the utility (for MAX) of being in that state, assuming that both players play optimally from there to the end of the game.
- The minimax value of a terminal state is just its utility

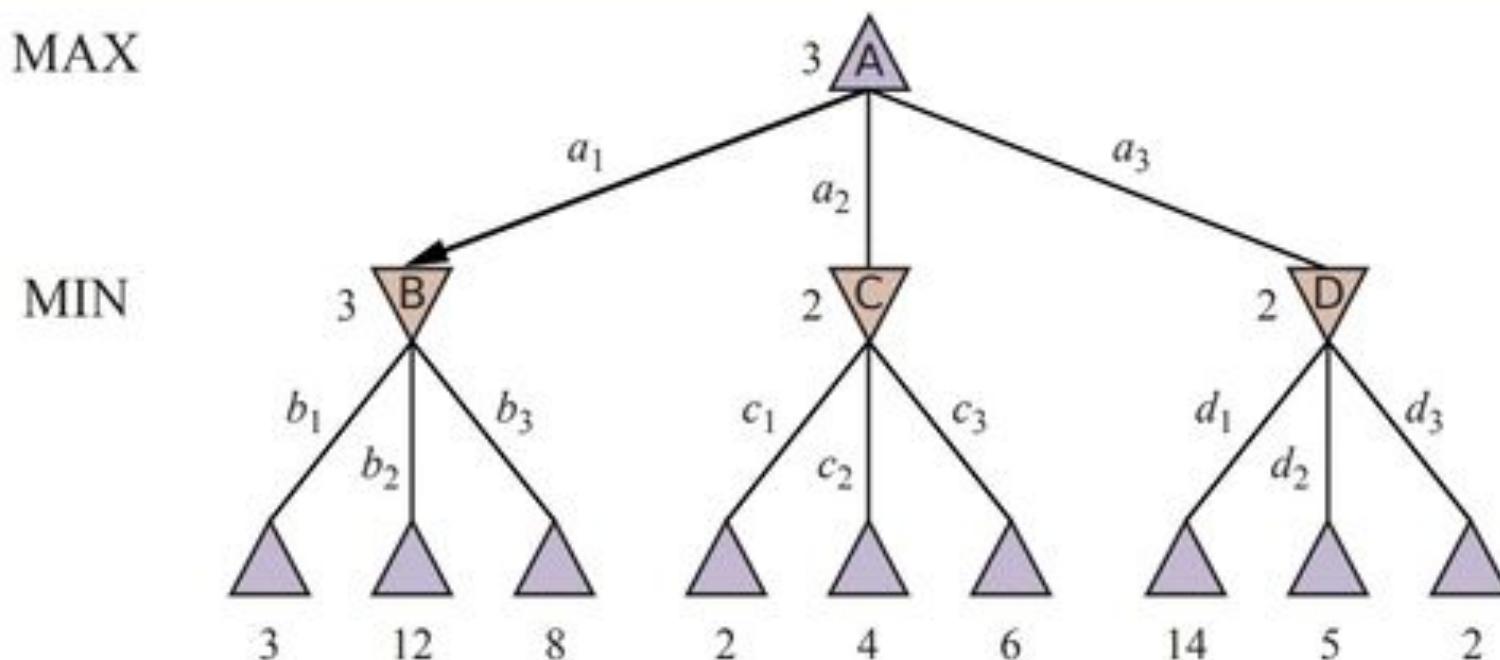
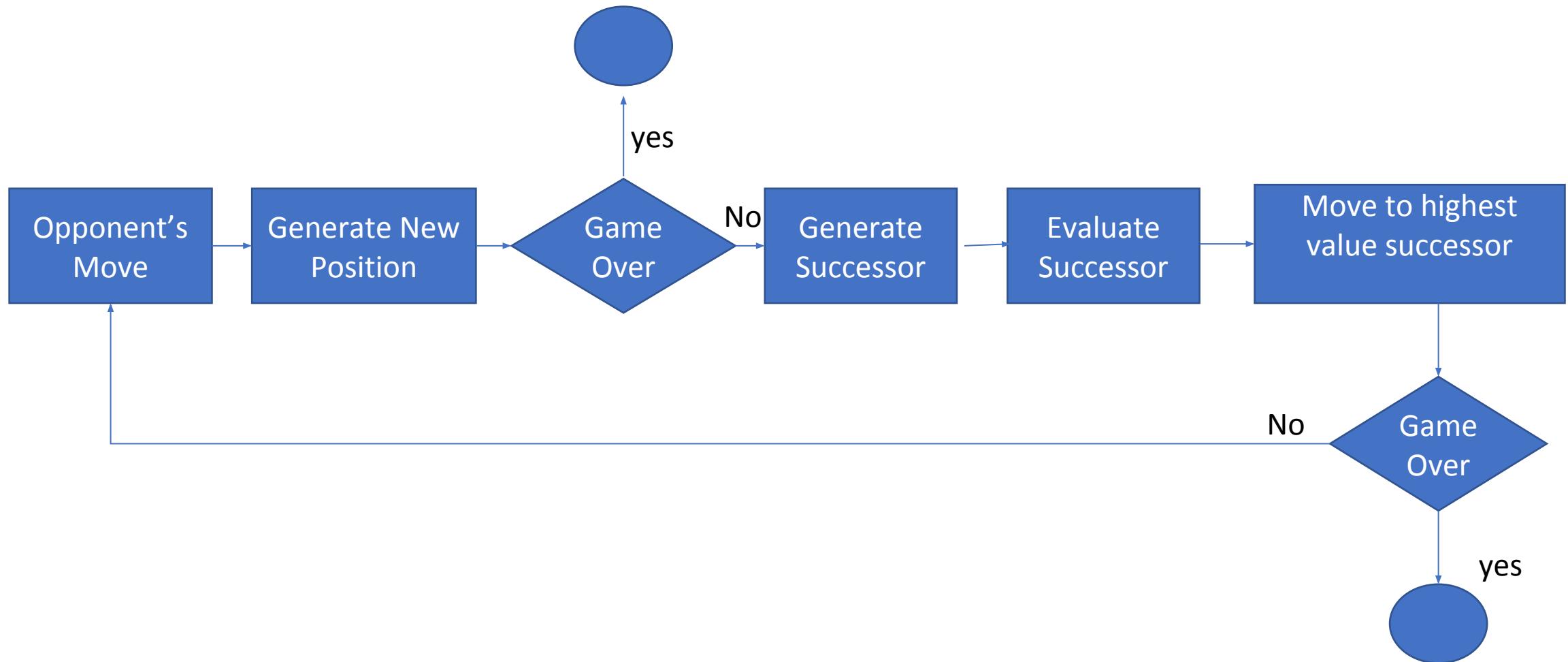


Figure 5.2 A two-ply game tree. The \triangle nodes are “MAX nodes,” in which it is MAX’s turn to move, and the ∇ nodes are “MIN nodes.” The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX’s best move at the root is a_1 , because it leads to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

Two Player Game



GAME PLAYING

CHAPTER 6

Outline

- ◊ Games
- ◊ Perfect play
 - minimax decisions
 - α - β pruning
- ◊ Resource limits and approximate evaluation
- ◊ Games of chance
- ◊ Games of imperfect information

Games vs. search problems

“Unpredictable” opponent \Rightarrow solution is a **strategy** specifying a move for every possible opponent reply

Time limits \Rightarrow unlikely to find goal, must approximate

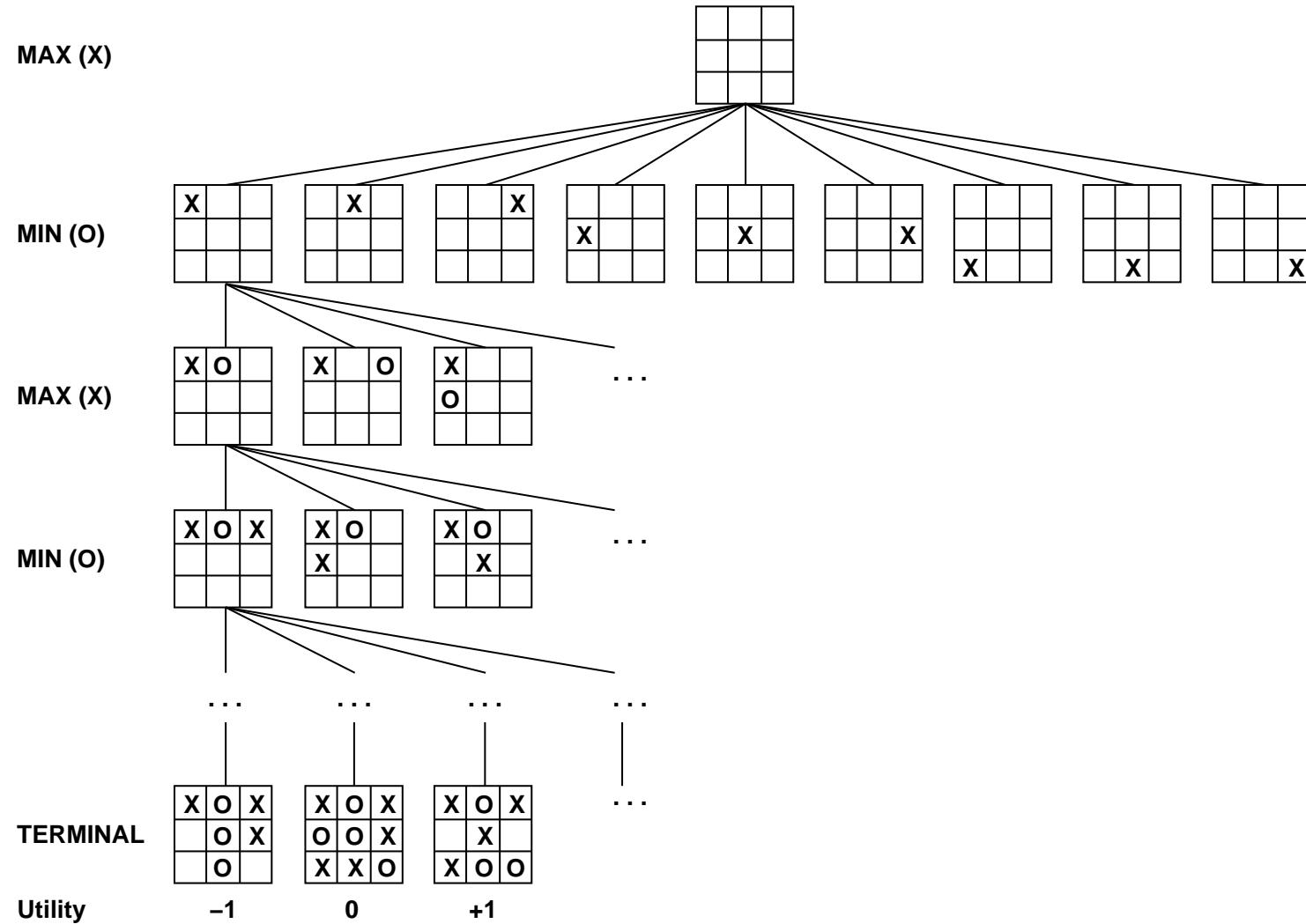
Plan of attack:

- Computer considers possible lines of play (Babbage, 1846)
- Algorithm for perfect play (Zermelo, 1912; Von Neumann, 1944)
- Finite horizon, approximate evaluation (Zuse, 1945; Wiener, 1948; Shannon, 1950)
- First chess program (Turing, 1951)
- Machine learning to improve evaluation accuracy (Samuel, 1952–57)
- Pruning to allow deeper search (McCarthy, 1956)

Types of games

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Game tree (2-player, deterministic, turns)

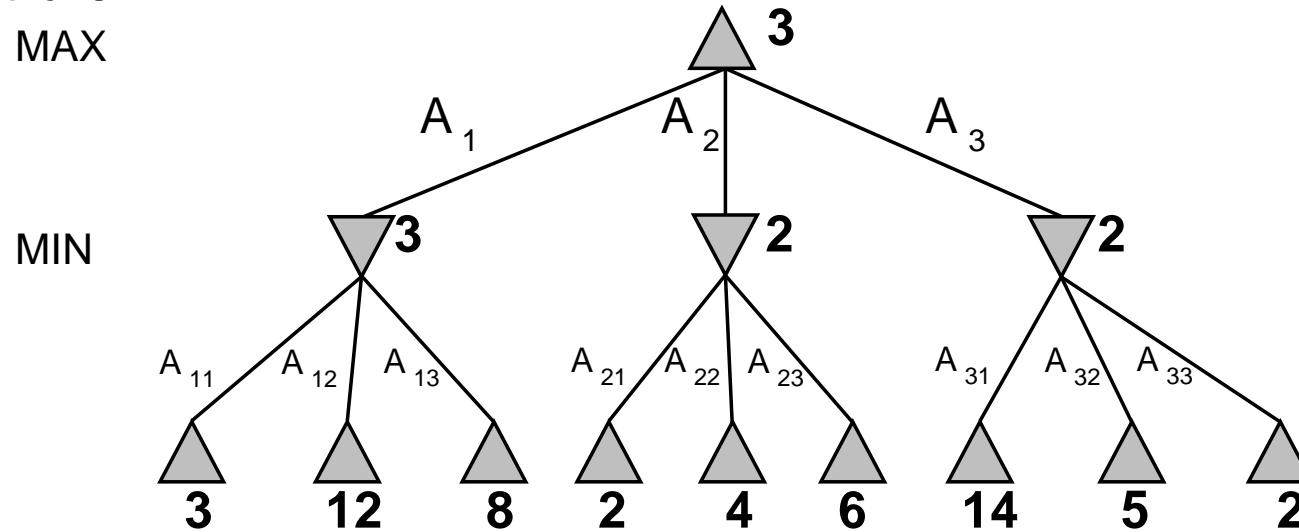


Minimax

Perfect play for deterministic, perfect-information games

Idea: choose move to position with highest **minimax value**
= best achievable payoff against best play

E.g., 2-ply game:



Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v  $\leftarrow -\infty$ 
    for a, s in SUCCESSORS(state) do v  $\leftarrow \text{MAX}(\text{MAX-VALUE}(\text{RESULT}(\text{ACTION}(\iota, \text{state}))), v)$ 
    return v
```

```
function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v  $\leftarrow \infty$ 
    for a, s in SUCCESSORS(state) do v  $\leftarrow \text{MIN}(\text{MIN-VALUE}(\text{RESULT}(\text{ACTION}(\iota, \text{state}))), v)$ 
    return v
```

Properties of minimax

Complete??

Properties of minimax

Complete?? Only if tree is finite (chess has specific rules for this).

NB a finite strategy can exist even in an infinite tree!

Optimal??

Properties of minimax

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity??

Properties of minimax

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity?? $O(b^m)$

Space complexity??

Properties of minimax

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

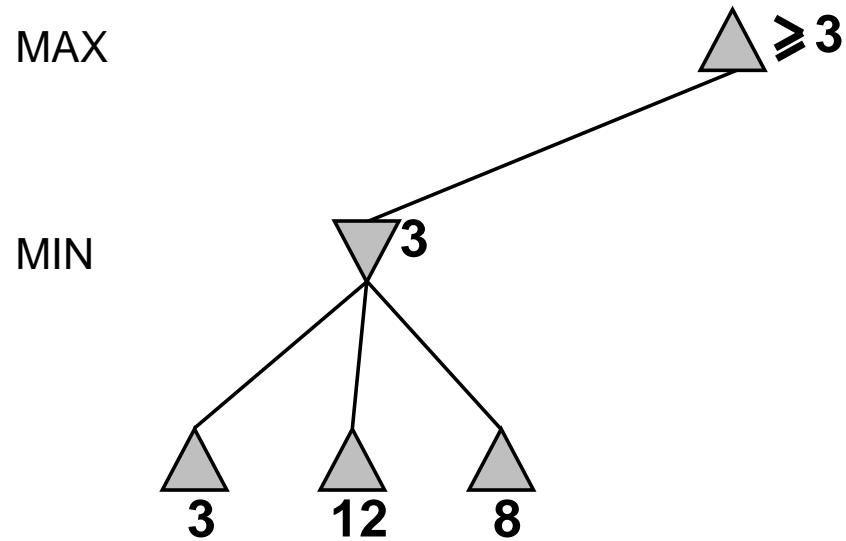
Time complexity?? $O(b^m)$

Space complexity?? $O(bm)$ (depth-first exploration)

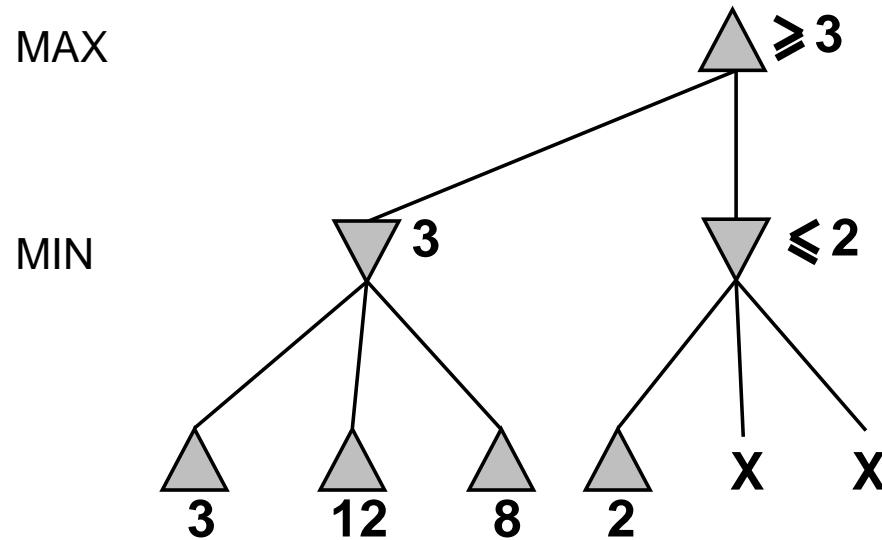
For chess, $b \approx 35$, $m \approx 100$ for “reasonable” games
⇒ exact solution completely infeasible

But do we need to explore every path?

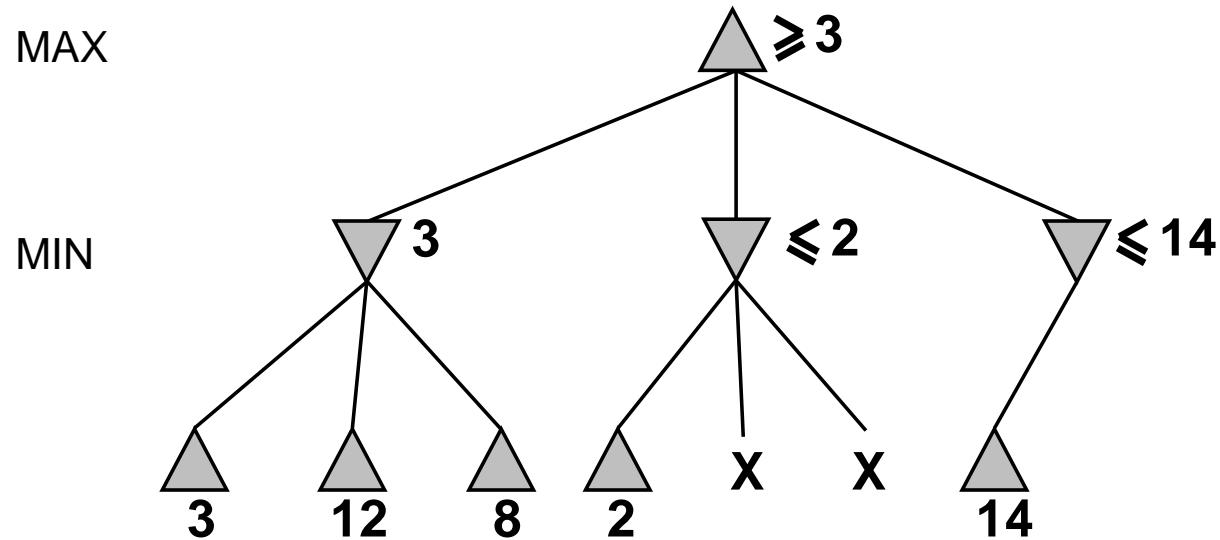
$\alpha-\beta$ pruning example



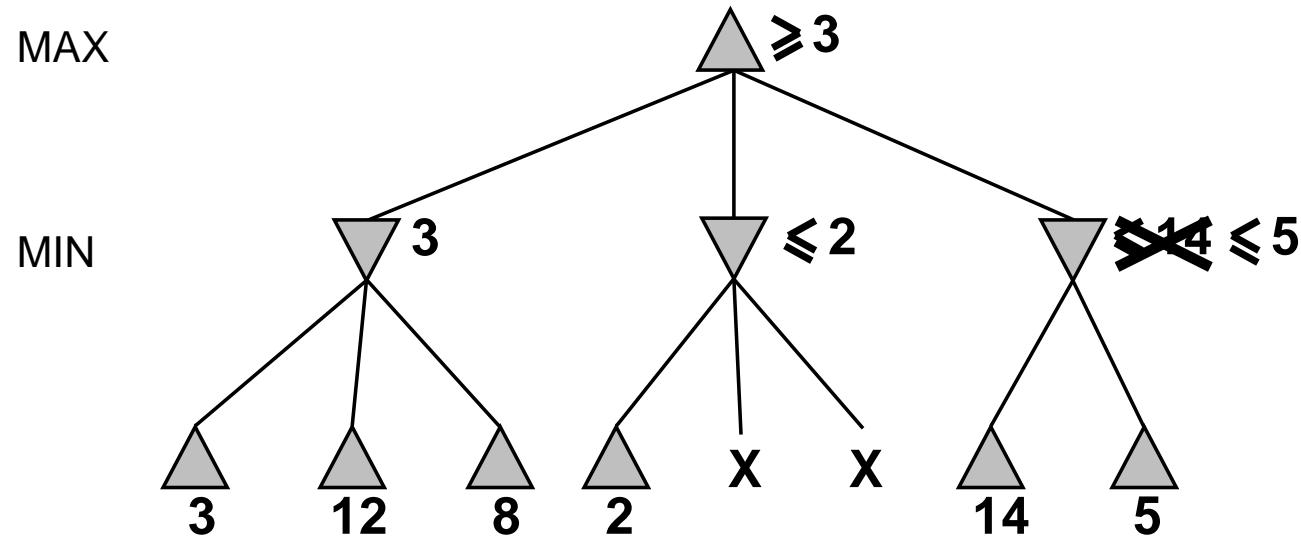
$\alpha-\beta$ pruning example



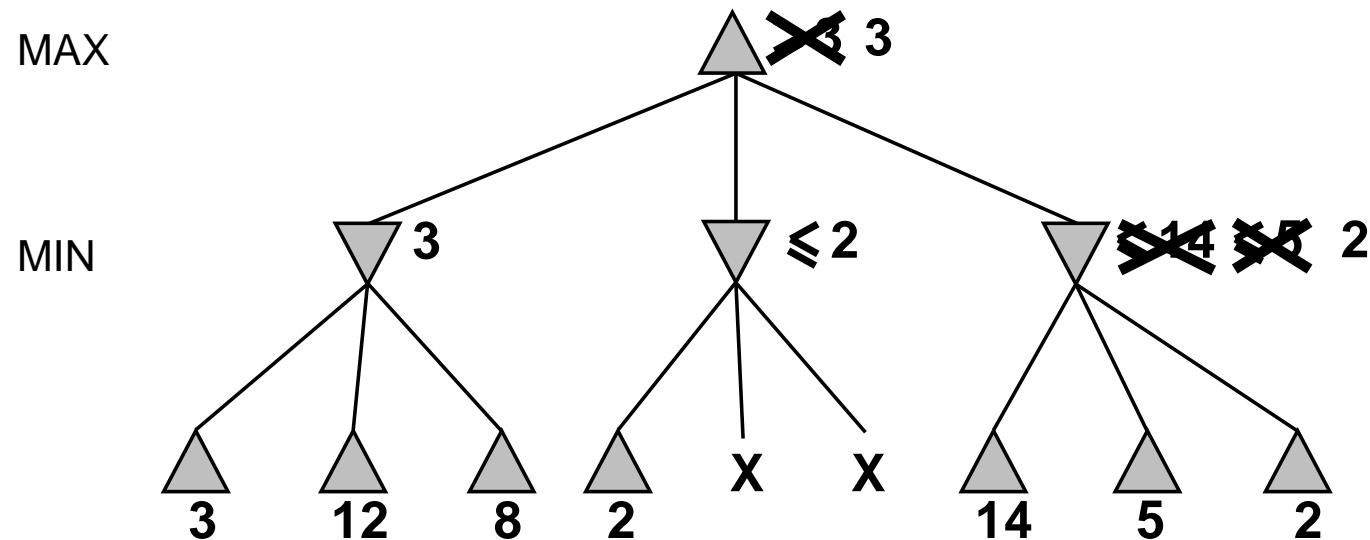
$\alpha-\beta$ pruning example



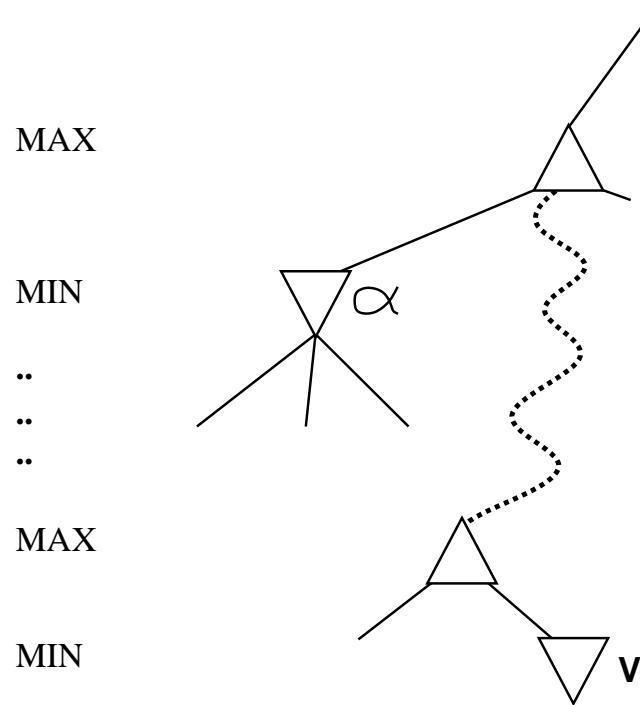
$\alpha-\beta$ pruning example



$\alpha-\beta$ pruning example



Why is it called α - β ?



α is the best value (to MAX) found so far off the current path

If V is worse than α , MAX will avoid it \Rightarrow prune that branch

Define β similarly for MIN

The α - β algorithm

```
function ALPHA-BETA-DECISION(state) returns an action
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
function MAX-VALUE(state, α, β) returns a utility value
```

inputs: *state*, current state in game

α , the value of the best alternative for MAX along the path to *state*

β , the value of the best alternative for MIN along the path to *state*

```
if TERMINAL-TEST(state) then return UTILITY(state)
```

$v \leftarrow -\infty$

```
for a, s in SUCCESSORS(state) do
```

$v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\textit{s}, \alpha, \beta))$

 if $v \geq \beta$ then return *v*

$\alpha \leftarrow \text{MAX}(\alpha, v)$

```
return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
```

same as MAX-VALUE but with roles of α, β reversed

Properties of α - β

Pruning **does not** affect final result

Good move ordering improves effectiveness of pruning

With “perfect ordering,” time complexity = $O(b^{m/2})$
⇒ **doubles** solvable depth

A simple example of the value of reasoning about which computations are relevant (a form of **metareasoning**)

Unfortunately, 35^{50} is still impossible!

Resource limits

Standard approach:

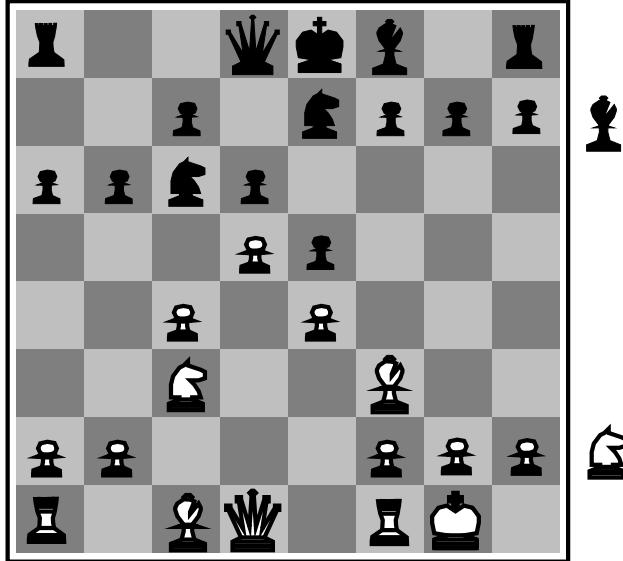
- Use CUTOFF-TEST instead of TERMINAL-TEST
 - e.g., depth limit (perhaps add quiescence search)
- Use EVAL instead of UTILITY
 - i.e., evaluation function that estimates desirability of position

Suppose we have 100 seconds, explore 10^4 nodes/second

$$\Rightarrow 10^6 \text{ nodes per move} \approx 35^{8/2}$$

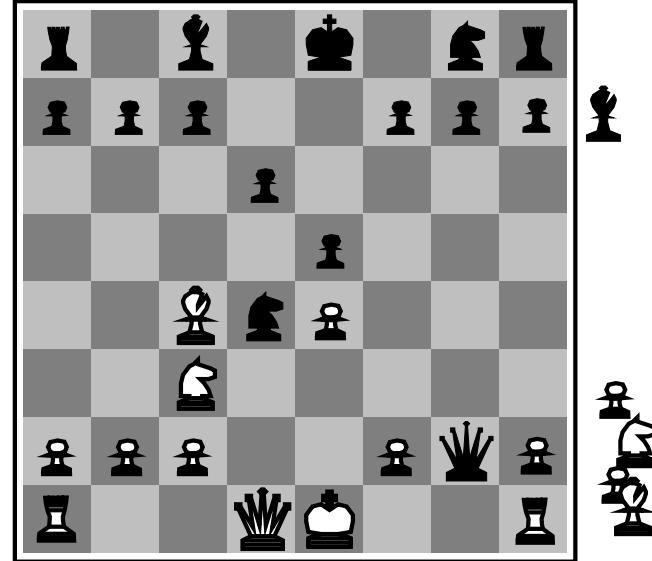
$\Rightarrow \alpha-\beta$ reaches depth 8 \Rightarrow pretty good chess program

Evaluation functions



Black to move

White slightly better



White to move

Black winning

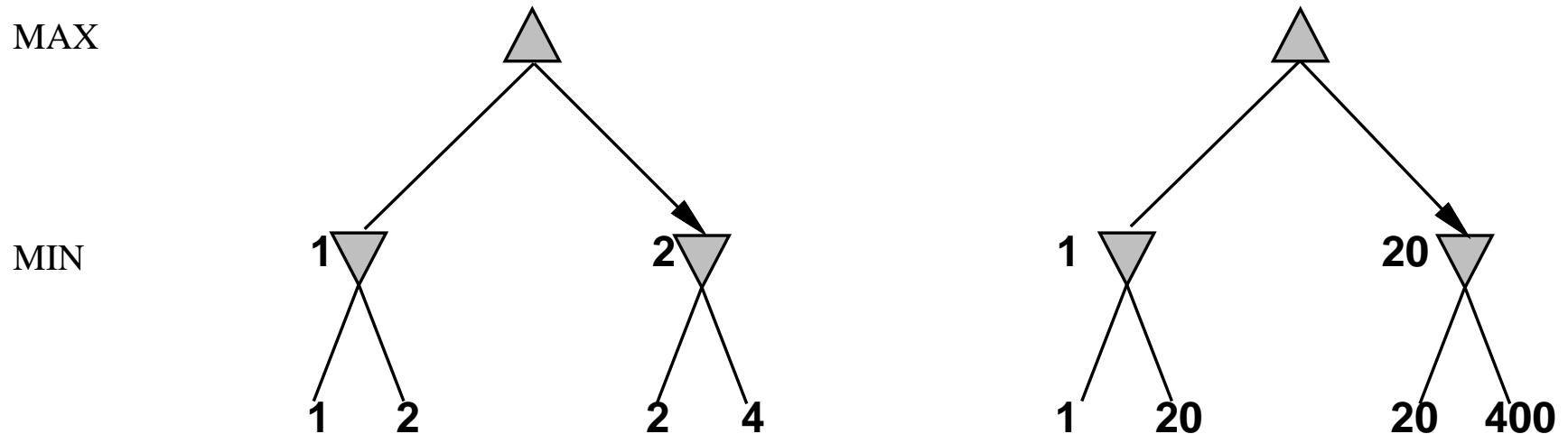
For chess, typically linear weighted sum of features

$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$$

e.g., $w_1 = 9$ with

$f_1(s) = (\text{number of white queens}) - (\text{number of black queens})$, etc.

Digression: Exact values don't matter



Behaviour is preserved under any **monotonic** transformation of EVAL

Only the order matters:

payoff in deterministic games acts as an **ordinal utility function**

Deterministic games in practice

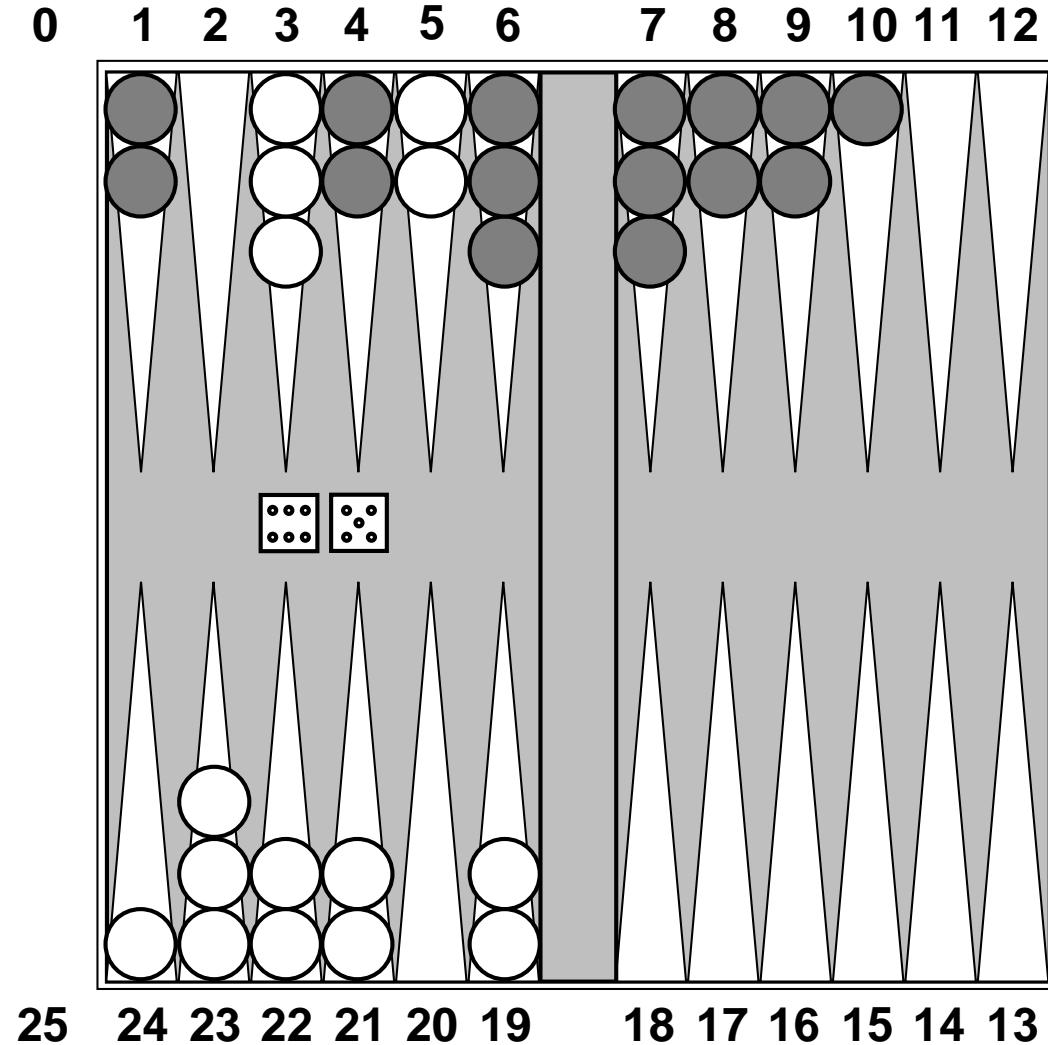
Checkers: Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions.

Chess: Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue searches 200 million positions per second, uses very sophisticated evaluation, and undisclosed methods for extending some lines of search up to 40 ply.

Othello: human champions refuse to compete against computers, who are too good.

Go: human champions refuse to compete against computers, who are too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.

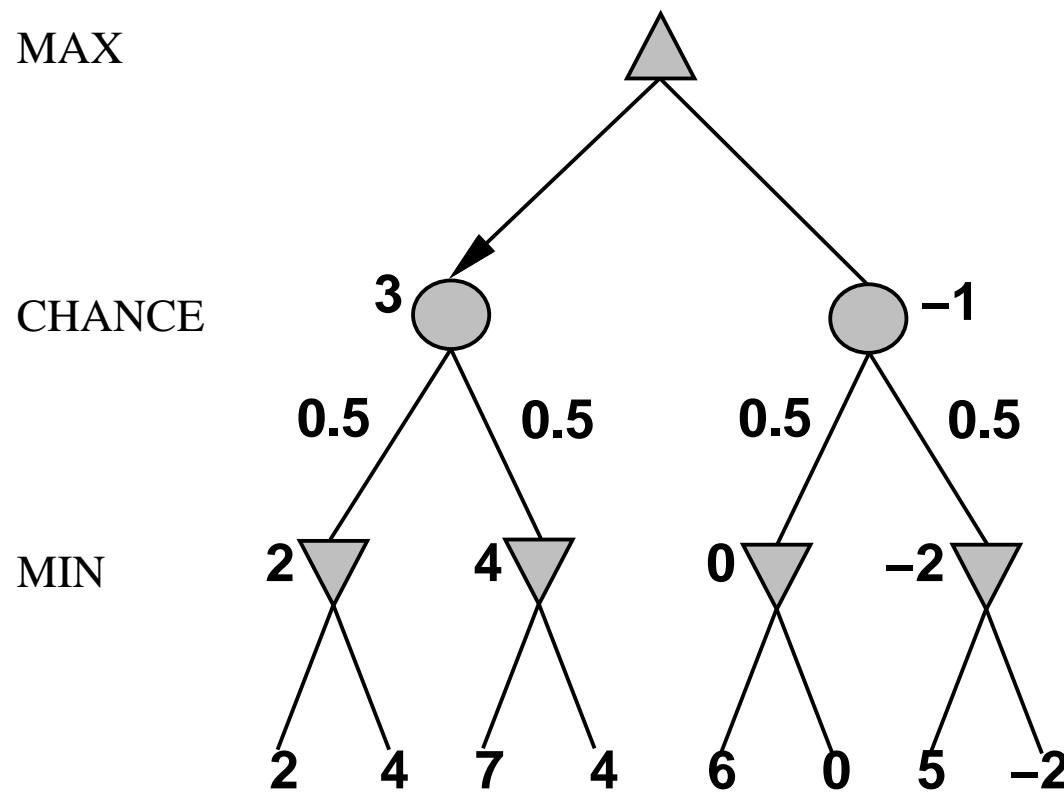
Nondeterministic games: backgammon



Nondeterministic games in general

In nondeterministic games, chance introduced by dice, card-shuffling

Simplified example with coin-flipping:



Algorithm for nondeterministic games

EXPECTIMINIMAX gives perfect play

Just like MINIMAX, except we must also handle chance nodes:

...

```
if state is a MAX node then
    return the highest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a MIN node then
    return the lowest EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
if state is a chance node then
    return average of EXPECTIMINIMAX-VALUE of SUCCESSORS(state)
...
```

Nondeterministic games in practice

Dice rolls increase b : 21 possible rolls with 2 dice

Backgammon ≈ 20 legal moves (can be 6,000 with 1-1 roll)

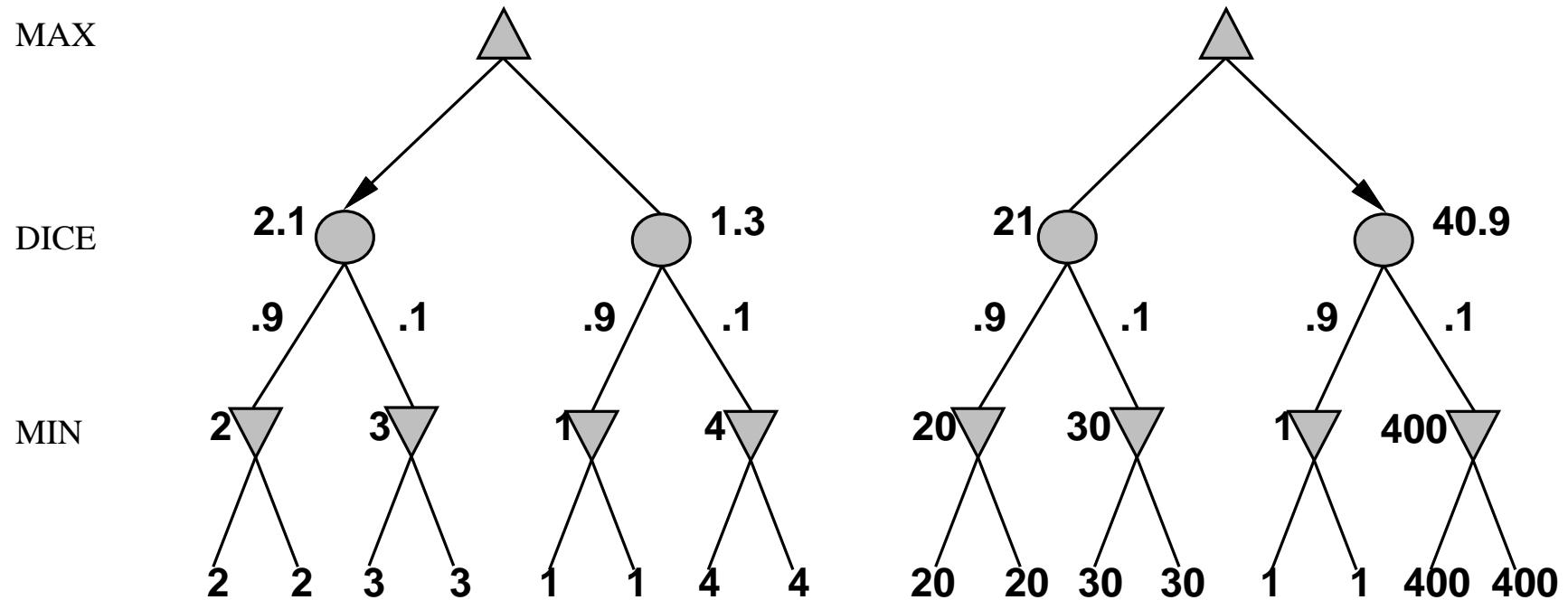
$$\text{depth } 4 = 20 \times (21 \times 20)^3 \approx 1.2 \times 10^9$$

As depth increases, probability of reaching a given node shrinks
⇒ value of lookahead is diminished

α - β pruning is much less effective

TDGAMMON uses depth-2 search + very good EVAL
 \approx world-champion level

Digression: Exact values DO matter



Behaviour is preserved only by positive linear transformation of EVAL

Hence EVAL should be proportional to the expected payoff

Games of imperfect information

E.g., card games, where opponent's initial cards are unknown

Typically we can calculate a probability for each possible deal

Seems just like having one big dice roll at the beginning of the game^{*}

Idea: compute the minimax value of each action in each deal,
then choose the action with highest expected value over all deals^{*}

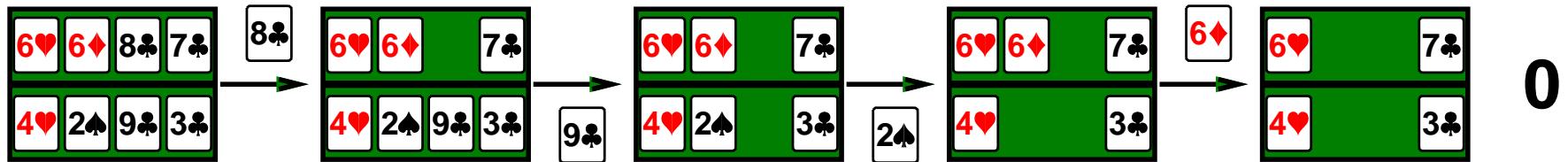
Special case: if an action is optimal for all deals, it's optimal.^{*}

GIB, current best bridge program, approximates this idea by

- 1) generating 100 deals consistent with bidding information
- 2) picking the action that wins most tricks on average

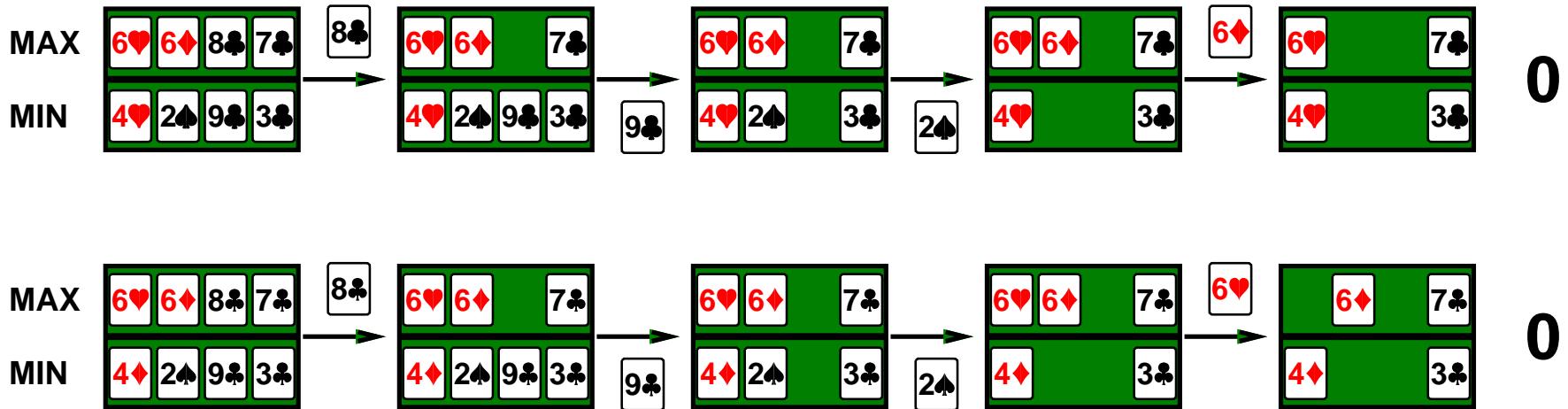
Example

Four-card bridge/whist/hearts hand, MAX to play first



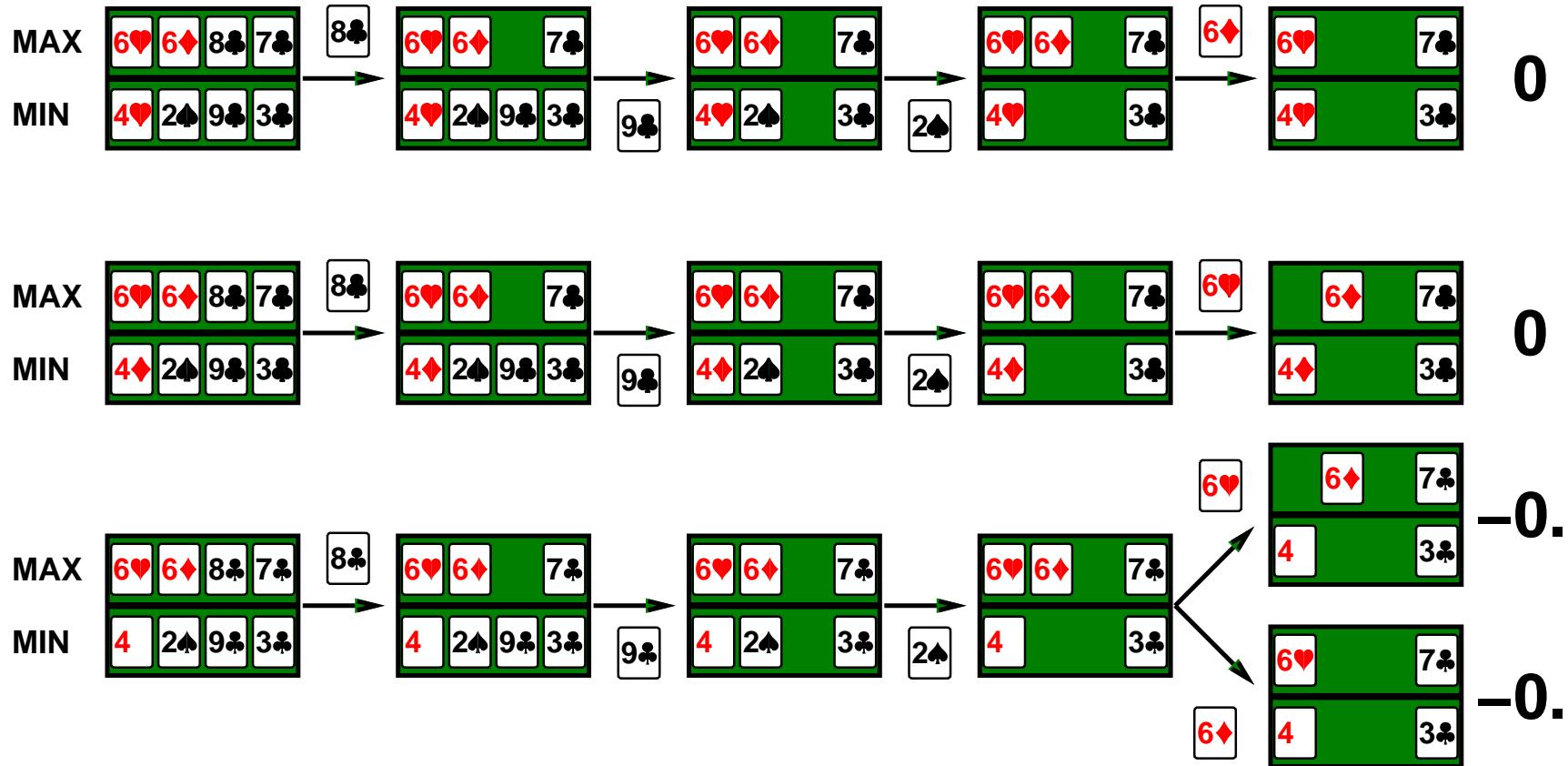
Example

Four-card bridge/whist/hearts hand, MAX to play first



Example

Four-card bridge/whist/hearts hand, MAX to play first



Commonsense example

Road A leads to a small heap of gold pieces

Road B leads to a fork:

take the left fork and you'll find a mound of jewels;

take the right fork and you'll be run over by a bus.

Commonsense example

Road A leads to a small heap of gold pieces

Road B leads to a fork:

take the left fork and you'll find a mound of jewels;

take the right fork and you'll be run over by a bus.

Road A leads to a small heap of gold pieces

Road B leads to a fork:

take the left fork and you'll be run over by a bus;

take the right fork and you'll find a mound of jewels.

Commonsense example

Road A leads to a small heap of gold pieces

Road B leads to a fork:

- take the left fork and you'll find a mound of jewels;
- take the right fork and you'll be run over by a bus.

Road A leads to a small heap of gold pieces

Road B leads to a fork:

- take the left fork and you'll be run over by a bus;
- take the right fork and you'll find a mound of jewels.

Road A leads to a small heap of gold pieces

Road B leads to a fork:

- guess correctly and you'll find a mound of jewels;
- guess incorrectly and you'll be run over by a bus.

Proper analysis

* Intuition that the value of an action is the average of its values in all actual states is **WRONG**

With partial observability, value of an action depends on the **information state or belief state** the agent is in

Can generate and search a tree of information states

Leads to rational behaviors such as

- ◇ Acting to obtain information
- ◇ Signalling to one's partner
- ◇ Acting randomly to minimize information disclosure

Summary

Games are fun to work on! (and dangerous)

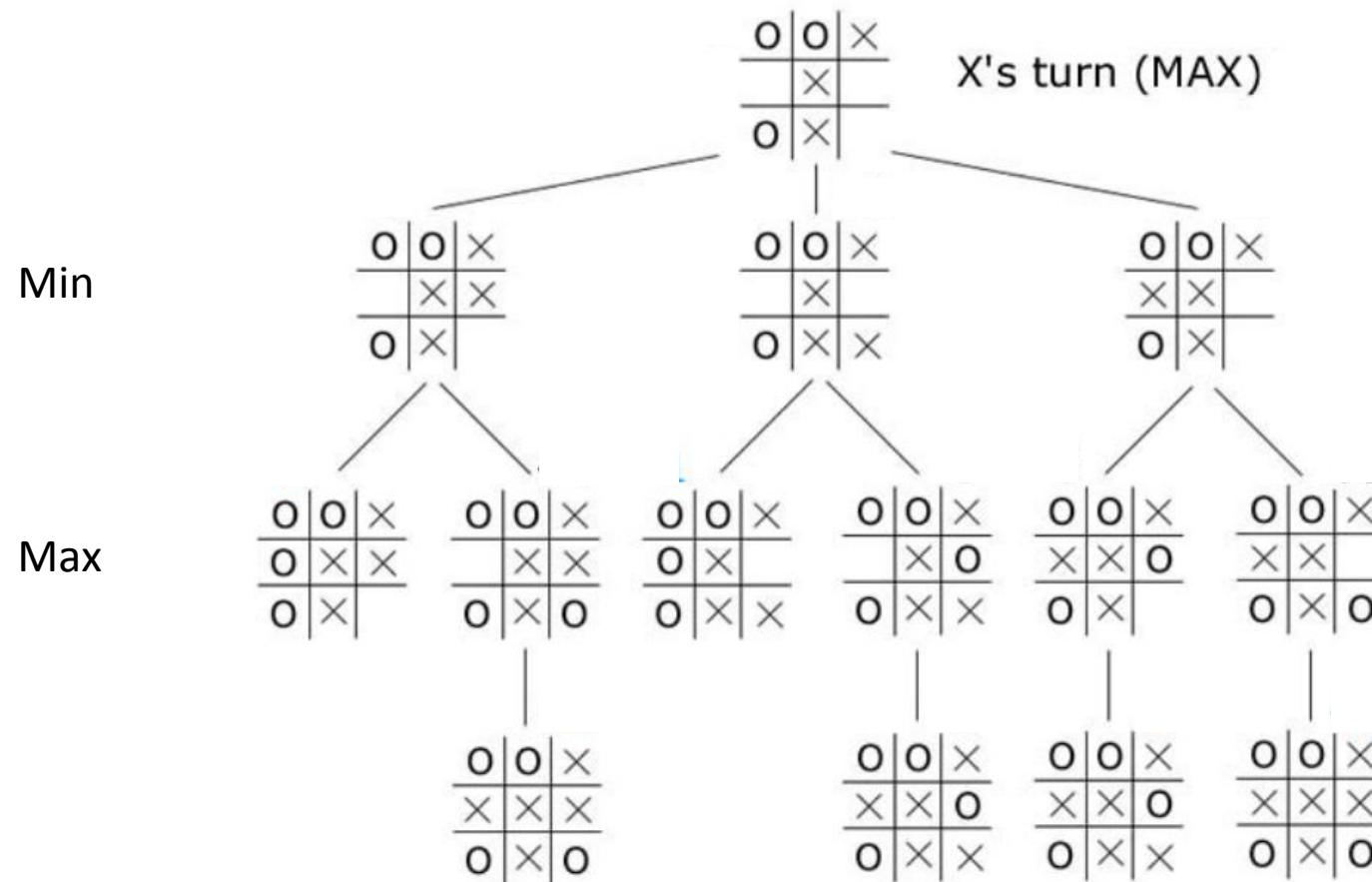
They illustrate several important points about AI

- ◊ perfection is unattainable \Rightarrow must approximate
- ◊ good idea to think about what to think about
- ◊ uncertainty constrains the assignment of values to states
- ◊ optimal decisions depend on information state, not real state

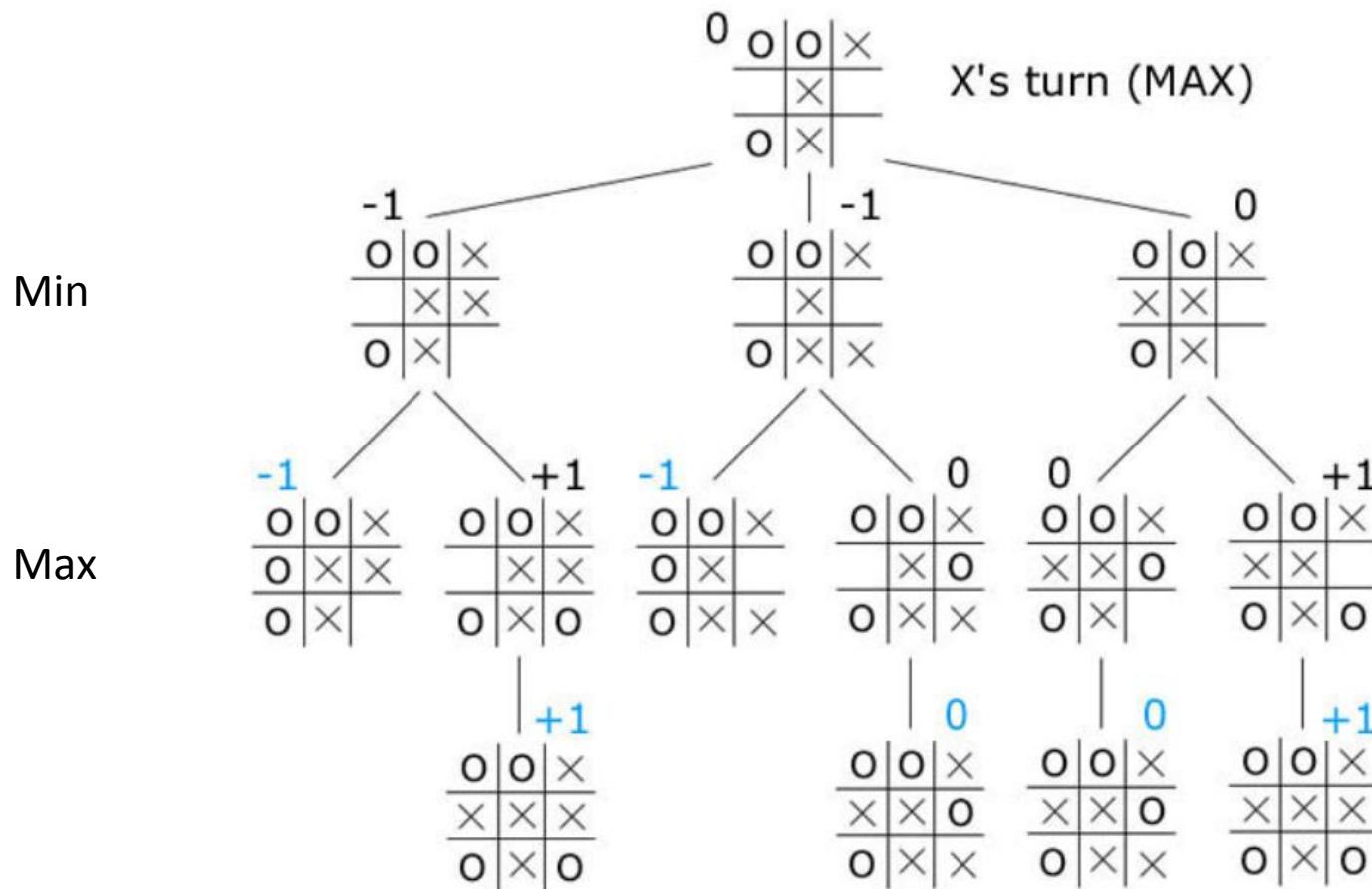
Games are to AI as grand prix racing is to automobile design

Minimax and Alpha Beta Pruning

Game tree for Tic-Tac-Toe



Game tree for Tic-Tac-Toe



Minmax Algorithm

```
minimax(player,board)
```

```
    if(game over in current board position)
```

```
        return winner
```

```
    children = all legal moves for player from this board
```

```
    if(max's turn)
```

```
        return maximal score of calling minimax on all the children
```

```
    else (min's turn)
```

```
        return minimal score of calling minimax on all the children
```

Depth-Limited Minmax Pseudocode

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := -∞
        for each child of node do
            value := max(value, minimax(child, depth - 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth - 1, TRUE))
        return value
```

Properties of minimax

Complete?? Yes, if tree is finite (chess has specific rules for this)

Optimal?? Yes, against an optimal opponent. Otherwise??

Time complexity?? $O(b^m)$

Space complexity?? $O(bm)$ (depth-first exploration)

For chess, $b \approx 35$, $m \approx 100$ for “reasonable” games
⇒ exact solution completely infeasible

But do we need to explore every path?

Alpha-Beta Pruning

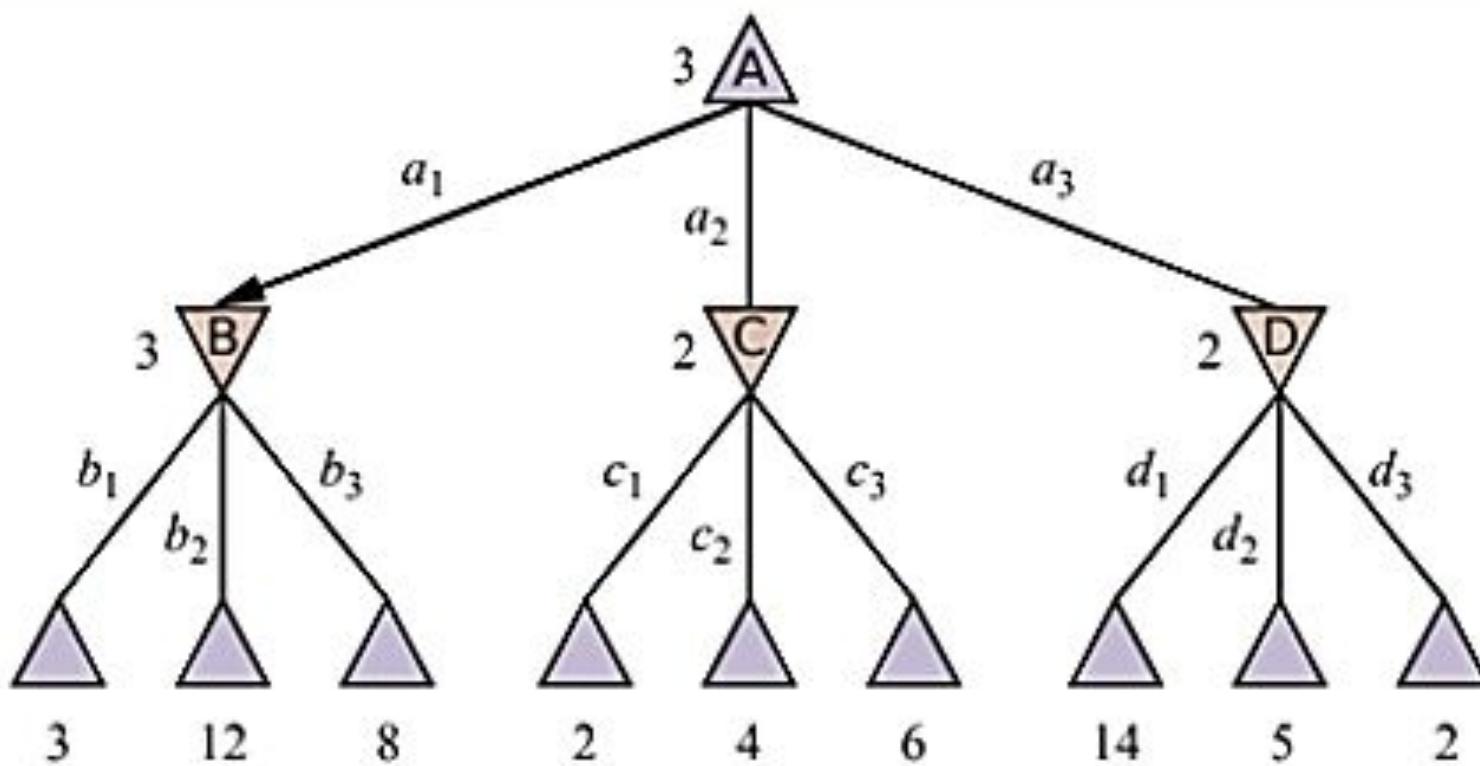
- The number of game states is exponential in the depth of the tree.
- No algorithm can completely eliminate the exponent.
- But we can sometimes cut it in half, computing the correct minimax decision without examining every state by **pruning** large parts of the tree that make no difference to the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

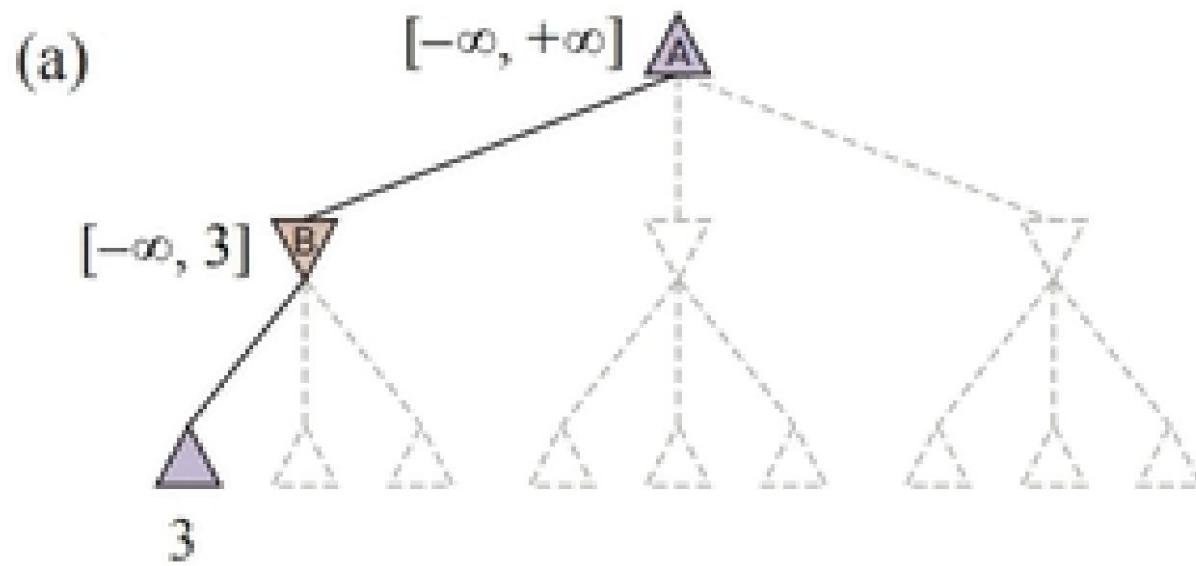
Alpha beta pruning

- Children inherit parent's alpha and beta values via passing them as arguments during recursive calls (not while returning).
- Alpha can only be updated during max's turn
- Alpha = the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX.
 - Think: alpha= "at least."
- Beta can only be updated during min's turn
- Beta= the value of the best (i.e., lowe: lue) choice we have found so far at any choice point along the path for MIN.
 - Think: Beta = "at most."
- When Alpha 'is greater than equal to' beta we don't look into the remaining children (AKA **pruning**)

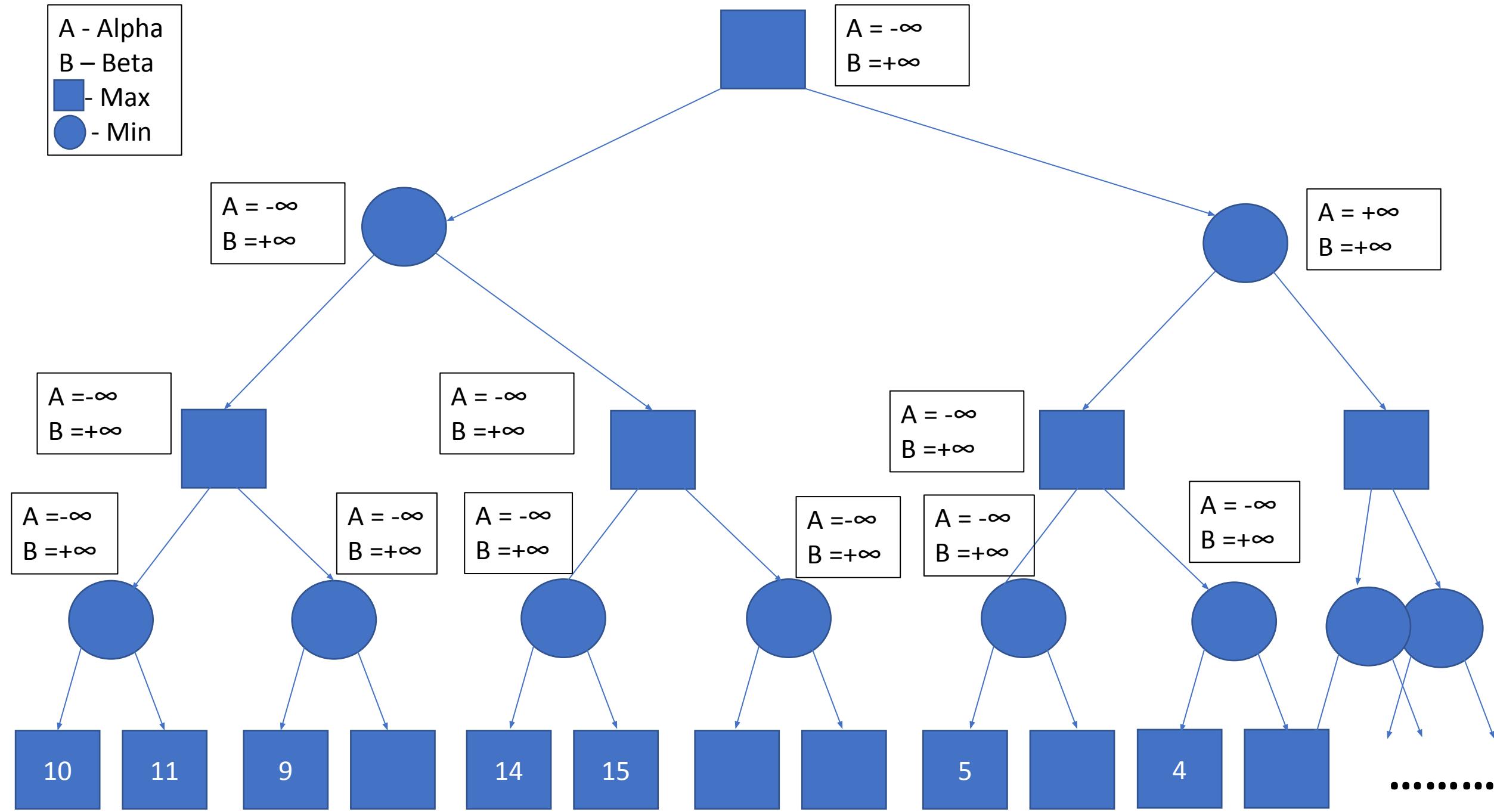
MAX

MIN

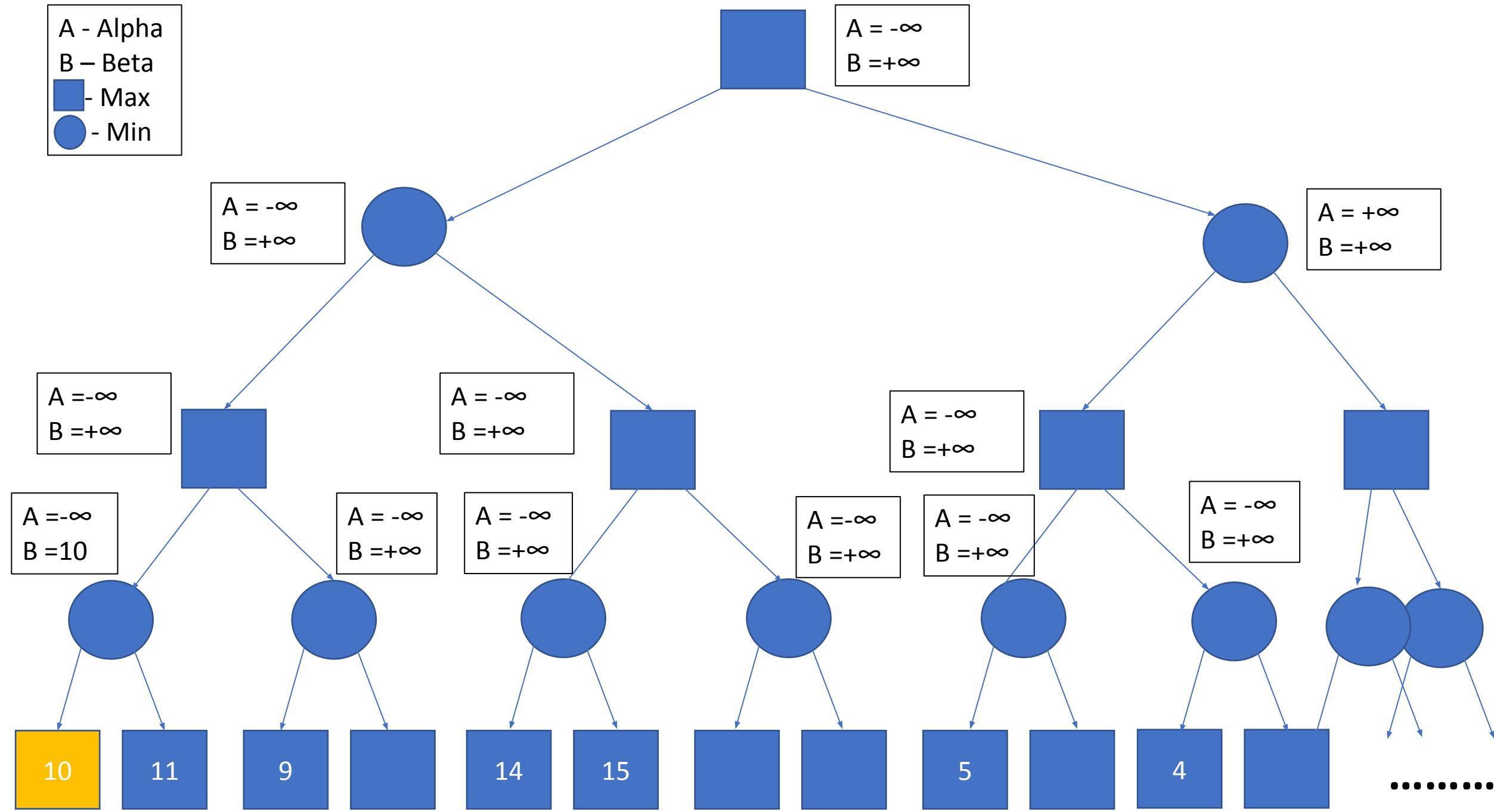




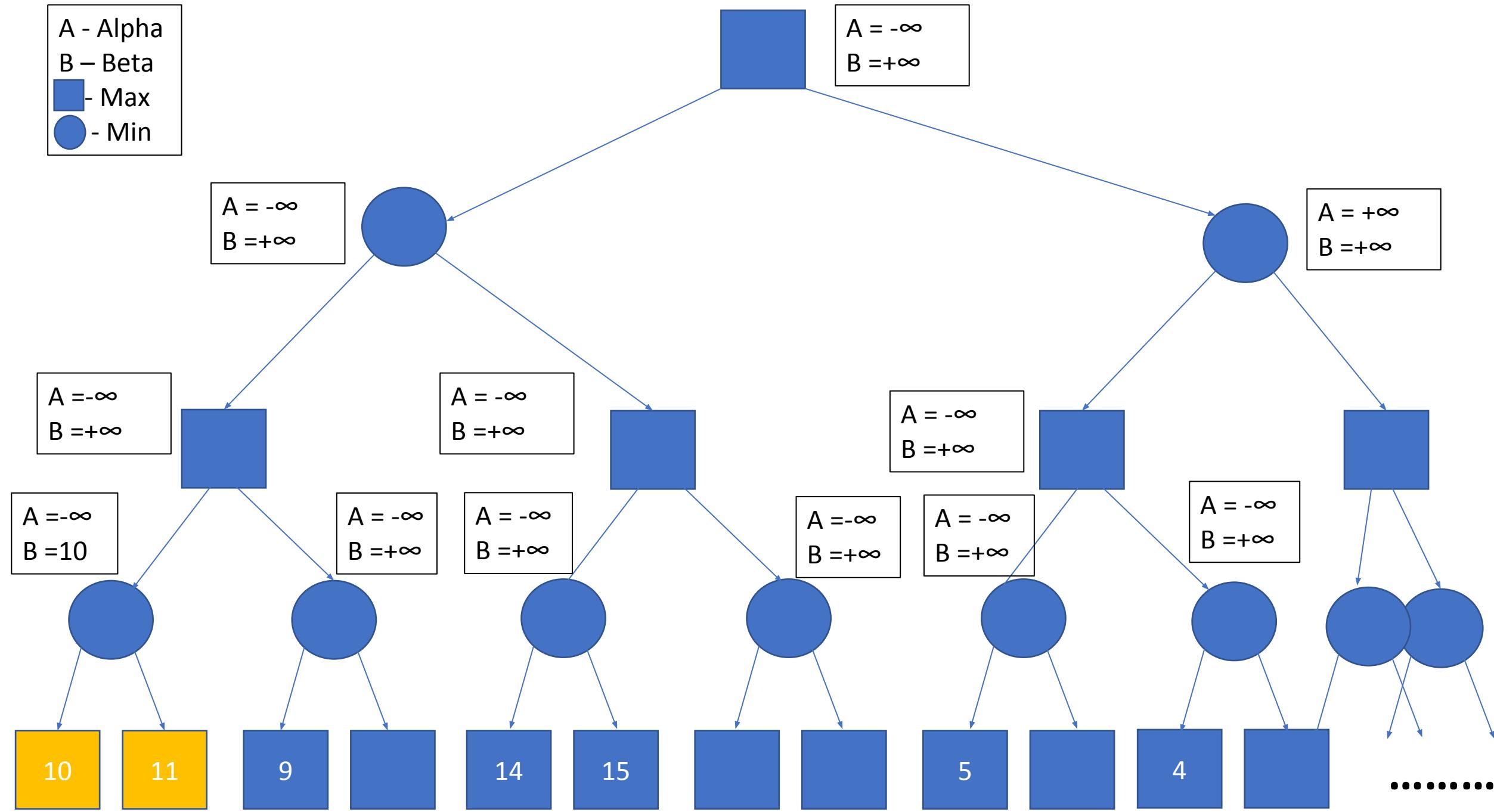
A - Alpha
B - Beta
■ - Max
● - Min



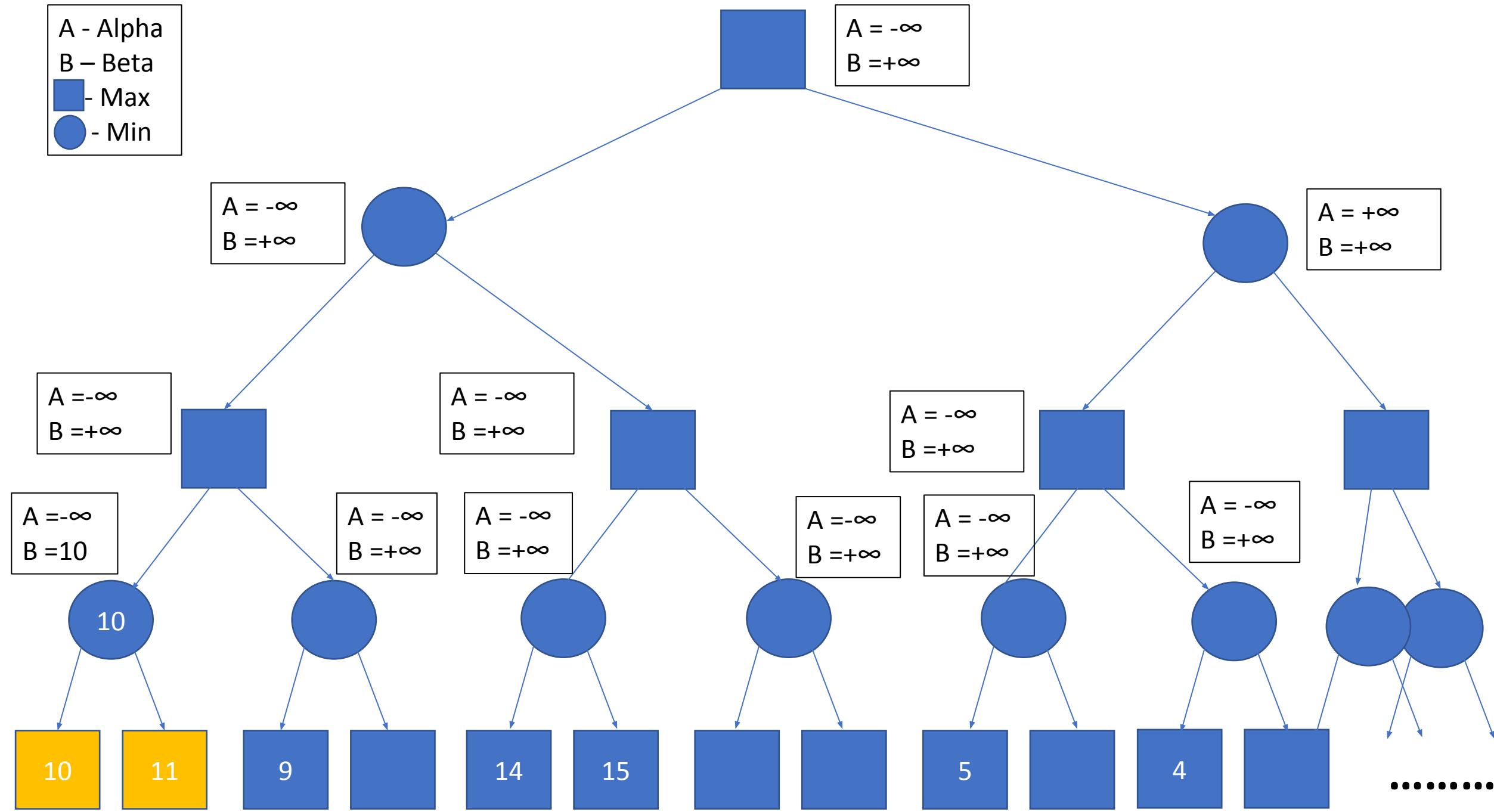
A - Alpha
B - Beta
■ - Max
● - Min



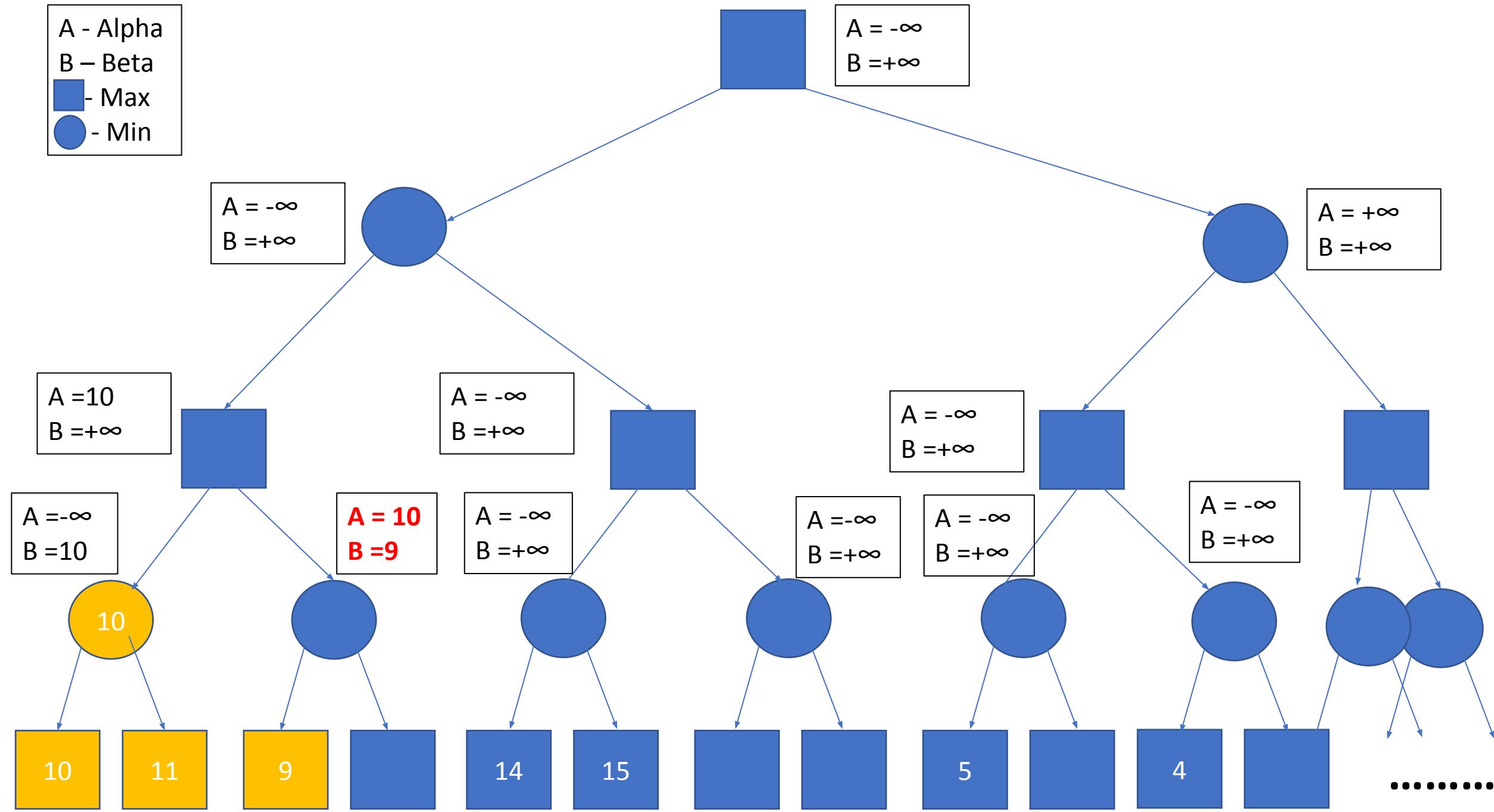
A - Alpha
B - Beta
■ - Max
● - Min



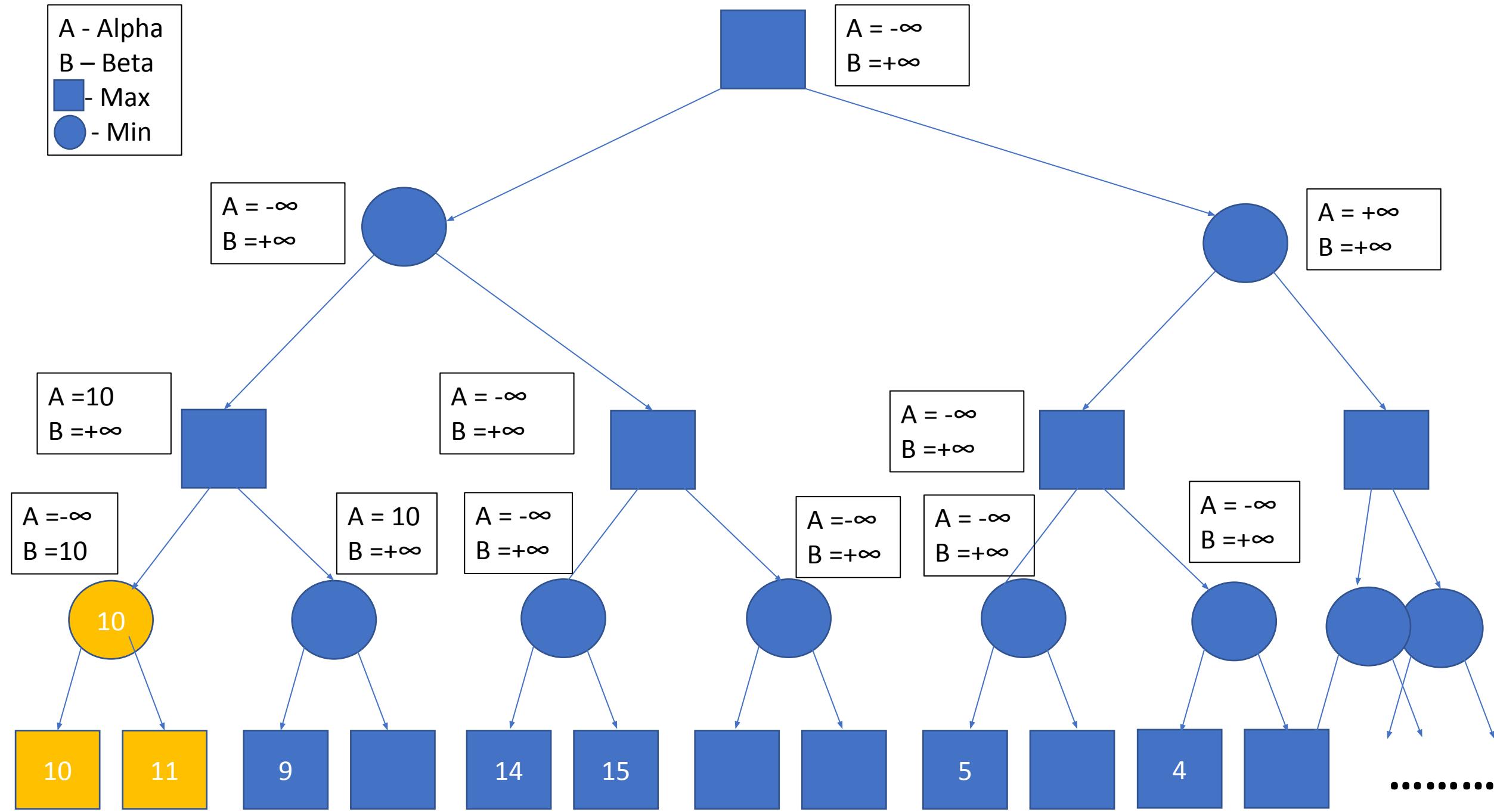
A - Alpha
B - Beta
■ - Max
● - Min



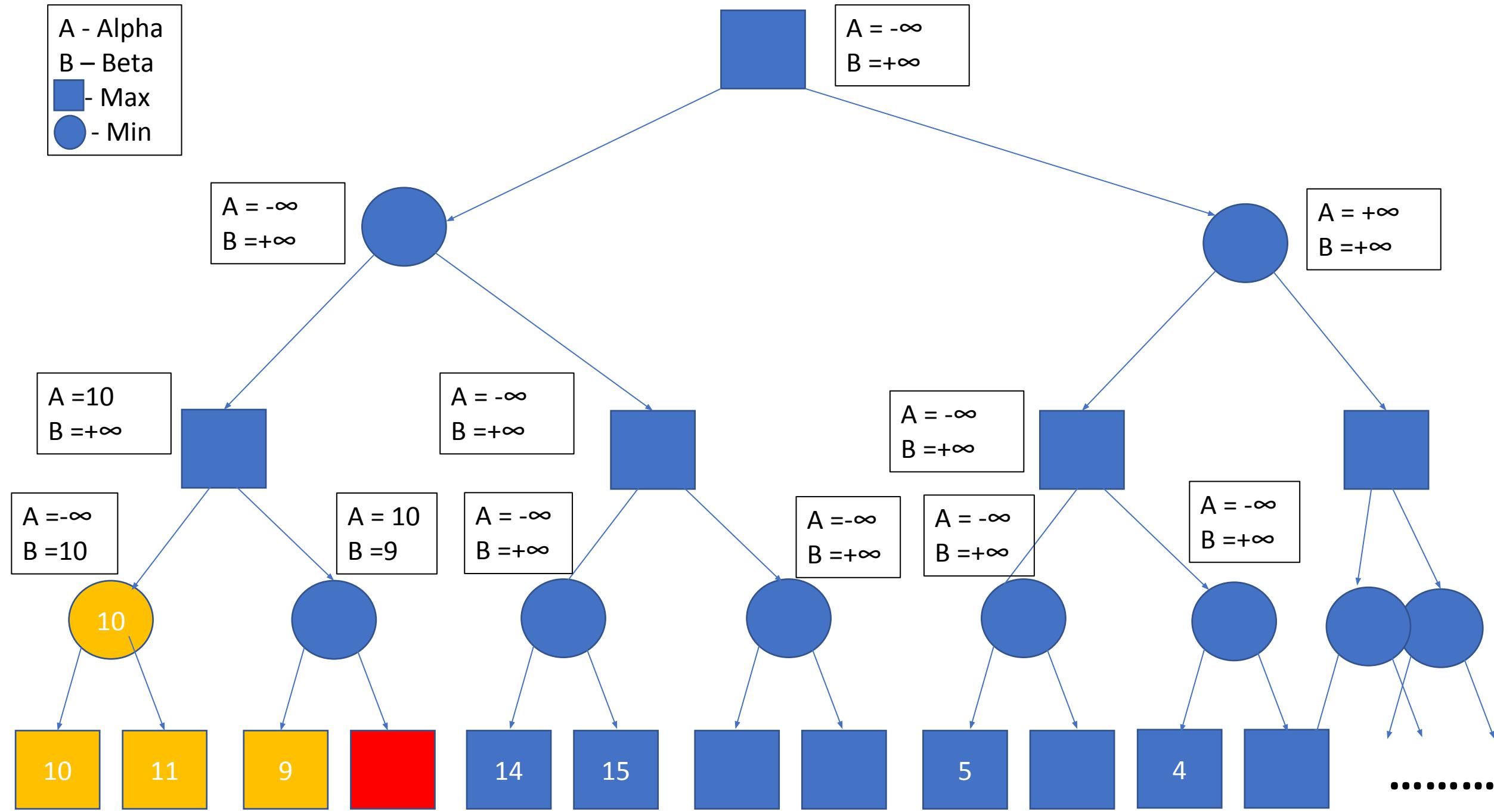
A - Alpha
B - Beta
■ - Max
● - Min



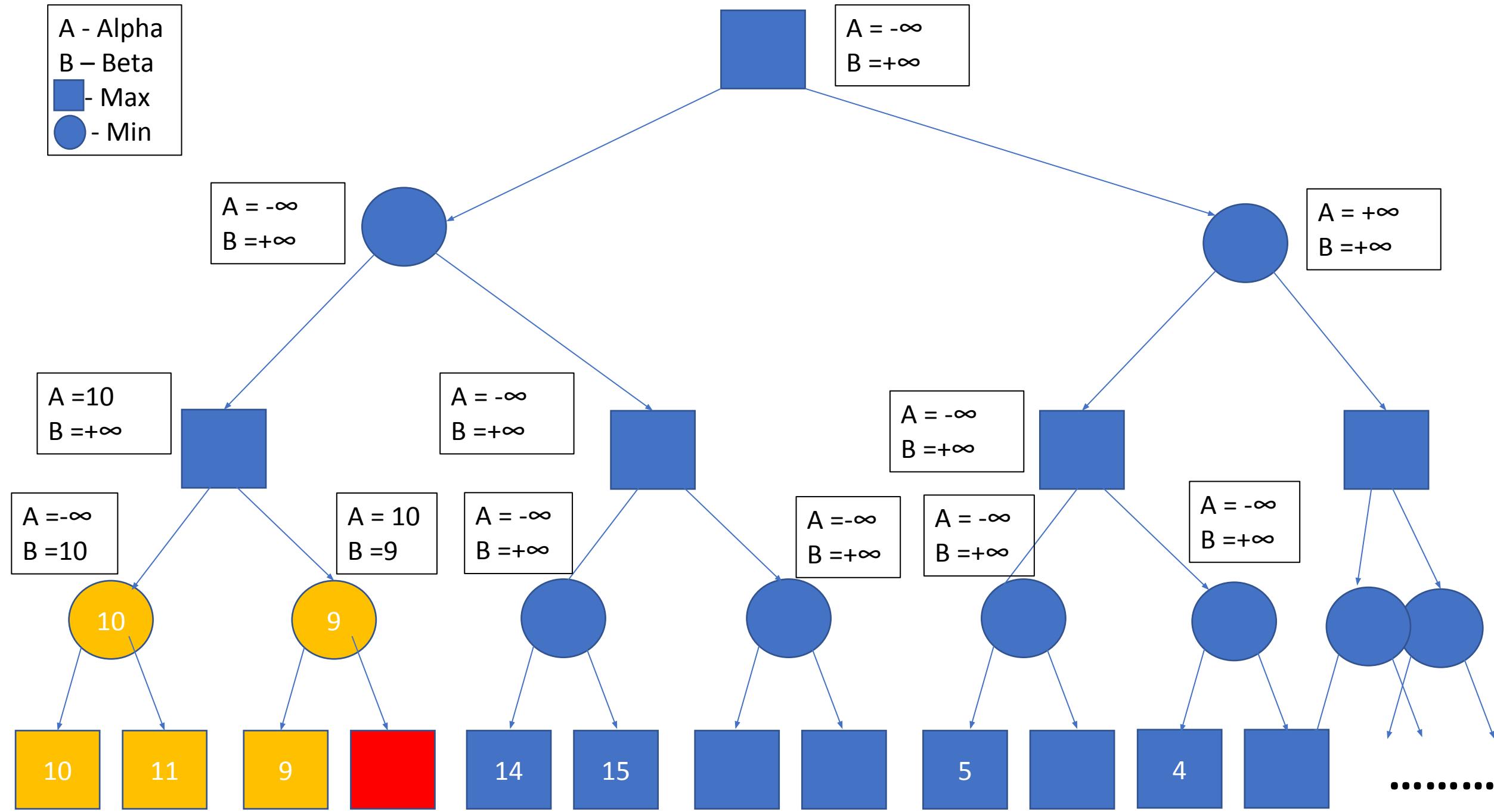
A - Alpha
B - Beta
■ - Max
● - Min



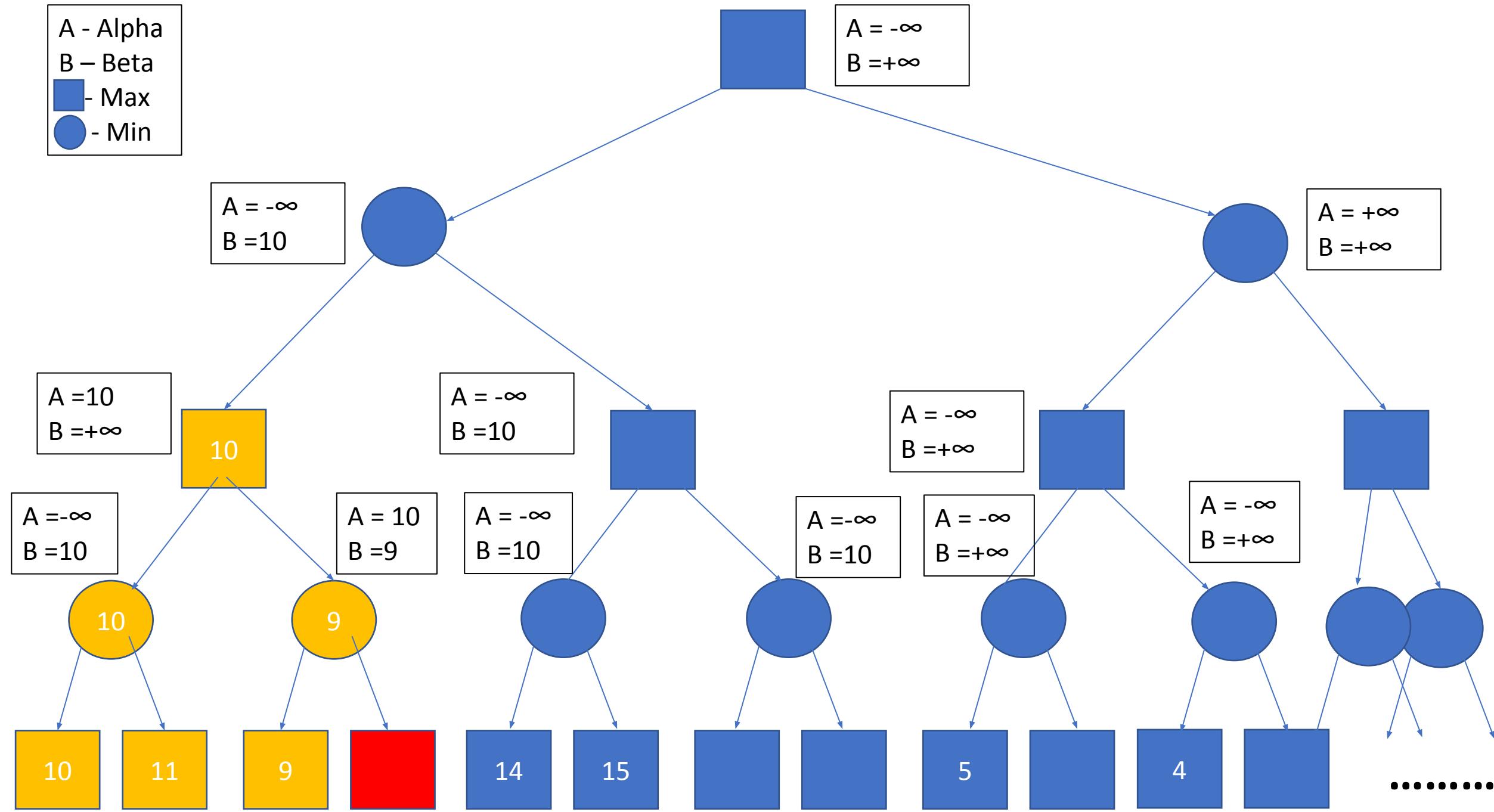
A - Alpha
B - Beta
■ - Max
● - Min



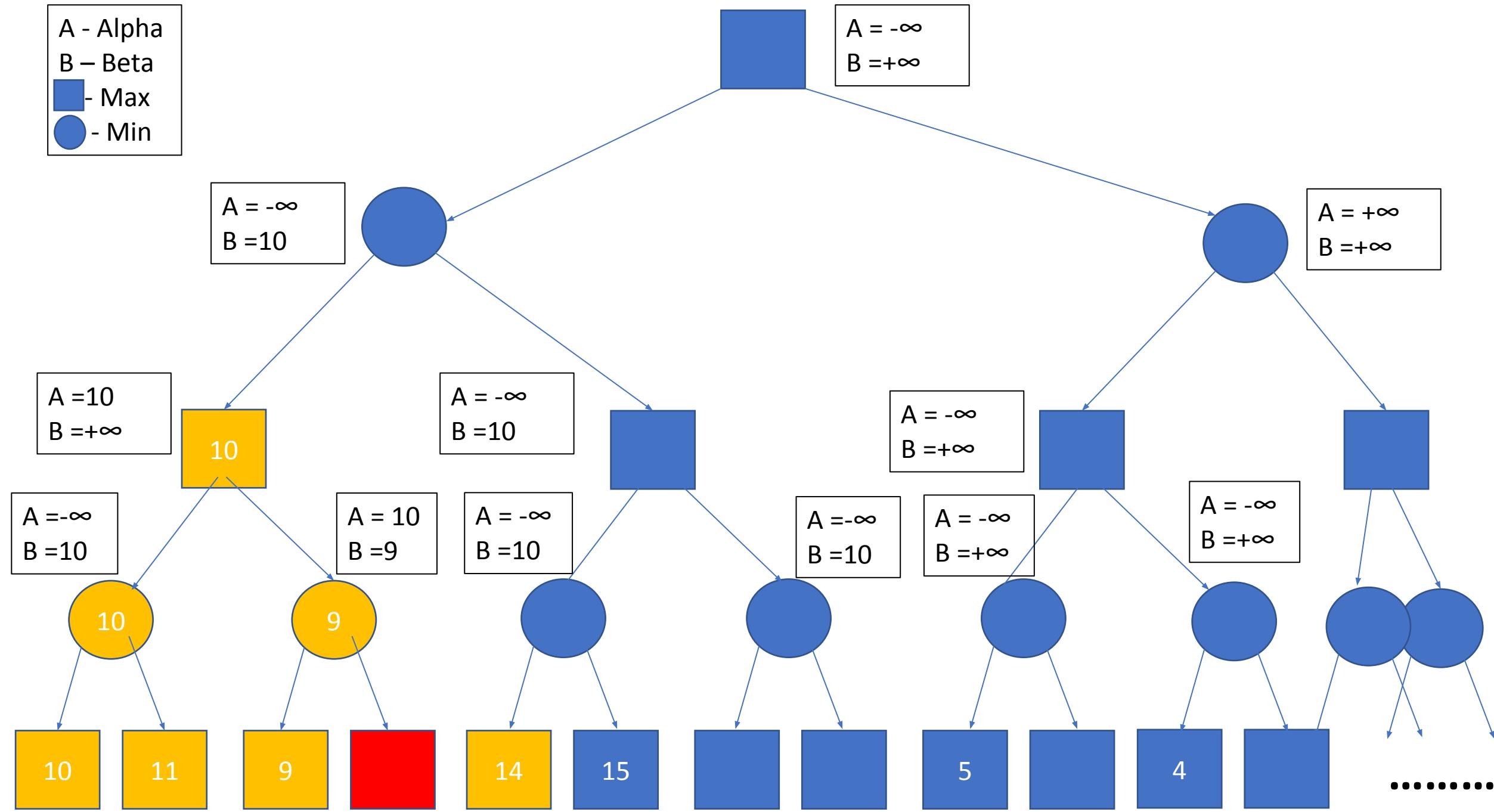
A - Alpha
B - Beta
■ - Max
● - Min



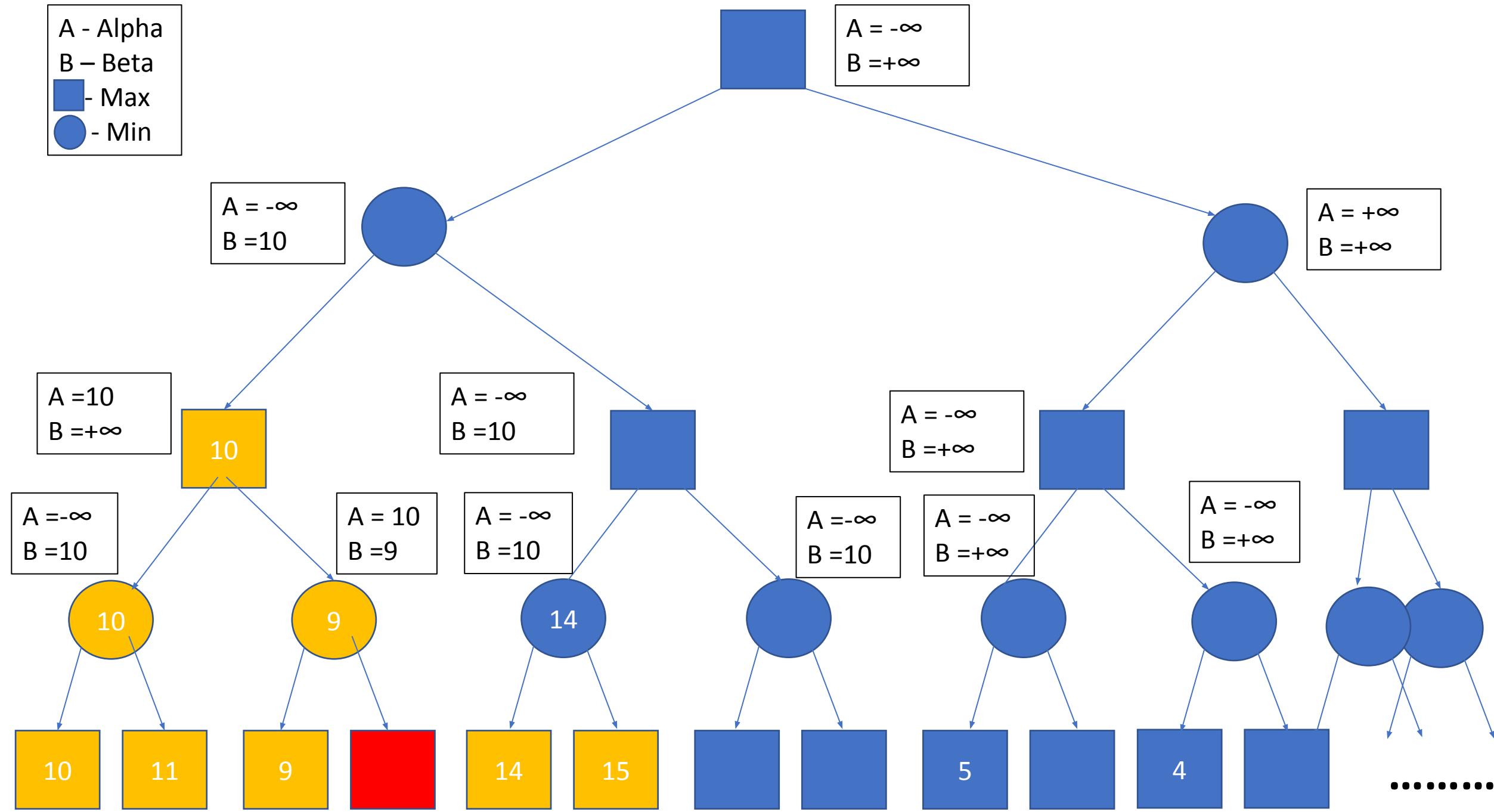
A - Alpha
B - Beta
■ - Max
● - Min



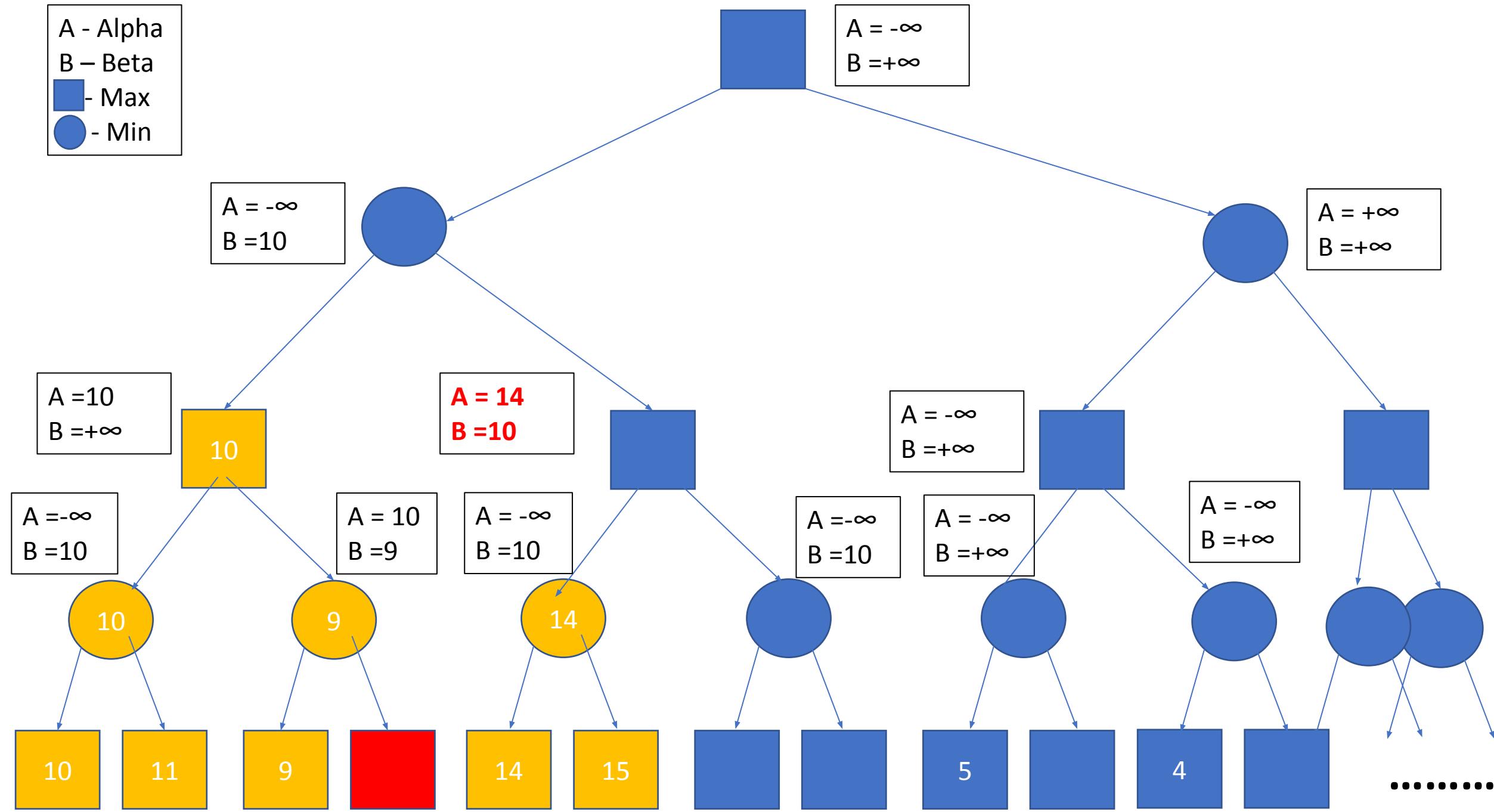
A - Alpha
B - Beta
■ - Max
● - Min



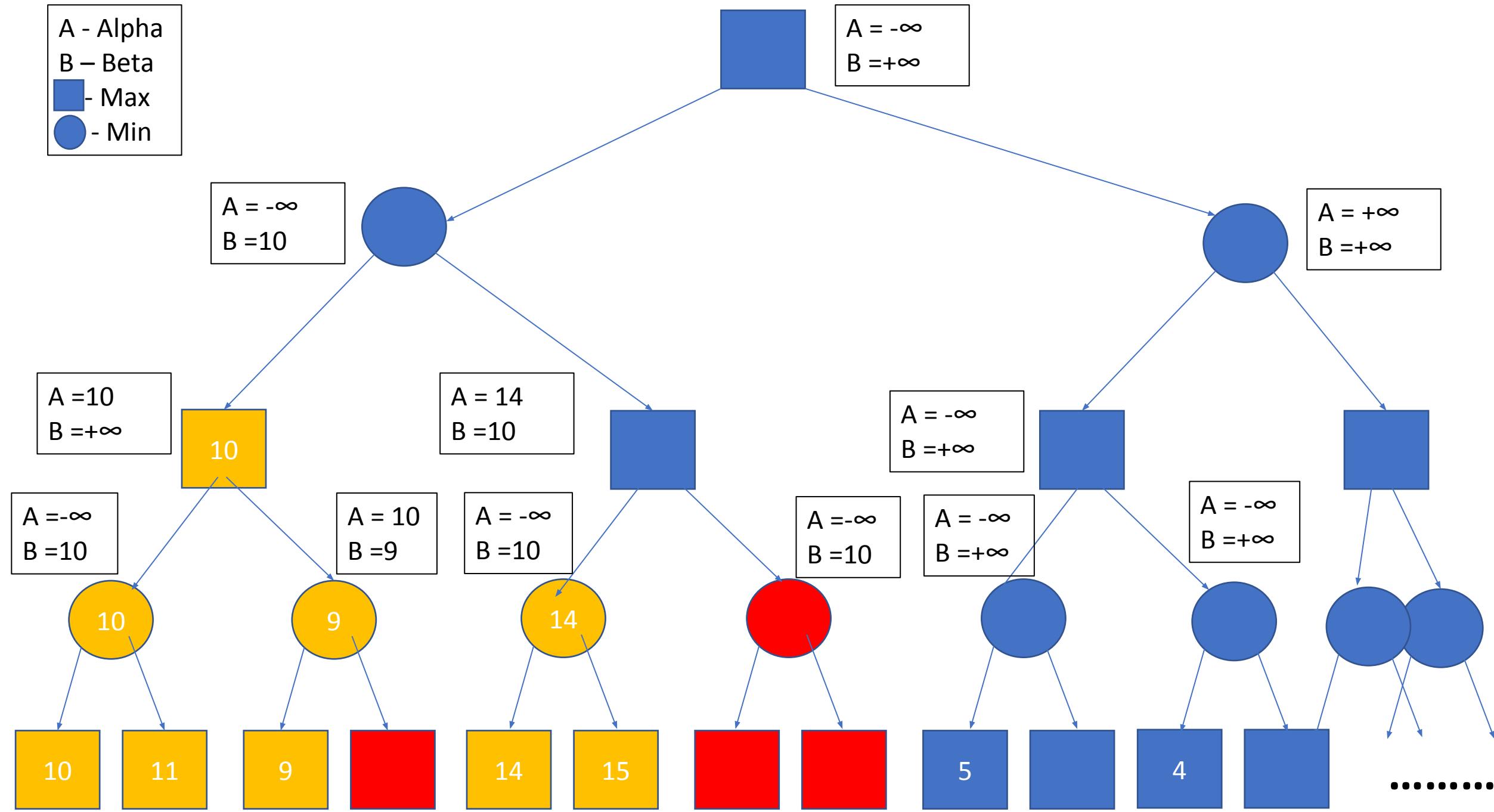
A - Alpha
B - Beta
■ - Max
● - Min



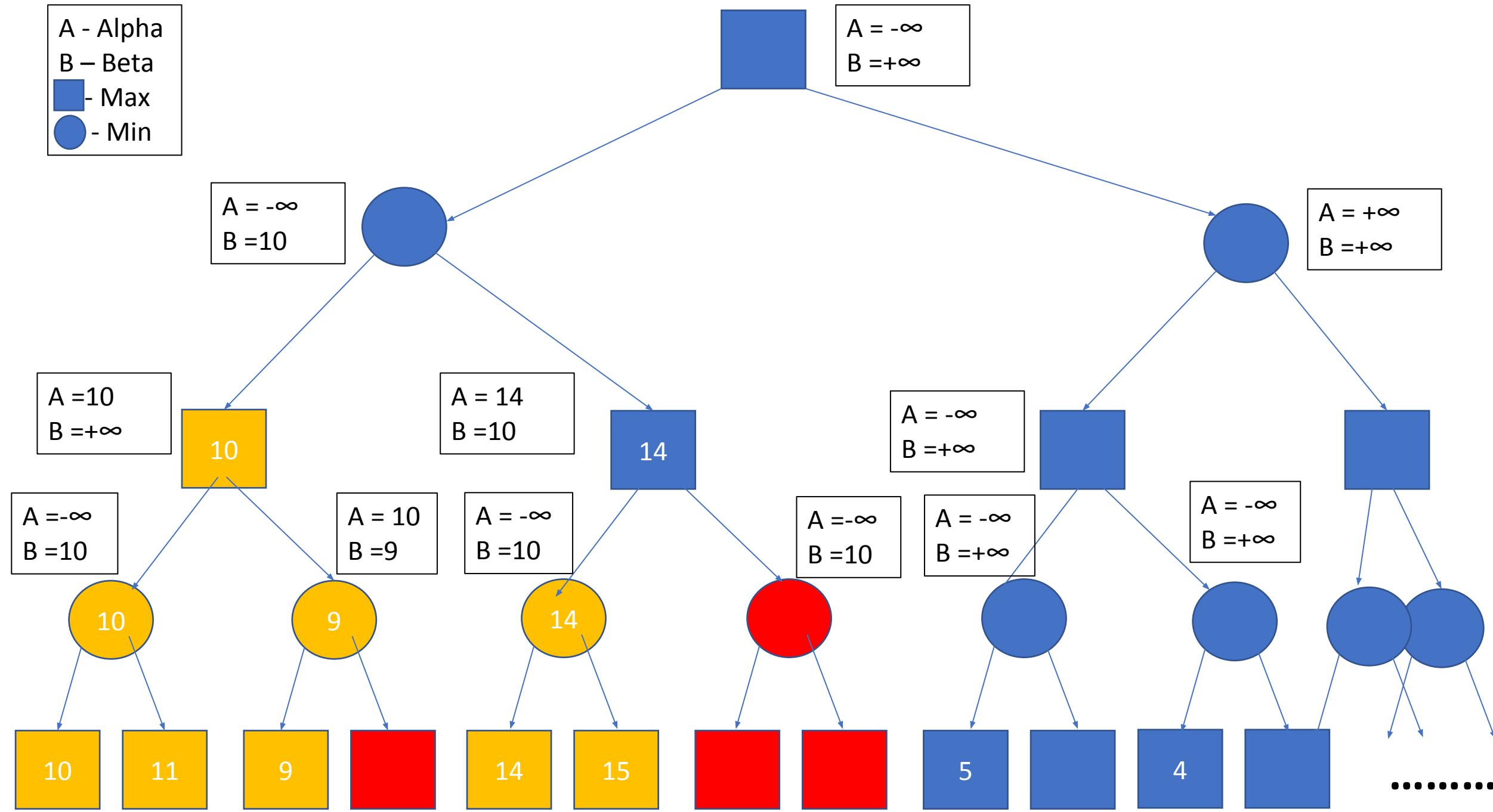
A - Alpha
B - Beta
■ - Max
● - Min



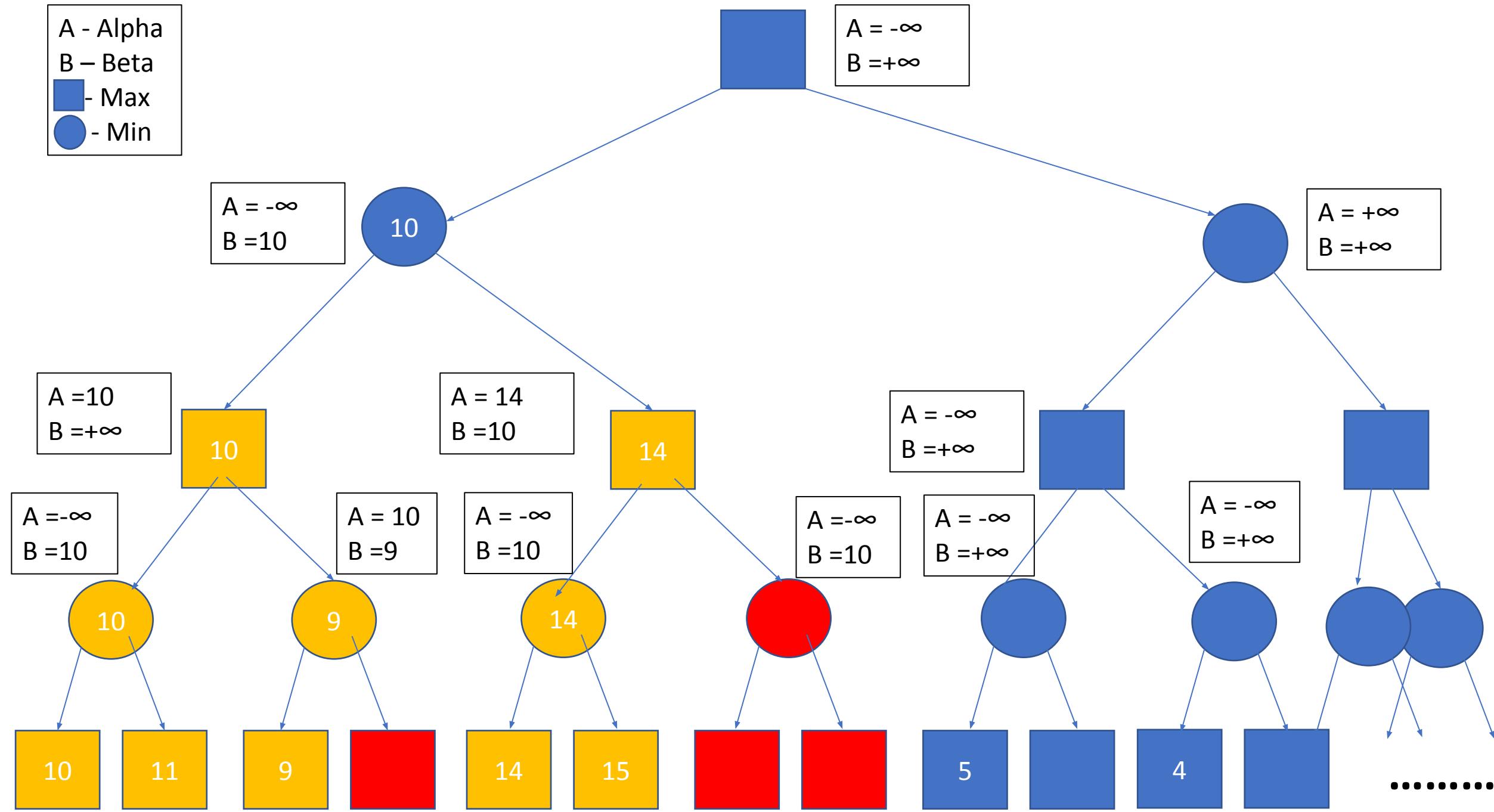
A - Alpha
B - Beta
■ - Max
● - Min

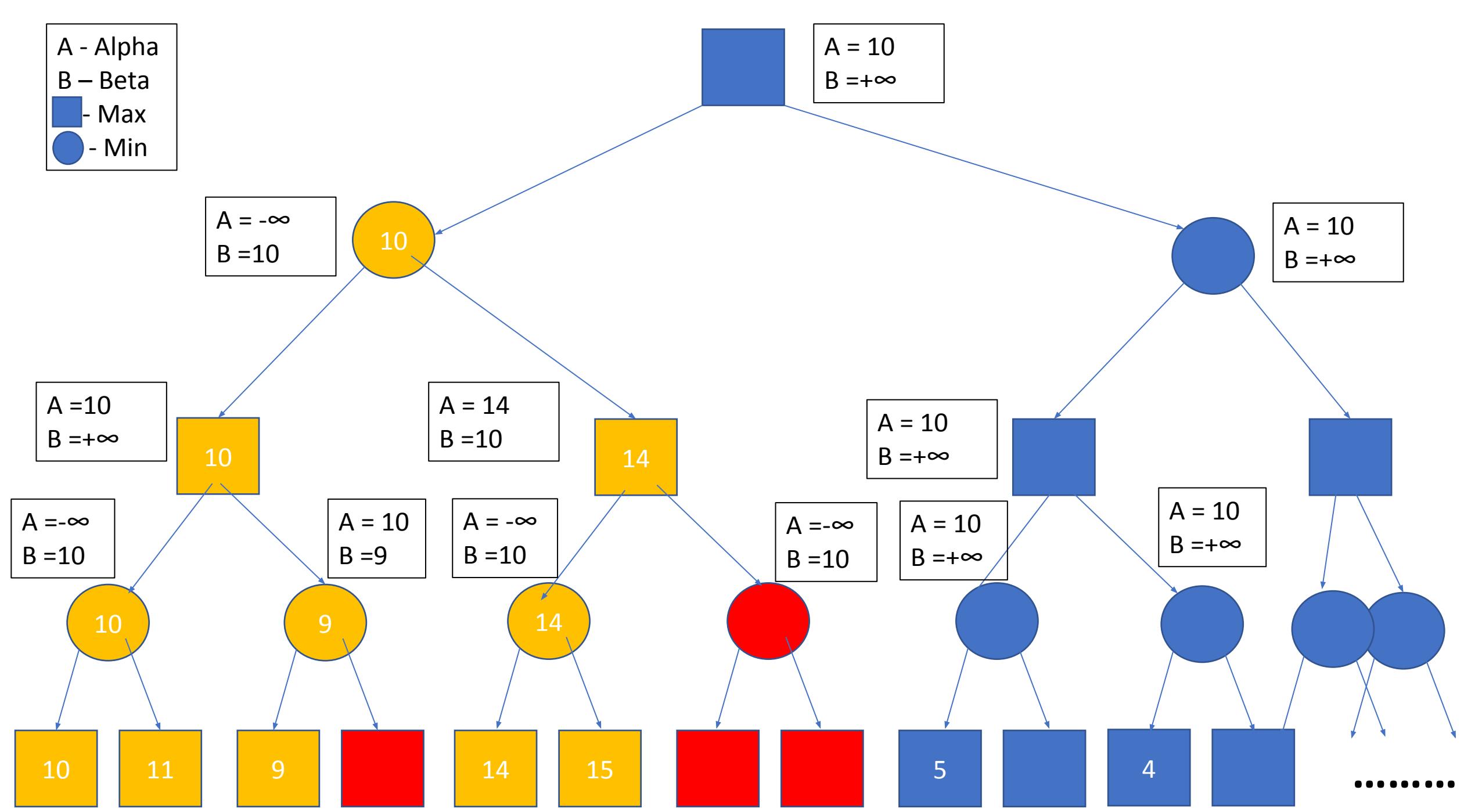


A - Alpha
B - Beta
- Max
- Min

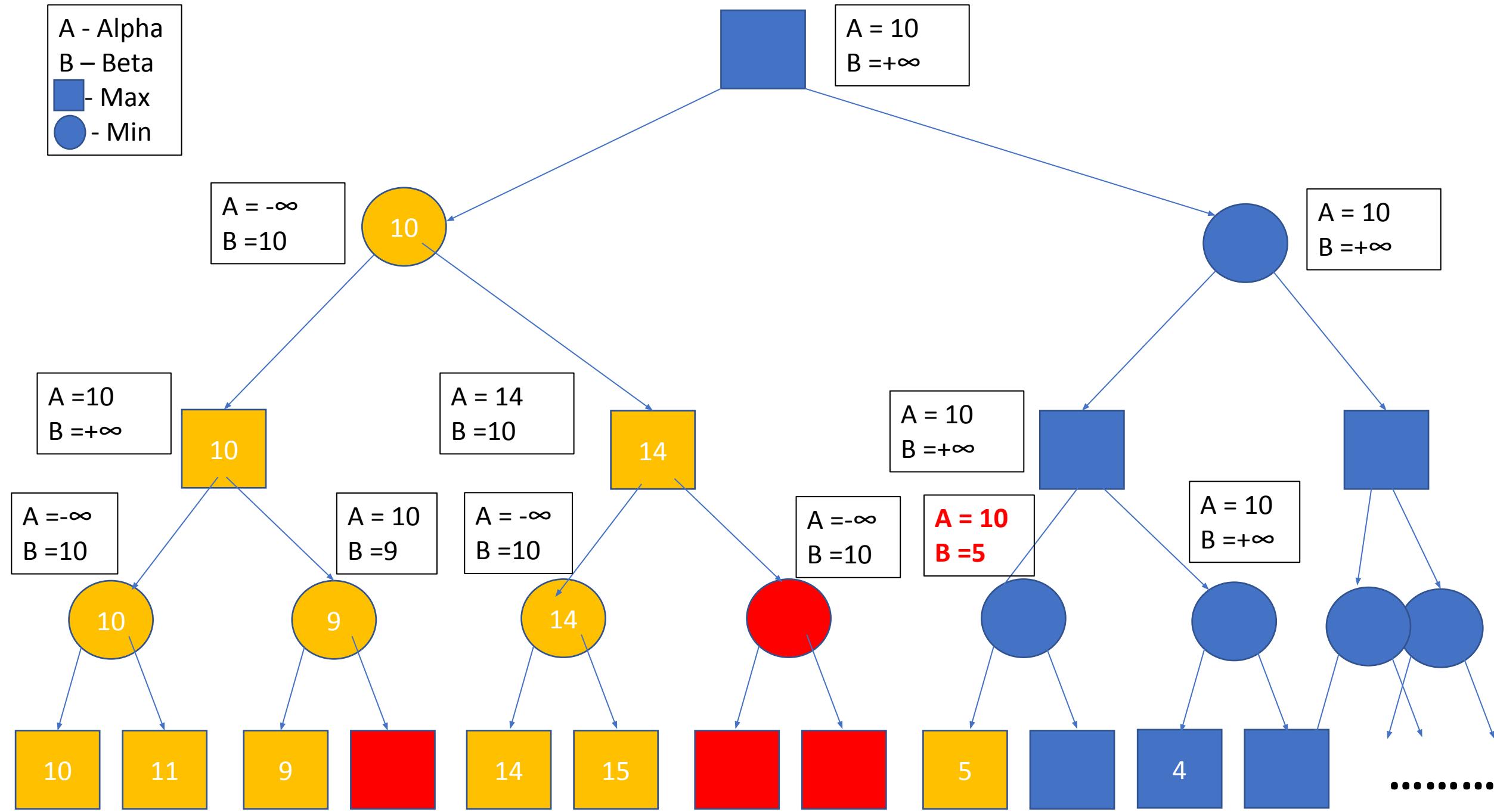


A - Alpha
B - Beta
- Max
- Min

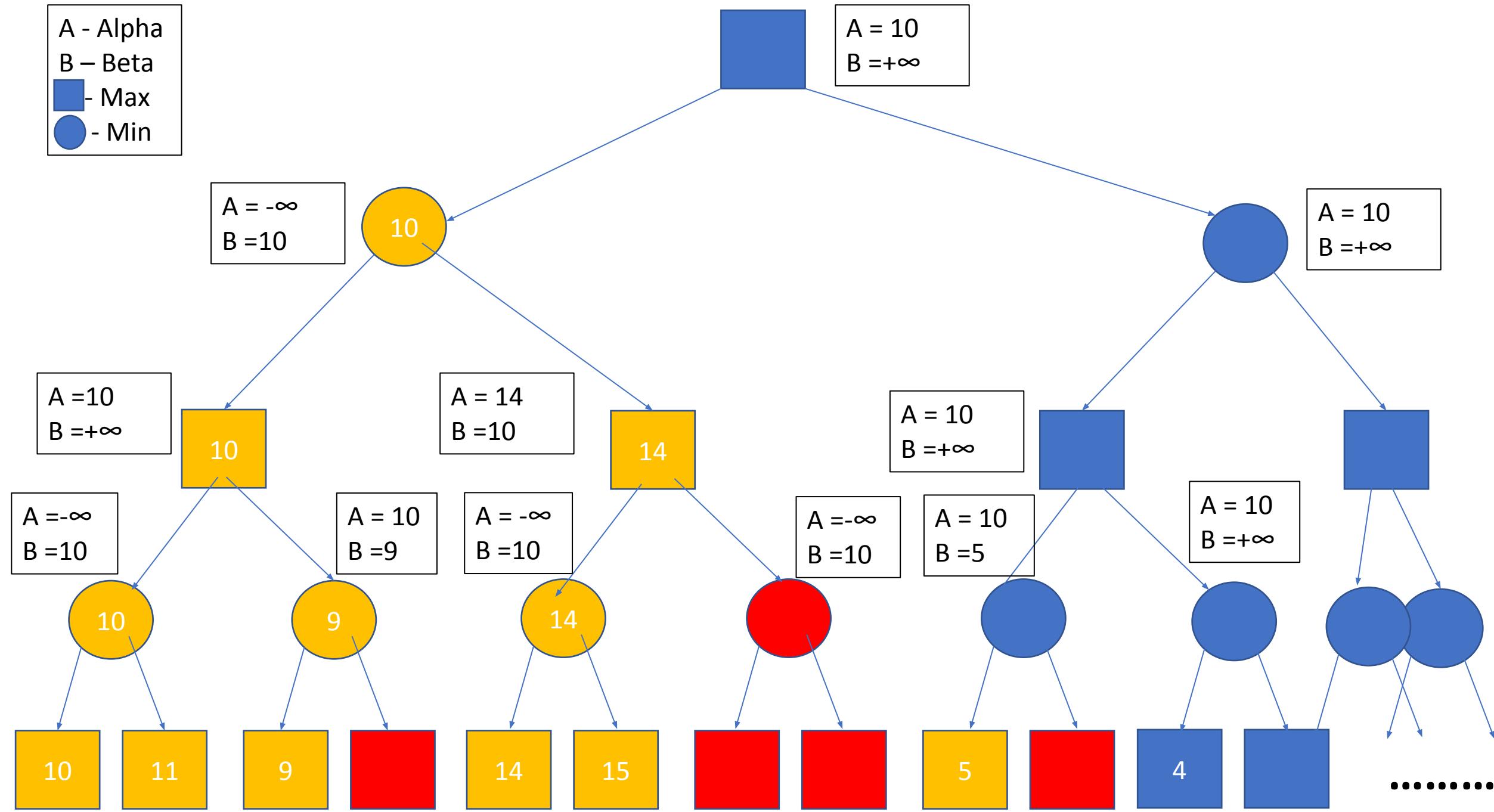




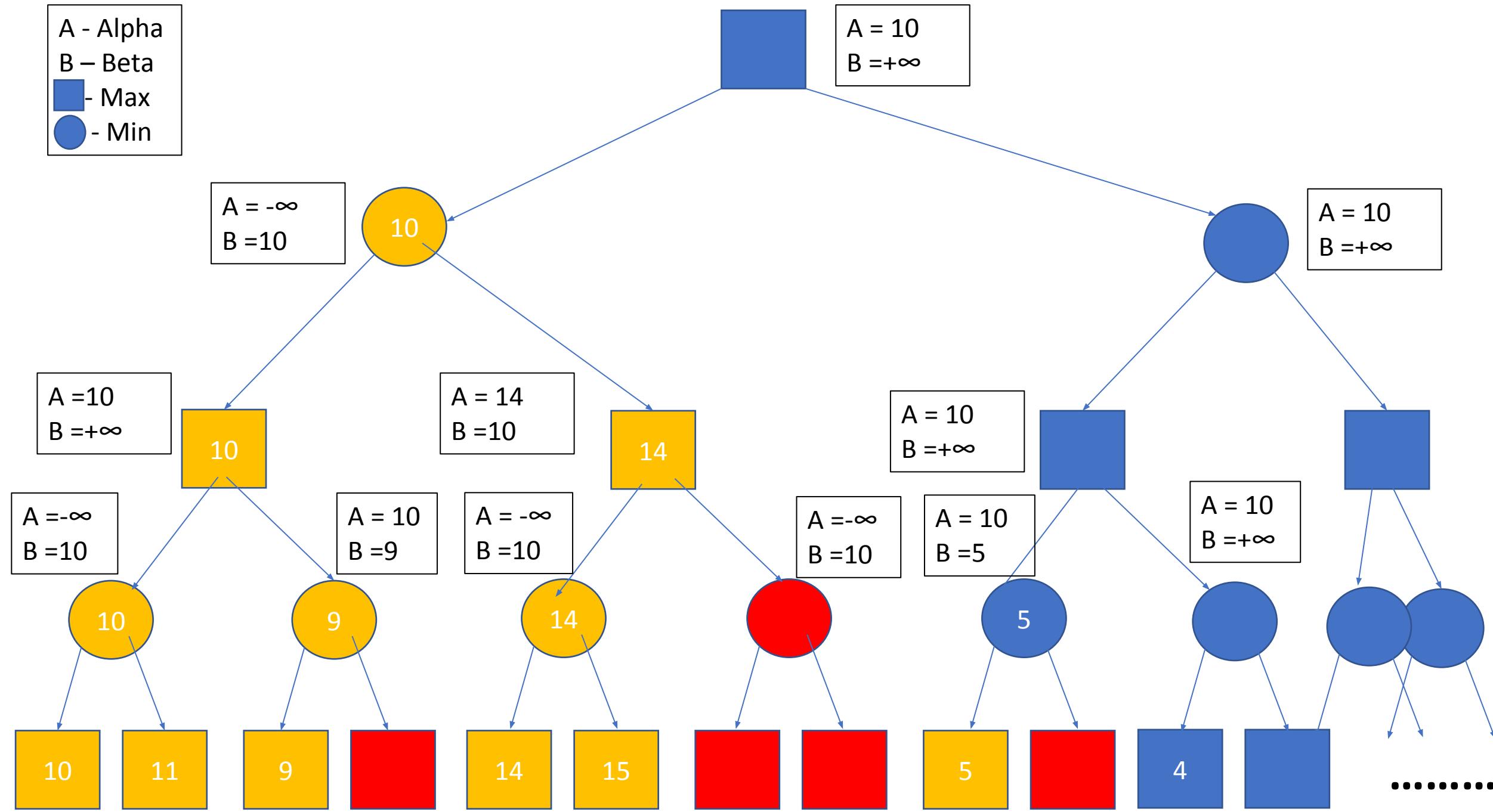
A - Alpha
B - Beta
- Max
- Min



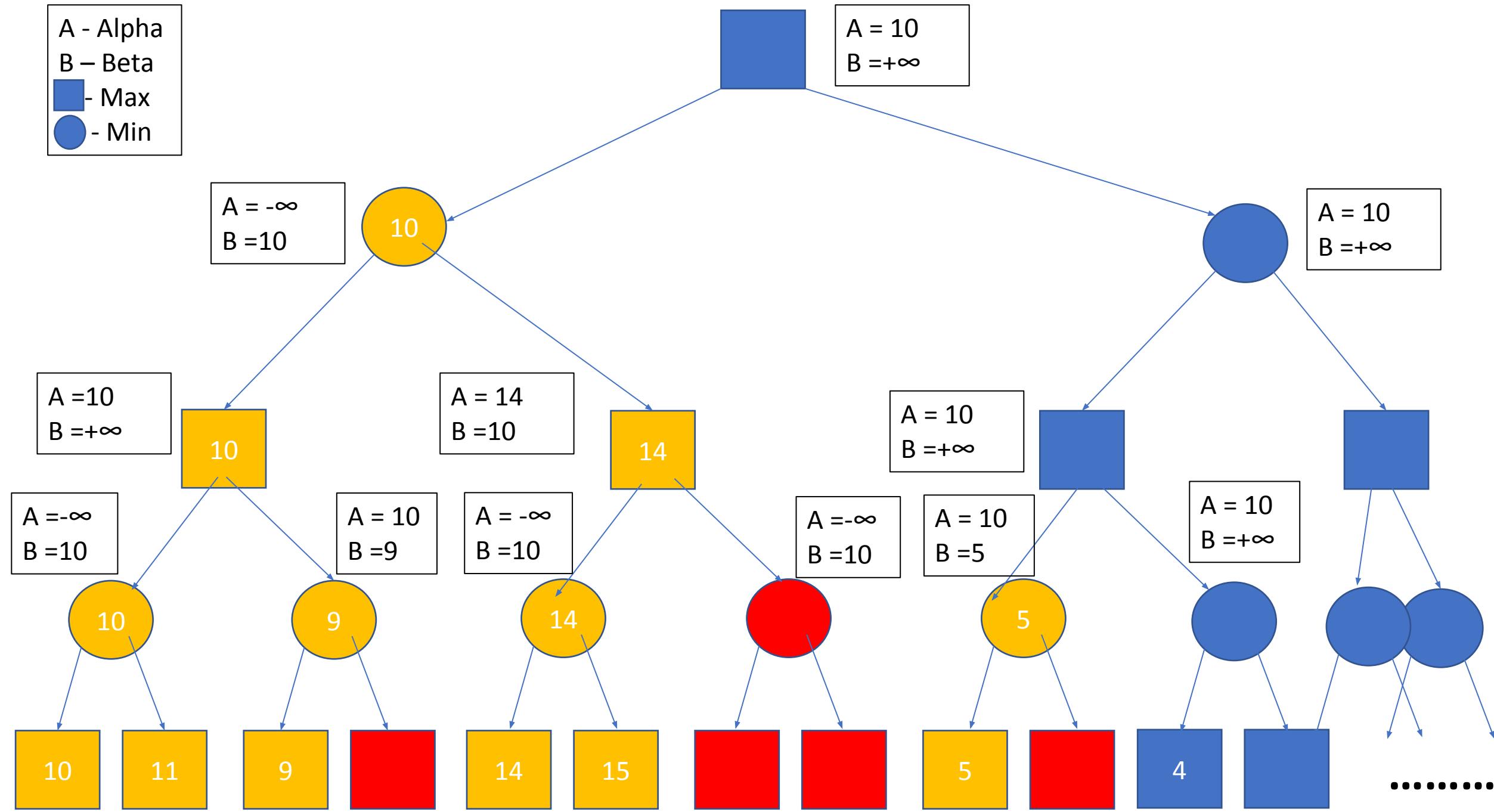
A - Alpha
B - Beta
- Max
- Min



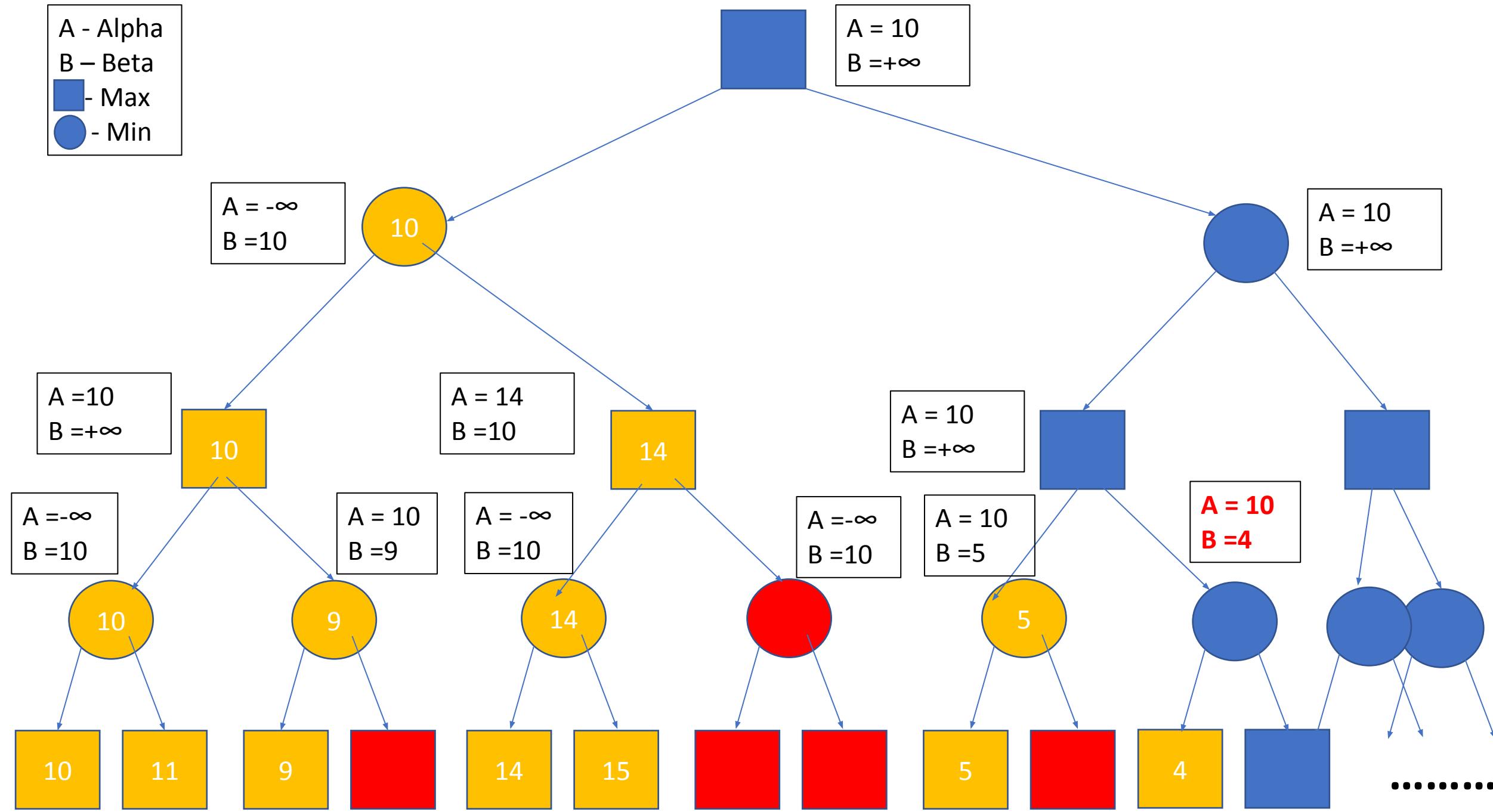
A - Alpha
B - Beta
- Max
- Min



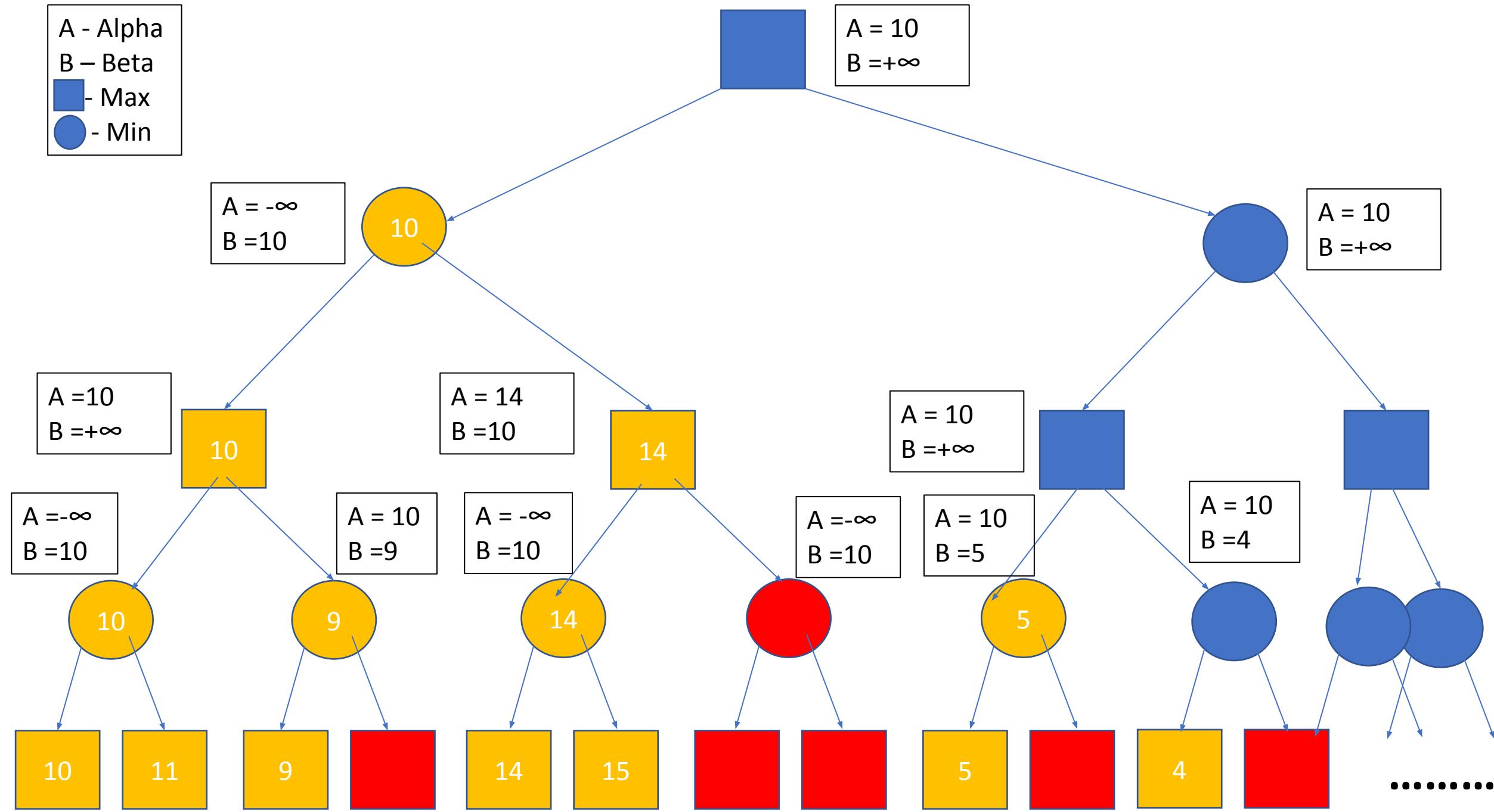
A - Alpha
B - Beta
- Max
- Min



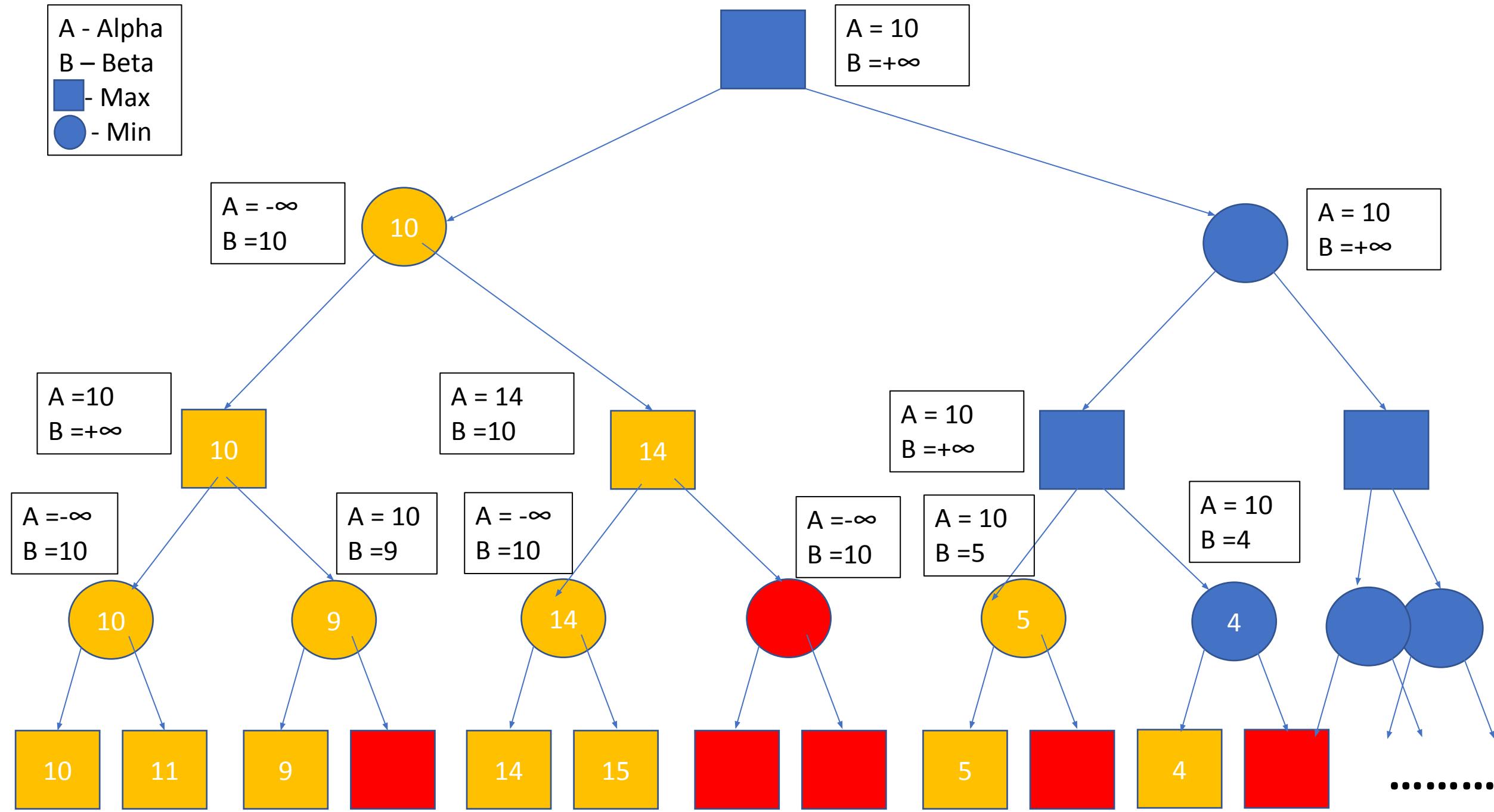
A - Alpha
B - Beta
— Max
● - Min



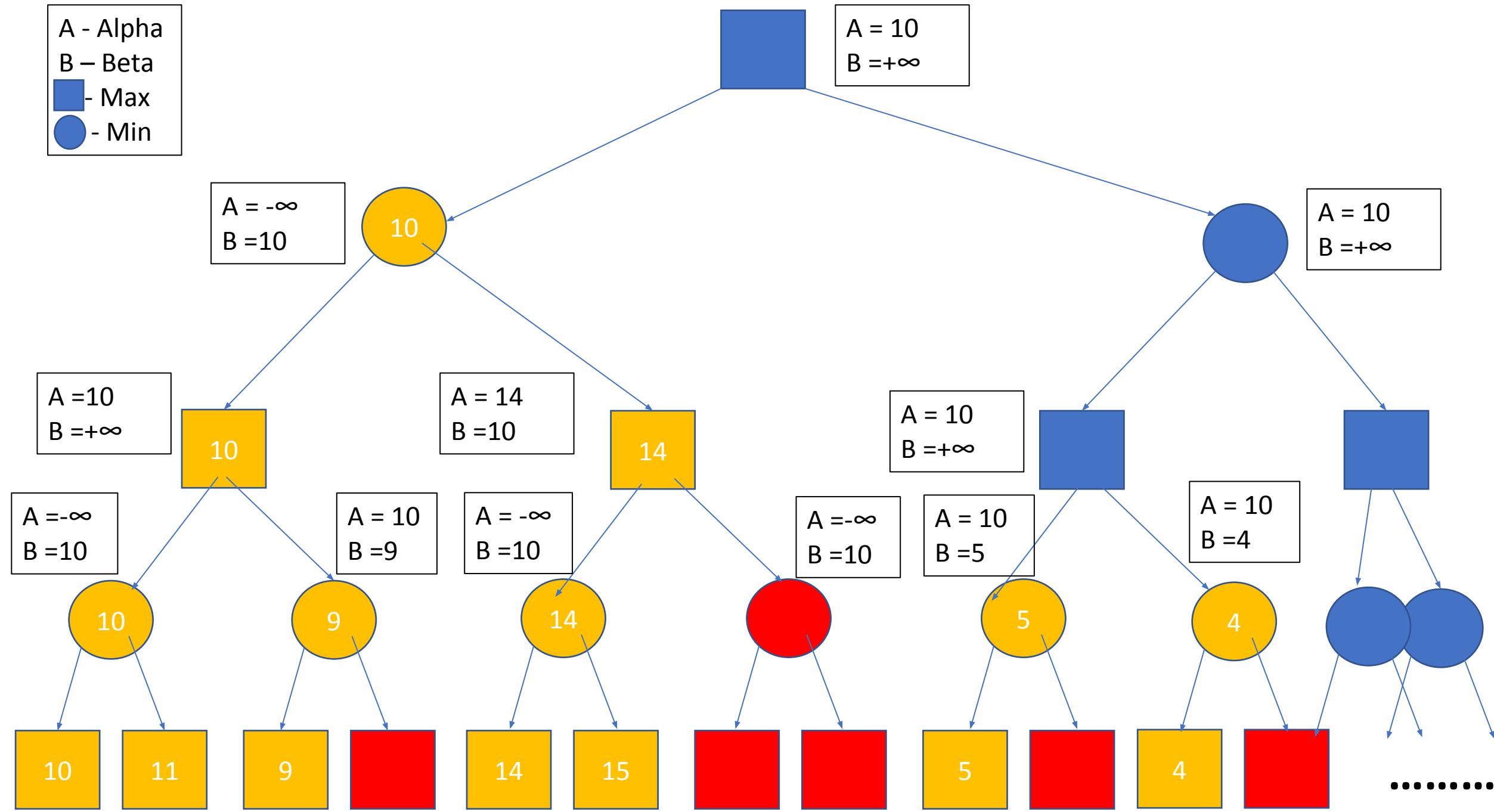
A - Alpha
B - Beta
— Max
● - Min



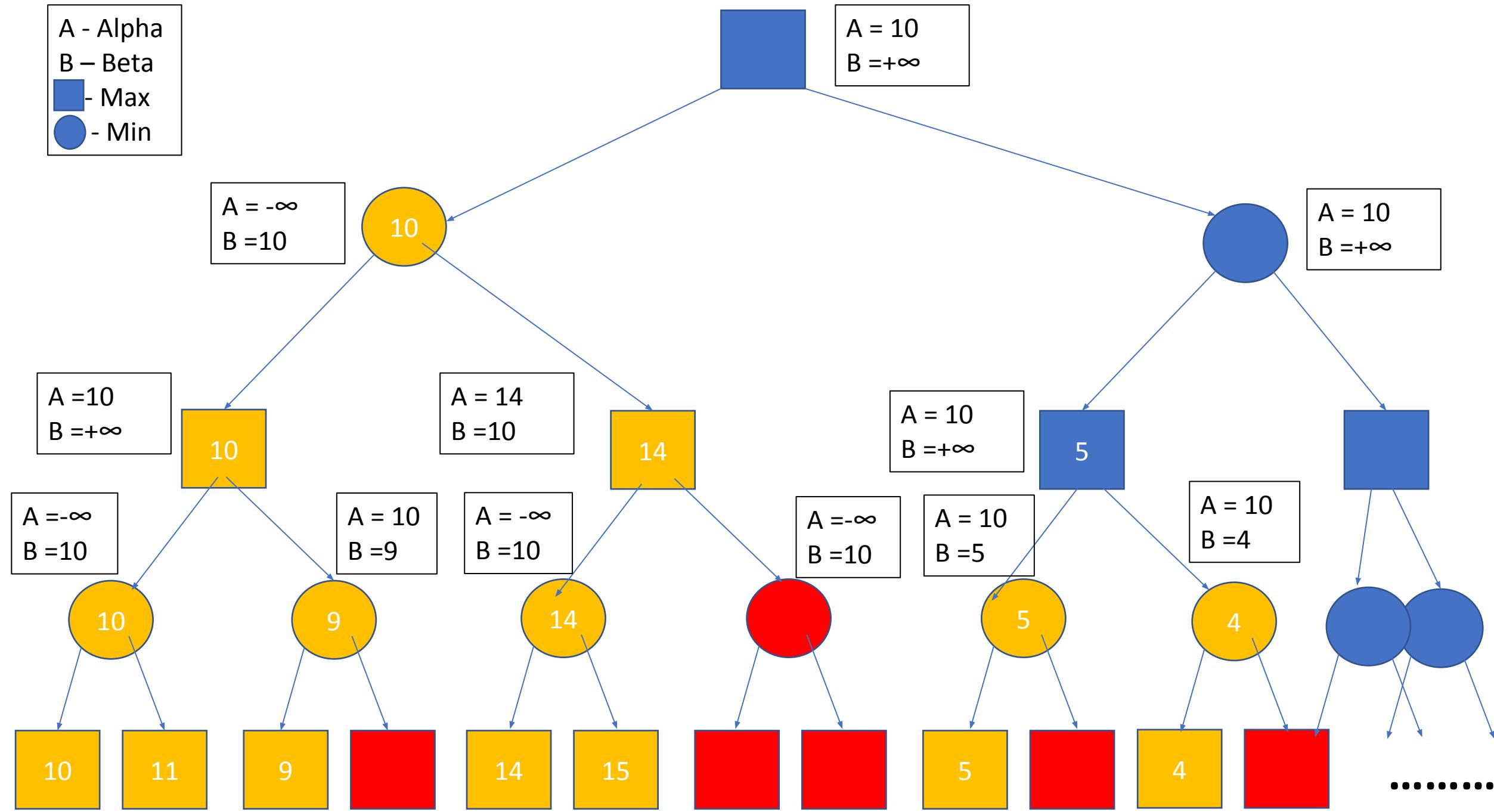
A - Alpha
B - Beta
— Max
● - Min



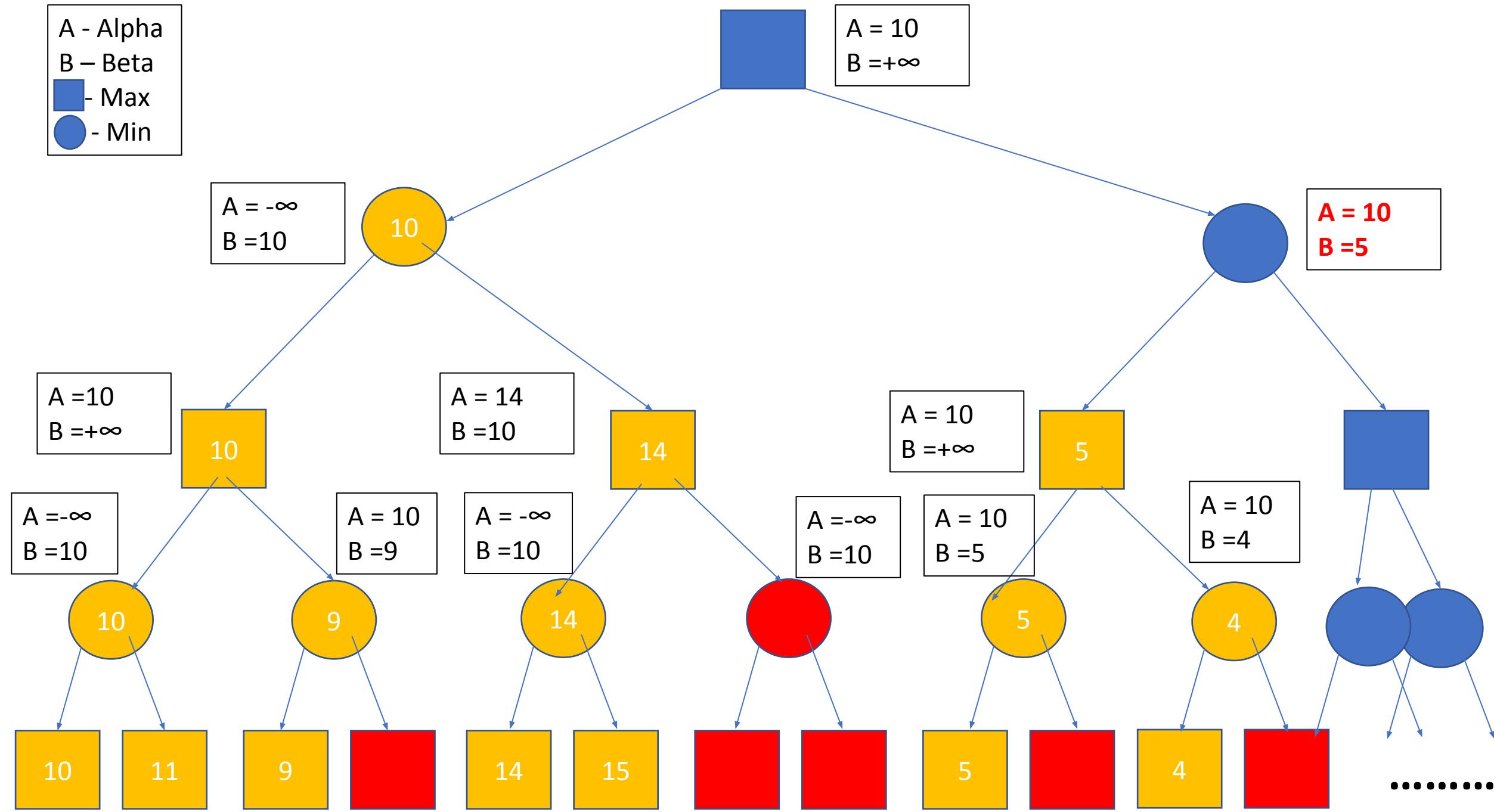
A - Alpha
B - Beta
— Max
● - Min



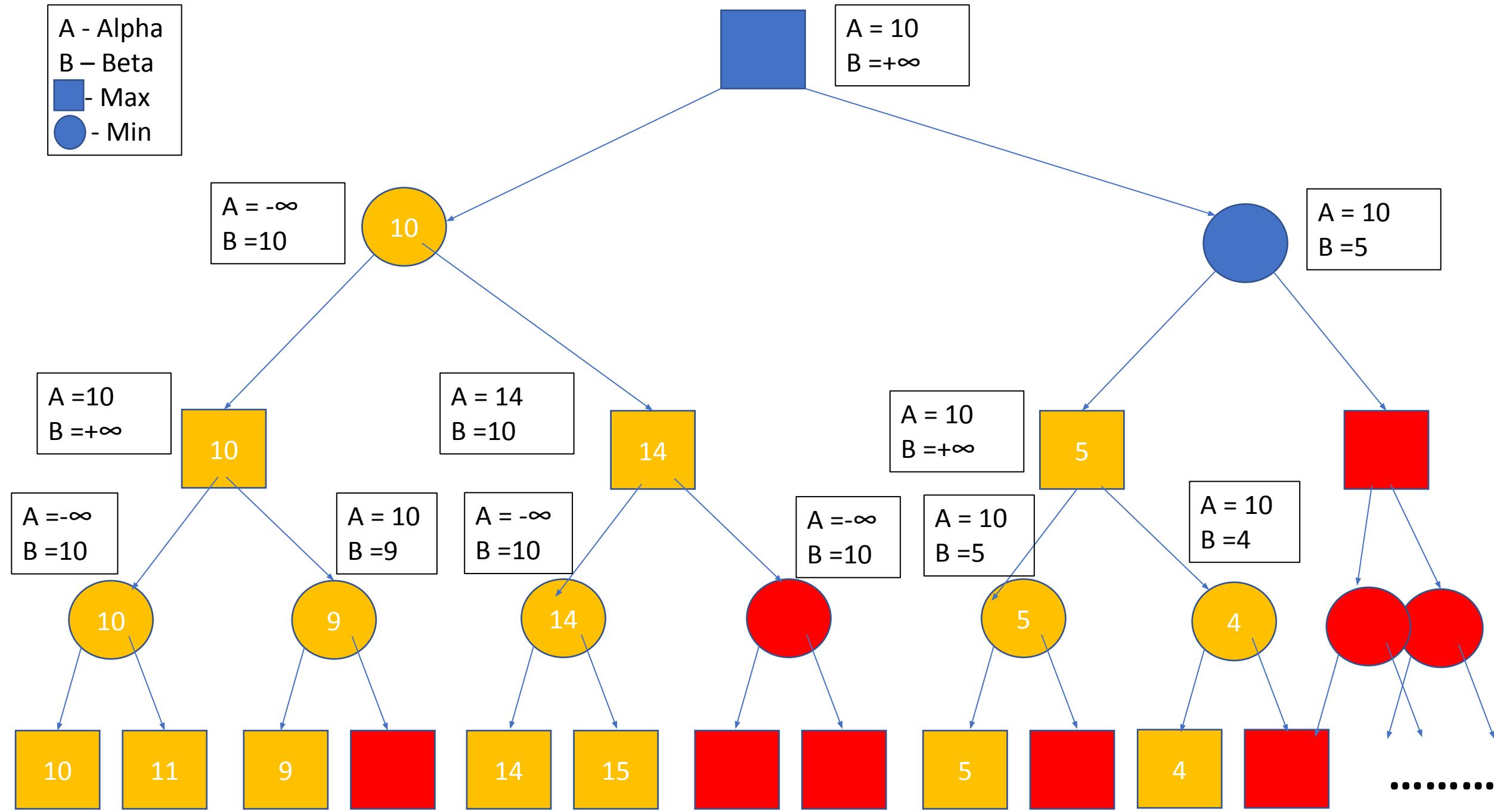
A - Alpha
B - Beta
— Max
● - Min



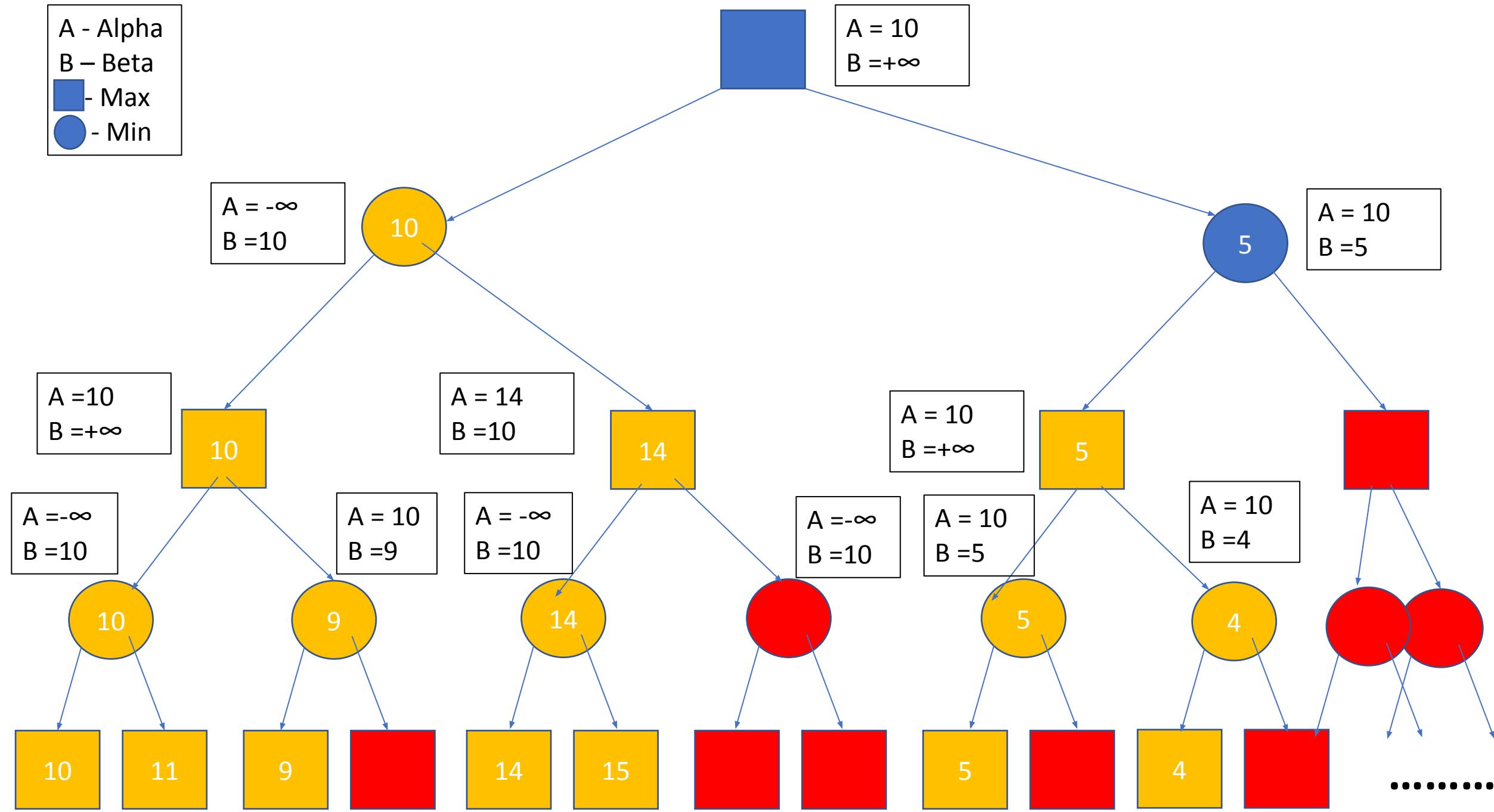
A - Alpha
B - Beta
— Max
● - Min



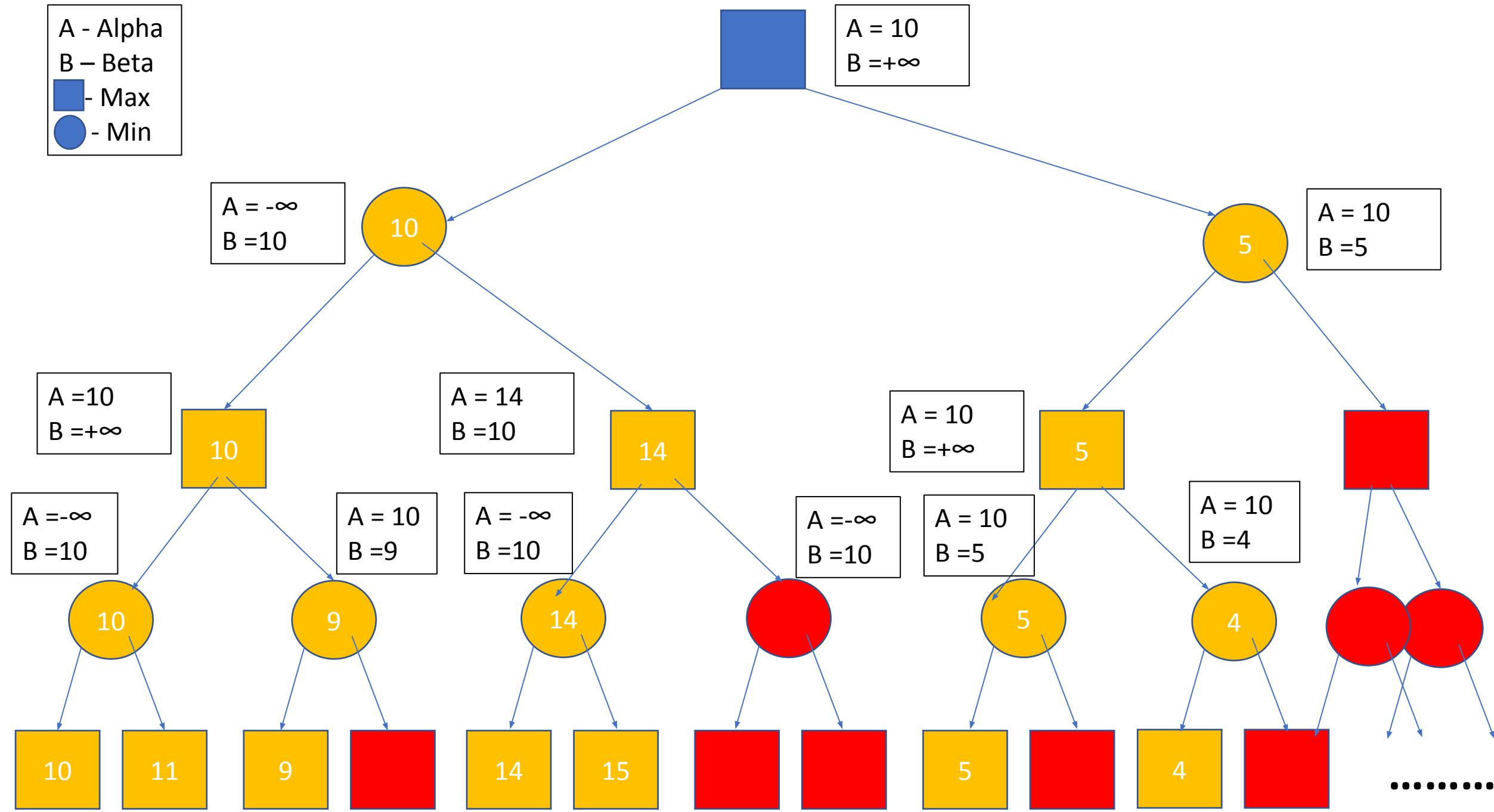
A - Alpha
B - Beta
— Max
● - Min



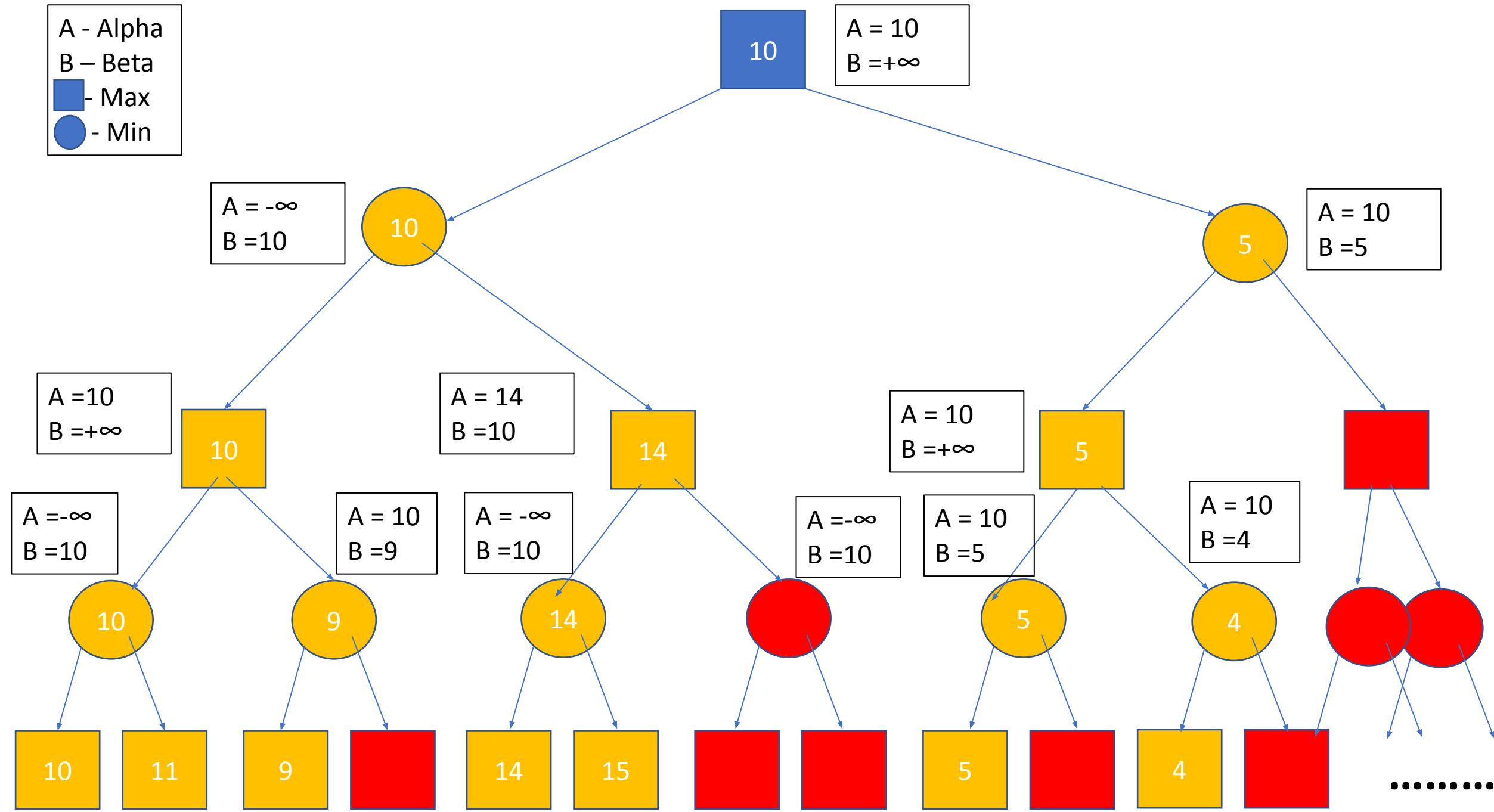
A - Alpha
B - Beta
— Max
● - Min



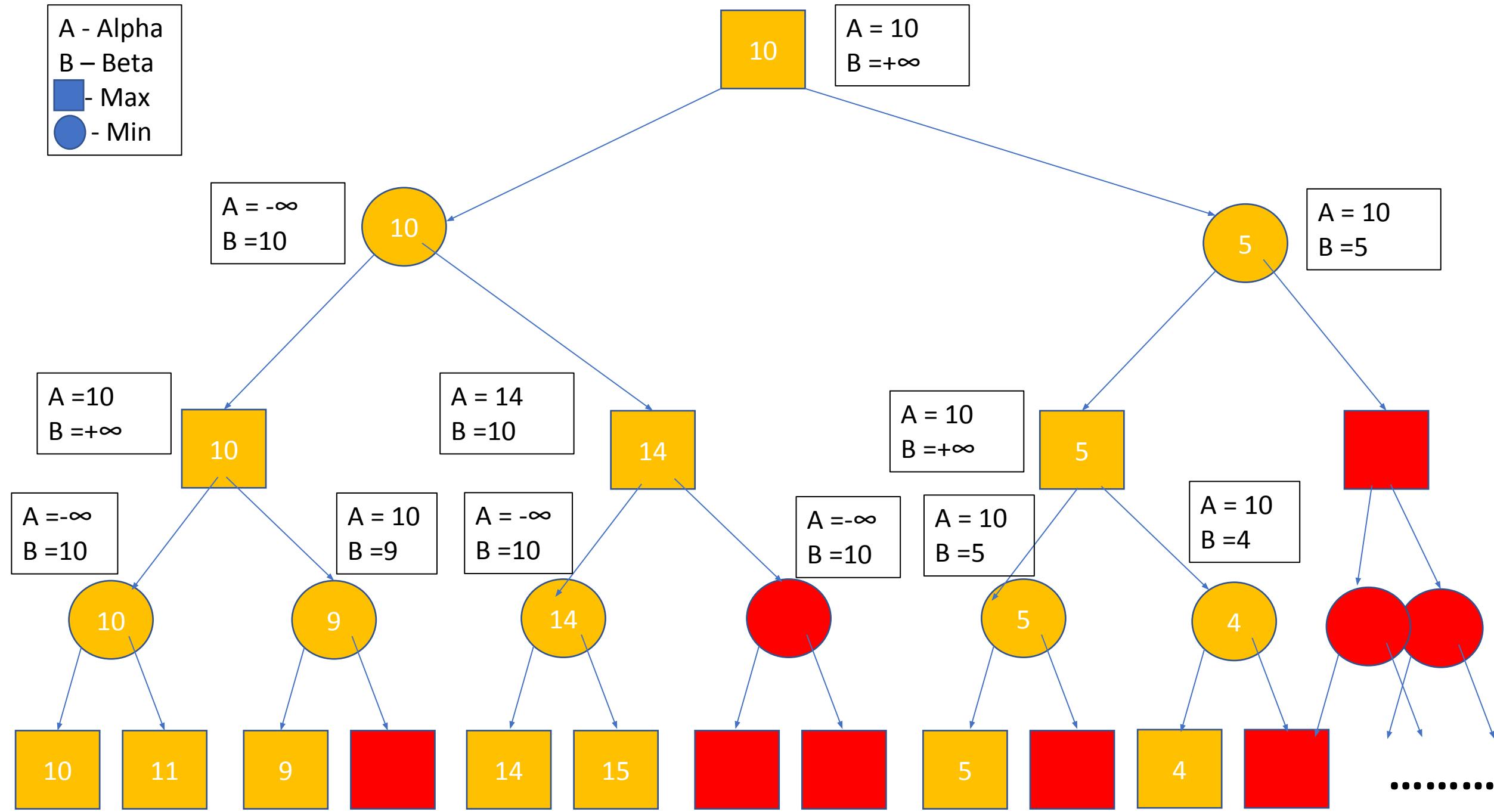
A - Alpha
B - Beta
— Max
● - Min



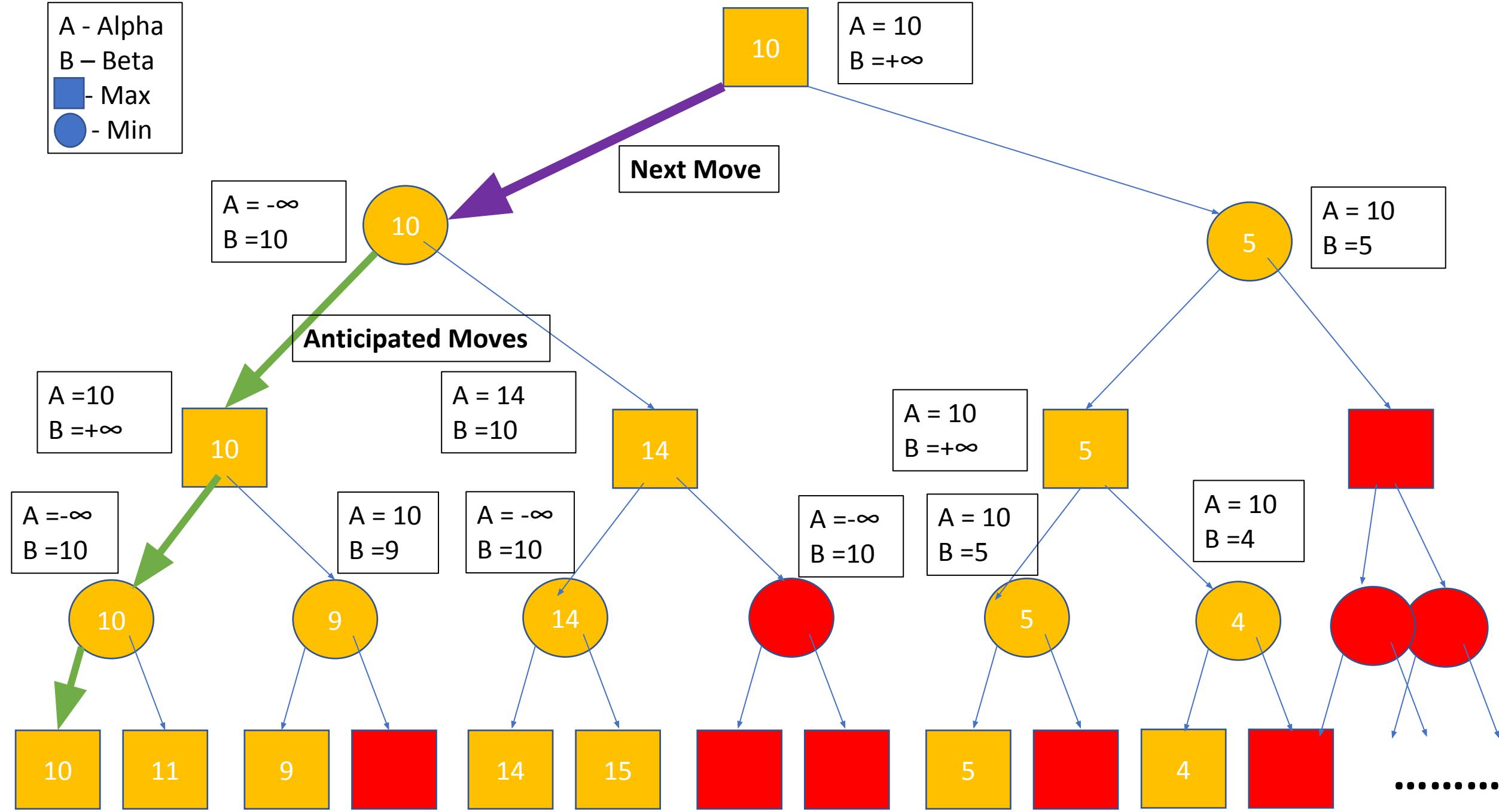
A - Alpha
B - Beta
— Max
● - Min



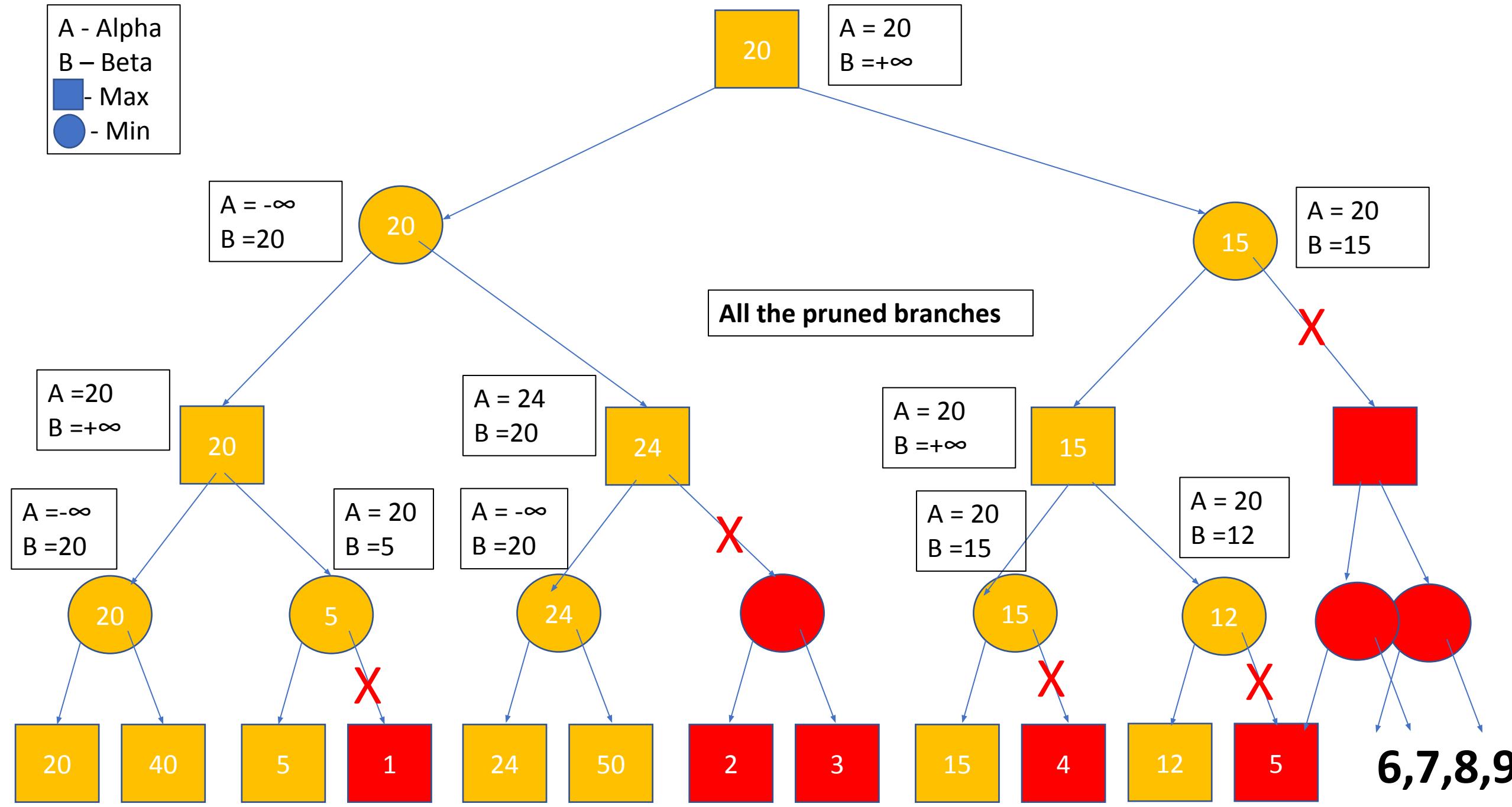
A - Alpha
B - Beta
— Max
● - Min



A - Alpha
B - Beta
— Max
● - Min



A - Alpha
B - Beta
— Max
● - Min



Pseudocode for the alpha-beta algorithm

```
evaluate (node, alpha, beta)
    if node is a leaf
        return the heuristic value of node
    if node is a minimizing node
        for each child of node
            beta = min (beta, evaluate (child, alpha, beta))
            if beta <= alpha
                return beta
        return beta
    if node is a maximizing node
        for each child of node
            alpha = max (alpha, evaluate (child, alpha, beta))
            if beta <= alpha
                return alpha
    return alpha
```

LOGICAL AGENTS

Knowledge-based agents

- Knowledge based agents use a process of reasoning over an internal representation of knowledge to decide what actions to take.
- Knowledge-based agents can accept new tasks in the form of explicitly described goals;
 - Can achieve competence quickly by being told or learning new knowledge about the environment;
 - Can adapt to changes in the environment by updating the relevant knowledge.

Knowledge-based agents

- The central component of a knowledge-based agent is its knowledge base, or KB.
- A knowledge base is a set of **sentences**.
- Each **sentence** is expressed in a language called a **knowledge representation language** and represents some assertion about the world.
- When the sentence is taken as being given without being derived from other sentences, we call it an **axiom**.

- TELL and ASK, are to add new sentences to the knowledge base and a way to query what is known, respectively.
- Both operations involve **inference**—that is, deriving new sentences from old.

Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously.

- KB may initially contain some background knowledge.

```
function KB-AGENT(percept) returns an action
  persistent: KB, a knowledge base
    t, a counter, initially 0, indicating time
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action  $\leftarrow$  ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t  $\leftarrow$  t + 1
  return action
```

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

- A knowledge-based agent can be built simply by TELLing it what it needs to know.
- Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment.
- This is called the **declarative approach** to system building.
- In contrast, the **procedural approach** encodes desired behaviors directly as program code.

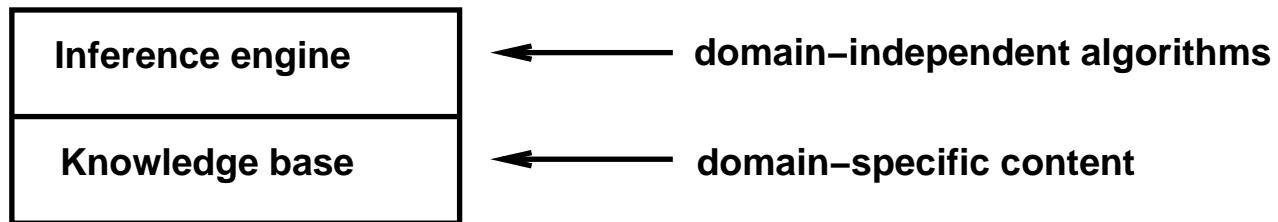
LOGICAL AGENTS

CHAPTER 7

Outline

- ◊ Knowledge-based agents
- ◊ Wumpus world
- ◊ Logic in general—models and entailment
- ◊ Propositional (Boolean) logic
- ◊ Equivalence, validity, satisfiability
- ◊ Inference rules and theorem proving
 - forward chaining
 - backward chaining
 - resolution

Knowledge bases



Knowledge base = set of sentences in a **formal** language

Declarative approach to building an agent (or other system):

TELL it what it needs to know

Then it can ASK itself what to do—answers should follow from the KB

Agents can be viewed at the **knowledge level**

i.e., **what they know**, regardless of how implemented

Or at the **implementation level**

i.e., data structures in KB and algorithms that manipulate them

A simple knowledge-based agent

```
function KB-AGENT(percept) returns an action
    static: KB, a knowledge base
          t, a counter, initially 0, indicating time
    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action  $\leftarrow$  ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t  $\leftarrow$  t + 1
    return action
```

The agent must be able to:

Represent states, actions, etc.

Incorporate new percepts

Update internal representations of the world

Deduce hidden properties of the world

Deduce appropriate actions

Wumpus World PEAS description

Performance measure

gold +1000, death -1000

-1 per step, -10 for using the arrow

Environment

Squares adjacent to wumpus are smelly

Squares adjacent to pit are breezy

Glitter iff gold is in the same square

Shooting kills wumpus if you are facing it

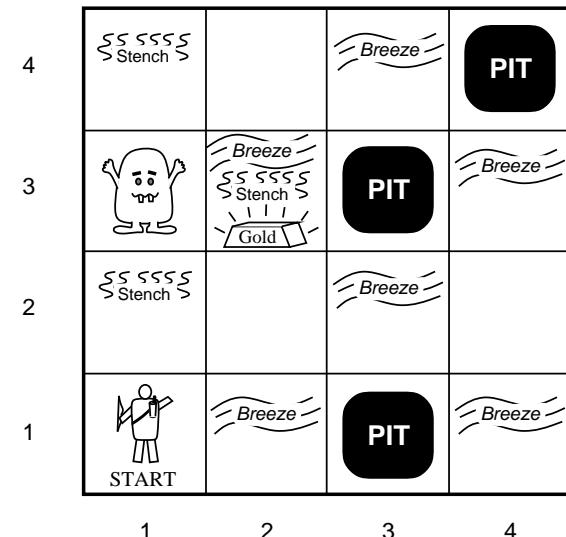
Shooting uses up the only arrow

Grabbing picks up gold if in same square

Releasing drops the gold in same square

Actuators Left turn, Right turn,
Forward, Grab, Release, Shoot

Sensors Breeze, Glitter, Smell



Wumpus world characterization

Observable??

Wumpus world characterization

Observable?? No—only local perception

Deterministic??

Wumpus world characterization

Observable?? No—only local perception

Deterministic?? Yes—outcomes exactly specified

Episodic??

Wumpus world characterization

Observable?? No—only local perception

Deterministic?? Yes—outcomes exactly specified

Episodic?? No—sequential at the level of actions

Static??

Wumpus world characterization

Observable?? No—only local perception

Deterministic?? Yes—outcomes exactly specified

Episodic?? No—sequential at the level of actions

Static?? Yes—Wumpus and Pits do not move

Discrete??

Wumpus world characterization

Observable?? No—only local perception

Deterministic?? Yes—outcomes exactly specified

Episodic?? No—sequential at the level of actions

Static?? Yes—Wumpus and Pits do not move

Discrete?? Yes

Single-agent??

Wumpus world characterization

Observable?? No—only local perception

Deterministic?? Yes—outcomes exactly specified

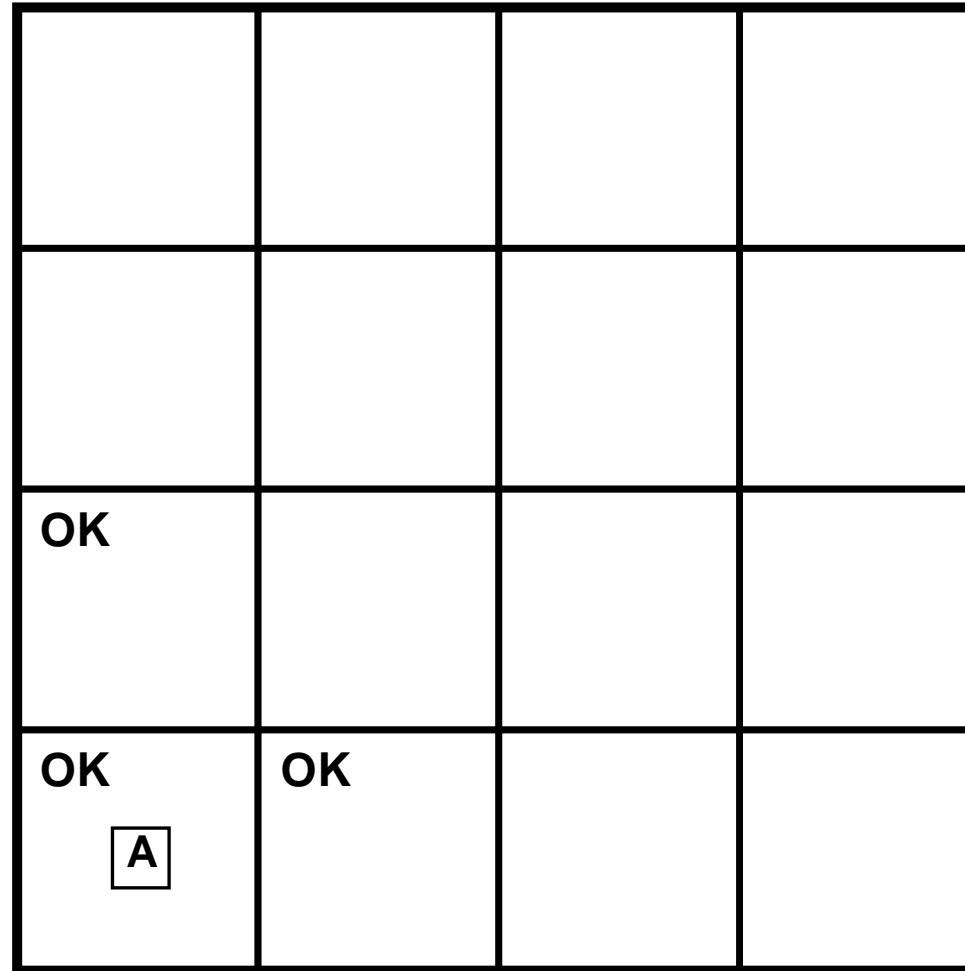
Episodic?? No—sequential at the level of actions

Static?? Yes—Wumpus and Pits do not move

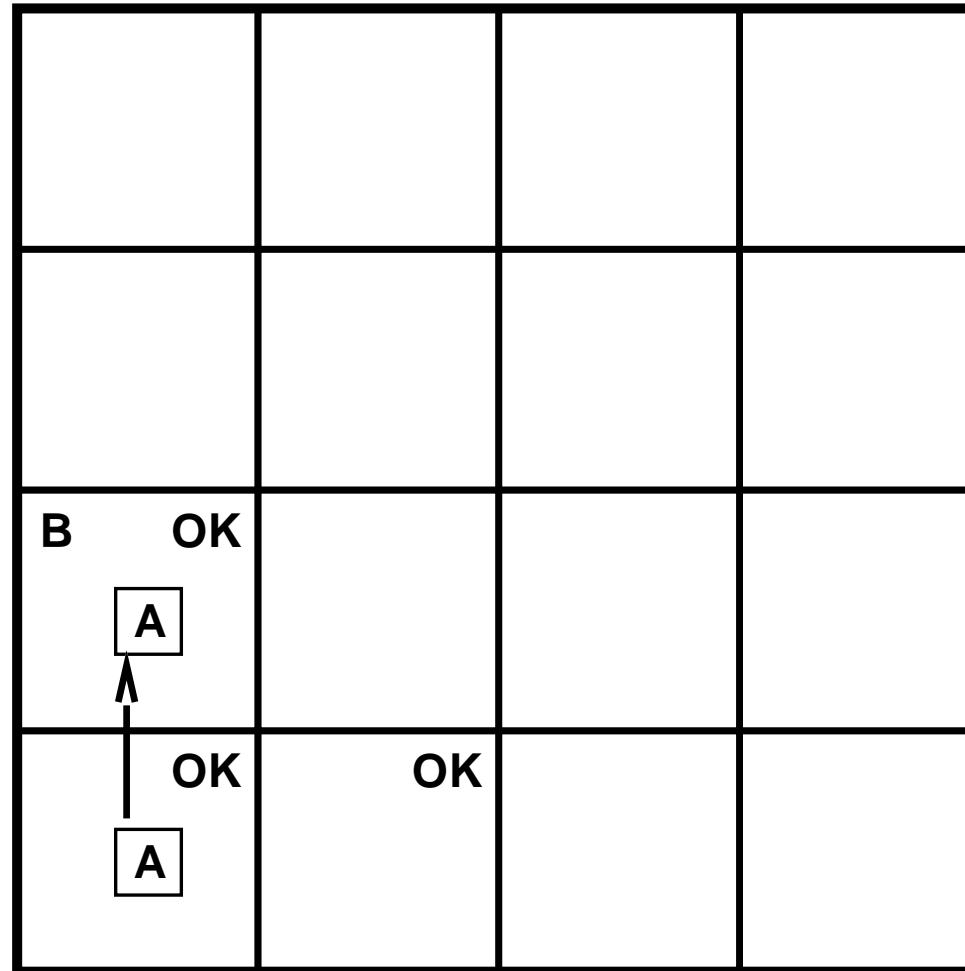
Discrete?? Yes

Single-agent?? Yes—Wumpus is essentially a natural feature

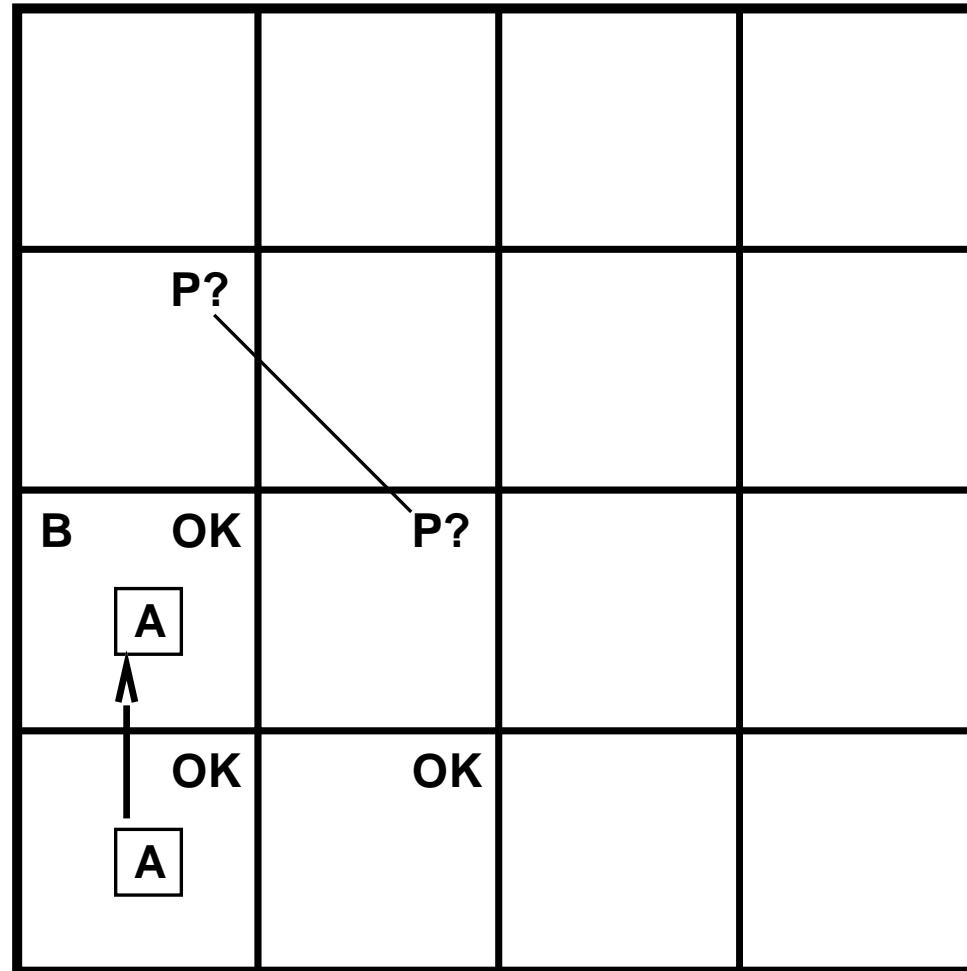
Exploring a wumpus world



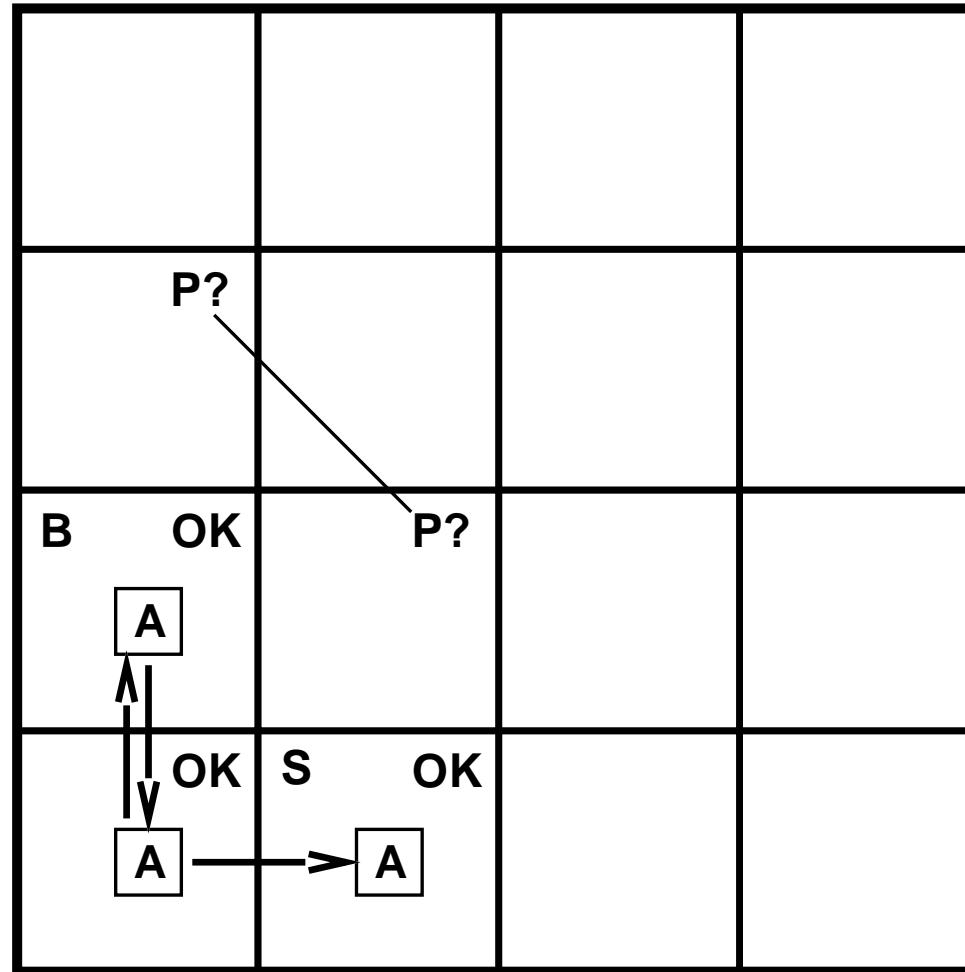
Exploring a wumpus world



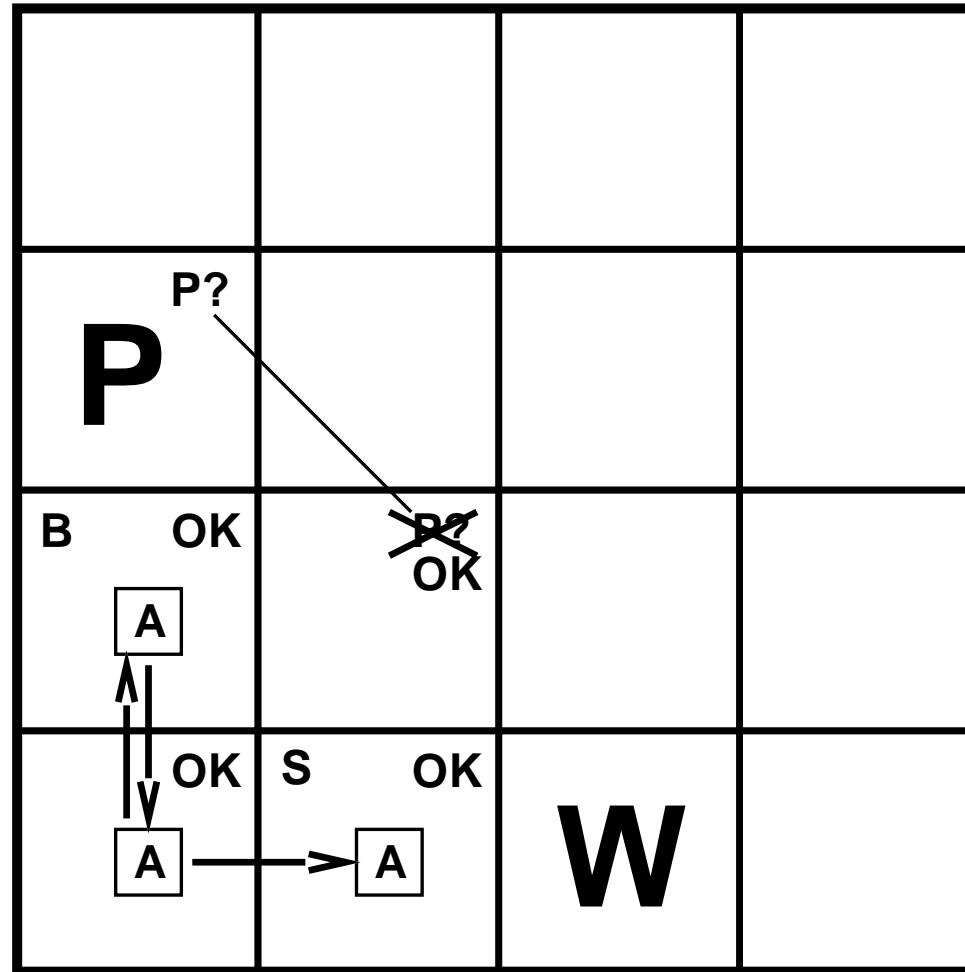
Exploring a wumpus world



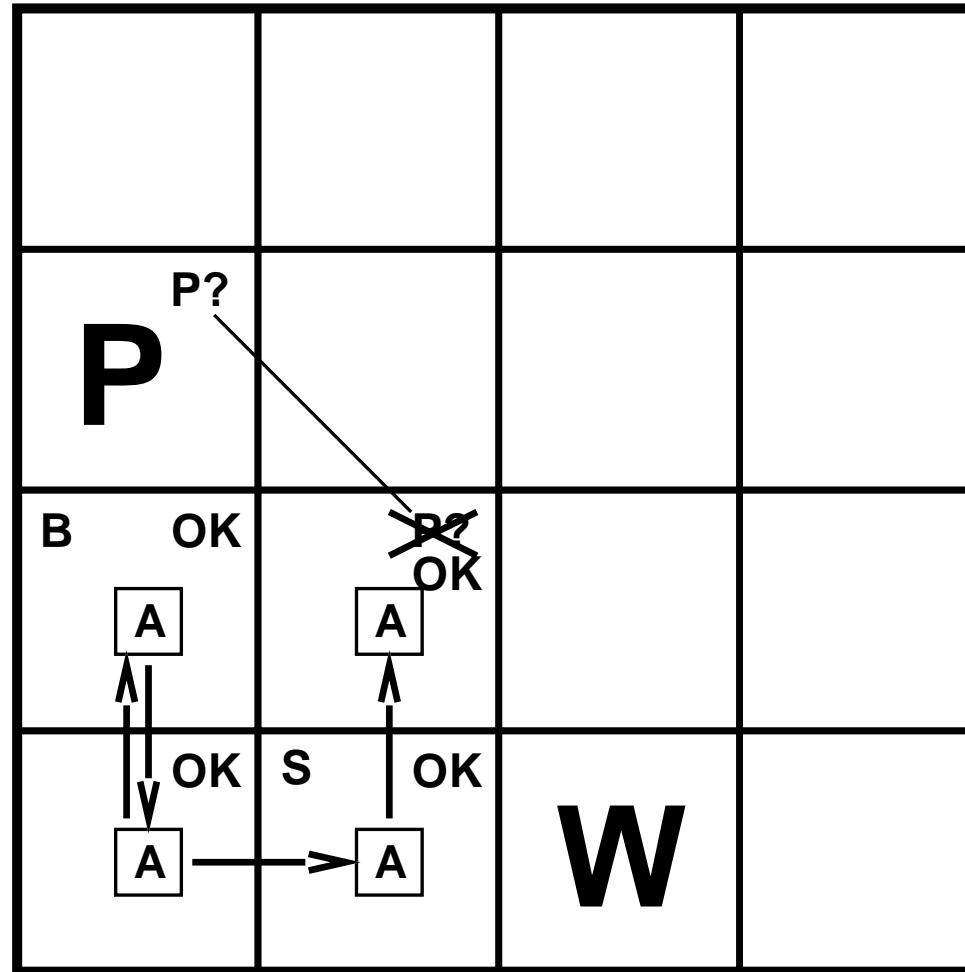
Exploring a wumpus world



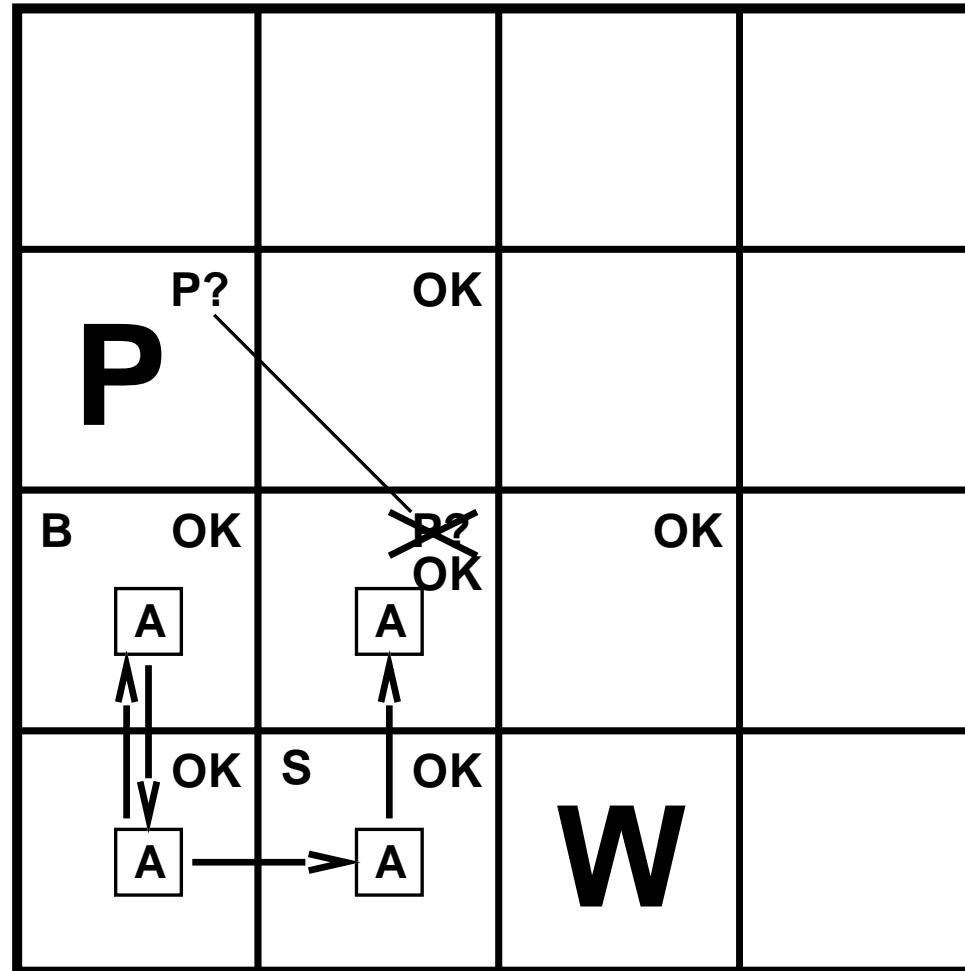
Exploring a wumpus world



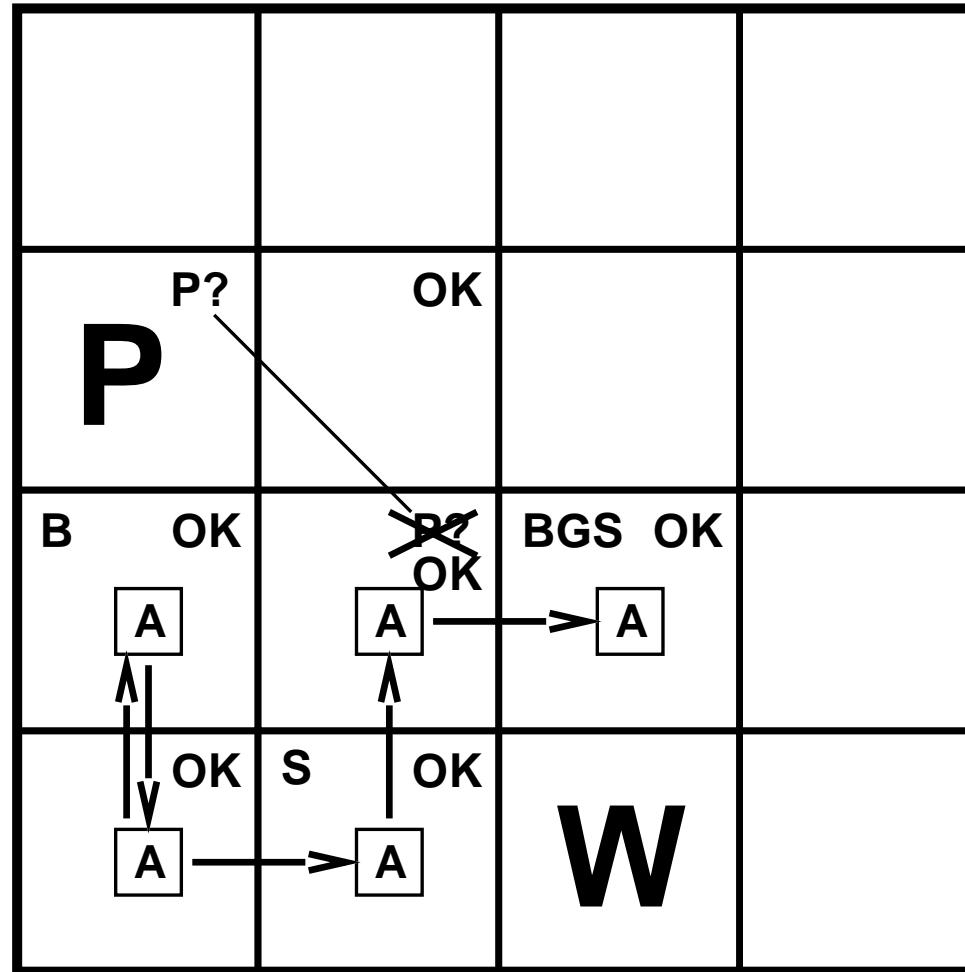
Exploring a wumpus world



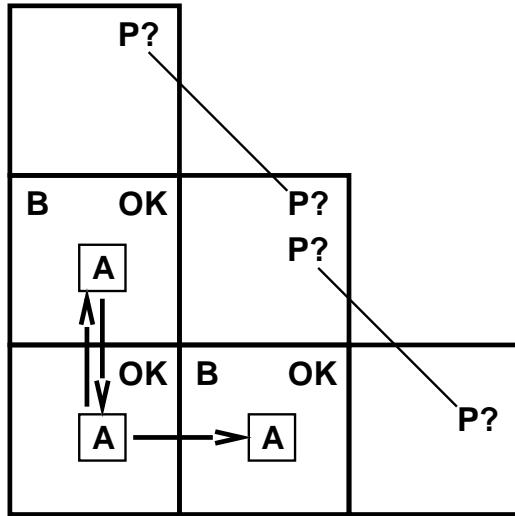
Exploring a wumpus world



Exploring a wumpus world

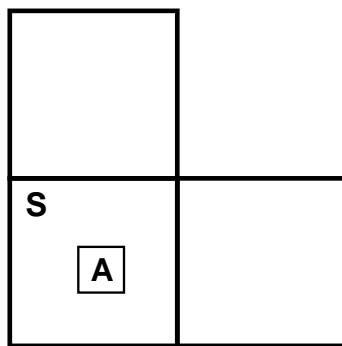


Other tight spots



Breeze in (1,2) and (2,1)
⇒ no safe actions

Assuming pits uniformly distributed,
(2,2) has pit w/ prob 0.86, vs. 0.31



Smell in (1,1)
⇒ cannot move

Can use a strategy of **coercion**:

shoot straight ahead

wumpus was there ⇒ dead ⇒ safe

wumpus wasn't there ⇒ safe

Logic in general

Logics are formal languages for representing information such that conclusions can be drawn

Syntax defines the sentences in the language

Semantics define the “meaning” of sentences;
i.e., define truth of a sentence in a world

E.g., the language of arithmetic

$x + 2 \geq y$ is a sentence; $x2 + y >$ is not a sentence

$x + 2 \geq y$ is true iff the number $x + 2$ is no less than the number y

$x + 2 \geq y$ is true in a world where $x = 7, y = 1$

$x + 2 \geq y$ is false in a world where $x = 0, y = 6$

Entailment

Entailment means that one thing **follows from** another:

$$KB \models \alpha$$

Knowledge base KB entails sentence α

if and only if

α is true in all worlds where KB is true

E.g., the KB containing “the Giants won” and “the Reds won”
entails “Either the Giants won or the Reds won”

E.g., $x + y = 4$ entails $4 = x + y$

Entailment is a relationship between sentences (i.e., **syntax**)
that is based on **semantics**

Note: brains process **syntax** (of some sort)

Models

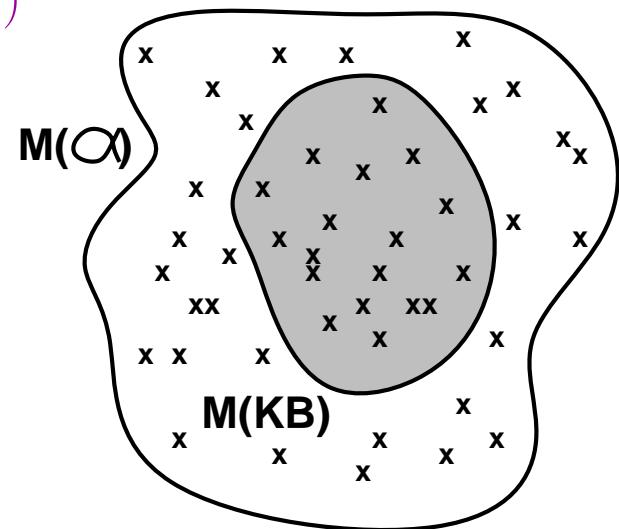
Logicians typically think in terms of **models**, which are formally structured worlds with respect to which truth can be evaluated

We say m is a model of a sentence α if α is true in m

$M(\alpha)$ is the set of all models of α

Then $KB \models \alpha$ if and only if $M(KB) \subseteq M(\alpha)$

E.g. $KB =$ Giants won and Reds won
 $\alpha =$ Giants won

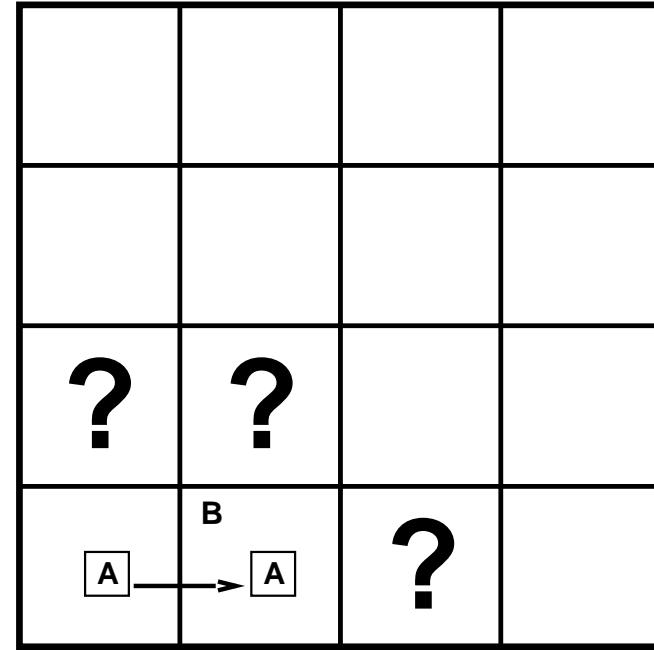


Entailment in the wumpus world

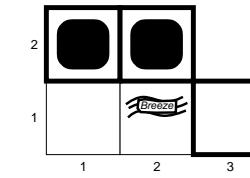
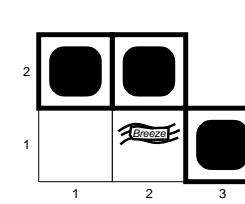
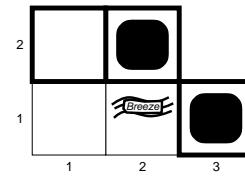
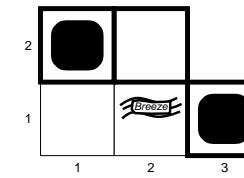
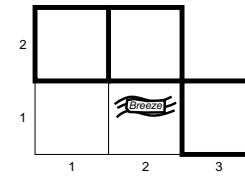
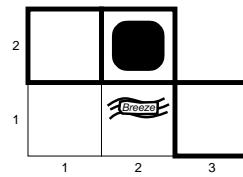
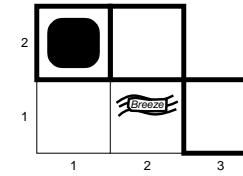
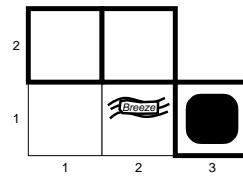
Situation after detecting nothing in [1,1],
moving right, breeze in [2,1]

Consider possible models for ?s
assuming only pits

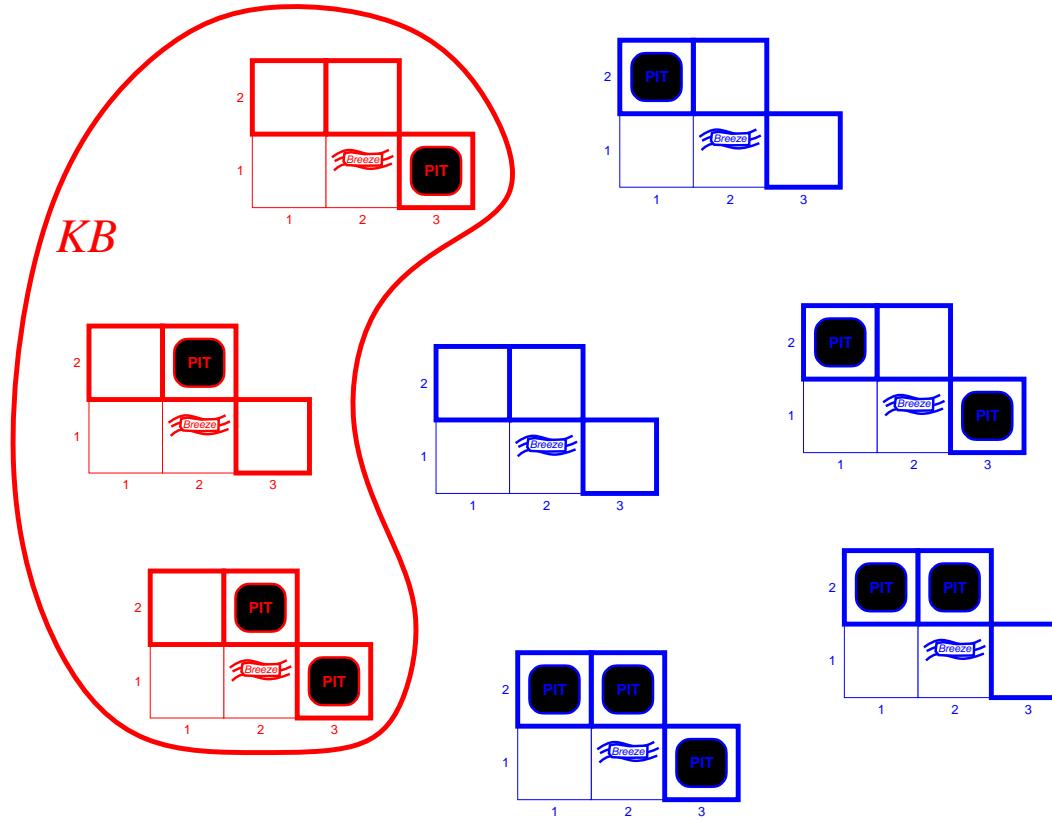
3 Boolean choices \Rightarrow 8 possible models



Wumpus models

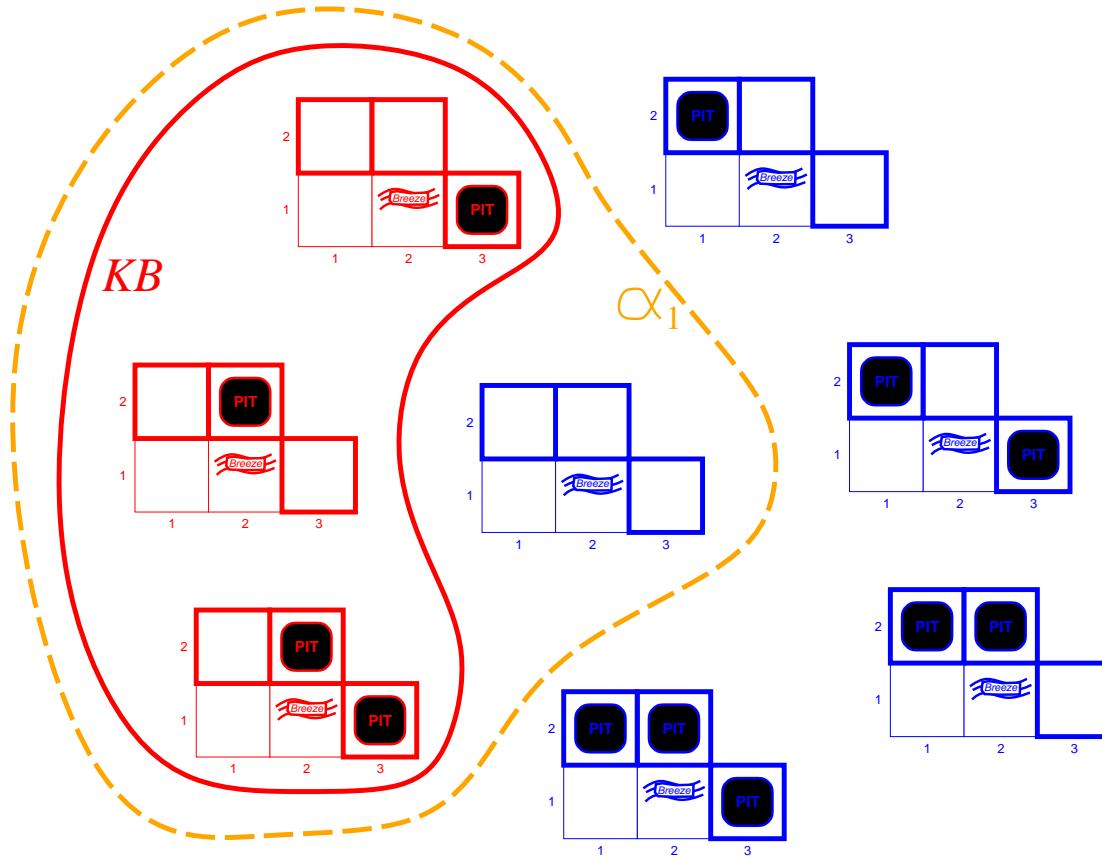


Wumpus models



KB = wumpus-world rules + observations

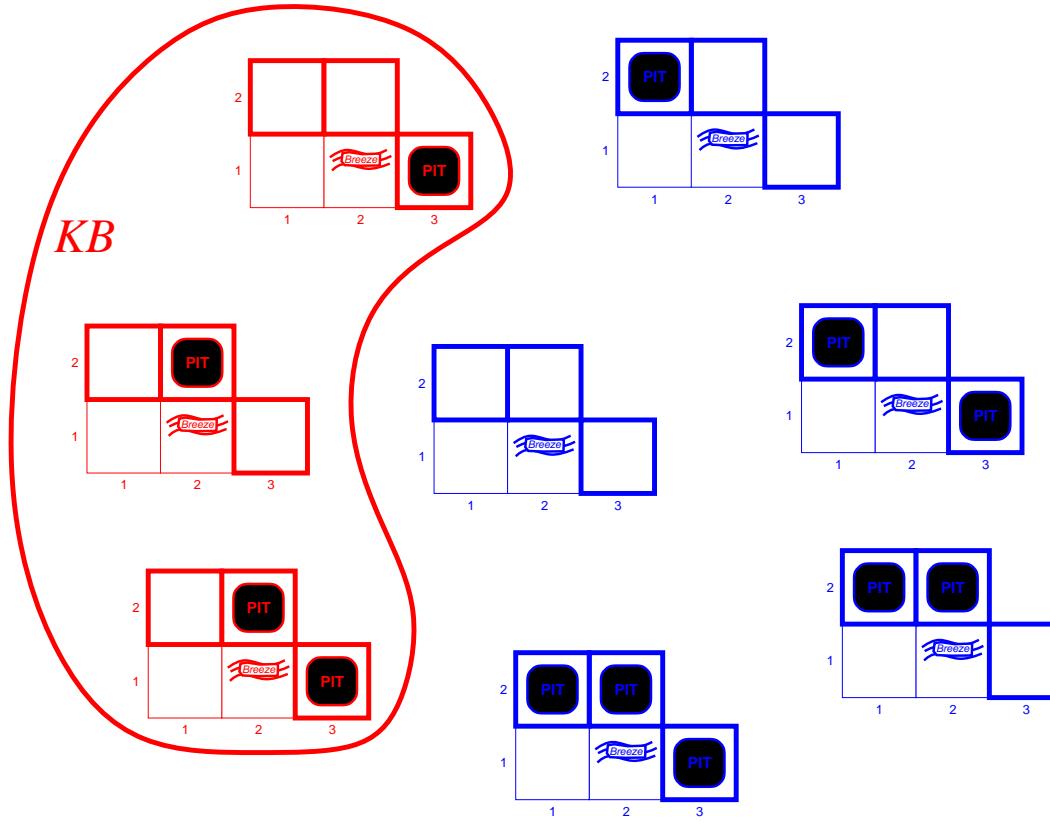
Wumpus models



KB = wumpus-world rules + observations

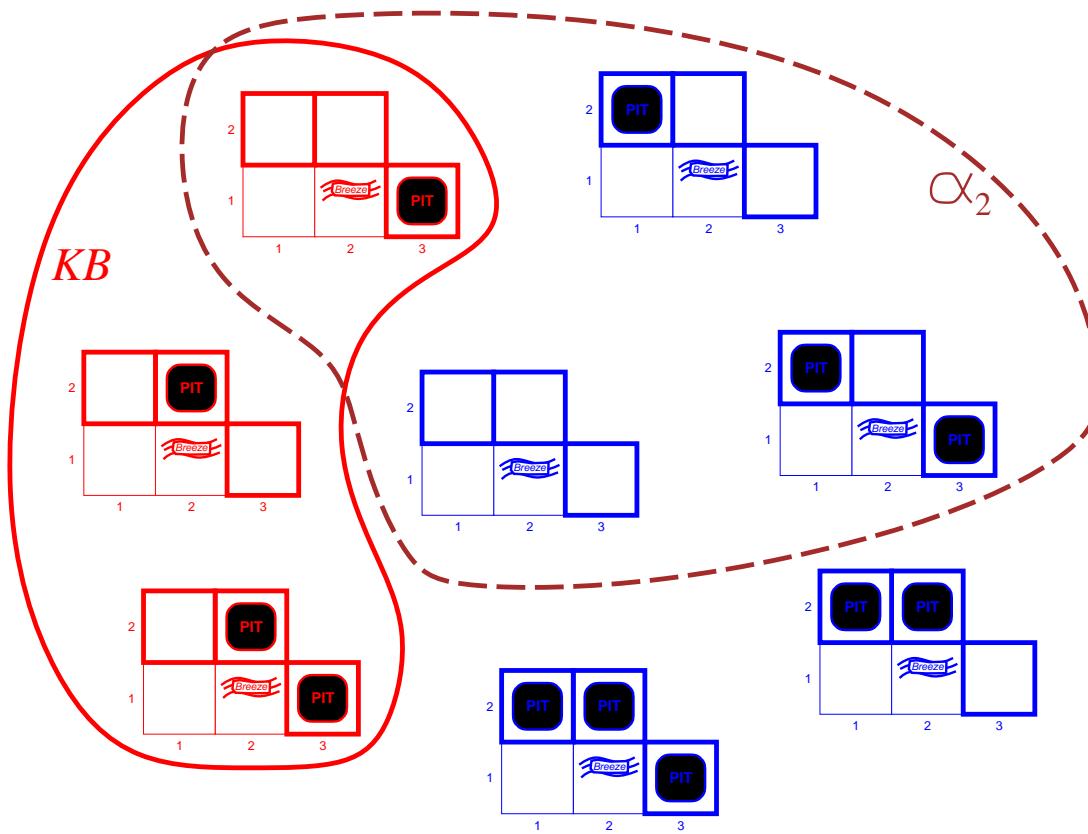
α_1 = “[1,2] is safe”, $KB \models \alpha_1$, proved by model checking

Wumpus models



KB = wumpus-world rules + observations

Wumpus models



$KB = \text{wumpus-world rules} + \text{observations}$

$\alpha_2 = \text{"[2,2] is safe"}, KB \not\models \alpha_2$

Inference

$KB \vdash_i \alpha$ = sentence α can be derived from KB by procedure i

Consequences of KB are a haystack; α is a needle.

Entailment = needle in haystack; inference = finding it

Soundness: i is sound if

whenever $KB \vdash_i \alpha$, it is also true that $KB \models \alpha$

Completeness: i is complete if

whenever $KB \models \alpha$, it is also true that $KB \vdash_i \alpha$

Preview: we will define a logic (first-order logic) which is expressive enough to say almost anything of interest, and for which there exists a sound and complete inference procedure.

That is, the procedure will answer any question whose answer follows from what is known by the KB .

Propositional logic: Syntax

Propositional logic is the simplest logic—illustrates basic ideas

The proposition symbols P_1, P_2 etc are sentences

If S is a sentence, $\neg S$ is a sentence (negation)

If S_1 and S_2 are sentences, $S_1 \wedge S_2$ is a sentence (conjunction)

If S_1 and S_2 are sentences, $S_1 \vee S_2$ is a sentence (disjunction)

If S_1 and S_2 are sentences, $S_1 \Rightarrow S_2$ is a sentence (implication)

If S_1 and S_2 are sentences, $S_1 \Leftrightarrow S_2$ is a sentence (biconditional)

Propositional logic: Semantics

Each model specifies true/false for each proposition symbol

E.g. $P_{1,2}$ $P_{2,2}$ $P_{3,1}$
true true false

(With these symbols, 8 possible models, can be enumerated automatically.)

Rules for evaluating truth with respect to a model m :

$\neg S$	is true iff	S	is false	
$S_1 \wedge S_2$	is true iff	S_1	is true and	S_2 is true
$S_1 \vee S_2$	is true iff	S_1	is true or	S_2 is true
$S_1 \Rightarrow S_2$	is true iff	S_1	is false or	S_2 is true
i.e.,	is false iff	S_1	is true and	S_2 is false
$S_1 \Leftrightarrow S_2$	is true iff	$S_1 \Rightarrow S_2$	is true and	$S_2 \Rightarrow S_1$ is true

Simple recursive process evaluates an arbitrary sentence, e.g.,

$$\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1}) = \text{true} \wedge (\text{false} \vee \text{true}) = \text{true} \wedge \text{true} = \text{true}$$

Truth tables for connectives

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>true</i>
<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>false</i>
<i>true</i>	<i>false</i>	<i>false</i>	<i>false</i>	<i>true</i>	<i>false</i>	<i>false</i>
<i>true</i>	<i>true</i>	<i>false</i>	<i>true</i>	<i>true</i>	<i>true</i>	<i>true</i>

Wumpus world sentences

Let $P_{i,j}$ be true if there is a pit in $[i, j]$.

Let $B_{i,j}$ be true if there is a breeze in $[i, j]$.

$\neg P_{1,1}$

$\neg B_{1,1}$

$B_{2,1}$

“Pits cause breezes in adjacent squares”

Wumpus world sentences

Let $P_{i,j}$ be true if there is a pit in $[i, j]$.

Let $B_{i,j}$ be true if there is a breeze in $[i, j]$.

$$\neg P_{1,1}$$

$$\neg B_{1,1}$$

$$B_{2,1}$$

“Pits cause breezes in adjacent squares”

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

$$B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

“A square is breezy **if and only if** there is an adjacent pit”

Truth tables for inference

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

Enumerate rows (different assignments to symbols),
if KB is true in row, check that α is too

Inference by enumeration

Depth-first enumeration of all models is sound and complete

```
function TT-ENTAILS?(KB,  $\alpha$ ) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
           $\alpha$ , the query, a sentence in propositional logic
  symbols  $\leftarrow$  a list of the proposition symbols in KB and  $\alpha$ 
  return TT-CHECK-ALL(KB,  $\alpha$ , symbols, [])



---


function TT-CHECK-ALL(KB,  $\alpha$ , symbols, model) returns true or false
  if EMPTY?(symbols) then
    if PL-TRUE?(KB, model) then return PL-TRUE?( $\alpha$ , model)
    else return true
  else do
    P  $\leftarrow$  FIRST(symbols); rest  $\leftarrow$  REST(symbols)
    return TT-CHECK-ALL(KB,  $\alpha$ , rest, EXTEND(P, true, model)) and
           TT-CHECK-ALL(KB,  $\alpha$ , rest, EXTEND(P, false, model))
```

$O(2^n)$ for n symbols; problem is **co-NP-complete**

Logical equivalence

Two sentences are **logically equivalent** iff true in same models:

$\alpha \equiv \beta$ if and only if $\alpha \models \beta$ and $\beta \models \alpha$

$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha)$	commutativity of \wedge
$(\alpha \vee \beta) \equiv (\beta \vee \alpha)$	commutativity of \vee
$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma))$	associativity of \wedge
$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma))$	associativity of \vee
$\neg(\neg \alpha) \equiv \alpha$	double-negation elimination
$(\alpha \Rightarrow \beta) \equiv (\neg \beta \Rightarrow \neg \alpha)$	contraposition
$(\alpha \Rightarrow \beta) \equiv (\neg \alpha \vee \beta)$	implication elimination
$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha))$	biconditional elimination
$\neg(\alpha \wedge \beta) \equiv (\neg \alpha \vee \neg \beta)$	De Morgan
$\neg(\alpha \vee \beta) \equiv (\neg \alpha \wedge \neg \beta)$	De Morgan
$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma))$	distributivity of \wedge over \vee
$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma))$	distributivity of \vee over \wedge

Validity and satisfiability

A sentence is **valid** if it is true in **all** models,

e.g., True , $A \vee \neg A$, $A \Rightarrow A$, $(A \wedge (A \Rightarrow B)) \Rightarrow B$

Validity is connected to inference via the Deduction Theorem:

$KB \models \alpha$ if and only if $(KB \Rightarrow \alpha)$ is valid

A sentence is **satisfiable** if it is true in **some** model

e.g., $A \vee B$, C

A sentence is **unsatisfiable** if it is true in **no** models

e.g., $A \wedge \neg A$

Satisfiability is connected to inference via the following:

$KB \models \alpha$ if and only if $(KB \wedge \neg \alpha)$ is unsatisfiable

i.e., prove α by *reductio ad absurdum*

Proof methods

Proof methods divide into (roughly) two kinds:

Application of inference rules

- Legitimate (sound) generation of new sentences from old
- **Proof** = a sequence of inference rule applications
 - Can use inference rules as operators in a standard search alg.
- Typically require translation of sentences into a **normal form**

Model checking

- truth table enumeration (always exponential in n)
- improved backtracking, e.g., Davis–Putnam–Logemann–Loveland
- heuristic search in model space (sound but incomplete)
 - e.g., min-conflicts-like hill-climbing algorithms

Forward and backward chaining

Horn Form (restricted)

KB = **conjunction** of **Horn clauses**

Horn clause =

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

E.g., $C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$

Modus Ponens (for Horn Form): complete for Horn KBs

$$\frac{\alpha_1, \dots, \alpha_n, \quad \alpha_1 \wedge \dots \wedge \alpha_n \Rightarrow \beta}{\beta}$$

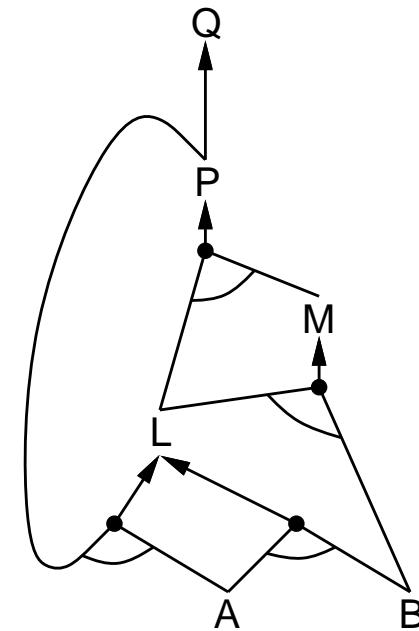
Can be used with **forward chaining** or **backward chaining**.

These algorithms are very natural and run in **linear** time

Forward chaining

Idea: fire any rule whose premises are satisfied in the KB ,
add its conclusion to the KB , until query is found

$$\begin{aligned} P &\Rightarrow Q \\ L \wedge M &\Rightarrow P \\ B \wedge L &\Rightarrow M \\ A \wedge P &\Rightarrow L \\ A \wedge B &\Rightarrow L \\ A \\ B \end{aligned}$$



Forward chaining algorithm

function PL-FC-ENTAILS?(*KB*, *q*) **returns** *true* or *false*

inputs: *KB*, the knowledge base, a set of propositional Horn clauses
 q, the query, a proposition symbol

local variables: *count*, a table, indexed by clause, initially the number of premises
 inferred, a table, indexed by symbol, each entry initially *false*
 agenda, a list of symbols, initially the symbols known in *KB*

while *agenda* is not empty **do**

p \leftarrow POP(*agenda*)

unless *inferred*[*p*] **do**

inferred[*p*] \leftarrow *true*

for each Horn clause *c* in whose premise *p* appears **do**

 decrement *count*[*c*]

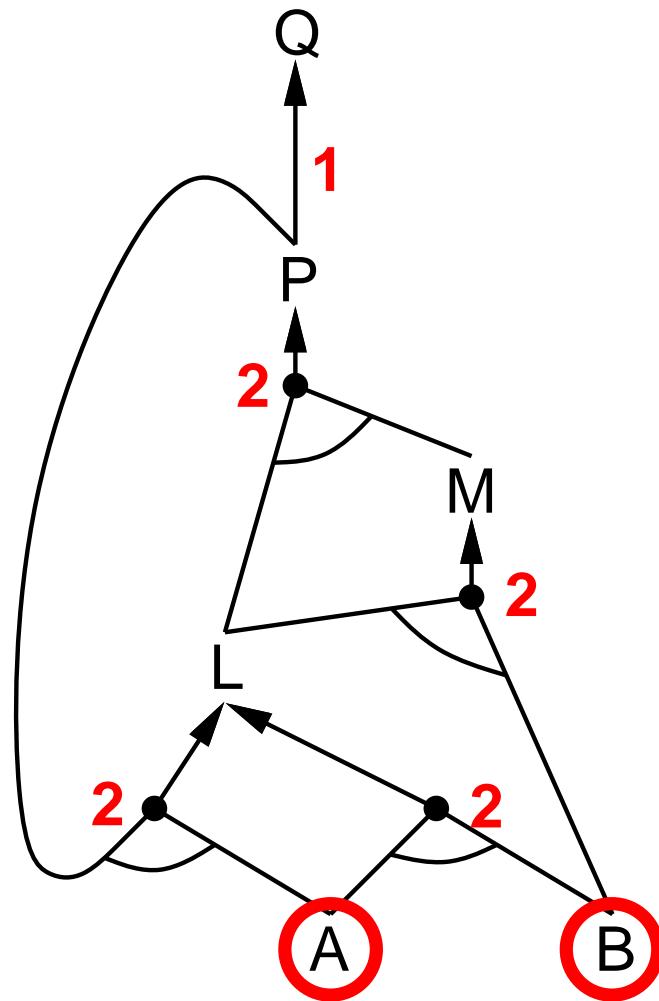
if *count*[*c*] = 0 **then do**

if HEAD[*c*] = *q* **then return** *true*

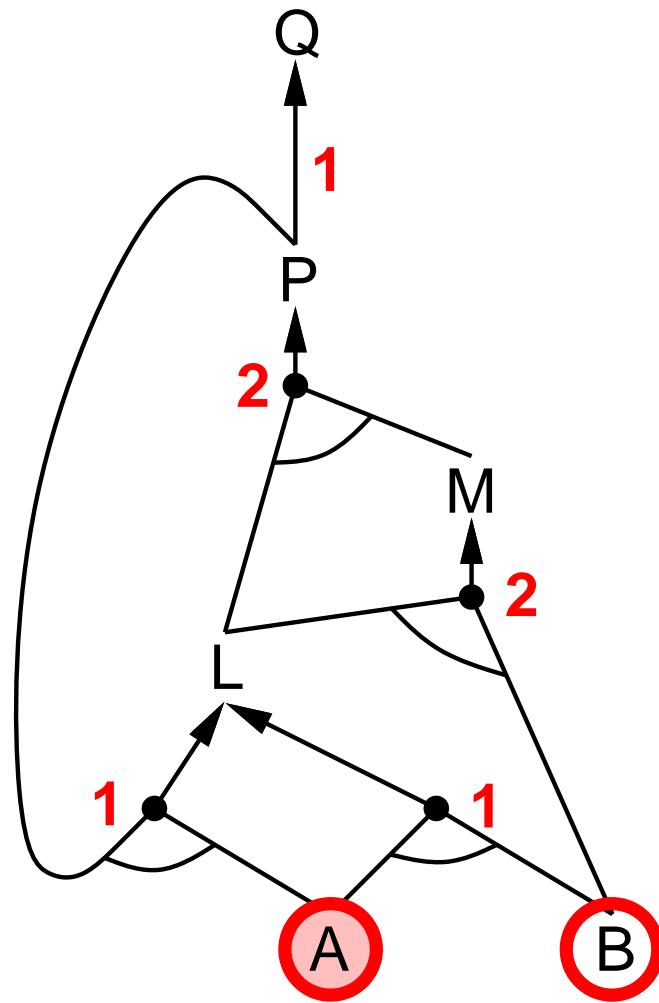
 PUSH(HEAD[*c*], *agenda*)

return *false*

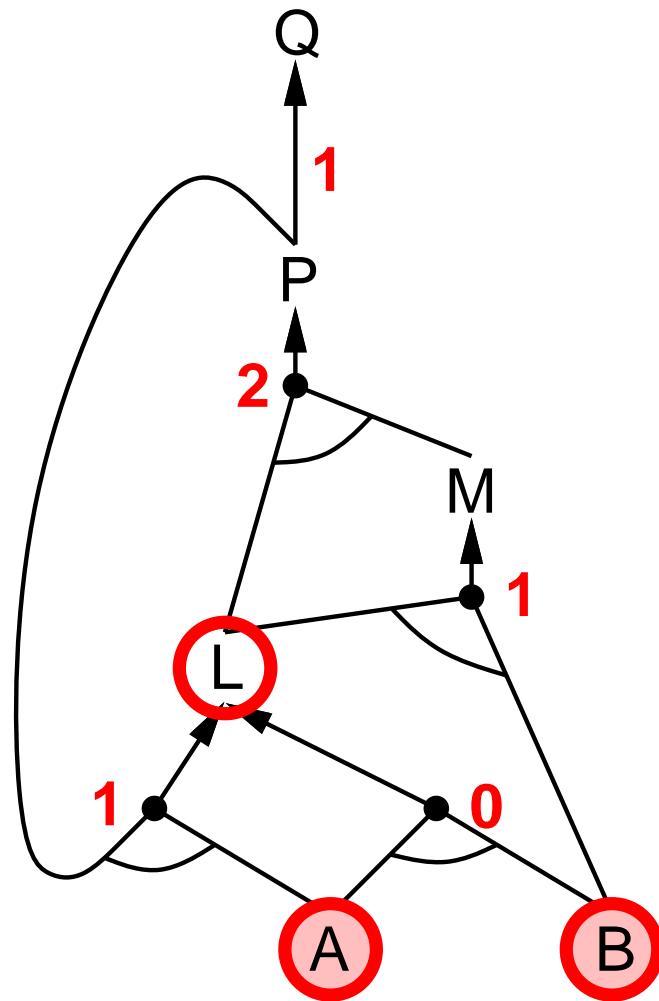
Forward chaining example



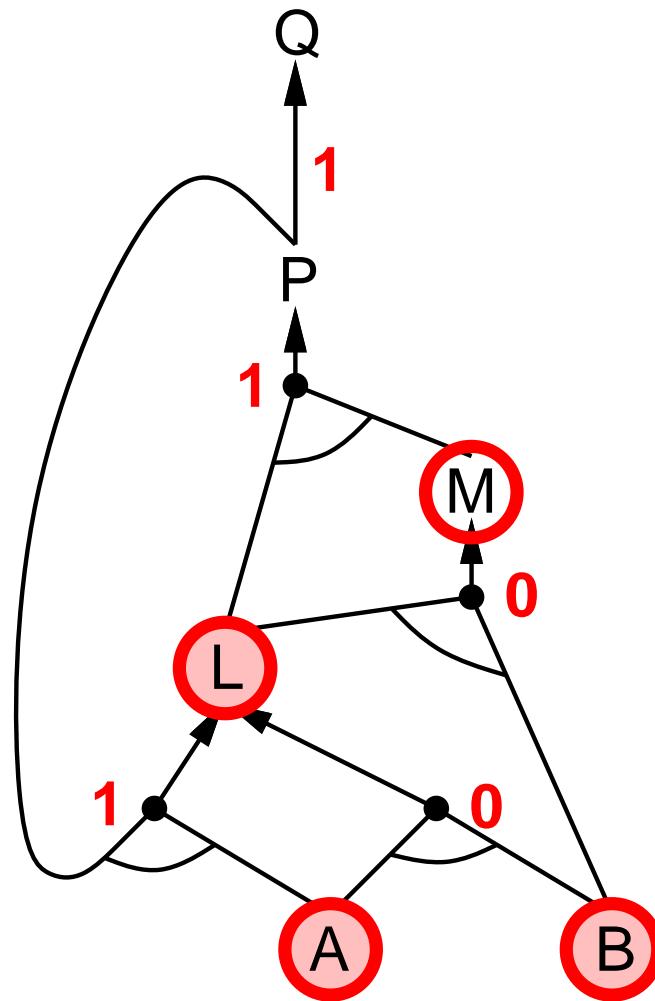
Forward chaining example



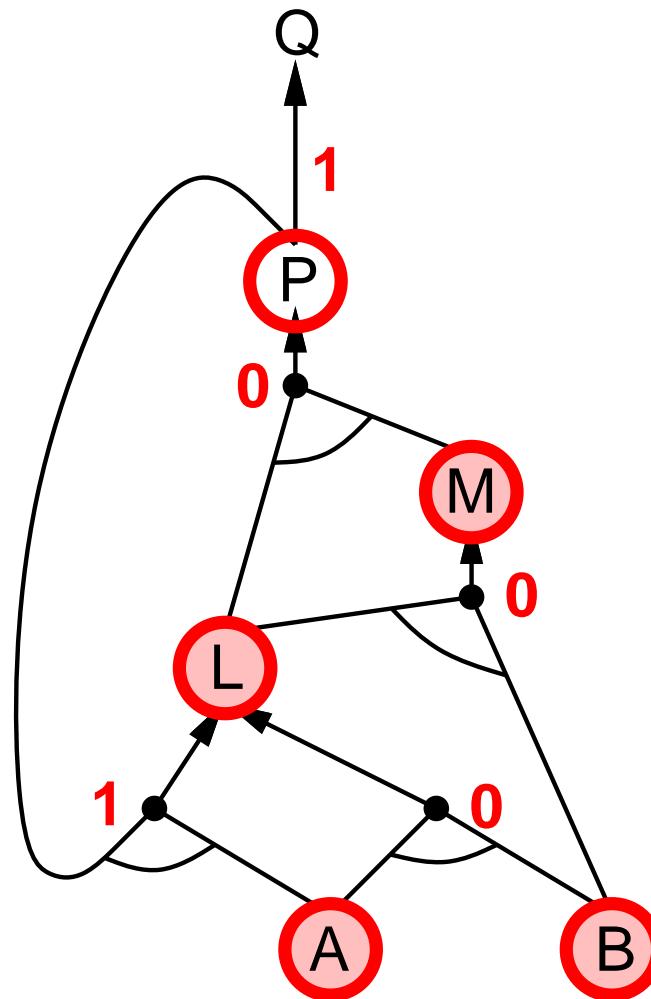
Forward chaining example



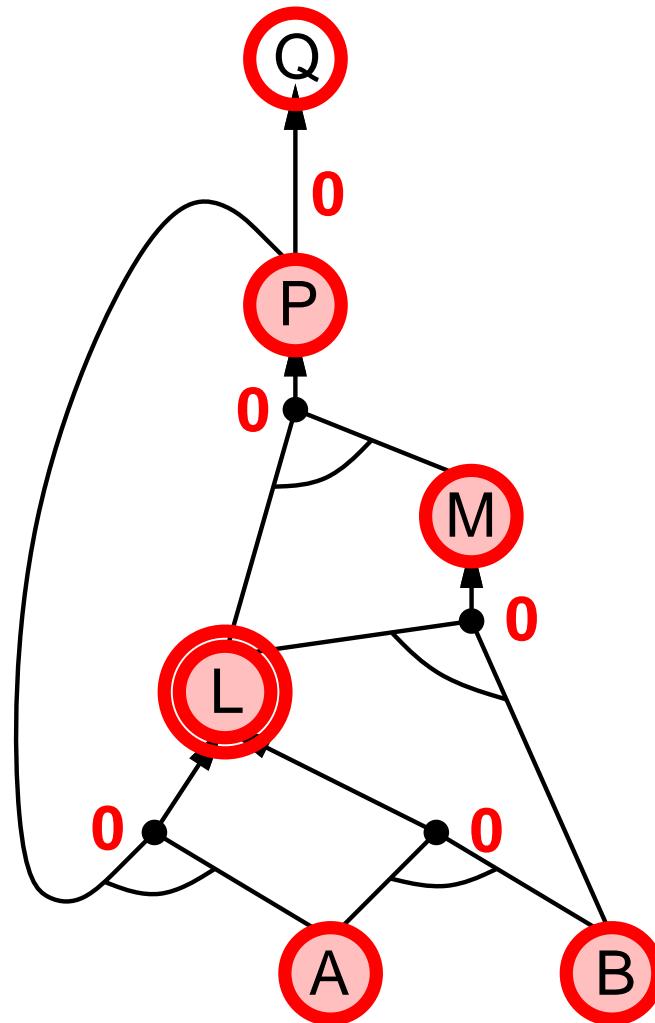
Forward chaining example



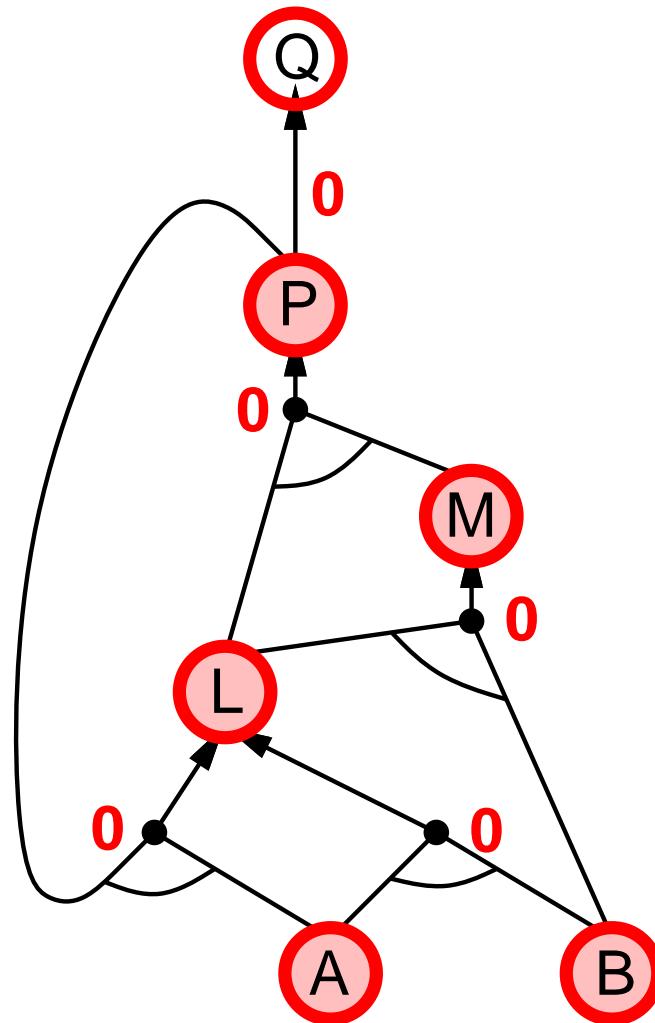
Forward chaining example



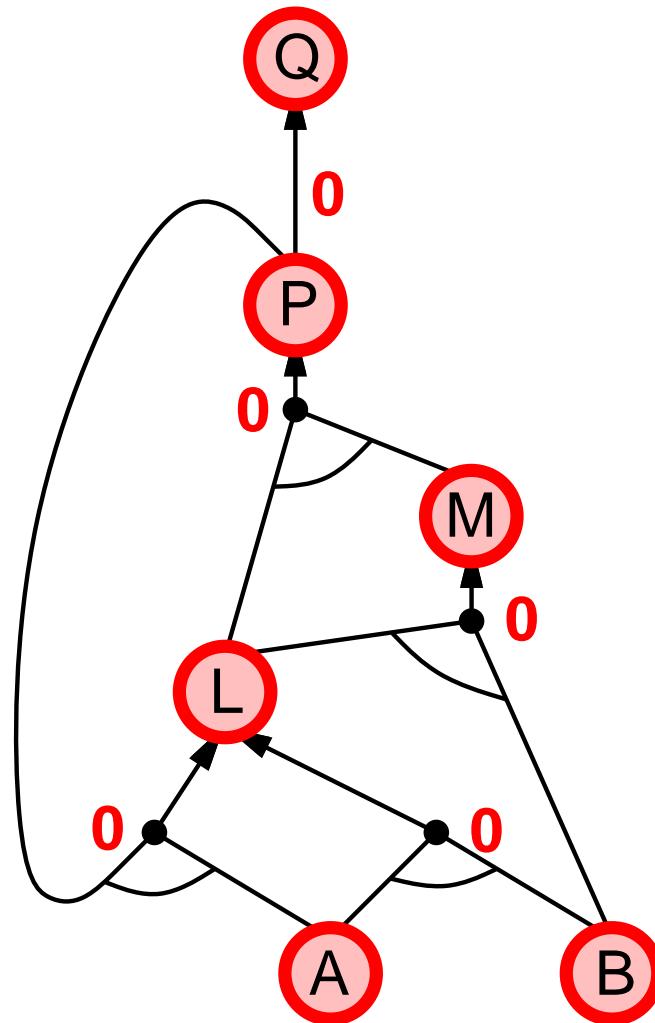
Forward chaining example



Forward chaining example



Forward chaining example



Proof of completeness

FC derives every atomic sentence that is entailed by KB

1. FC reaches a **fixed point** where no new atomic sentences are derived
2. Consider the final state as a model m , assigning true/false to symbols
3. Every clause in the original KB is true in m
Proof: Suppose a clause $a_1 \wedge \dots \wedge a_k \Rightarrow b$ is false in m
Then $a_1 \wedge \dots \wedge a_k$ is true in m and b is false in m
Therefore the algorithm has not reached a fixed point!
4. Hence m is a model of KB
5. If $KB \models q$, q is true in **every** model of KB , including m

General idea: construct any model of KB by sound inference, check α

Backward chaining

Idea: work backwards from the query q :

- to prove q by BC,

- check if q is known already, or

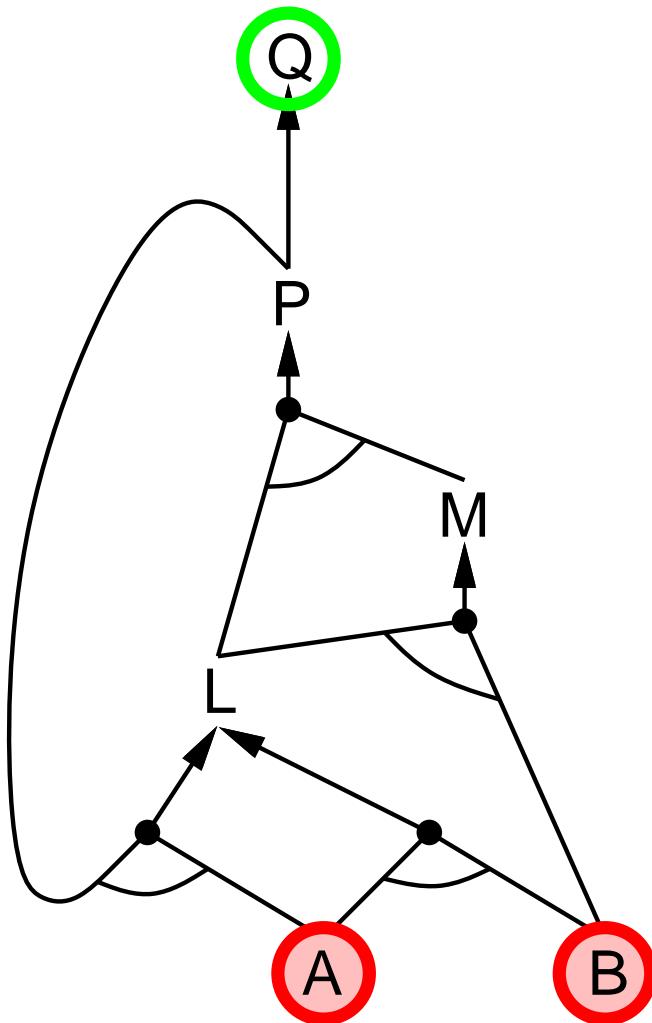
- prove by BC all premises of some rule concluding q

Avoid loops: check if new subgoal is already on the goal stack

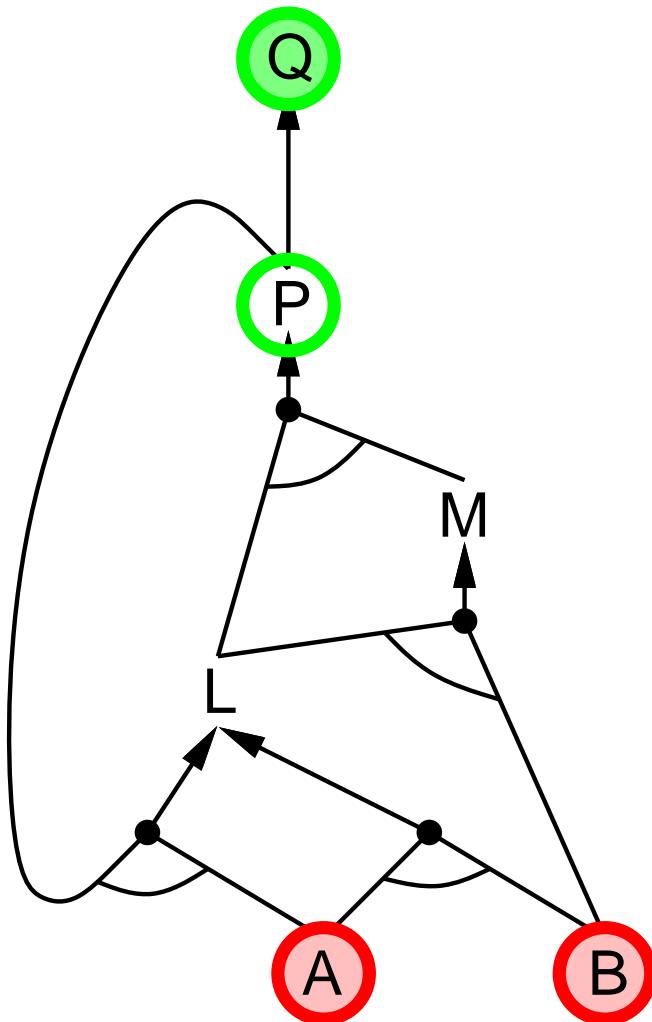
Avoid repeated work: check if new subgoal

- 1) has already been proved true, or
- 2) has already failed

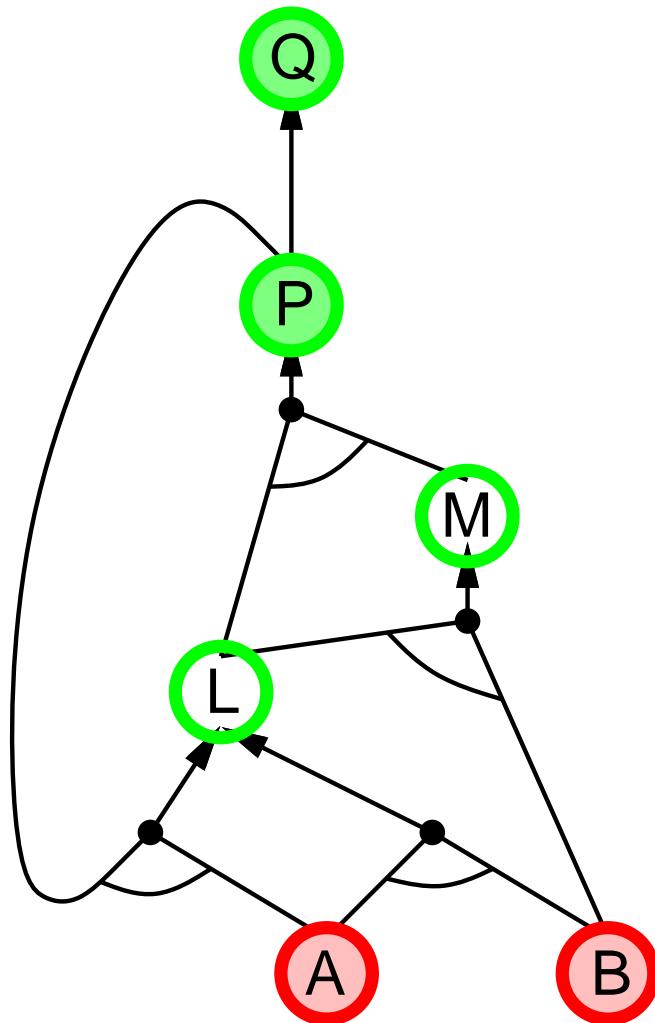
Backward chaining example



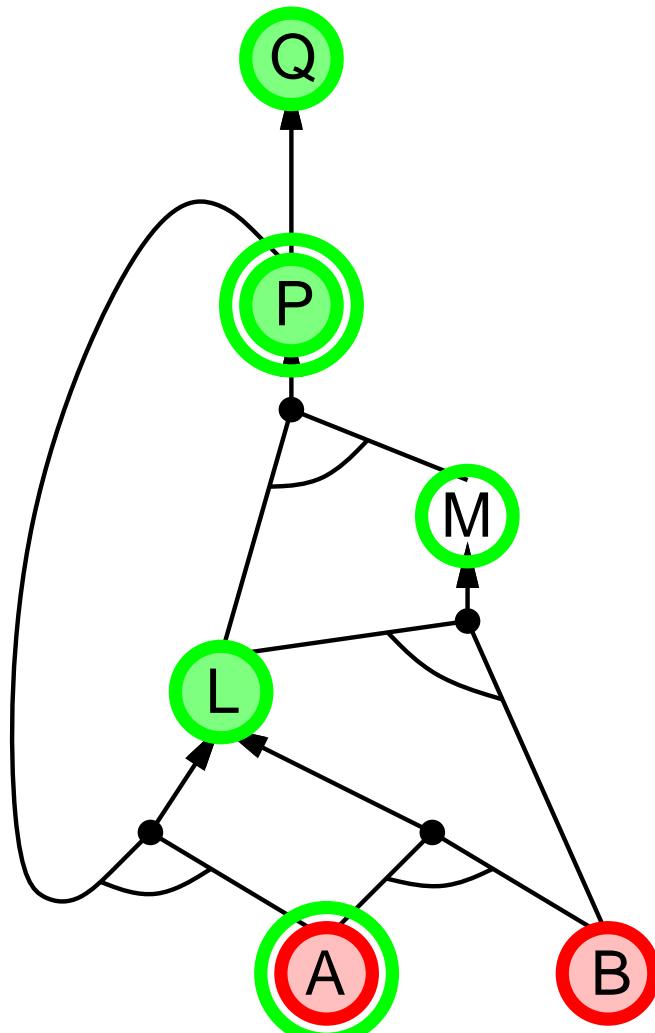
Backward chaining example



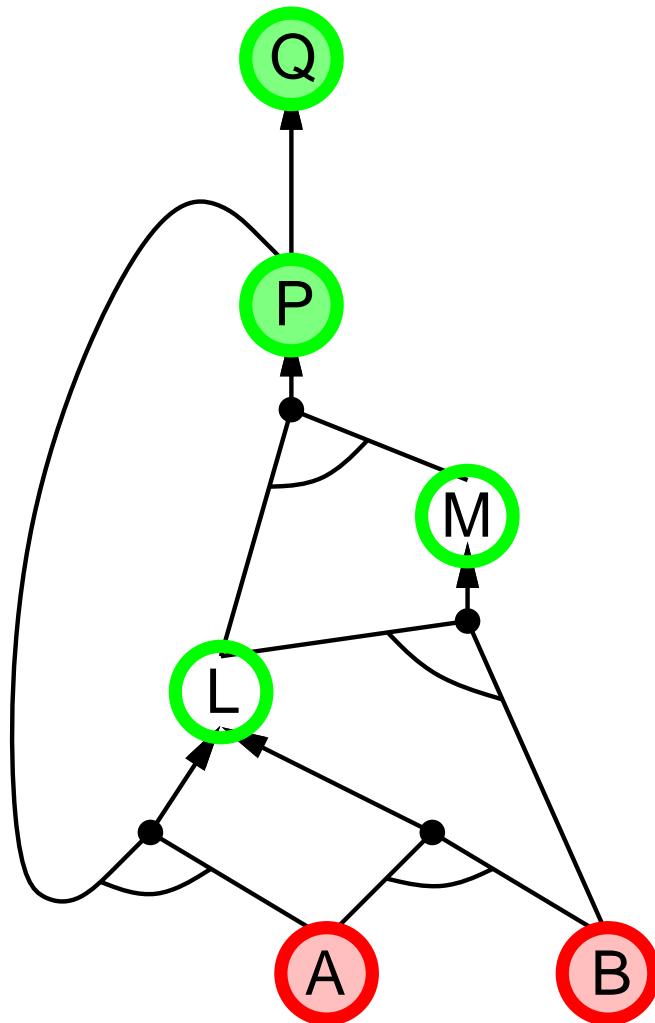
Backward chaining example



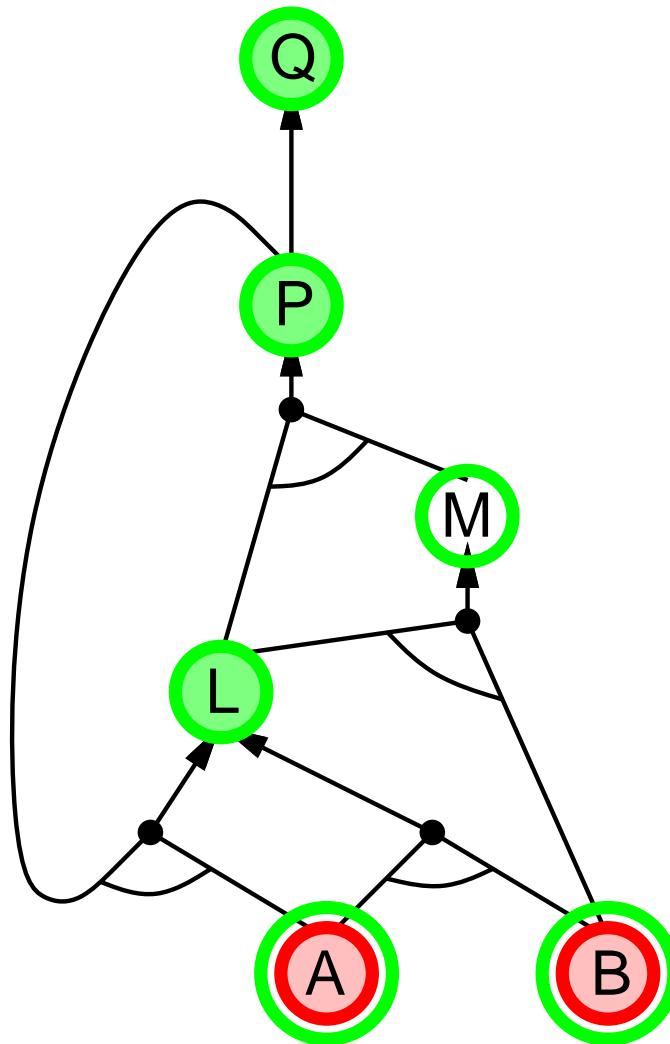
Backward chaining example



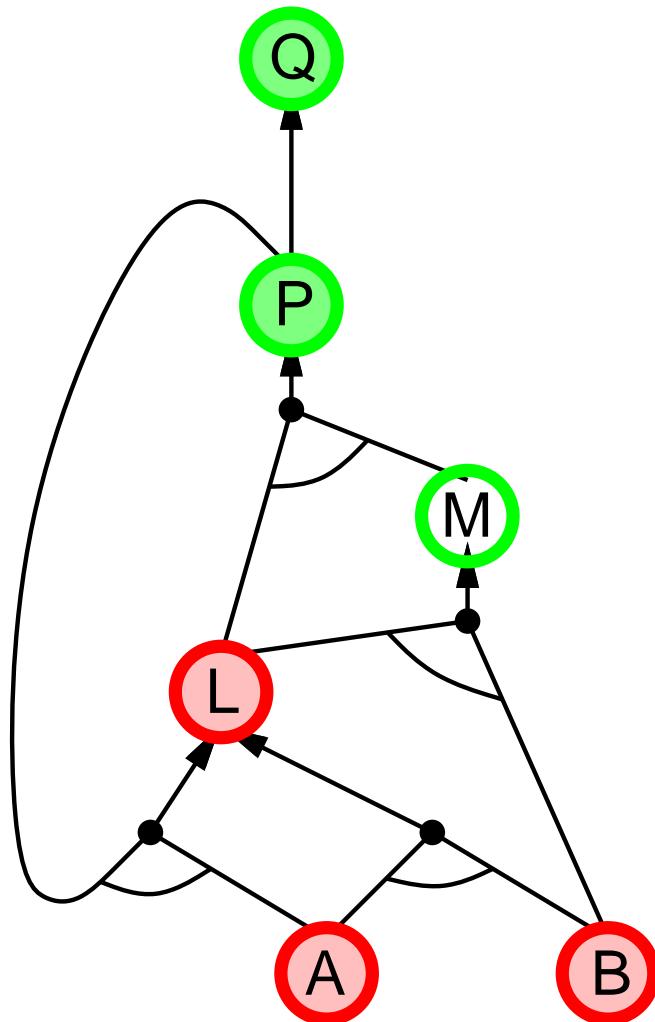
Backward chaining example



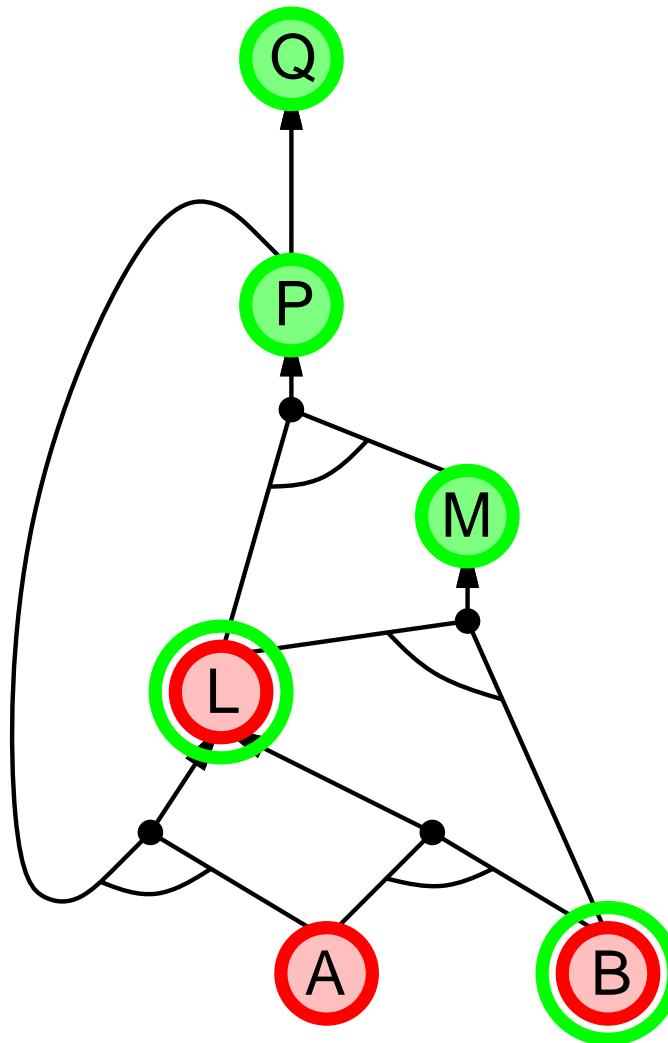
Backward chaining example



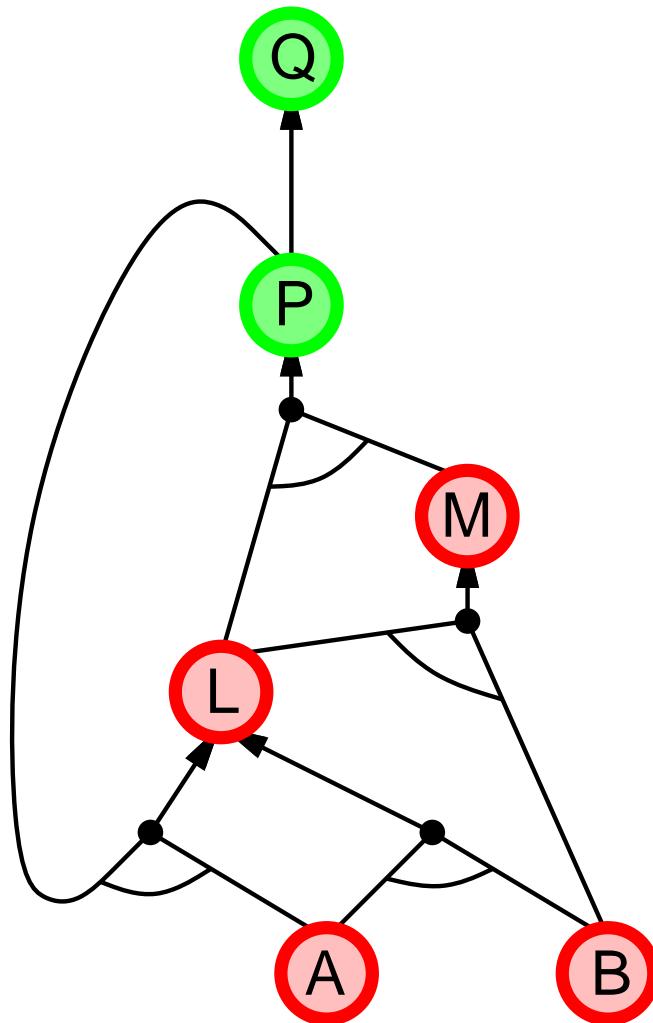
Backward chaining example



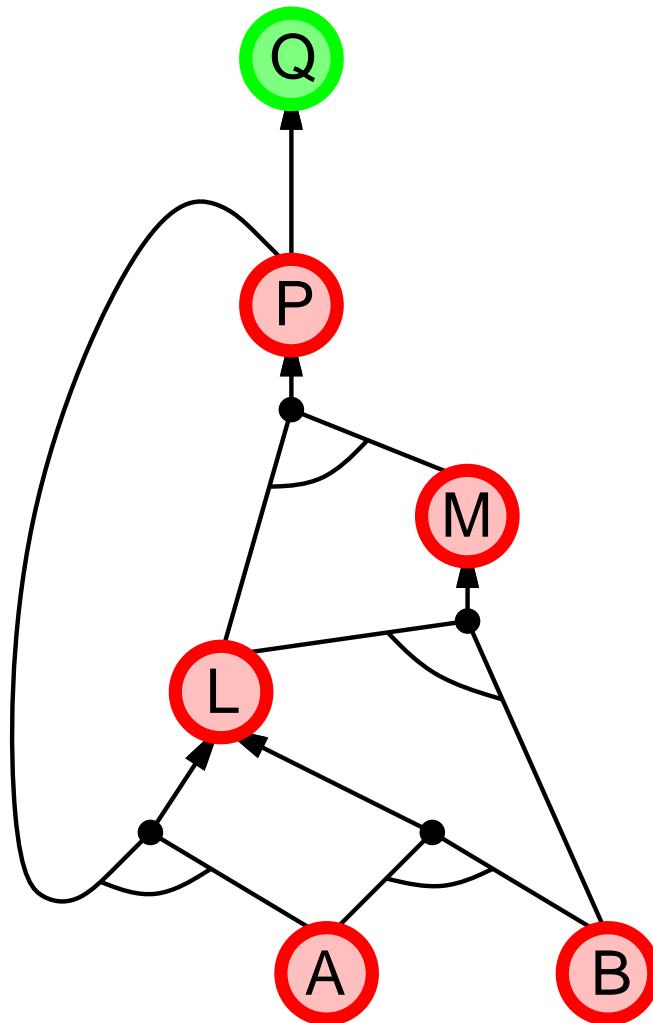
Backward chaining example



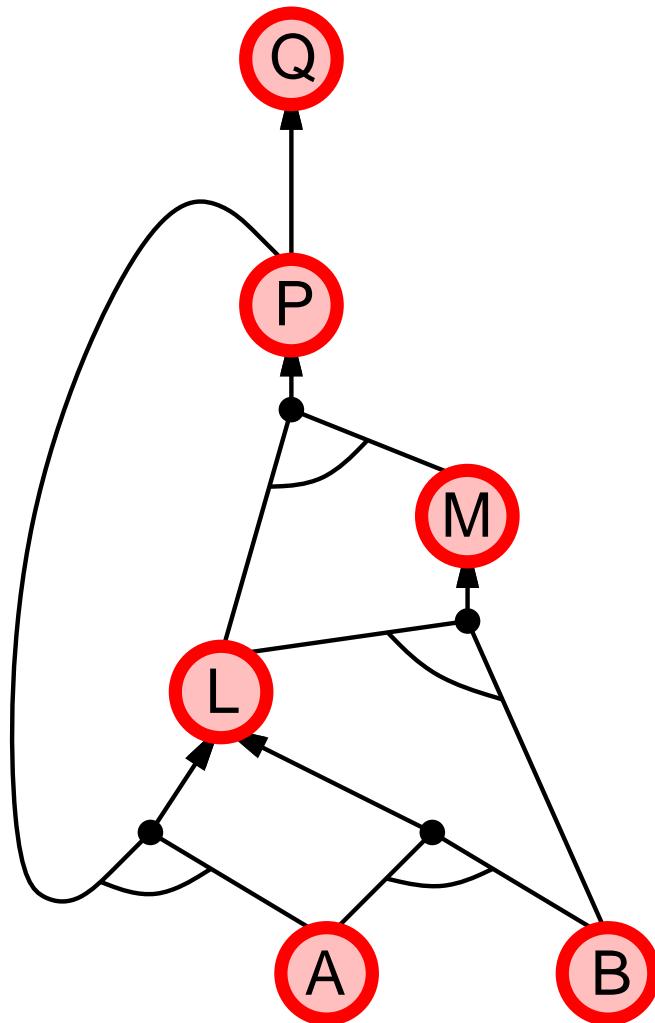
Backward chaining example



Backward chaining example



Backward chaining example



Forward vs. backward chaining

FC is **data-driven**, cf. automatic, unconscious processing,
e.g., object recognition, routine decisions

May do lots of work that is irrelevant to the goal

BC is **goal-driven**, appropriate for problem-solving,
e.g., Where are my keys? How do I get into a PhD program?

Complexity of BC can be **much less** than linear in size of KB

Resolution

Conjunctive Normal Form (CNF—universal)

conjunction of disjunctions of literals clauses

E.g., $(A \vee \neg B) \wedge (B \vee \neg C \vee \neg D)$

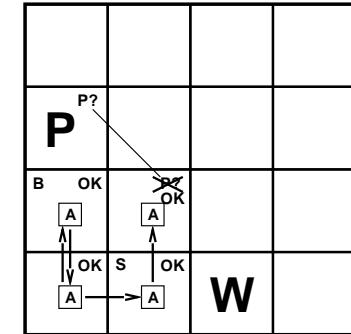
Resolution inference rule (for CNF): complete for propositional logic

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

where ℓ_i and m_j are complementary literals. E.g.,

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

Resolution is sound and complete for propositional logic



Conversion to CNF

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

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

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

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

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

3. Move \neg inwards using de Morgan's rules and double-negation:

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

4. Apply distributivity law (\vee over \wedge) and flatten:

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

Resolution algorithm

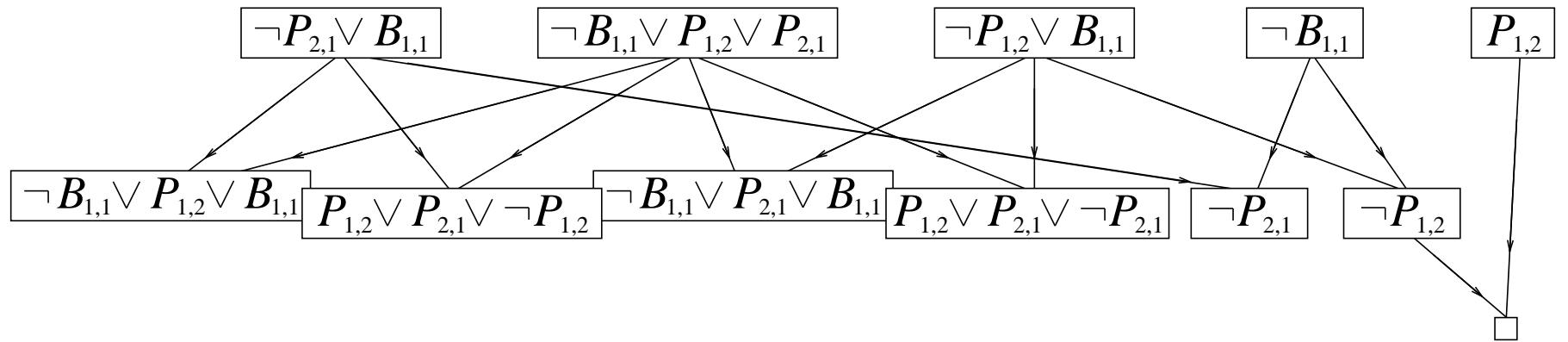
Proof by contradiction, i.e., show $KB \wedge \neg\alpha$ unsatisfiable

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

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

Resolution example

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



Summary

Logical agents apply inference to a knowledge base
to derive new information and make decisions

Basic concepts of logic:

- syntax: formal structure of sentences
- semantics: truth of sentences wrt models
- entailment: necessary truth of one sentence given another
- inference: deriving sentences from other sentences
- soundness: derivations produce only entailed sentences
- completeness: derivations can produce all entailed sentences

Wumpus world requires the ability to represent partial and negated information, reason by cases, etc.

Forward, backward chaining are linear-time, complete for Horn clauses

Resolution is complete for propositional logic

Propositional logic lacks expressive power

First Order Logic

The wumpus world example

Assertions in FOL

- A **domain** is just some part of the world about which we wish to express some knowledge.
- Sentences are added to a knowledge base using **TELL**, exactly as in propositional logic. Such sentences are called **assertions**.
- For example, we can assert that John is a king, Richard is a person, and all kings are persons:
 - `TELL(KB, King(John)) .`
 - `TELL(KB, Person(Richard)).`
 - `TELL(KB, Vx King(x) = Person(x)).`
- We can ask questions of the knowledge base using **ASK**. For example,
 - `ASK (KB, King(John))` returns true.
- Questions asked with ASK are called **queries** or **goals**.

- If we want to know what value of x makes the sentence true, we will need a different function, which we call **ASKVARS**,
 - $\text{ASKVARS}(\text{KB}, \text{Person}(x))$
- and which yields a stream of answers.
- In this case there will be two answers: $\{x/\text{John}\}$ and $\{x/\text{Richard}\}$. Such an answer is called a **substitution or binding list**.

The wumpus world

- Recall that wumpus world agent receives percepts.
- The percepts will be given to the agent program in the form of a list of five symbols;
- For example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get
 - Percept([Stench, Breeze, None, None, None]).
- The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred.
 - Percept([Stench, Breeze, Glitter, None, None],5)
- Percept is a binary predicate, and Stench and so on are constants placed in a list.

- **Actions:** Turn(Right), Turn(Left), Forward, Shoot, Grab, Climb.
- To determine which is best, the agent program executes the query
 - ASKVARs(KB, BestAction(a,5))
 which returns a binding list such as {a/Grab}.

$$\forall t, s, g, w, c \ Percept([s, Breeze, g, w, c], t) \Rightarrow Breeze(t)$$

$$\forall t, s, g, w, c \ Percept([s, None, g, w, c], t) \Rightarrow \neg Breeze(t)$$

$$\forall t, s, b, w, c \ Percept([s, b, Glitter, w, c], t) \Rightarrow Glitter(t)$$

$$\forall t, s, b, w, c \ Percept([s, b, None, w, c], t) \Rightarrow \neg Glitter(t)$$

$\exists t \ Glitter(t) \Rightarrow \text{BestAction(Grab, }t\text{)}.$

- **Objects:** squares, pits, and the wumpus.
- We could name each square—Square1, 5 and so on, but adjacent would have to be an “extra” fact, and we would need one such fact for each pair of squares.
- Adjacency of any two squares can be defined as,
- $\exists_{x,y,a,b} \text{Adjacent}([x,y],[a,b])=?$

- **Objects:** squares, pits, and the wumpus.
- We could name each square—Square1, 5 and so on, but adjacent would have to be an “extra” fact, and we would need one such fact for each pair of squares.
- Adjacency of any two squares can be defined as

$$\forall x, y, a, b \text{ } \textit{Adjacent}([x, y], [a, b]) \Leftrightarrow \\ (x = a \wedge (y = b - 1 \vee y = b + 1)) \vee (y = b \wedge (x = a - 1 \vee x = a + 1)).$$

- It is simpler to use a unary predicate Pit that is true of squares containing pits.
- Finally, since there is exactly one wumpus, a constant Wumpus is just as good as a unary predicate.
- Agents Location: At(Agent,s,r) to mean that the agent is at square s at time 1.
- We can fix the wumpus to a specific location forever with
 - $\exists t \text{ At(Wumpus, [1,3],t)}$.

- Given its current location, the agent can infer properties of the square from properties of its current percept.
- For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$$\forall s, t \ At(\text{Agent}, s, t) \wedge \text{Breeze}(t) \Rightarrow \text{Breezy}(s)$$

- Consider the following statements:

$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}).$$

$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

- What they infer in natural language?**

- Given its current location, the agent can infer properties of the square from properties of its current percept.
- For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$$\forall s, t \ At(\text{Agent}, s, t) \wedge \text{Breeze}(t) \Rightarrow \text{Breezy}(s)$$

- Consider the following statements:

$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}).$$

$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

- What they infer in natural language?:**
 - A square is breezy if and only if there is a pit in a neighboring square.

- Given its current location, the agent can infer properties of the square from properties of its current percept.
- For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$$\forall s, t \ At(\text{Agent}, s, t) \wedge \text{Breeze}(t) \Rightarrow \text{Breezy}(s)$$

- Consider the following statements:

$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}).$$

$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

- What they infer in natural language?**
 - A square is breezy if and only if there is a pit in a neighboring square.
- How do you represent the above sentence in FOL?**

- Given its current location, the agent can infer properties of the square from properties of its current percept.
- For example, if the agent is at a square and perceives a breeze, then that square is breezy:

$$\forall s, t \ At(\text{Agent}, s, t) \wedge \text{Breeze}(t) \Rightarrow \text{Breezy}(s)$$

- Consider the following statements:

$$R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}).$$

$$R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

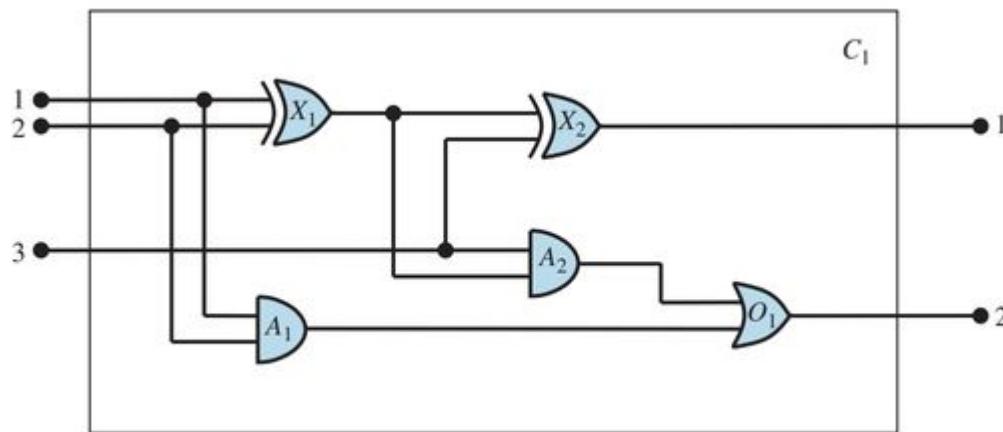
- What they infer in natural language?**
 - A square is breezy if and only if there is a pit in a neighboring square.
- How do you represent the above sentence in FOL?**

$$\forall s \ \text{Breezy}(s) \Leftrightarrow \exists r \ \text{Adjacent}(r, s) \wedge \text{Pit}(r)$$

The knowledge engineering process

1. Identify the questions.
2. Assemble the relevant knowledge.
3. Decide on a vocabulary of predicates, functions, and constants.
4. Encode general knowledge of the domain
5. Encode the specific problem instance
6. Pose queries to the inference procedure
7. Debug the knowledge base.

Example Problem: Full Adder Circuit



FIRST-ORDER LOGIC

CHAPTER 8

Outline

- ◊ Why FOL?
- ◊ Syntax and semantics of FOL
- ◊ Fun with sentences
- ◊ Wumpus world in FOL

Pros and cons of propositional logic

- 😊 Propositional logic is **declarative**: pieces of syntax correspond to facts
- 😊 Propositional logic allows partial/disjunctive/negated information
(unlike most data structures and databases)
- 😊 Propositional logic is **compositional**:
meaning of $B_{1,1} \wedge P_{1,2}$ is derived from meaning of $B_{1,1}$ and of $P_{1,2}$
- 😊 Meaning in propositional logic is **context-independent**
(unlike natural language, where meaning depends on context)
- 😢 Propositional logic has very limited expressive power
(unlike natural language)
E.g., cannot say “pits cause breezes in adjacent squares”
except by writing one sentence for each square

First-order logic

Whereas propositional logic assumes world contains **facts**, first-order logic (like natural language) assumes the world contains

- **Objects**: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries . . .
- **Relations**: red, round, bogus, prime, multistoried . . ., brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, . . .
- **Functions**: father of, best friend, third inning of, one more than, end of . . .

Logics in general

Language	Ontological Commitment	Epistemological Commitment
Propositional logic	facts	true/false/unknown
First-order logic	facts, objects, relations	true/false/unknown
Temporal logic	facts, objects, relations, times	true/false/unknown
Probability theory	facts	degree of belief
Fuzzy logic	facts + degree of truth	known interval value

Syntax of FOL: Basic elements

Constants *KingJohn, 2, UCB, ...*

Predicates *Brother, >, ...*

Functions *Sqrt, LeftLegOf, ...*

Variables *x, y, a, b, ...*

Connectives $\wedge \vee \neg \Rightarrow \Leftrightarrow$

Equality $=$

Quantifiers $\forall \exists$

Atomic sentences

Atomic sentence = *predicate*($term_1, \dots, term_n$)
or $term_1 = term_2$

Term = *function*($term_1, \dots, term_n$)
or *constant* or *variable*

E.g., *Brother(KingJohn, RichardTheLionheart)*
> (Length(LeftLegOf(Richard)), Length(LeftLegOf(KingJohn)))

Complex sentences

Complex sentences are made from atomic sentences using connectives

$$\neg S, \quad S_1 \wedge S_2, \quad S_1 \vee S_2, \quad S_1 \Rightarrow S_2, \quad S_1 \Leftrightarrow S_2$$

E.g. *Sibling(KingJohn, Richard) \Rightarrow Sibling(Richard, KingJohn)*

$$>(1, 2) \vee \leq(1, 2)$$

$$>(1, 2) \wedge \neg >(1, 2)$$

Truth in first-order logic

Sentences are true with respect to a **model** and an **interpretation**

Model contains ≥ 1 objects (**domain elements**) and relations among them

Interpretation specifies referents for

constant symbols \rightarrow objects

predicate symbols \rightarrow relations

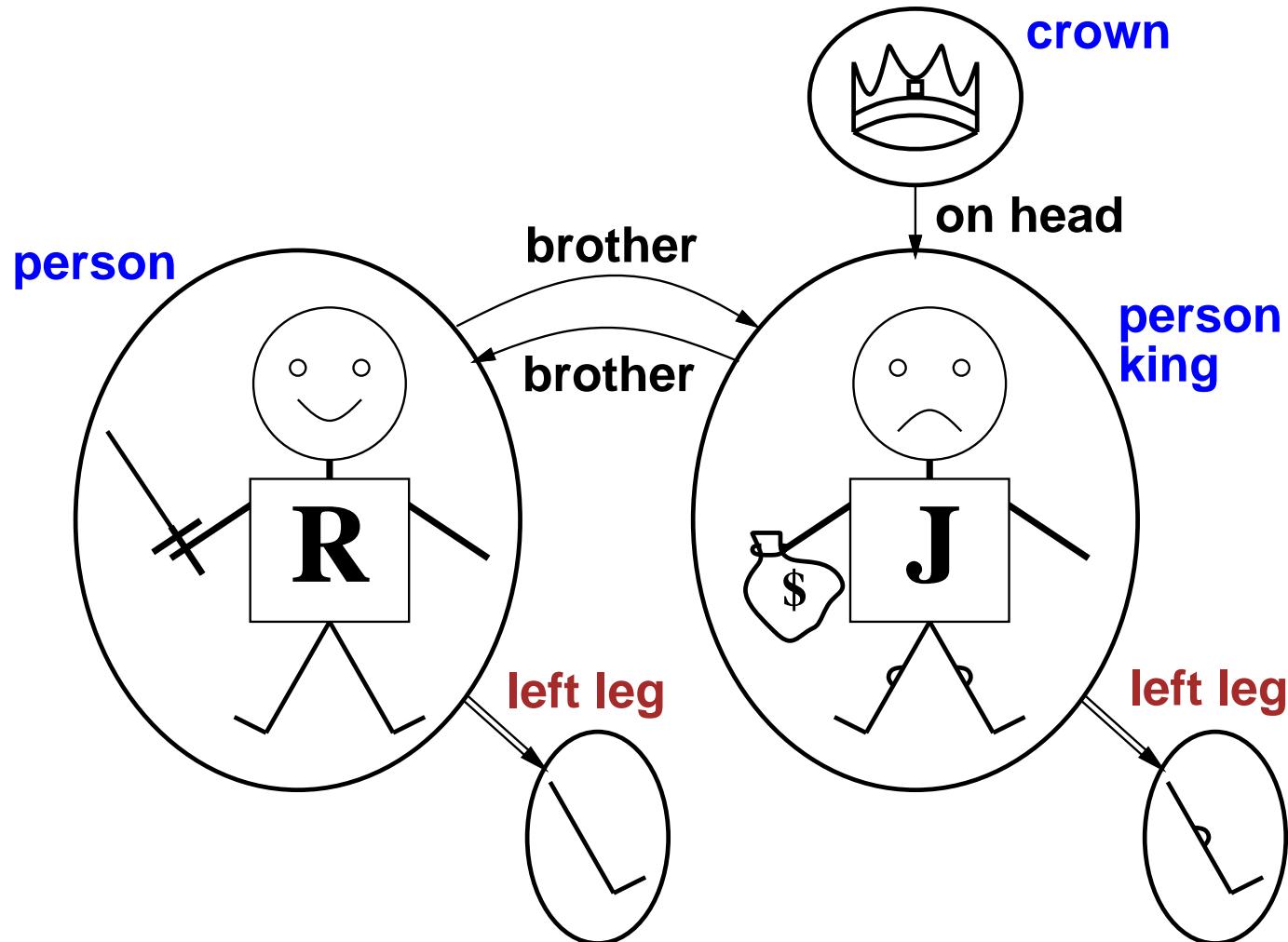
function symbols \rightarrow functional relations

An atomic sentence *predicate*(*term*₁, …, *term*_{*n*}) is true

iff the **objects** referred to by *term*₁, …, *term*_{*n*}

are in the **relation** referred to by *predicate*

Models for FOL: Example



Truth example

Consider the interpretation in which

Richard → Richard the Lionheart

John → the evil King John

Brother → the brotherhood relation

Under this interpretation, *Brother(Richard, John)* is true just in case Richard the Lionheart and the evil King John are in the brotherhood relation in the model

Models for FOL: Lots!

Entailment in propositional logic can be computed by enumerating models

We **can** enumerate the FOL models for a given KB vocabulary:

For each number of domain elements n from 1 to ∞

 For each k -ary predicate P_k in the vocabulary

 For each possible k -ary relation on n objects

 For each constant symbol C in the vocabulary

 For each choice of referent for C from n objects . . .

Computing entailment by enumerating FOL models is not easy!

Universal quantification

$\forall \langle variables \rangle \ \langle sentence \rangle$

Everyone at Berkeley is smart:

$\forall x \ At(x, Berkeley) \Rightarrow Smart(x)$

$\forall x \ P$ is true in a model m iff P is true with x being each possible object in the model

Roughly speaking, equivalent to the conjunction of instantiations of P

$$\begin{aligned} & (At(KingJohn, Berkeley) \Rightarrow Smart(KingJohn)) \\ & \wedge (At(Richard, Berkeley) \Rightarrow Smart(Richard)) \\ & \wedge (At(Berkeley, Berkeley) \Rightarrow Smart(Berkeley)) \\ & \wedge \dots \end{aligned}$$

A common mistake to avoid

Typically, \Rightarrow is the main connective with \forall

Common mistake: using \wedge as the main connective with \forall :

$$\forall x \ At(x, Berkeley) \wedge Smart(x)$$

means “Everyone is at Berkeley and everyone is smart”

Existential quantification

$\exists \langle \text{variables} \rangle \ \langle \text{sentence} \rangle$

Someone at Stanford is smart:

$\exists x \ At(x, \text{Stanford}) \wedge \text{Smart}(x)$

$\exists x \ P$ is true in a model m iff P is true with x being some possible object in the model

Roughly speaking, equivalent to the disjunction of instantiations of P

$$\begin{aligned} & (At(\text{KingJohn}, \text{Stanford}) \wedge \text{Smart}(\text{KingJohn})) \\ \vee & (At(\text{Richard}, \text{Stanford}) \wedge \text{Smart}(\text{Richard})) \\ \vee & (At(\text{Stanford}, \text{Stanford}) \wedge \text{Smart}(\text{Stanford})) \\ \vee & \dots \end{aligned}$$

Another common mistake to avoid

Typically, \wedge is the main connective with \exists

Common mistake: using \Rightarrow as the main connective with \exists :

$$\exists x \ At(x, Stanford) \Rightarrow Smart(x)$$

is true if there is anyone who is not at Stanford!

Properties of quantifiers

$\forall x \ \forall y$ is the same as $\forall y \ \forall x$ (why??)

$\exists x \ \exists y$ is the same as $\exists y \ \exists x$ (why??)

$\exists x \ \forall y$ is **not** the same as $\forall y \ \exists x$

$\exists x \ \forall y \ Loves(x, y)$

“There is a person who loves everyone in the world”

$\forall y \ \exists x \ Loves(x, y)$

“Everyone in the world is loved by at least one person”

Quantifier duality: each can be expressed using the other

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

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

Fun with sentences

Brothers are *siblings*

Fun with sentences

Brothers are *siblings*

$$\forall x, y \text{ } Brother(x, y) \Rightarrow Sibling(x, y).$$

“*Sibling*” is symmetric

Fun with sentences

Brothers are *siblings*

$$\forall x, y \text{ } Brother(x, y) \Rightarrow Sibling(x, y).$$

“*Sibling*” is symmetric

$$\forall x, y \text{ } Sibling(x, y) \Leftrightarrow Sibling(y, x).$$

One's mother is one's female parent

Fun with sentences

Brothers are *siblings*

$$\forall x, y \text{ } Brother(x, y) \Rightarrow Sibling(x, y).$$

“*Sibling*” is symmetric

$$\forall x, y \text{ } Sibling(x, y) \Leftrightarrow Sibling(y, x).$$

One's mother is one's female parent

$$\forall x, y \text{ } Mother(x, y) \Leftrightarrow (Female(x) \wedge Parent(x, y)).$$

A first cousin is a child of a parent's sibling

Fun with sentences

Brothers are siblings

$$\forall x, y \text{ } Brother(x, y) \Rightarrow Sibling(x, y).$$

“Sibling” is symmetric

$$\forall x, y \text{ } Sibling(x, y) \Leftrightarrow Sibling(y, x).$$

One's mother is one's female parent

$$\forall x, y \text{ } Mother(x, y) \Leftrightarrow (Female(x) \wedge Parent(x, y)).$$

A first cousin is a child of a parent's sibling

$$\forall x, y \text{ } FirstCousin(x, y) \Leftrightarrow \exists p, ps \text{ } Parent(p, x) \wedge Sibling(ps, p) \wedge Parent(ps, y)$$

Equality

$\text{term}_1 = \text{term}_2$ is true under a given interpretation
if and only if term_1 and term_2 refer to the same object

E.g., $1 = 2$ and $\forall x \times (\text{Sqrt}(x), \text{Sqrt}(x)) = x$ are satisfiable
 $2 = 2$ is valid

E.g., definition of (full) *Sibling* in terms of *Parent*:

$$\forall x, y \text{ } \textit{Sibling}(x, y) \Leftrightarrow [\neg(x = y) \wedge \exists m, f \neg(m = f) \wedge \\ \textit{Parent}(m, x) \wedge \textit{Parent}(f, x) \wedge \textit{Parent}(m, y) \wedge \textit{Parent}(f, y)]$$

Interacting with FOL KBs

Suppose a wumpus-world agent is using an FOL KB
and perceives a smell and a breeze (but no glitter) at $t = 5$:

$\text{Tell}(KB, \text{Percept}([\text{Smell}, \text{Breeze}, \text{None}], 5))$

$\text{Ask}(KB, \exists a \text{ Action}(a, 5))$

I.e., does KB entail any particular actions at $t = 5$?

Answer: Yes, $\{a/\text{Shoot}\}$ \leftarrow substitution (binding list)

Given a sentence S and a substitution σ ,

$S\sigma$ denotes the result of plugging σ into S ; e.g.,

$S = \text{Smarter}(x, y)$

$\sigma = \{x/\text{Hillary}, y/\text{Bill}\}$

$S\sigma = \text{Smarter}(\text{Hillary}, \text{Bill})$

$\text{Ask}(KB, S)$ returns some/all σ such that $KB \models S\sigma$

Knowledge base for the wumpus world

“Perception”

$$\forall b, g, t \ Percept([Smell, b, g], t) \Rightarrow Smelt(t)$$

$$\forall s, b, t \ Percept([s, b, Glitter], t) \Rightarrow AtGold(t)$$

Reflex: $\forall t \ AtGold(t) \Rightarrow Action(Grab, t)$

Reflex with internal state: do we have the gold already?

$$\forall t \ AtGold(t) \wedge \neg Holding(Gold, t) \Rightarrow Action(Grab, t)$$

$Holding(Gold, t)$ cannot be observed

\Rightarrow keeping track of change is essential

Deducing hidden properties

Properties of locations:

$$\forall x, t \ At(\text{Agent}, x, t) \wedge \text{Smelt}(t) \Rightarrow \text{Smelly}(x)$$

$$\forall x, t \ At(\text{Agent}, x, t) \wedge \text{Breeze}(t) \Rightarrow \text{Breezy}(x)$$

Squares are breezy near a pit:

Diagnostic rule—infer cause from effect

$$\forall y \ \text{Breezy}(y) \Rightarrow \exists x \ \text{Pit}(x) \wedge \text{Adjacent}(x, y)$$

Causal rule—infer effect from cause

$$\forall x, y \ \text{Pit}(x) \wedge \text{Adjacent}(x, y) \Rightarrow \text{Breezy}(y)$$

Neither of these is complete—e.g., the causal rule doesn't say whether squares far away from pits can be breezy

Definition for the *Breezy* predicate:

$$\forall y \ \text{Breezy}(y) \Leftrightarrow [\exists x \ \text{Pit}(x) \wedge \text{Adjacent}(x, y)]$$

Keeping track of change

Facts hold in **situations**, rather than eternally

E.g., *Holding(Gold, Now)* rather than just *Holding(Gold)*

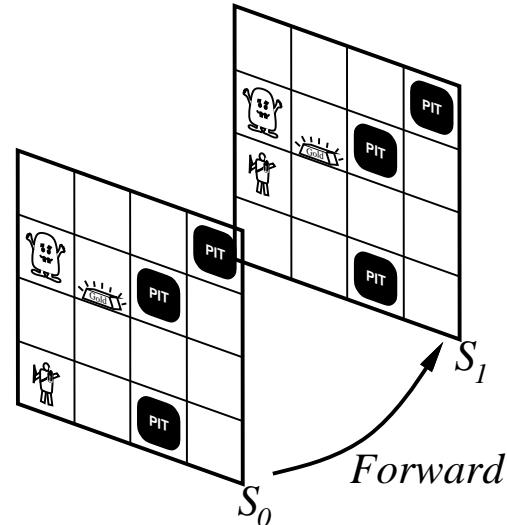
Situation calculus is one way to represent change in FOL:

Adds a situation argument to each non-eternal predicate

E.g., *Now* in *Holding(Gold, Now)* denotes a situation

Situations are connected by the *Result* function

Result(a, s) is the situation that results from doing *a* in *s*



Describing actions I

“Effect” axiom—describe changes due to action

$$\forall s \ AtGold(s) \Rightarrow Holding(Gold, Result(Grab, s))$$

“Frame” axiom—describe **non-changes** due to action

$$\forall s \ HaveArrow(s) \Rightarrow HaveArrow(Result(Grab, s))$$

Frame problem: find an elegant way to handle non-change

- (a) representation—avoid frame axioms
- (b) inference—avoid repeated “copy-overs” to keep track of state

Qualification problem: true descriptions of real actions require endless caveats—what if gold is slippery or nailed down or . . .

Ramification problem: real actions have many secondary consequences—what about the dust on the gold, wear and tear on gloves, . . .

Describing actions II

Successor-state axioms solve the representational frame problem

Each axiom is “about” a **predicate** (not an action per se):

P true afterwards \Leftrightarrow [an action made P true
 $\vee P$ true already and no action made P false]

For holding the gold:

$$\begin{aligned} \forall a, s \ Holding(Gold, Result(a, s)) &\Leftrightarrow \\ &[(a = Grab \wedge AtGold(s)) \\ &\vee (Holding(Gold, s) \wedge a \neq Release)] \end{aligned}$$

Making plans

Initial condition in KB:

$At(Agent, [1, 1], S_0)$

$At(Gold, [1, 2], S_0)$

Query: $Ask(KB, \exists s \ Holding(Gold, s))$

i.e., in what situation will I be holding the gold?

Answer: $\{s / Result(Grab, Result(Forward, S_0))\}$

i.e., go forward and then grab the gold

This assumes that the agent is interested in plans starting at S_0 and that S_0 is the only situation described in the KB

Making plans: A better way

Represent **plans** as action sequences $[a_1, a_2, \dots, a_n]$

$\text{PlanResult}(p, s)$ is the result of executing p in s

Then the query $\text{Ask}(KB, \exists p \text{ Holding(Gold, PlanResult}(p, S_0)))$
has the solution $\{p/[Forward, Grab]\}$

Definition of PlanResult in terms of Result :

$$\forall s \text{ } \text{PlanResult}([], s) = s$$

$$\forall a, p, s \text{ } \text{PlanResult}([a|p], s) = \text{PlanResult}(p, \text{Result}(a, s))$$

Planning systems are special-purpose reasoners designed to do this type of inference more efficiently than a general-purpose reasoner

Summary

First-order logic:

- objects and relations are semantic primitives
- syntax: constants, functions, predicates, equality, quantifiers

Increased expressive power: sufficient to define wumpus world

Situation calculus:

- conventions for describing actions and change in FOL
- can formulate planning as inference on a situation calculus KB

Introduction to Learning

Introduction to Learning

- An agent is learning if it improves its performance after making observations about the world.
- Learning can range from the trivial, such as jotting down a shopping list, to the profound, as when Albert Einstein inferred a new theory of the universe.
- When the agent is a computer, we call it **machine learning**: a
 - computer observes some data,
 - builds a model based on the data,
 - uses the model as both a hypothesis about the world and a piece of software that can solve problems.

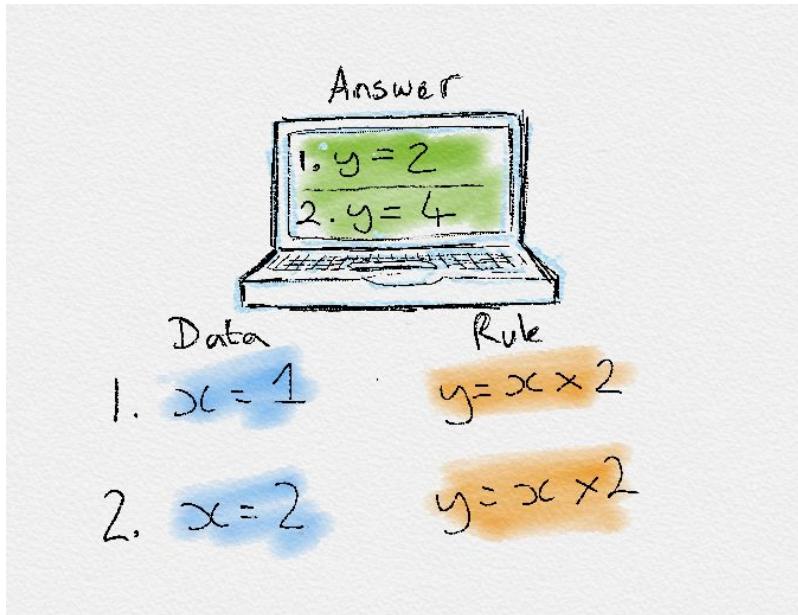
Why Machine Learning?

- Why would we want a machine to learn?
- Why not just program it the right way to begin with?

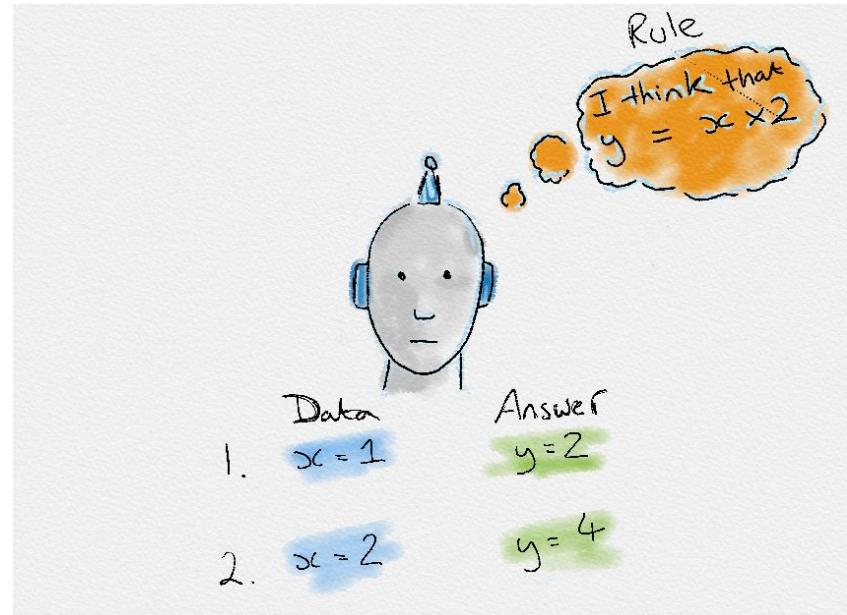
What is ML?

- Machine learning (ML) is the study of computer algorithms that improve automatically through experience.
- Machine-learning algorithms use statistics to find patterns in massive amounts of data.
- Traditionally, software engineering combined human created rules with data to create answers to a problem. Instead, machine learning uses data and answers to discover the rules behind a problem – **F. Chollet, Deep Learning with Python**

What is ML?



Traditional Programming



Machine Learning

Terminologies used in ML

- ML systems learn how to make inference from the input data samples to produce useful predictions on un-seen (test) data.
- Input data:
 - labelled examples: A labelled example includes feature(s) and the label. {features, label}: (x, y)
For e.g.:

Features:	Label
Normal RBC, Normal HgB	Healthy
Low RBC, Low HgB	Anaemic
 - unlabelled examples: An unlabelled example contains features but not the label. {features, ?}: (x, ?)
 - For e.g.:

Features:	Label
Housing type: 4BHK, Price: 40,000	
Housing type: 4BHK, Price: 15,000	
Housing type: 2BHK, Price: 25,000	

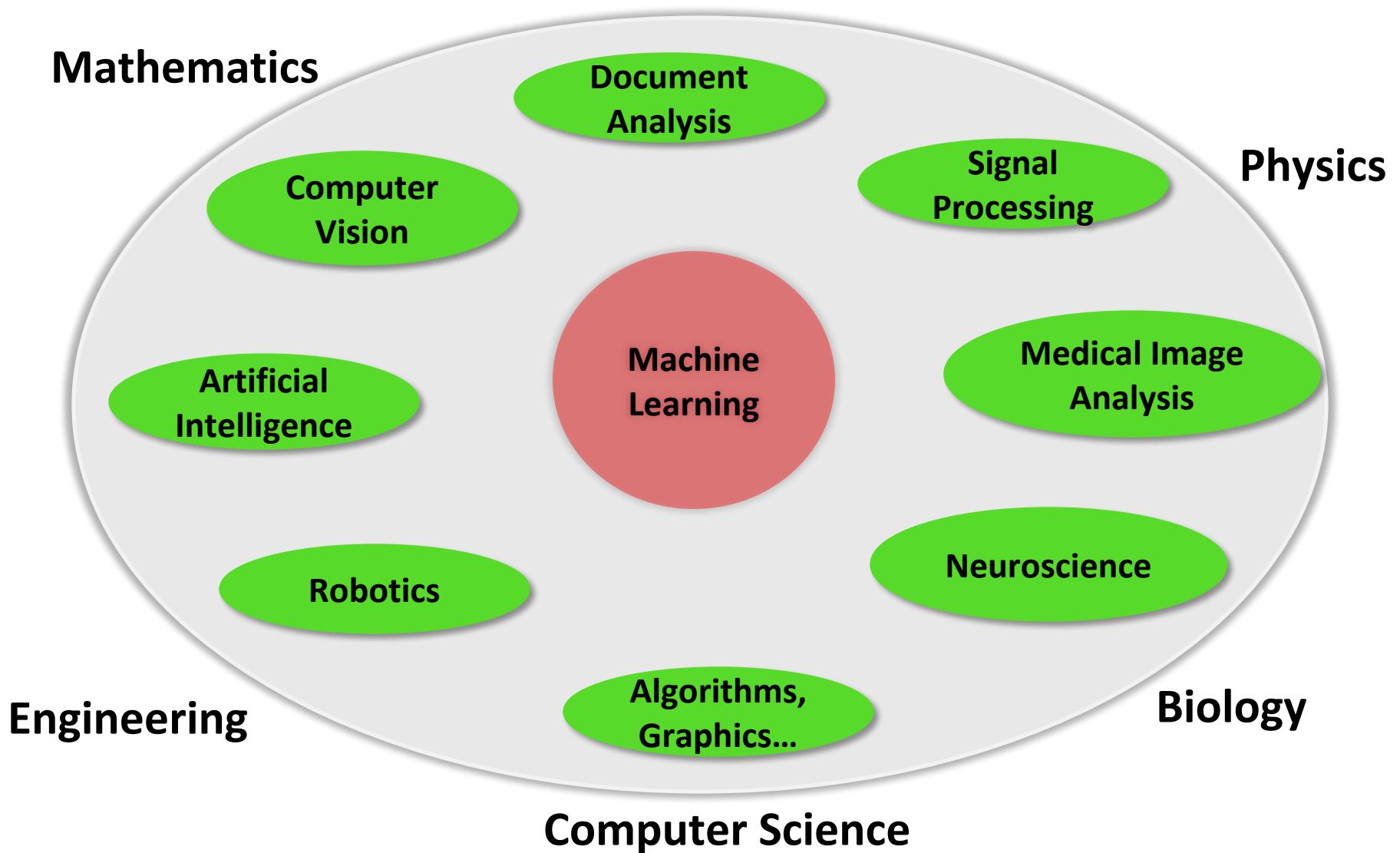
Terminologies used in ML

- Machine Learning Model:
 - A ML model defines the relationship between the features and label.
 - For e.g.: An anaemia diagnostic model might associate certain features strongly with “anaemic” or “healthy”, and predict the labels based on the association rules it inferred.
 - Two Phases of ML model development
 - **Training** means creating or **learning** the model.
 - **Testing/Inference** means applying the trained model to unlabelled examples.

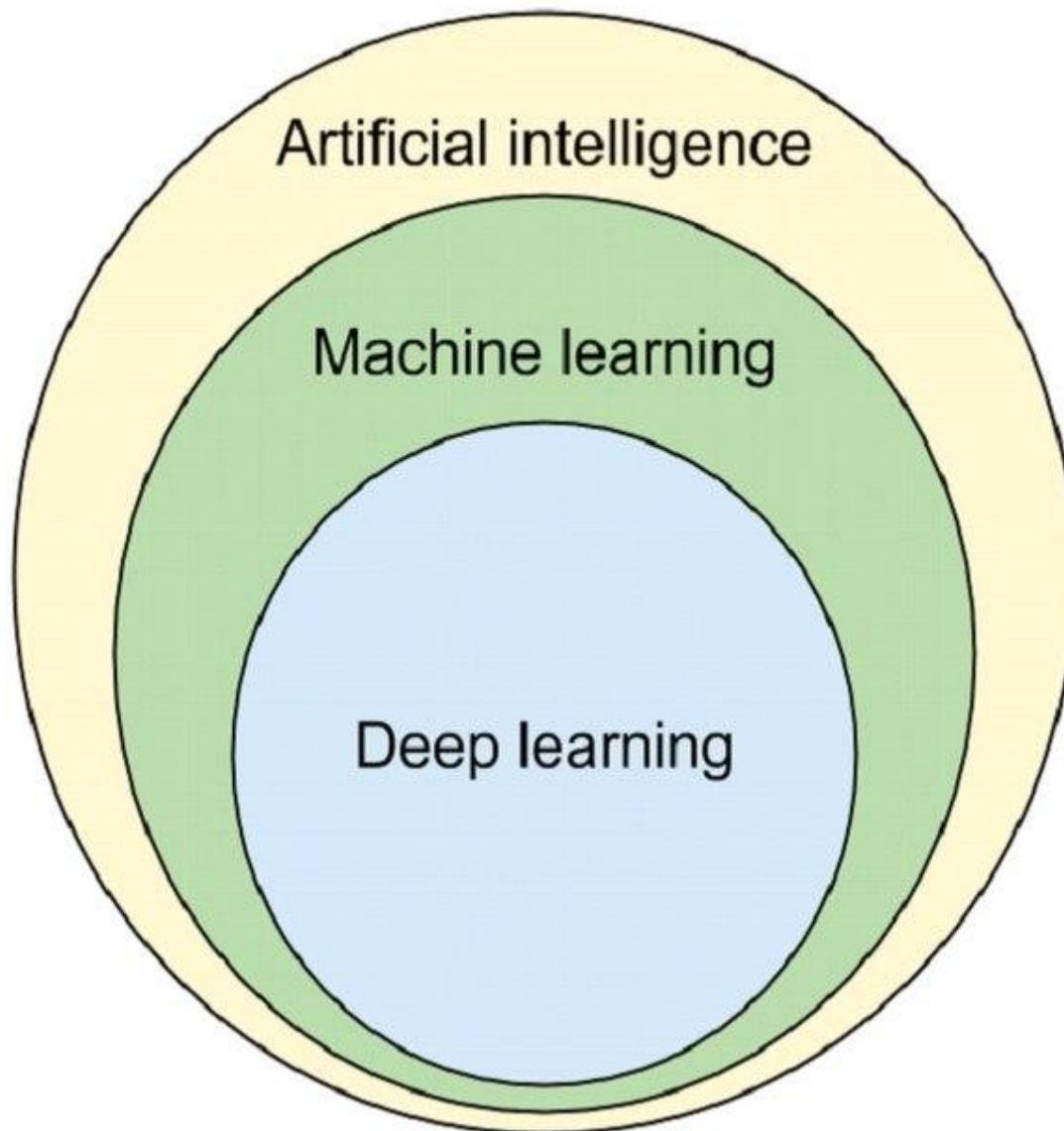
Applications

- Hand-written digit recognition
- Speech recognition
- Face detection
- Object classification
- Email spam detection
- Computational biology
- Autonomous cars
- Computer-aided diagnosis

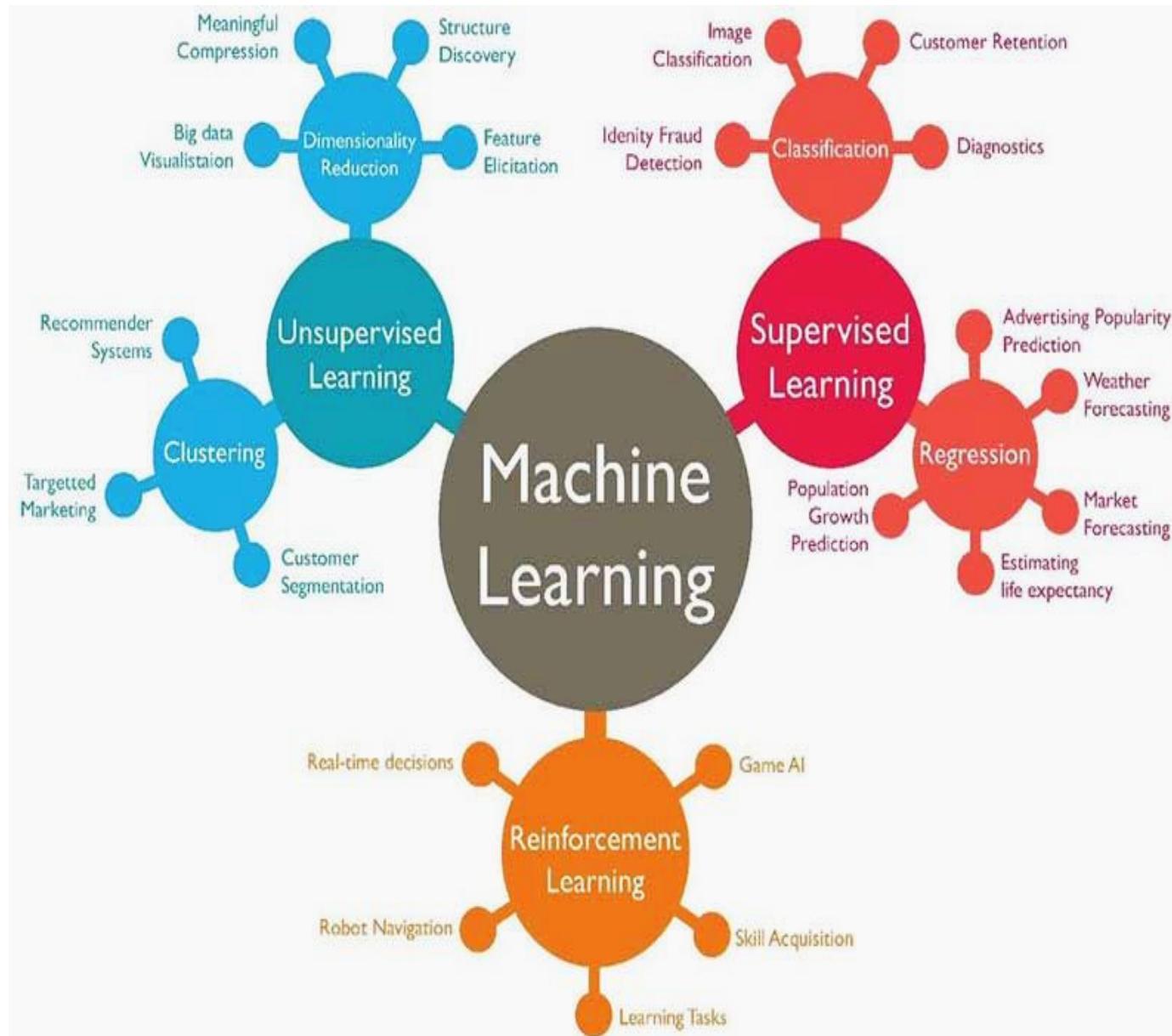
Relation with Other Fields



Relation with AI, ML and DL



Different Machine Learning Paradigms



Bayesian Learning

- ML works with data and hypotheses.
- Here, the data are evidence—that is, instantiations of some or all of the random variables describing the domain.
- The hypotheses are probabilistic theories of how the domain works, including logical theories as a special case.
- **Bayesian learning** simply calculates the probability of each hypothesis, given the data, and makes predictions on that basis.
 - i.e, the predictions are made by using all the hypotheses, weighted by their probabilities, rather than by using just a single “best” hypothesis.

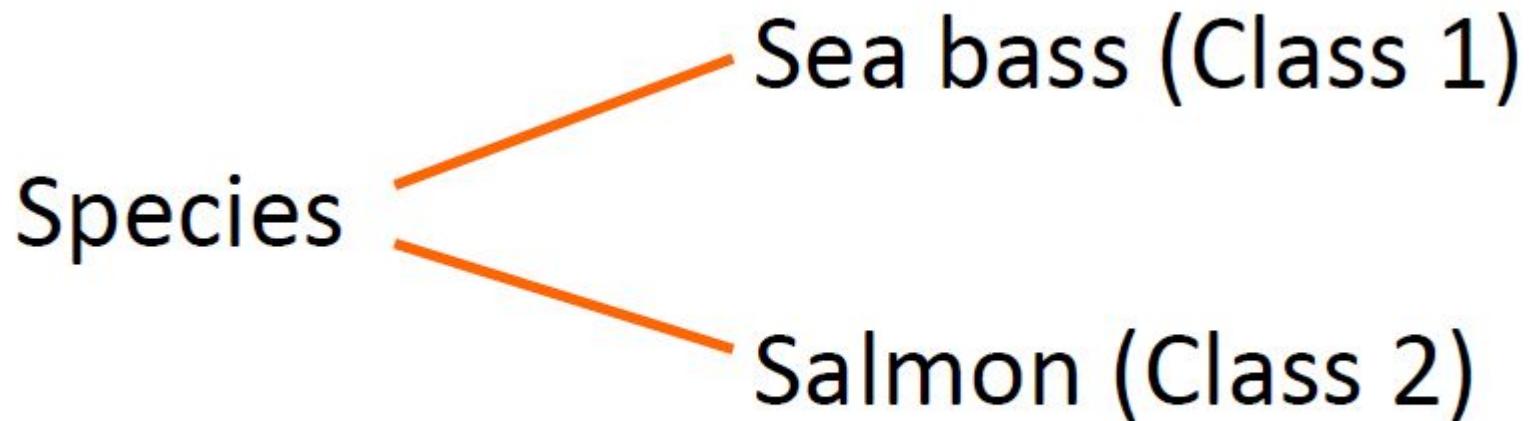
Bayesian Learning

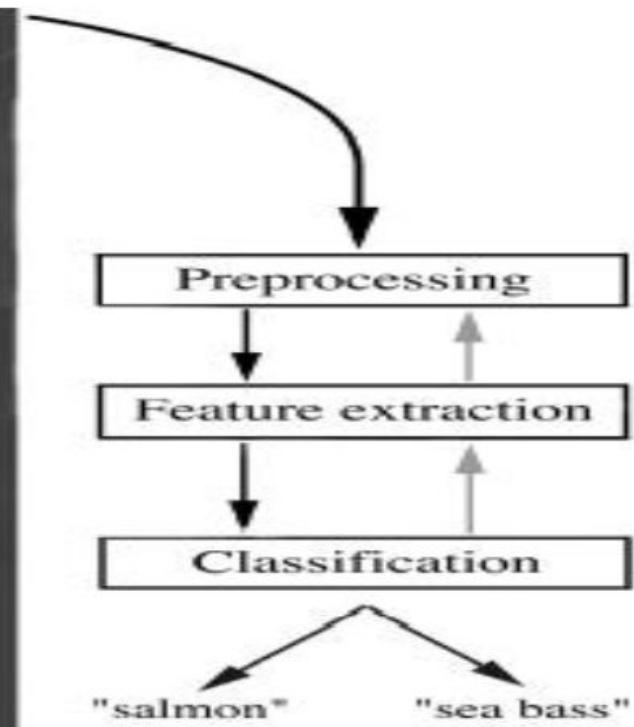
- **Marginal Probability:** The probability of an event irrespective of the outcomes of other random variables, e.g. $P(A)$.
- **Joint Probability:** Probability of two (or more) simultaneous events, e.g. $P(A \text{ and } B)$ or $P(A, B)$.
- **Conditional Probability:** Probability of one (or more) event given the occurrence of another event, e.g. $P(A \text{ given } B)$ or $P(A | B)$

- The joint probability can be calculated using the conditional probability; for example:
$$P(A, B) = P(A | B) * P(B)$$
- This is called the product rule. Importantly, the joint probability is symmetrical, meaning that:
$$P(A, B) = P(B, A)$$
- The conditional probability can be calculated using the joint probability; for example:
$$P(A | B) = P(A, B) / P(B)$$
- The conditional probability is not symmetrical; for example:
$$P(A | B) \neq P(B | A)$$

An Example

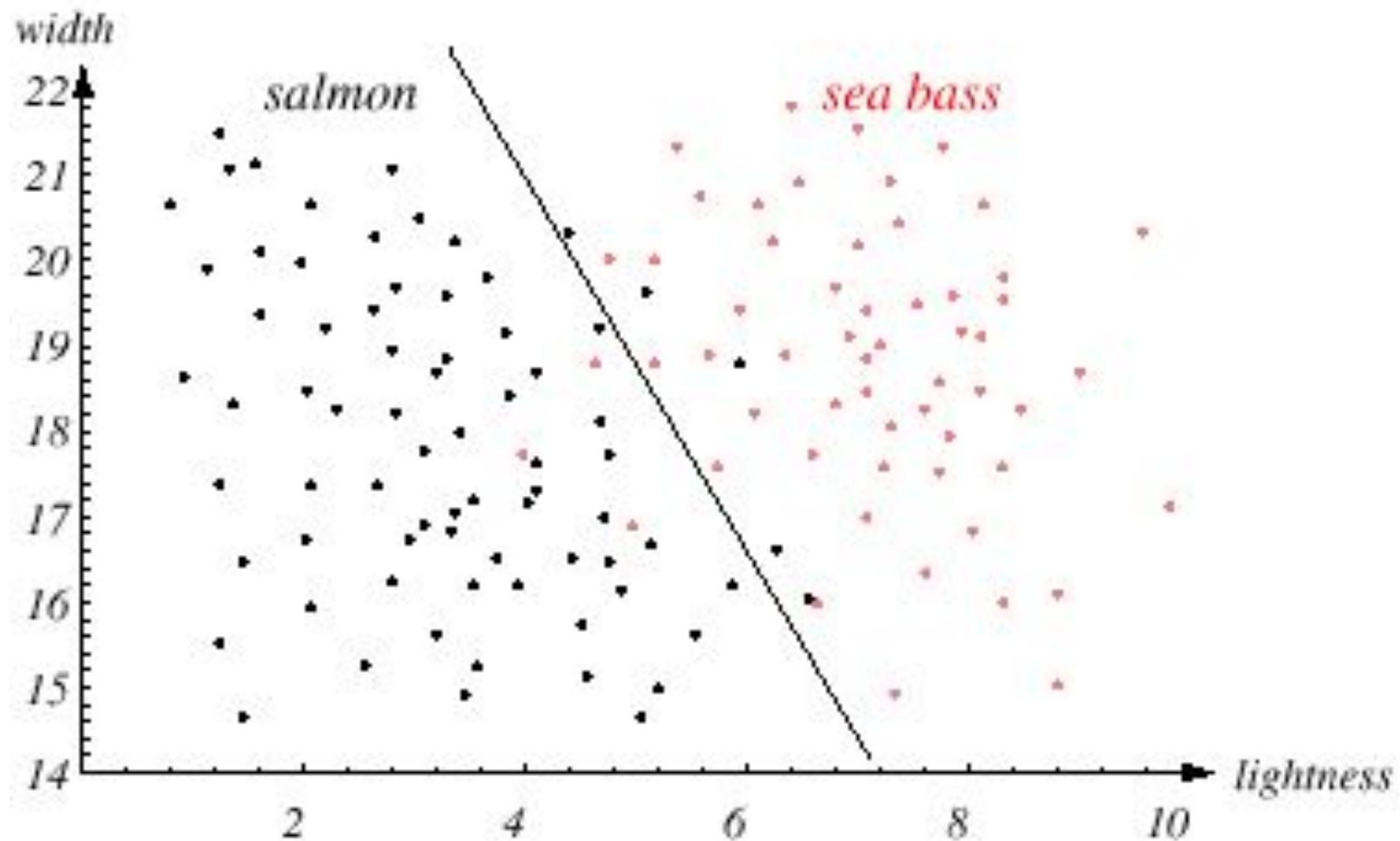
- “Sorting incoming Fish on a conveyor according to species using optical sensing”





Problem Analysis

- Set up a camera and take some sample images to extract features like
 - Length of the fish
 - Lightness (based on the gray level)
 - Width of the fish



- The sea bass/salmon example (a two class problem)
- For example if we randomly catch 100 fishes and out of this if 75 are *sea bass* and 25 are *salmon*.
-
- Let the rule, in this case is: For any fish say its class is *sea bass*.
- **What is the error rate of this rule?**
- This information which is independent of feature values is called **apriori** knowledge.

Let the two classes are ω_1 and ω_2

- $P(\omega_1) + P(\omega_2) = 1$
- State of nature (class) is a random variable
- If $P(\omega_1) = P(\omega_2)$, we say it is of uniform priors
 - The catch of salmon and sea bass is equi-probable

- Decision rule with only the prior information
 - Decide ω_1 if $P(\omega_1) > P(\omega_2)$, otherwise decide ω_2
- *This is not a good classifier.*
- *We should take feature values into account !*
- *If x is the pattern we want to classify, then use the rule:*

If $P(\omega_1 | x) > P(\omega_2 | x)$ then assign class ω_1
Else assign class ω_2
- *$P(\omega_1 | x)$ is called posteriori probability of class ω_1 given that the pattern is x .*

Bayes rule

- From data it might be possible for us to estimate $p(x | \mathcal{C}_j)$, where $i = 1$ or 2 . These are called class-conditional distributions.
- Also it is easy to find apriori probabilities $P(\mathcal{C}_1)$ and $P(\mathcal{C}_2)$. How this can be done?
- Bayes rule combines apriori probability with class conditional distributions to find posteriori probabilities.

Bayes rule

$$P(B|A) = \frac{P(A, B)}{P(A)} = \frac{P(A|B) * P(B)}{P(A)}$$

This is Bayes Rule



Bayes, Thomas (1763) An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society of London*, **53**:370-418

$$P(\omega_j | x) = \frac{p(x | \omega_j) \cdot P(\omega_j)}{p(x)}$$

– Where in case of two categories

$$p(x) = \sum_{j=1}^{j=2} p(x | \omega_j) P(\omega_j)$$

Likelihood . Prior

– Posterior = $\frac{\text{Likelihood . Prior}}{\text{Evidence}}$

Decision given the posterior probabilities

X is an observation for which:

if $P(\omega_1 | x) > P(\omega_2 | x)$  True state of nature = ω_1
if $P(\omega_1 | x) < P(\omega_2 | x)$  True state of nature = ω_2

Therefore:

whenever we observe a particular x, the probability of error is :

$P(\text{error} | x) = P(\omega_1 | x) \text{ if we decide } \omega_2$

$P(\text{error} | x) = P(\omega_2 | x) \text{ if we decide } \omega_1$

- Minimizing the probability of error
- Decide ω_1 if $P(\omega_1 | x) > P(\omega_2 | x)$;
otherwise decide ω_2

Therefore:

$$P(\text{error} | x) = \min [P(\omega_1 | x), P(\omega_2 | x)]$$

(error of Bayes decision)

Consider a one dimensional two class problem. The feature used is color of fish. Color can be either white or dark $P(\omega_1) = 0.75$, $P(\omega_2) = 0.25$, $P(\text{white} | \omega_1) = 0.2$, $P(\text{white} | \omega_2) = 0.6$, $P(\text{dark} | \omega_1) = 0.8$, $P(\text{dark} | \omega_2) = 0.4$. Find $P(\text{error})$ of the Bayes Classifier.

$$P(\text{white}) = P(\text{white} | \omega_1)P(\omega_1) + P(\text{white} | \omega_2)P(\omega_2)$$

$$P(\text{white}) = 0.2 * 0.75 + 0.6 * 0.25 = 0.3$$

Consider a one dimensional two class problem. The feature used is color of fish. Color can be either white or dark $P(\omega_1) = 0.75$, $P(\omega_2) = 0.25$, $P(\text{white} | \omega_1) = 0.2$, $P(\text{white} | \omega_2) = 0.6$, $P(\text{dark} | \omega_1) = 0.8$, $P(\text{dark} | \omega_2) = 0.4$ Find $P(\text{error})$ of the Bayes Classifier.

$$P(\text{white}) = P(\text{white} | \omega_1)P(\omega_1) + P(\text{white} | \omega_2)P(\omega_2)$$

$$P(\text{white}) = 0.2 * 0.75 + 0.6 * 0.25 = 0.3$$

$$P(\text{dark}) = P(\text{dark} | \omega_1)P(\omega_1) + P(\text{dark} | \omega_2)P(\omega_2)$$

$$P(\text{dark}) = 0.8 * 0.75 + 0.4 * 0.25 = 0.7$$

Consider a one dimensional two class problem. The feature used is color of fish. Color can be either white or dark $P(\omega_1) = 0.75$, $P(\omega_2) = 0.25$, $P(\text{white} | \omega_1) = 0.2$, $P(\text{white} | \omega_2) = 0.6$, $P(\text{dark} | \omega_1) = 0.8$, $P(\text{dark} | \omega_2) = 0.4$. Find $P(\text{error})$ of the Bayes Classifier.

$$P(\text{white}) = P(\text{white} | \omega_1)P(\omega_1) + P(\text{white} | \omega_2)P(\omega_2)$$

$$P(\text{white}) = 0.2 * 0.75 + 0.6 * 0.25 = 0.3$$

$$P(\text{dark}) = P(\text{dark} | \omega_1)P(\omega_1) + P(\text{dark} | \omega_2)P(\omega_2)$$

$$P(\text{dark}) = 0.8 * 0.75 + 0.4 * 0.25 = 0.7$$

$$P(\omega_1 | \text{white}) = \frac{P(\text{white} | \omega_1)P(\omega_1)}{P(\text{white})} = \frac{0.2 * 0.75}{0.3} = 0.5$$

$$P(\omega_2 | \text{white}) = \frac{P(\text{white} | \omega_2)P(\omega_2)}{P(\text{white})} = \frac{0.6 * 0.25}{0.3} = 0.5$$

Consider a one dimensional two class problem. The feature used is color of fish. Color can be either white or dark $P(\omega_1) = 0.75$, $P(\omega_2) = 0.25$, $P(\text{white} | \omega_1) = 0.2$, $P(\text{white} | \omega_2) = 0.6$, $P(\text{dark} | \omega_1) = 0.8$, $P(\text{dark} | \omega_2) = 0.4$. Find $P(\text{error})$ of the Bayes Classifier.

$$P(\text{white}) = P(\text{white} | \omega_1)P(\omega_1) + P(\text{white} | \omega_2)P(\omega_2)$$

$$P(\text{white}) = 0.2 * 0.75 + 0.6 * 0.25 = 0.3$$

$$P(\text{dark}) = P(\text{dark} | \omega_1)P(\omega_1) + P(\text{dark} | \omega_2)P(\omega_2)$$

$$P(\text{dark}) = 0.8 * 0.75 + 0.4 * 0.25 = 0.7$$

$$P(\omega_1 | \text{white}) = \frac{P(\text{white} | \omega_1)P(\omega_1)}{P(\text{white})} = \frac{0.2 * 0.75}{0.3} = 0.5$$

$$P(\omega_2 | \text{white}) = \frac{P(\text{white} | \omega_2)P(\omega_2)}{P(\text{white})} = \frac{0.6 * 0.25}{0.3} = 0.5$$

$$P(\omega_1 | \text{dark}) = \frac{P(\text{dark} | \omega_1)P(\omega_1)}{P(\text{dark})} = \frac{0.8 * 0.75}{0.7} = \frac{6}{7}$$

$$P(\omega_2 | \text{dark}) = \frac{P(\text{dark} | \omega_2)P(\omega_2)}{P(\text{dark})} = \frac{0.4 * 0.25}{0.7} = \frac{1}{7}$$

$$P(error) = P(error|white)P(white) + P(error|dark)P(dark)$$

$$P(error) = 0.5 * 0.3 + \frac{1}{7} * 0.7 = 0.25$$

- But, what is the error, if we use only apriori probabilities?

Since, $P(\omega_1) = 0.75$, $P(\omega_2) = 0.25$, every pattern is assigned to ω_1 , So the error,

$$P(error) = P(\omega_2 | white)P(white) + P(\omega_2 | dark)P(dark)$$

Since, $P(\omega_1) = 0.75$, $P(\omega_2) = 0.25$, every pattern is assigned to ω_1 , So the error,

$$P(\text{error}) = P(\omega_2|\text{white})P(\text{white}) + P(\omega_2|\text{dark})P(\text{dark})$$

$$P(\text{error}) = \frac{P(\text{white}|\omega_2)P(\omega_2)}{P(\text{white})}P(\text{white}) + \frac{P(\text{dark}|\omega_2)P(\omega_2)}{P(\text{dark})}P(\text{dark})$$

$$P(\text{error}) = (P(\text{white}|\omega_2) + P(\text{dark}|\omega_2))P(\omega_2)$$

$$P(\text{error}) = P(\omega_2) = 0.25$$

- Same error? Where is the advantage?!

Consider $P(\omega_1) = 0.5$, $P(\omega_2) = 0.5$

$$P(\text{white}) = P(\text{white}|\omega_1)P(\omega_1) + P(\text{white}|\omega_2)P(\omega_2)$$

$$P(\text{white}) = 0.2 * 0.5 + 0.6 * 0.5 = 0.4$$

$$P(\text{dark}) = P(\text{dark}|\omega_1)P(\omega_1) + P(\text{dark}|\omega_2)P(\omega_2)$$

$$P(\text{dark}) = 0.8 * 0.5 + 0.4 * 0.5 = 0.6$$

$$P(\omega_1|\text{white}) = \frac{P(\text{white}|\omega_1)P(\omega_1)}{P(\text{white})} = \frac{0.2 * 0.5}{0.4} = 0.25$$

$$P(\omega_2|\text{white}) = \frac{P(\text{white}|\omega_2)P(\omega_2)}{P(\text{white})} = \frac{0.6 * 0.5}{0.4} = 0.75$$

$$P(\omega_1|\text{dark}) = \frac{P(\text{dark}|\omega_1)P(\omega_1)}{P(\text{dark})} = \frac{0.8 * 0.5}{0.6} = \frac{2}{3}$$

$$P(\omega_2|\text{dark}) = \frac{P(\text{dark}|\omega_2)P(\omega_2)}{P(\text{dark})} = \frac{0.4 * 0.5}{0.6} = \frac{1}{3}$$

$$P(\text{error}) = P(\text{error}|\text{white})P(\text{white}) + P(\text{error}|\text{dark})P(\text{dark})$$

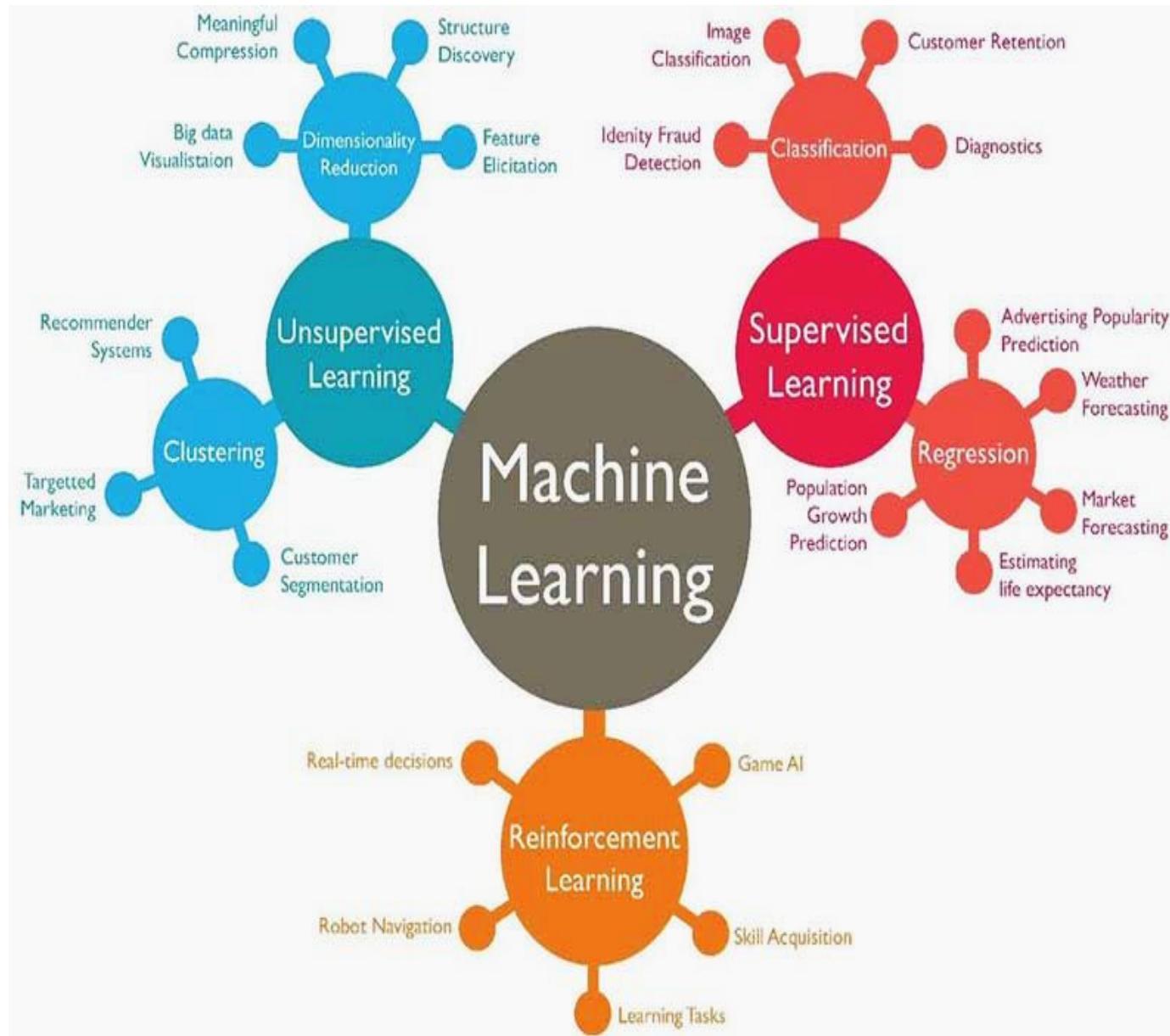
$$P(\text{error}) = 0.25 * 0.4 + \frac{1}{3} * 0.6 = 0.3$$

- But, $P(\text{error})$ based on apriori probabilities only is 0.5.
- Error based on the Bayes classifier is the lower bound.
 - Any classifier's error is greater than or equal to this.

- Read Duda and Hart book.

Supervised and Unsupervised Learning

Different Machine Learning Paradigms

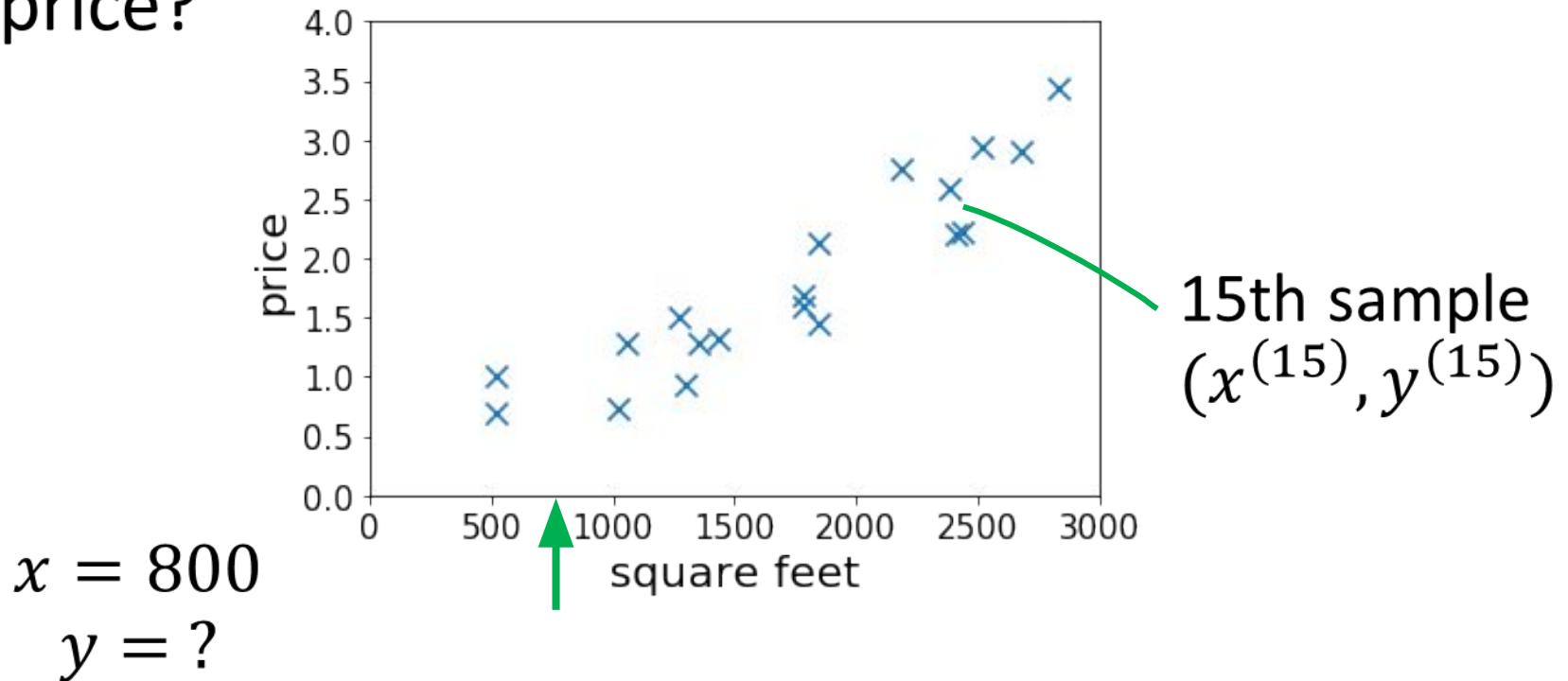


Supervised Learning

- To learn an unknown *target function f*
- Input: a *training set* of *labeled examples* (x_j, y_j) where $y_j = f(x_j)$
 - E.g., x_j is an image, $f(x_j)$ is the label “giraffe”
- Output: *hypothesis h* that is “close” to *f*, i.e., predicts well on unseen examples (“*test set*”)
- Many possible hypothesis families for *h*
 - Linear models, logistic regression, neural networks, decision trees, examples (nearest-neighbor) etc.

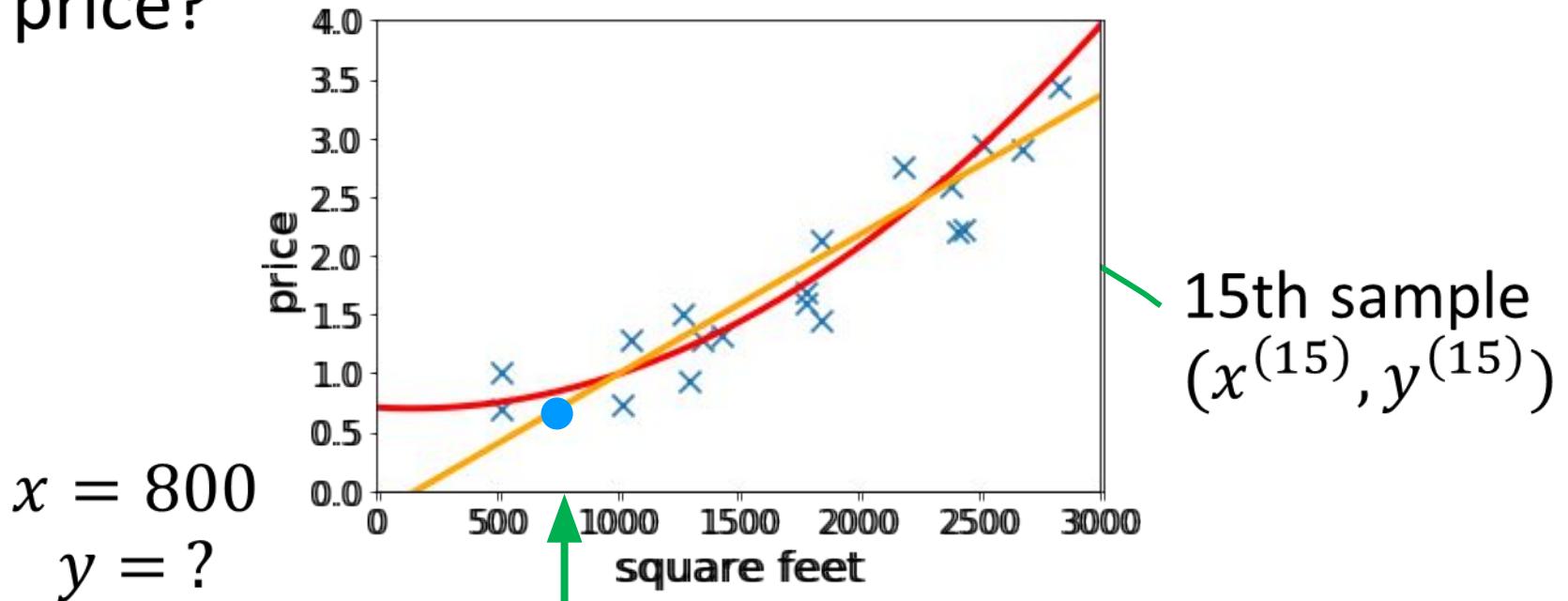
Housing Price Prediction

- Given: a dataset that contains n samples $(x^{(1)}, y^{(1)}), \dots (x^{(n)}, y^{(n)})$
- Task:** If a residence has x square feet, predict its price?



Housing Price Prediction

- Given: a dataset that contains n samples $(x^{(1)}, y^{(1)}), \dots (x^{(n)}, y^{(n)})$
- Task: If a residence has x square feet, predict its price?



- Solution: fitting linear/quadratic functions to the dataset.

High-dimensional Features

- $x \in \mathbb{R}^d$ for large d

- E.g.,

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ \vdots \\ x_d \end{bmatrix} \begin{array}{l} \text{--- living size} \\ \text{--- lot size} \\ \text{--- \# floors} \\ \text{--- condition} \\ \text{--- zip code} \\ \vdots \end{array} \xrightarrow{\hspace{1cm}} y \text{ --- price}$$

Supervised Learning in CV

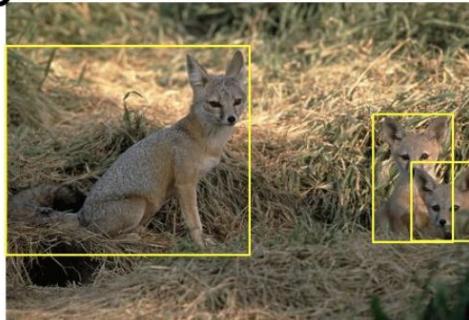
- Image Classification
 - x = raw pixels of the image, y = the main object



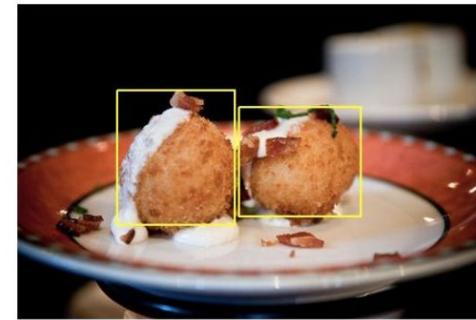
ImageNet Large Scale Visual Recognition Challenge. Russakovsky et al.'2015

Supervised Learning in CV

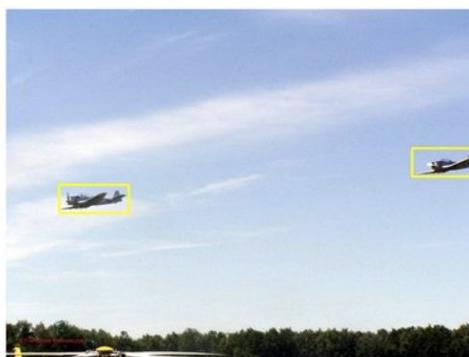
- Object localization and detection
 - x = raw pixels of the image, y = the bounding boxes



kit fox



croquette



airplane

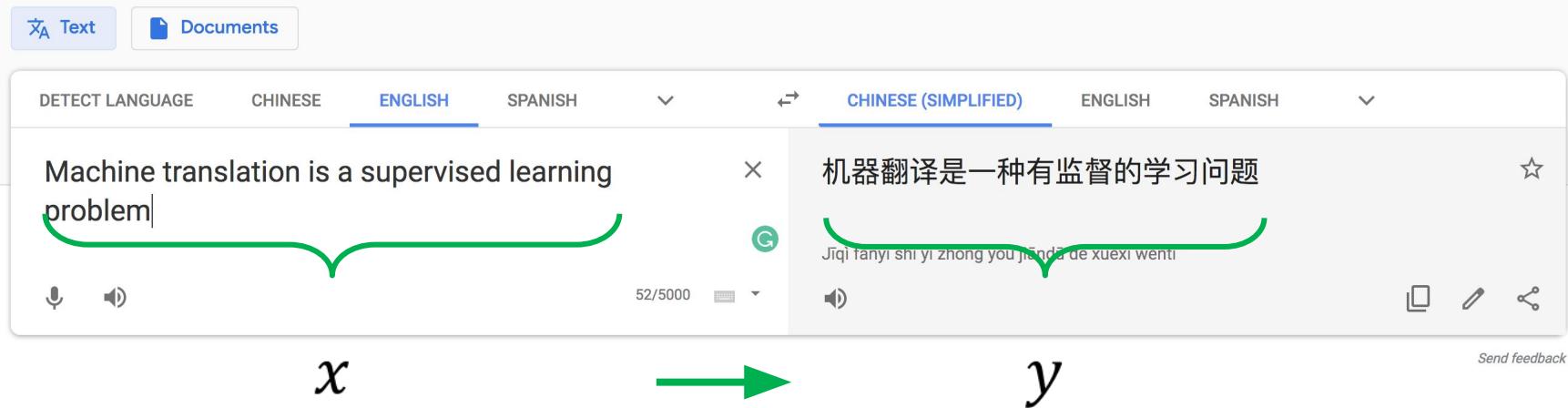


frog

Supervised Learning in Natural Language Processing

- Machine translation

Google Translate



- Note: This course only covers the basic and fundamental techniques of supervised learning (which are not enough for solving hard vision or NLP problems.)

Supervised Learning

Advantage

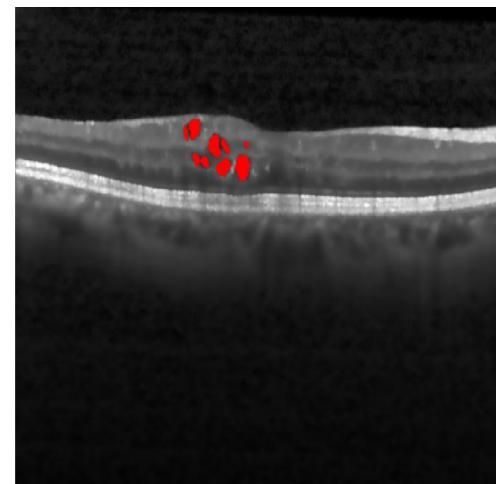
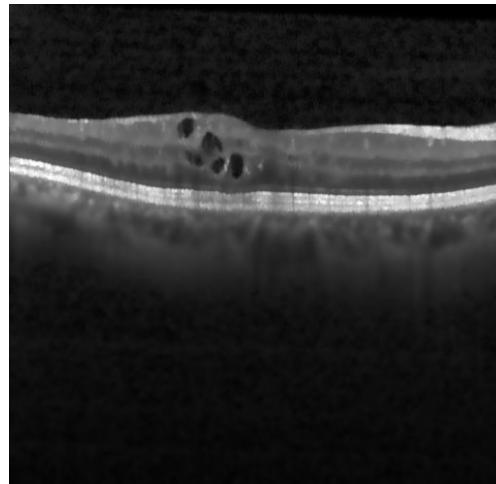
- You have full control over what the machine is learning.
- You can easily test and debug your learning machine.
 - Since the labelled data is available you can easily inspect its output and find out what errors it's making on what type of input data.

Supervised Learning

Disadvantage

- Collecting and labelling data is expensive and time-consuming.

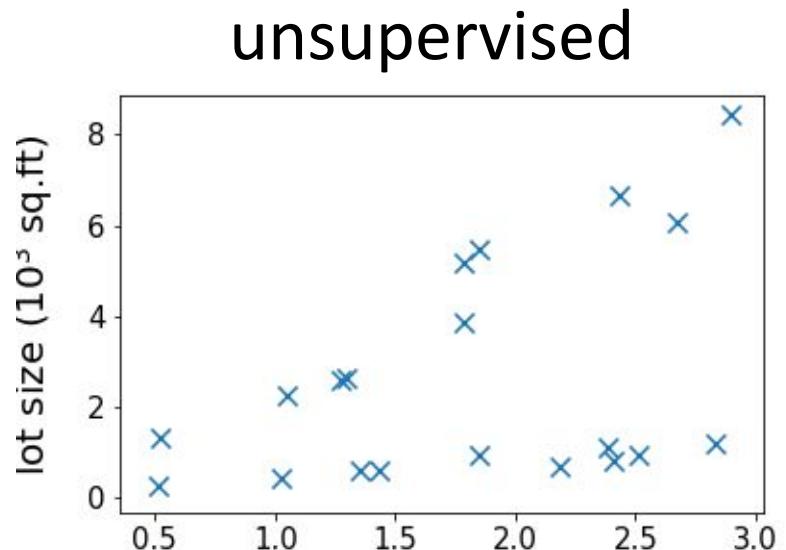
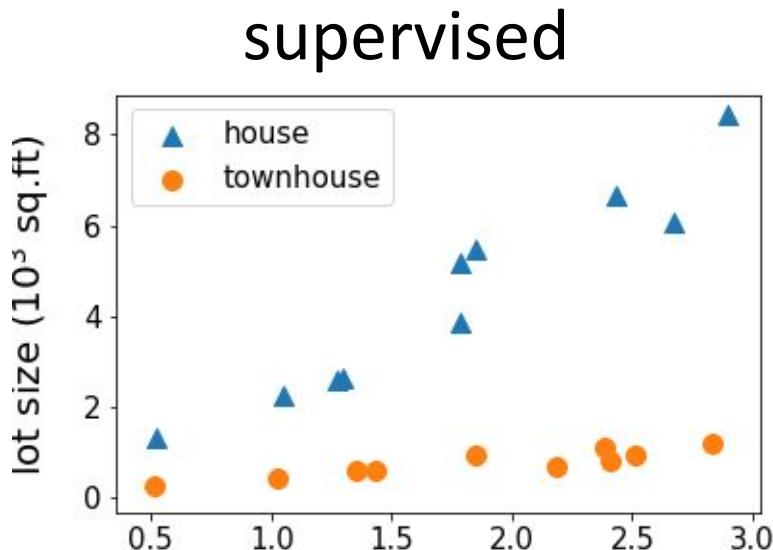
Example: Speech Recognition, Medical Image Analysis, etc.



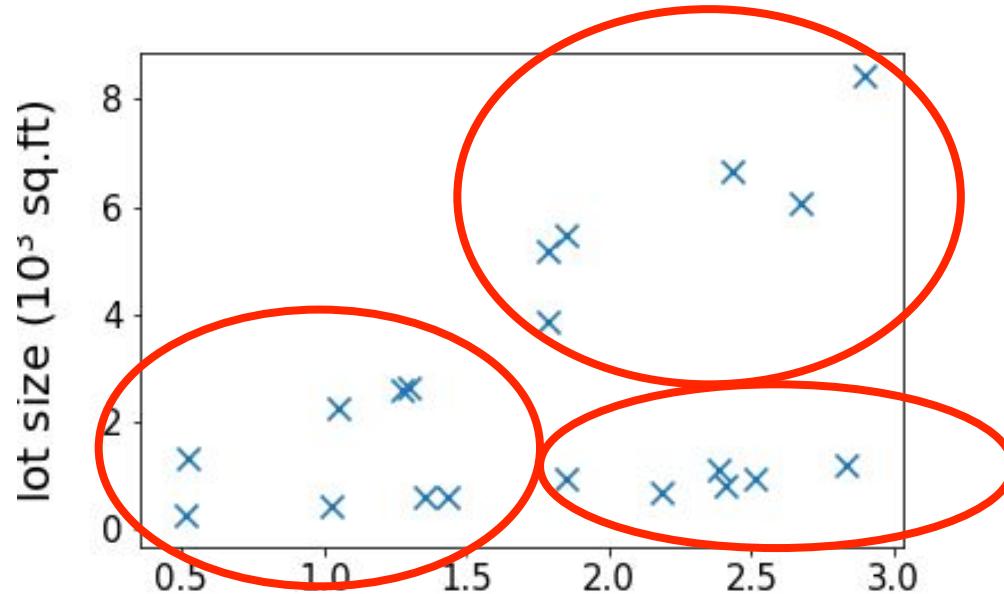
- Errors in your training data might confuse your algorithm and lower its accuracy. Garbage-in -> Garbage-out

Unsupervised Learning

- Dataset contains **no labels**: $x^{(1)}, \dots x^{(n)}$
- **Goal** (vaguely-posed): to find interesting structures in the data



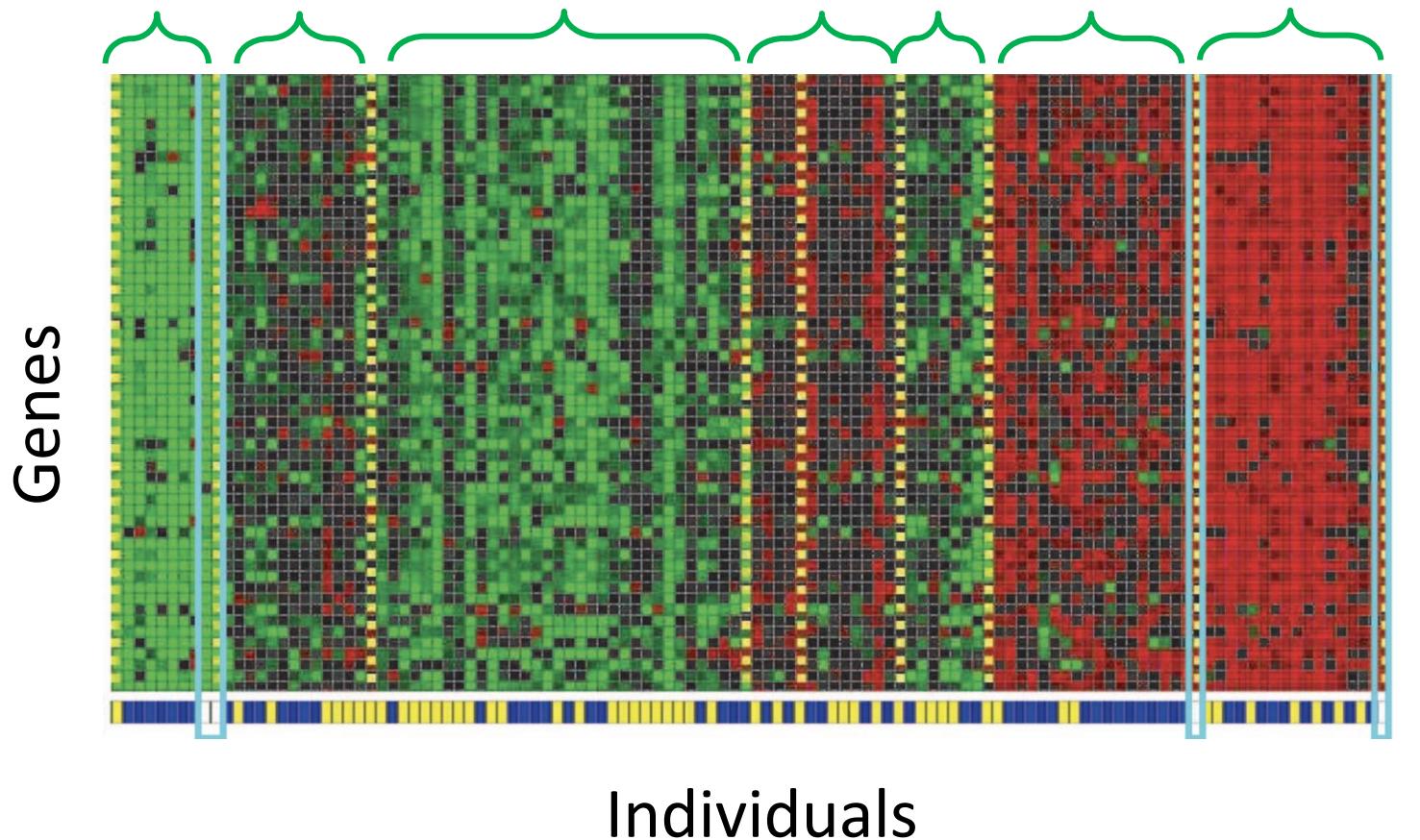
Clustering



Clustering Genes

Cluster 1

Cluster 7



Individuals

Identifying Regulatory Mechanisms using Individual Variation Reveals Key Role for Chromatin Modification. [Su-In Lee, Dana Pe'er, Aimee M. Dudley, George M. Church and Daphne Koller. '06]

Need for Unsupervised Learning

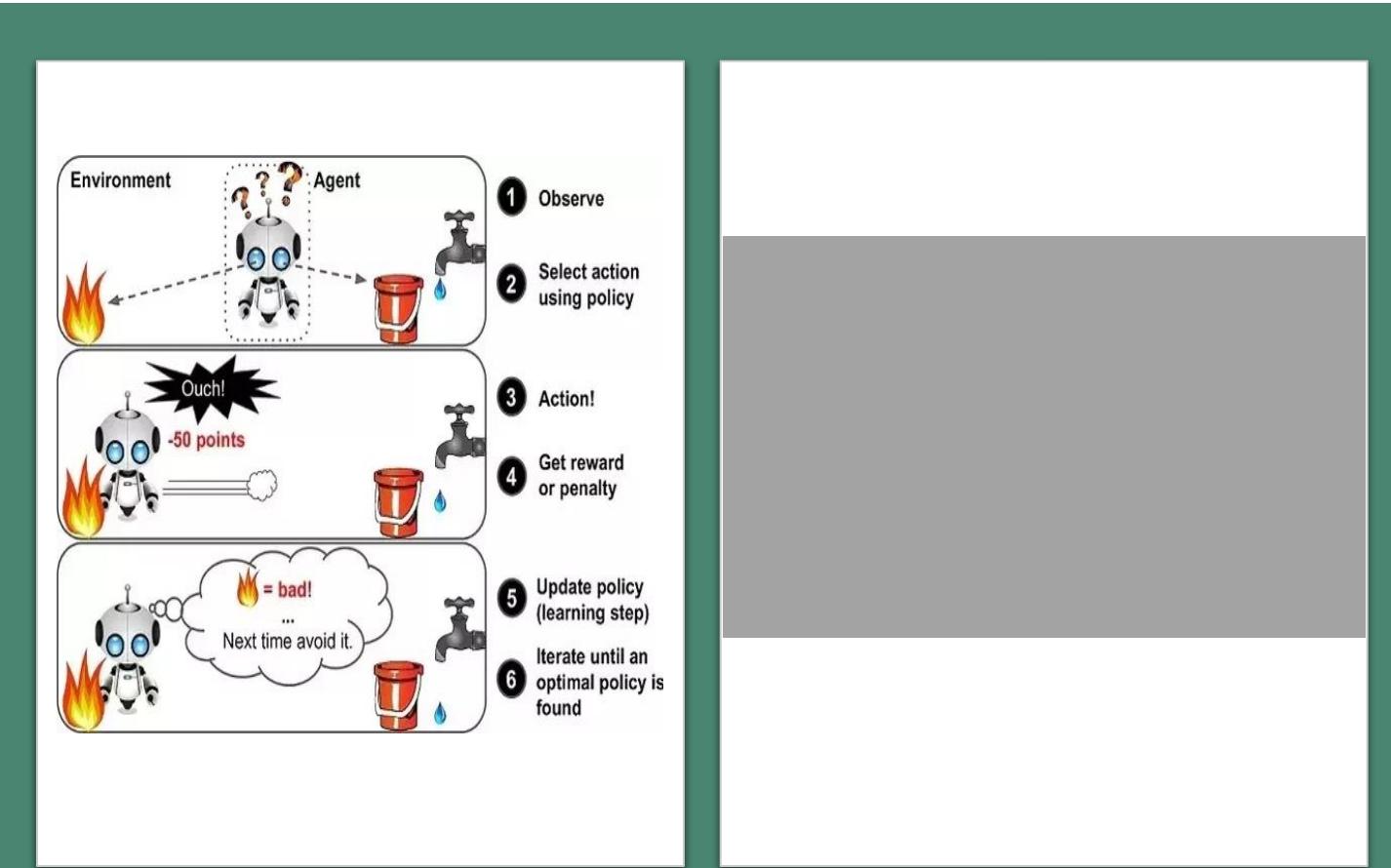
- Annotating large datasets is very costly and time consuming. Example: Speech Recognition, Medical Image Analysis, etc.
- There may be cases where we don't know how many/what classes the data divided into. Example: Data Mining, Sentimental Analysis.
- We may want to use clustering to gain some insight into the structure of the data before designing a classifier.

Disadvantages of Unsupervised Learning

- Unsupervised Learning is harder as compared to Supervised Learning. Since, making the inference is difficult due to unavailable labels.
- How do we know if results are meaningful since it has unlabelled data?
 - External evaluation- Expert analysis.
 - Internal evaluation- Objective function.

Reinforcement Learning

A reinforcement learning algorithm, or agent, learns by interacting with its environment. The agent receives rewards by performing correctly and penalties for performing incorrectly.



Reinforcement learning example: Fire fighting agent

Need for Reinforcement Learning

- Reinforcement learning can be used to solve very complex problems that cannot be solved by conventional techniques.
- In the absence of a training dataset, it is bound to learn from its experience.
- Reinforcement learning models can outperform humans in many tasks and learning process is similar to human learning.
- DeepMind's AlphaGo program, a reinforcement learning model, beat the world champion *Lee Sedol* at the game of *Go* in March 2016.

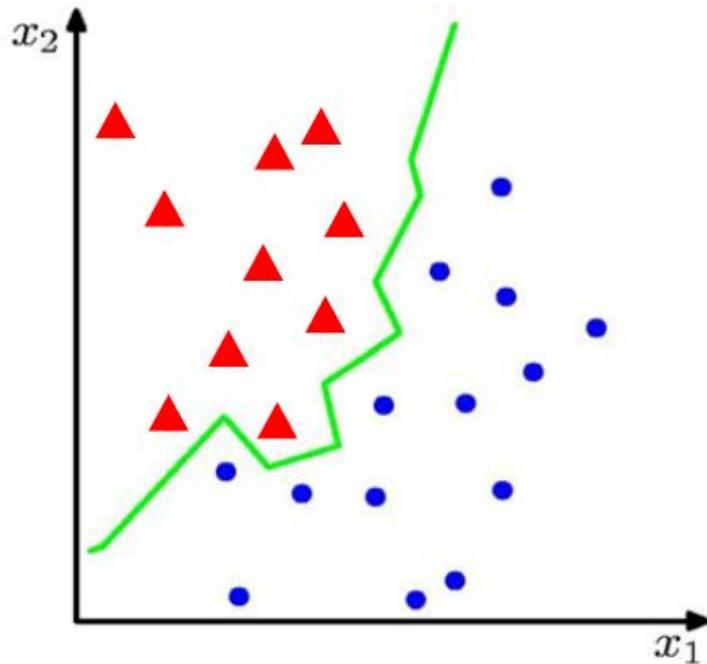
Disadvantages of Reinforcement Learning

- Reinforcement learning needs a lot of data and a lot of computation. It is data-hungry.
 - So for solving video games and puzzles it performs well.
- Reinforcement learning assumes the world is Markovian, which it is not.
 - The Markovian model describes a sequence of possible events in which the probability of each event depends only on the state attained in the previous event.

What is classification problem?

- Let there are two classes of objects.
 - Class 1: Set of dog pictures
 - Class 2: Set of cat pictures
- Problem is
 - Given a picture, you should say whether it is cat or dog.
 - For a human being it is easy..., but for a machine it is a non-trivial problem.

What is classification problem?



- Suppose we are given a training set of N observations (x_1, \dots, x_N) and (y_1, \dots, y_N) , $x_i \in \mathbb{R}^d$, $y_i \in \{-1, 1\}$
- Classification problem is to estimate $f(x)$ from this data such that

$$f(x_i) = y_i$$

Classification: Supervised Learning

Training Phase



We have shown a set of dog pictures and a set of cat pictures to a child.



Classification: Supervised Learning

Testing Phase



→ DOG

This picture as it is
may not be in the
training set

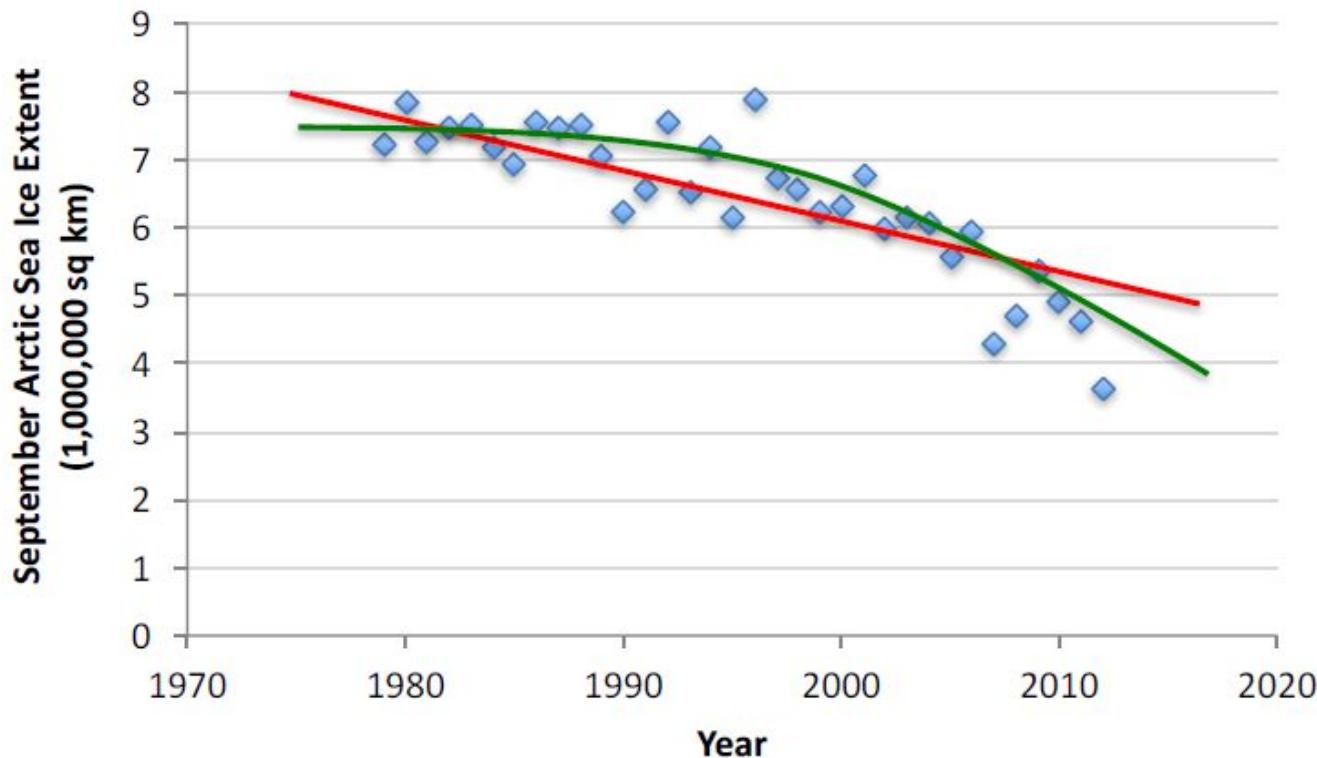
**Child has done more than
just remembering**

What is Learning?

- Child has learnt what is it that is common among dogs ... and, what is it that is common among cats... also, what are the distinguishing features/attributes.
- Child has learnt the pattern (regularity) behind all dogs and the pattern behind all cats.
- Child then recognized a test image as having a particular pattern that is unique to dogs.

What is Regression Problem?

- Given $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$
- Learn a function $f(x)$ to predict y given x
 - y is real-valued == regression



Popular ML algorithms

Classification

- Linear Classifiers
- Support Vector Machines
- Decision Trees
- K-Nearest Neighbor
- Random Forest

Regression

- Linear Regression
- Logistic Regression
- Polynomial Regression

Resources: Journals

- Journal of Machine Learning Research
www.jmlr.org
- Machine Learning
- IEEE Transactions on Neural Networks
- IEEE Transactions on Pattern Analysis and Machine Intelligence
- Annals of Statistics
- Journal of the American Statistical Association

Resources: Conferences

- International Conference on Machine learning (ICML)
- European Conference on Machine Learning (ECML)
- Neural Information Processing Systems (NIPS)
- Computational Learning
- International Joint Conference on Artificial Intelligence (IJCAI)
- ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)
- IEEE Int. Conf. on Data Mining (ICDM)

**Thank You:
Question?**

Supervised and unsupervised examples

Regression
k-means clustering

Introduction to Regression

- Regression analysis is the part of statistics that investigates the relationship between two or more variables related in a nondeterministic fashion.
- Linear regression attempts to model the relationship between two variables by fitting a linear equation to observed data.
- One variable is considered to be an explanatory variable, and the other is considered to be a dependent variable.
- For example, a modeler might want to relate the weights of individuals to their heights using a linear regression model.
- Before attempting to fit a linear model to observed data, a modeler should first determine whether or not there is a relationship between the variables of interest.

Introduction to Linear Regression

- This does not necessarily imply that one variable causes the other (for example, higher SAT scores do not cause higher college grades), but that there is some significant association between the two variables
- A scatterplot can be a helpful tool in determining the strength of the relationship between two variables.
- If there appears to be no association between the proposed explanatory and dependent variables (i.e., the scatterplot does not indicate any increasing or decreasing trends), then fitting a linear regression model to the data probably will not provide a useful model.

Introduction to Linear Regression

- The simplest deterministic mathematical relationship between two variables x and y is a linear relationship.

$$y = b_0 + b_1 x$$

- The set of pairs (x, y) for which determines a straight line with slope b_1 and y -intercept b_0 .
- More generally, the denoted by x and will be called the independent, predictor, or explanatory variable.
- For fixed x , the second variable will be random; we denote this random variable and its observed value by Y and y , respectively, and refer to it as the dependent or response variable.

Introduction to Linear Regression

- Usually observations will be made for a number of settings of the independent variable.
- Let x_1, x_2, \dots, x_n denote values of the independent variable for which observations are made, and let Y_i and y_i , respectively, denote the random variable and observed value associated with.
- The available bivariate data then consists of the n pairs $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$.
- A picture of this data called a **scatter plot** gives preliminary impressions about the nature of any relationship.
- In such a plot, each (x_i, y_i) is represented as a point plotted on a two-dimensional coordinate system.

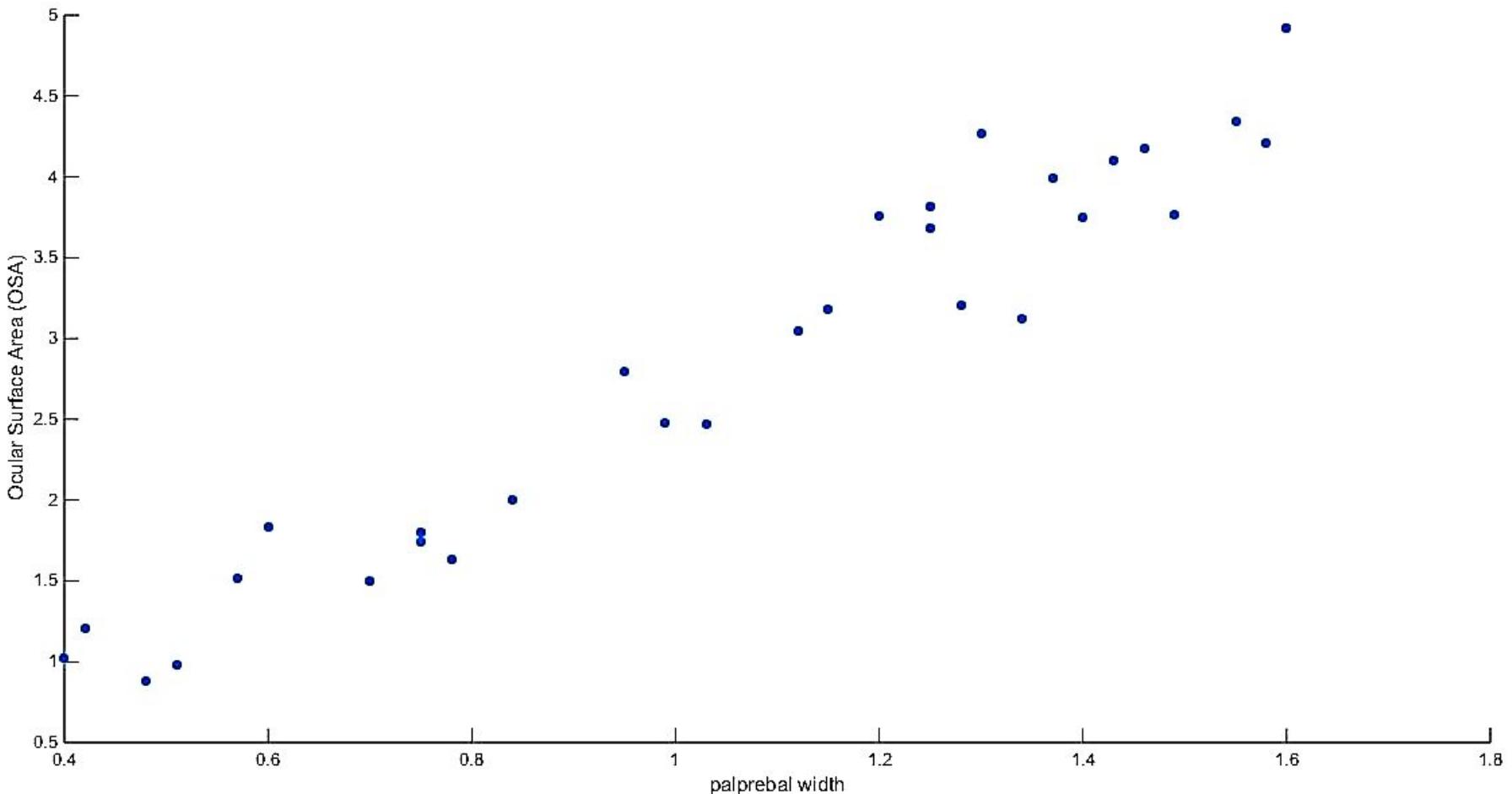
Scatter Plot Example: Linear Regression

Visual and musculoskeletal problems associated with the use of visual display terminals (VDTs) have become rather common in recent years. Some researchers have focused on vertical gaze direction as a source of eye strain and irritation. This direction is known to be closely related to ocular surface area (OSA), so a method of measuring OSA is needed. The accompanying representative data on $y = \text{OSA} (\text{cm}^2)$ and $x = \text{width of the palpebral fissure}$ (i.e., the horizontal width of the eye opening, in cm) is from the article "Analysis of Ocular Surface Area for Comfortable VDT Workstation Layout" (*Ergonomics*, 1996: 877–884). The order in which observations were obtained was not given, so for convenience they are listed in increasing order of x values.

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x_i	.40	.42	.48	.51	.57	.60	.70	.75	.75	.78	.84	.95	.99	1.03	1.12
y_i	1.02	1.21	.88	.98	1.52	1.83	1.50	1.80	1.74	1.63	2.00	2.80	2.48	2.47	3.05

i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
x_i	1.15	1.20	1.25	1.25	1.28	1.30	1.34	1.37	1.40	1.43	1.46	1.49	1.55	1.58	1.60
y_i	3.18	3.76	3.68	3.82	3.21	4.27	3.12	3.99	3.75	4.10	4.18	3.77	4.34	4.21	4.92

Scatter Plot Example



Scatter Plot Example: Linear Regression

- Several observations have identical x values yet different y values.
 - e.g., $x_8 = x_9 = 0.75$, but $y_8 = 1.80$ and $y_9 = 1.74$). Thus the value of y is not determined solely by x but also by various other factors.
- There is a strong tendency for y to increase as x increases. That is, larger values of OSA tend to be associated with larger values of fissure width—a positive relationship between the variables.
- It appears that the value of y could be predicted from x by *finding a line that is reasonably close to the points in the plot* (the authors of the cited article superimposed such a line on their plot).
- In other words, there is evidence of a substantial (though not perfect) linear relationship between the two variables.

Simple Linear Regression Model

- For the deterministic model , the actual observed value of y is a linear function of x.
- The appropriate generalization of this to a probabilistic model assumes that the expected value of Y is a linear function of x, but that for fixed x the variable Y differs from its expected value by a random amount.
- In a simple regression problem (a single x and a single y), the form of the model would be:

$$y = b_0 + b_1 * x$$

- In higher dimensions when we have more than one input (x), the line is called a plane or a hyper-plane. The representation therefore is the form of the equation and the specific values used for the coefficients (e.g. b_0 and b_1 in the above example).

Simple Linear Regression Model

- When a coefficient becomes zero, it effectively removes the influence of the input variable on the model and therefore from the prediction made from the model ($0 * x = 0$).
- The values b_0 and b_1 must be chosen so that they minimize the error. If sum of squared error is taken as a metric to evaluate the model, then goal to obtain a line that best reduces the error.

$$\text{Error} = \sum_{i=1}^n (\text{actual_output} - \text{predicted_output})^{** 2}$$

Simple Linear Regression Model

- For model with one predictor,

$$b_0 = \bar{y} - b_1 \bar{x}$$

and

$$b_1 = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

- Exploring 'b1':
 - If $b_1 > 0$, then x(predictor) and y (target) have a positive relationship. That is increase in x will increase y.
 - If $b_1 < 0$, then x(predictor) and y (target) have a negative relationship. That is increase in x will decrease y.

Simple Linear Regression Model

- Exploring ‘b0’
 - If the model does not include $x=0$, then the prediction will become meaningless with only b_0 .
 - For example, we have a dataset that relates height(x) and weight(y). Taking $x=0$ (that is height as 0), will make equation have only b_0 value which is completely meaningless as in real-time height and weight can never be zero. This resulted due to considering the model values beyond its scope.
 - If the model includes value 0, then ‘ b_0 ’ will be the average of all predicted values when $x=0$. But, setting zero for all the predictor variables is often impossible.
 - The value of b_0 guarantee that residual have mean zero. If there is no ‘ b_0 ’ term, then regression will be forced to pass over the origin. Both the regression co-efficient and prediction will be biased.

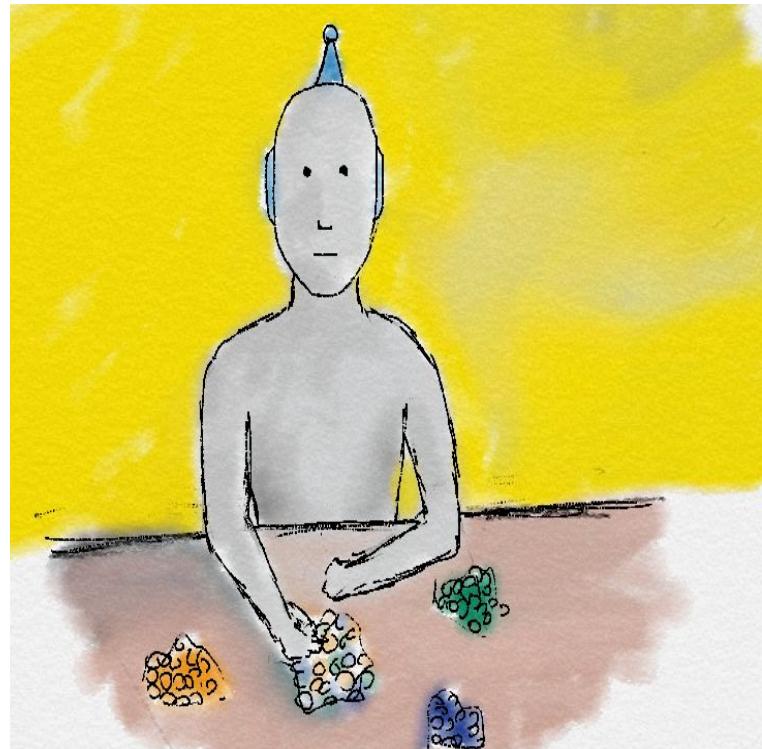
Additional Resources:

- <https://towardsdatascience.com/linear-regression-detailed-view-ea73175f6e86>
- <http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm>

Week	Covid Cases in thousands			$(\bar{x} - \bar{y})^2$	$(\bar{y} - \bar{z})^2$	$\text{mul}(\text{col 3 and col 4})$	
0	0.4			6.25	1.388469	7.638469	
1	0.56			16	6.4009	22.4009	
2	0.96			14.44	6.948496	21.3885	110.5536
3	1.61			14.0625	3.674122	17.73662	91.752
4	2.59			16	0.388711	16.38871	
5	3.35			25	0.000374	25.00037	

What is clustering?

- Clustering: the process of grouping a set of objects into classes of similar objects.
 - Objects within a cluster should be similar.
 - Objects from different clusters should be dissimilar.



k-means Clustering

- k-means is a partitional clustering algorithm
- Let the set of data points (or instances) D be $\{x_1, x_2, \dots, x_n\}$, where $x_i = (x_{i1}, x_{i2}, \dots, x_{ir})$ is a vector in a real-valued space $x \subseteq \mathbb{R}^r$, and r is the number of attributes (dimensions) in the data.
- The k-means algorithm partitions the given data into k clusters.
- Each cluster has a cluster centre, called centroid.
- k is specified by the user

k-means algorithm

- 1) Randomly choose k data points (**seeds**) to be the initial **centroids** or **cluster centers**.
- 2) Assign each data point to the closest **centroid**
- 3) Re-compute the **centroids** using the current cluster memberships.
- 4) Repeat the assignment (step 2) and update the centroids (step 3)
until **centroids remain same or no changes in clusters.**

Algorithm: k-means clustering algorithm

1: Select k points as initial centroids.

2: **repeat**

3: Form k clusters by assigning each point (x) to its closest centroid (m_j) using sum of squared error (SSE),

$$SSE = \sum_{j=1}^k \sum_{x \in C_j} dist(x, m_j)^2$$

C_j is the j th cluster, m_j is the centroid of cluster C_j (the mean vector of all the data points in C_j), and $dist(x, m_j)$ is the distance between data point x and centroid m_j .

4: Recompute the centroid (m_j) for each cluster (C_j).

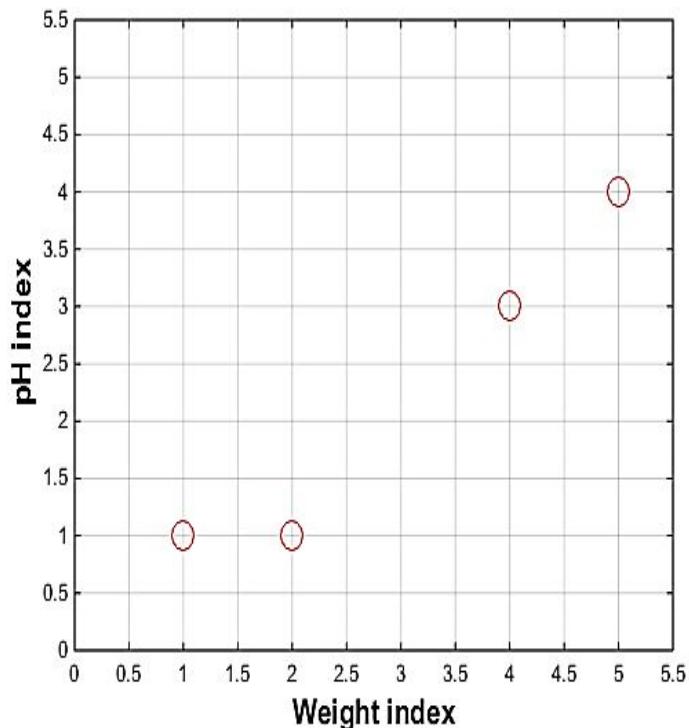
$$m_j = \frac{1}{n} \sum_{k=1, x \in C_j}^n x_k$$

5: **until** Centroids do not change.

Example of k means:

- Problem: Suppose we have 4 types of medicines and each has two attributes (pH and weight index). Our goal is to group these objects into $k=2$ group of medicine.

Medicine	Weight index	pH-index
A	1	1
B	2	1
C	4	3
D	5	4



Example of K-means. Iteration 1

Step 1: Use initial seed points for partitioning.

Seed Points: C1= $m_1 = (1,1)$ and C2= $m_2 = (2,1)$

$$SSE = \sum_{j=1}^k \sum_{x \in C_j} dist(x, m_j)^2$$

$$Euclidian\ dist(X, Y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_n - y_n)^2}$$

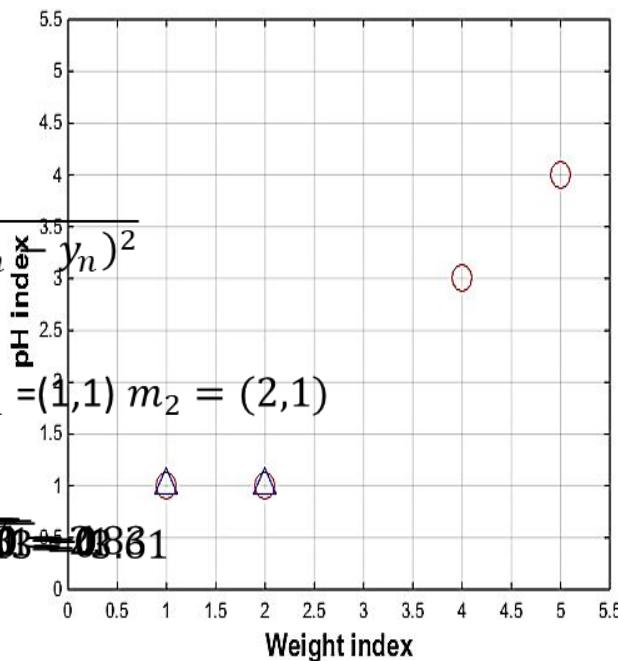
$$A=(1,1) \ m_1 =(1,1) \ m_2 = (2,1)$$

$$C=(4,3) \ m_1 =(1,1) \ m_2 = (2,1)$$

$$B=(2,1) \ m_1 =(1,1) \ m_2 = (2,1)$$

$$Euclidian\ dist(A, m_2) = \sqrt{(4-2)^2 + (1-1)^2} = \sqrt{0+4} = \sqrt{4} = 2.0$$

$$D^0 = \begin{bmatrix} 0 & 1 & 3.6 \\ 1 & 0 & 2.83 \end{bmatrix}$$



Example of K-means: iteration 1

Step 1: Use initial seed points for partitioning.

Seed Points:

$$c_1 = A, c_2 = B$$

$$\mathbf{D}^0 = \begin{bmatrix} 0 & 1 & 3.61 & 5 \\ 1 & 0 & 2.83 & 4.24 \\ A & B & C & D \end{bmatrix}$$

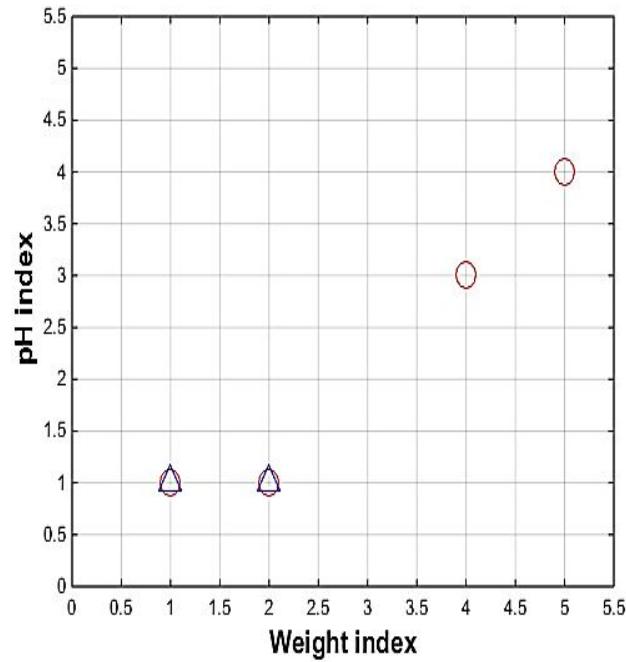
Euclidean distance

$$\begin{bmatrix} 1 & 2 & 4 & 5 \\ 1 & 1 & 3 & 4 \end{bmatrix} X$$
$$\begin{bmatrix} 1 & 2 & 4 & 5 \\ 1 & 1 & 3 & 4 \end{bmatrix} Y$$

$$d(D, c_1) = \sqrt{(5-1)^2 + (4-1)^2} = 5$$

$$d(D, c_2) = \sqrt{(5-2)^2 + (4-1)^2} = 4.24$$

Assign each object to the cluster with the nearest seed point (mean).



Example of K-means: iteration 1

Step 2: Renew membership based on new centroids..

Seed Points: $c_1 = A, c_2 = B$

$$D^0 = \begin{bmatrix} 0 & 1 & 3.61 & 5 \\ 1 & 0 & 2.83 & 4.24 \end{bmatrix}$$

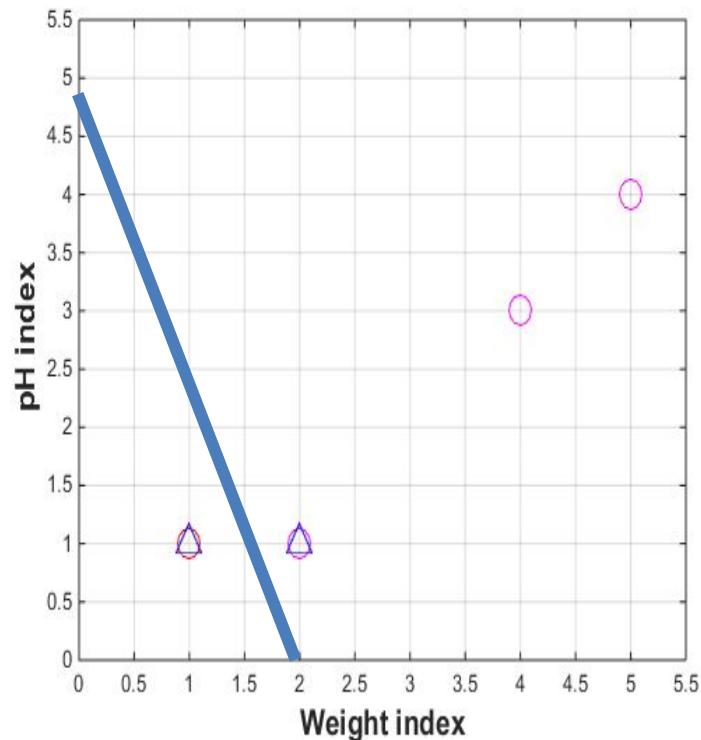
$A \quad B \quad C \quad D$ Euclidean distance

$$\begin{bmatrix} 1 & 2 & 4 & 5 \end{bmatrix} X$$
$$\begin{bmatrix} 1 & 1 & 3 & 4 \end{bmatrix} Y$$

$$d(D, c_1) = \sqrt{(5-1)^2 + (4-1)^2} = 5$$

$$d(D, c_2) = \sqrt{(5-2)^2 + (4-1)^2} = 4.24$$

Assign each object to the cluster with the nearest seed point (mean).



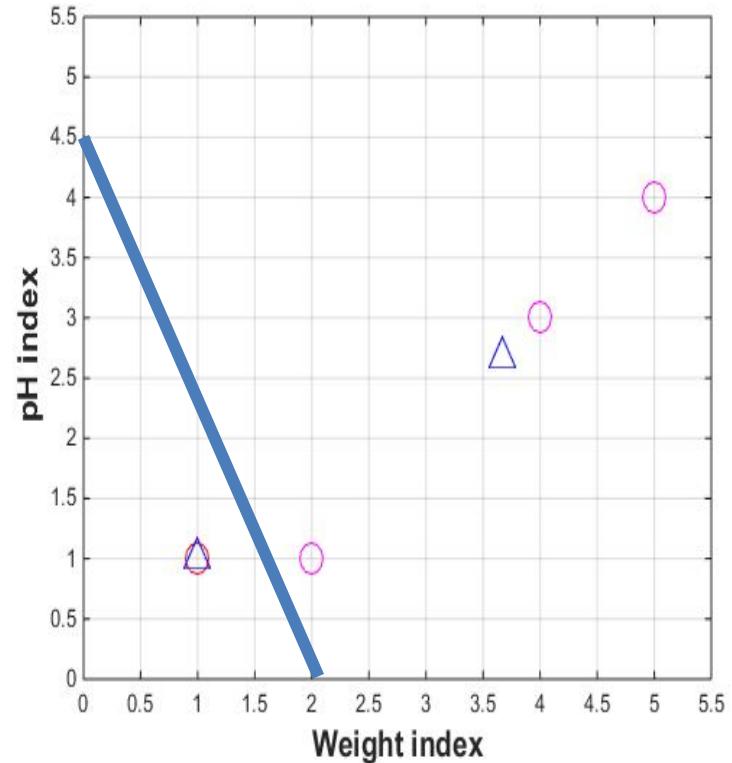
iteration 1

Step 3: Compute new centroids of the current partition.

Knowing the members of each cluster, compute the new centroid for each cluster based on these new membership assignment.

$$c_1 = (1, 1)$$

$$\begin{aligned} c_2 &= \left(\frac{2+4+5}{3}, \frac{1+3+4}{3} \right) \\ &= \left(\frac{11}{3}, \frac{8}{3} \right) = (3.66, 2.66) \end{aligned}$$



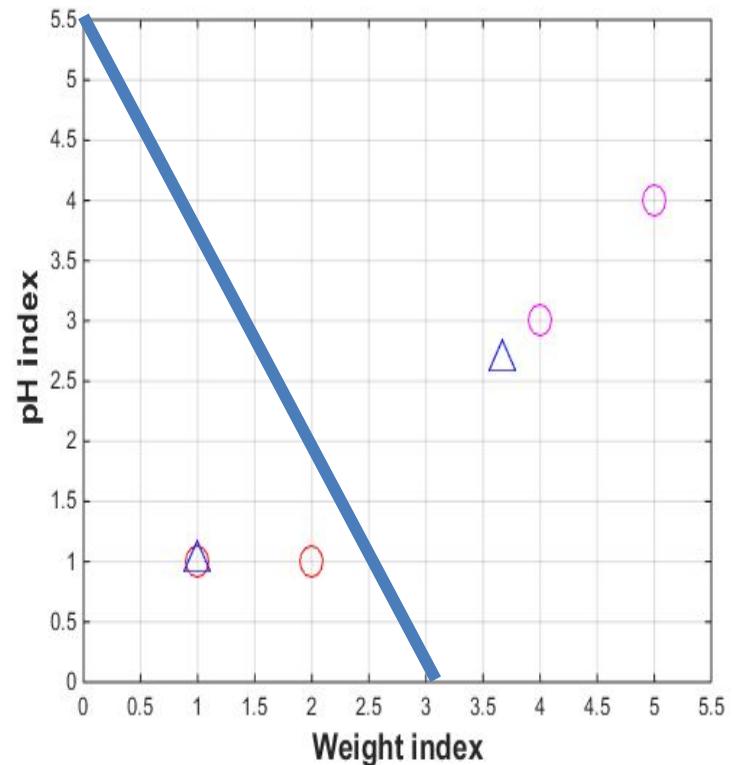
iteration 2

Step 2: Renew membership based on new centroids.

Compute the distance of all objects to the new centroids
Assign each object to the cluster with the nearest centroid

$$D^1 = \begin{bmatrix} 0 & 1 & 3.61 & 5 \\ 3.14 & 2.36 & 0.47 & 1.89 \end{bmatrix} \quad c_1 = (1,1) \text{ group-1} \\ c_2 = \left(\frac{11}{3}, \frac{8}{3}\right) \text{ group-2}$$

A	B	C	D
[1 2 4 5]	X		
[1 1 3 4]	Y		



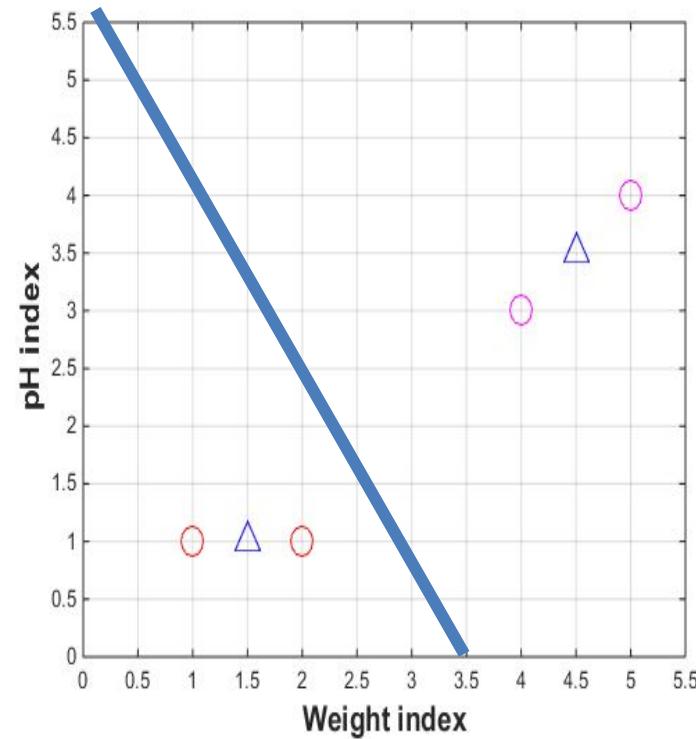
Example of k-means: iteration 2

Step 3: Compute new centroids of the current partition.

Knowing the members of each cluster, compute the new centroid for each cluster based on these new membership assignment.

$$c_1 = \left(\frac{1+2}{2}, \frac{1+1}{2} \right) = \left(1\frac{1}{2}, 1 \right)$$

$$c_2 = \left(\frac{4+5}{2}, \frac{3+4}{2} \right) = \left(4\frac{1}{2}, 3\frac{1}{2} \right)$$



iteration 3

Step 3: Repeat the first two steps until its convergence.

Compute the distance of all objects to the new centroid
Assign each object to the cluster with the nearest centroid

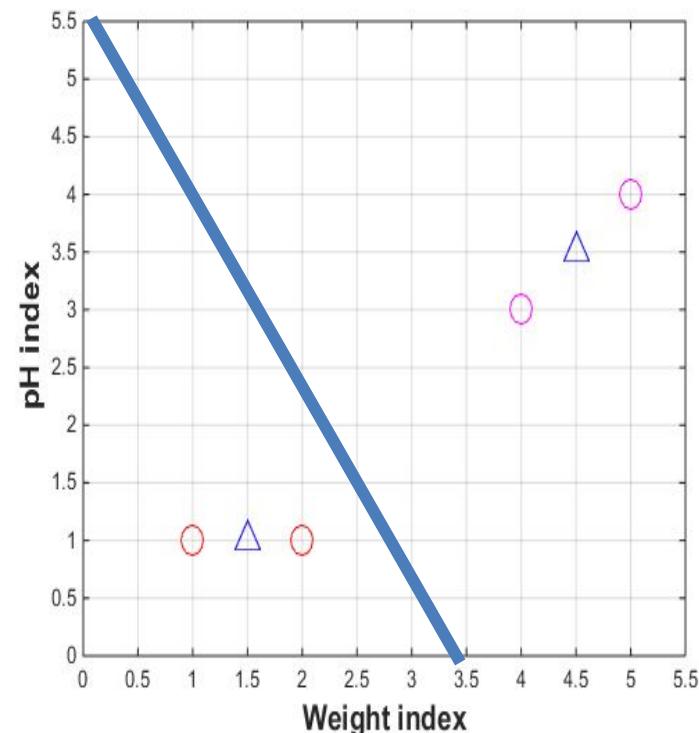
$$D^2 = \begin{bmatrix} 0.5 & 0.5 & 3.20 & 4.61 \\ 4.30 & 3.54 & 0.71 & 0.71 \end{bmatrix} \quad c_1 = (1\frac{1}{2}, 1) \text{ group-1} \\ c_2 = (4\frac{1}{2}, 3\frac{1}{2}) \text{ group-2}$$

A	B	C	D
1	2	4	5
1	1	3	4

$$X \quad Y$$

Knowing the members of each cluster, compute the new centroid for each cluster based on these new membership assignment.

Since there is no change in cluster assignments and centroids the algorithm terminates.



Applications of k-means clustering

- Behavioural segmentation:
- Inventory categorization:
- Sorting sensor measurements:
- Image Segmentation

INFERENCE IN FIRST-ORDER LOGIC

CHAPTER 9

Outline

- ◊ Reducing first-order inference to propositional inference
- ◊ Unification
- ◊ Generalized Modus Ponens
- ◊ Forward and backward chaining
- ◊ Logic programming
- ◊ Resolution

A brief history of reasoning

450B.C.	Stoics	propositional logic, inference (maybe)
322B.C.	Aristotle	“syllogisms” (inference rules), quantifiers
1565	Cardano	probability theory (propositional logic + uncertainty)
1847	Boole	propositional logic (again)
1879	Frege	first-order logic
1922	Wittgenstein	proof by truth tables
1930	Gödel	\exists complete algorithm for FOL
1930	Herbrand	complete algorithm for FOL (reduce to propositional)
1931	Gödel	$\neg\exists$ complete algorithm for arithmetic
1960	Davis/Putnam	“practical” algorithm for propositional logic
1965	Robinson	“practical” algorithm for FOL—resolution

Universal instantiation (UI)

Every instantiation of a universally quantified sentence is entailed by it:

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

for any variable v and ground term g

E.g., $\forall x \ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$ yields

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

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

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

:

Existential instantiation (EI)

For any sentence α , variable v , and constant symbol k
that does not appear elsewhere in the knowledge base:

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

E.g., $\exists x \ Crown(x) \wedge OnHead(x, John)$ yields

$$Crown(C_1) \wedge OnHead(C_1, John)$$

provided C_1 is a new constant symbol, called a Skolem constant

Another example: from $\exists x \ d(x^y)/dy = x^y$ we obtain

$$d(e^y)/dy = e^y$$

provided e is a new constant symbol

Existential instantiation contd.

UI can be applied several times to **add** new sentences;
the new KB is logically equivalent to the old

EI can be applied once to **replace** the existential sentence;
the new KB is **not** equivalent to the old,
but is satisfiable iff the old KB was satisfiable

Reduction to propositional inference

Suppose the KB contains just the following:

$$\begin{aligned} \forall x \ King(x) \wedge Greedy(x) &\Rightarrow Evil(x) \\ King(John) \\ Greedy(John) \\ Brother(Richard, John) \end{aligned}$$

Instantiating the universal sentence in **all possible ways**, we have

$$\begin{aligned} King(John) \wedge Greedy(John) &\Rightarrow Evil(John) \\ King(Richard) \wedge Greedy(Richard) &\Rightarrow Evil(Richard) \\ King(John) \\ Greedy(John) \\ Brother(Richard, John) \end{aligned}$$

The new KB is propositionalized: proposition symbols are

$$King(John), Greedy(John), Evil(John), King(Richard) \text{ etc.}$$

Reduction contd.

Claim: a ground sentence* is entailed by new KB iff entailed by original KB

Claim: every FOL KB can be propositionalized so as to preserve entailment

Idea: propositionalize KB and query, apply resolution, return result

Problem: with function symbols, there are infinitely many ground terms,

e.g., *Father(Father(Father(John)))*

Theorem: Herbrand (1930). If a sentence α is entailed by an FOL KB,
it is entailed by a **finite** subset of the propositional KB

Idea: For $n = 0$ to ∞ do

 create a propositional KB by instantiating with depth- n terms
 see if α is entailed by this KB

Problem: works if α is entailed, loops if α is not entailed

Theorem: Turing (1936), Church (1936), entailment in FOL is **semidecidable**

Problems with propositionalization

Propositionalization seems to generate lots of irrelevant sentences.

E.g., from

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

$$King(John)$$

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

$$Brother(Richard, John)$$

it seems obvious that $Evil(John)$, but propositionalization produces lots of facts such as $Greedy(Richard)$ that are irrelevant

With p k -ary predicates and n constants, there are $p \cdot n^k$ instantiations

With function symbols, it gets much much worse!

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(John)$ and $Greedy(y)$

$\theta = \{x/John, y/John\}$ works

$\text{UNIFY}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(John, x)$	$Knows(John, Jane)$	
$Knows(John, x)$	$Knows(y, OJ)$	
$Knows(John, x)$	$Knows(y, Mother(y))$	
$Knows(John, x)$	$Knows(x, OJ)$	

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(John)$ and $Greedy(y)$

$\theta = \{x/John, y/John\}$ works

$\text{UNIFY}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(John, x)$	$Knows(John, Jane)$	$\{x/Jane\}$
$Knows(John, x)$	$Knows(y, OJ)$	
$Knows(John, x)$	$Knows(y, Mother(y))$	
$Knows(John, x)$	$Knows(x, OJ)$	

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(John)$ and $Greedy(y)$

$\theta = \{x/John, y/John\}$ works

$\text{UNIFY}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(John, x)$	$Knows(John, Jane)$	$\{x/Jane\}$
$Knows(John, x)$	$Knows(y, OJ)$	$\{x/OJ, y/John\}$
$Knows(John, x)$	$Knows(y, Mother(y))$	
$Knows(John, x)$	$Knows(x, OJ)$	

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(John)$ and $Greedy(y)$

$\theta = \{x/John, y/John\}$ works

$\text{UNIFY}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(John, x)$	$Knows(John, Jane)$	$\{x/Jane\}$
$Knows(John, x)$	$Knows(y, OJ)$	$\{x/OJ, y/John\}$
$Knows(John, x)$	$Knows(y, Mother(y))$	$\{y/John, x/Mother(John)\}$
$Knows(John, x)$	$Knows(x, OJ)$	

Unification

We can get the inference immediately if we can find a substitution θ such that $King(x)$ and $Greedy(x)$ match $King(John)$ and $Greedy(y)$

$\theta = \{x/John, y/John\}$ works

$\text{UNIFY}(\alpha, \beta) = \theta$ if $\alpha\theta = \beta\theta$

p	q	θ
$Knows(John, x)$	$Knows(John, Jane)$	$\{x/Jane\}$
$Knows(John, x)$	$Knows(y, OJ)$	$\{x/OJ, y/John\}$
$Knows(John, x)$	$Knows(y, Mother(y))$	$\{y/John, x/Mother(John)\}$
$Knows(John, x)$	$Knows(x, OJ)$	$fail$

Standardizing apart eliminates overlap of variables, e.g., $Knows(z_{17}, OJ)$

Generalized Modus Ponens (GMP)

$$\frac{p_1', \ p_2', \ \dots, \ p_n', \ (p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q)}{q\theta} \quad \text{where } p_i'\theta = p_i\theta \text{ for all } i$$

p_1' is *King(John)* p_1 is *King(x)*
 p_2' is *Greedy(y)* p_2 is *Greedy(x)*
 θ is $\{x/John, y/John\}$ q is *Evil(x)*
 $q\theta$ is *Evil(John)*

GMP used with KB of definite clauses (**exactly** one positive literal)
All variables assumed universally quantified

Soundness of GMP

Need to show that

$$p_1', \dots, p_n', (p_1 \wedge \dots \wedge p_n \Rightarrow q) \models q\theta$$

provided that $p_i'\theta = p_i\theta$ for all i

Lemma: For any definite clause p , we have $p \models p\theta$ by UI

1. $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \models (p_1 \wedge \dots \wedge p_n \Rightarrow q)\theta = (p_1\theta \wedge \dots \wedge p_n\theta \Rightarrow q\theta)$
2. $p_1', \dots, p_n' \models p_1' \wedge \dots \wedge p_n' \models p_1'\theta \wedge \dots \wedge p_n'\theta$
3. From 1 and 2, $q\theta$ follows by ordinary Modus Ponens

Example knowledge base

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

Prove that Col. West is a criminal

Example knowledge base contd.

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

Example knowledge base contd.

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

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

Nono ... has some missiles

Example knowledge base contd.

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

$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$

Nono ... has some missiles, i.e., $\exists x \ Owns(Nono, x) \wedge Missile(x)$:

$Owns(Nono, M_1)$ and $Missile(M_1)$

... all of its missiles were sold to it by Colonel West

Example knowledge base contd.

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

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

Nono ... has some missiles, i.e., $\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$:

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

... all of its missiles were sold to it by Colonel West

$$\forall x \text{ Missle}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$$

Missiles are weapons:

Example knowledge base contd.

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

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

Nono ... has some missiles, i.e., $\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$:

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

... all of its missiles were sold to it by Colonel West

$$\forall x \text{ Missle}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$$

Missiles are weapons:

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

An enemy of America counts as “hostile”:

Example knowledge base contd.

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

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

Nono ... has some missiles, i.e., $\exists x \text{ Owns}(\text{Nono}, x) \wedge \text{Missile}(x)$:

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

... all of its missiles were sold to it by Colonel West

$$\forall x \text{ Missle}(x) \wedge \text{Owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$$

Missiles are weapons:

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

An enemy of America counts as “hostile”:

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

West, who is American ...

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

The country Nono, an enemy of America ...

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

Forward chaining algorithm

```
function FOL-FC-ASK( $KB, \alpha$ ) returns a substitution or false
    repeat until  $new$  is empty
         $new \leftarrow \{ \}$ 
        for each sentence  $r$  in  $KB$  do
             $(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow \text{STANDARDIZE-APART}(r)$ 
            for each  $\theta$  such that  $(p_1 \wedge \dots \wedge p_n)\theta = (p'_1 \wedge \dots \wedge p'_n)\theta$ 
                for some  $p'_1, \dots, p'_n$  in  $KB$ 
                     $q' \leftarrow \text{SUBST}(\theta, q)$ 
                    if  $q'$  is not a renaming of a sentence already in  $KB$  or  $new$  then do
                        add  $q'$  to  $new$ 
                         $\phi \leftarrow \text{UNIFY}(q', \alpha)$ 
                        if  $\phi$  is not fail then return  $\phi$ 
                    add  $new$  to  $KB$ 
    return false
```

Forward chaining proof

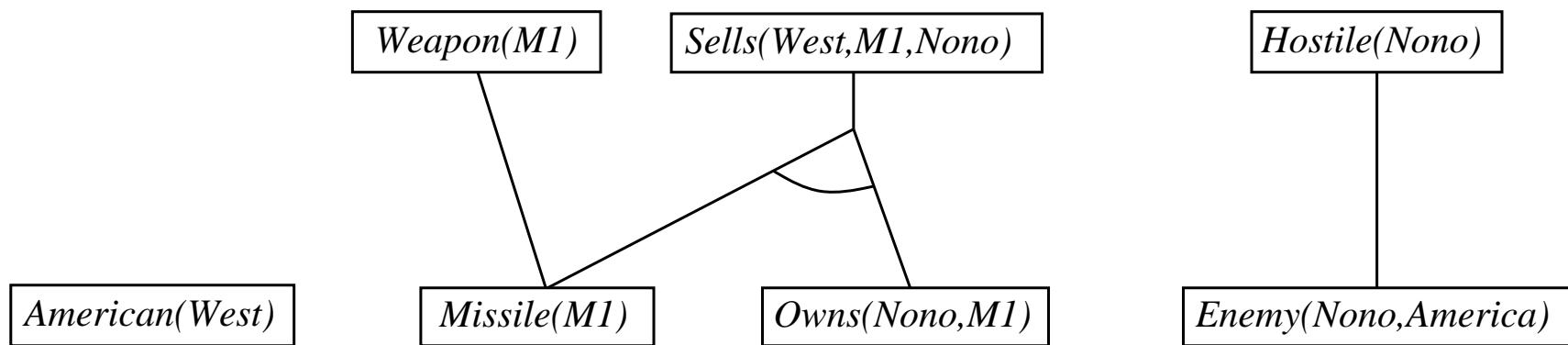
American(West)

Missile(M1)

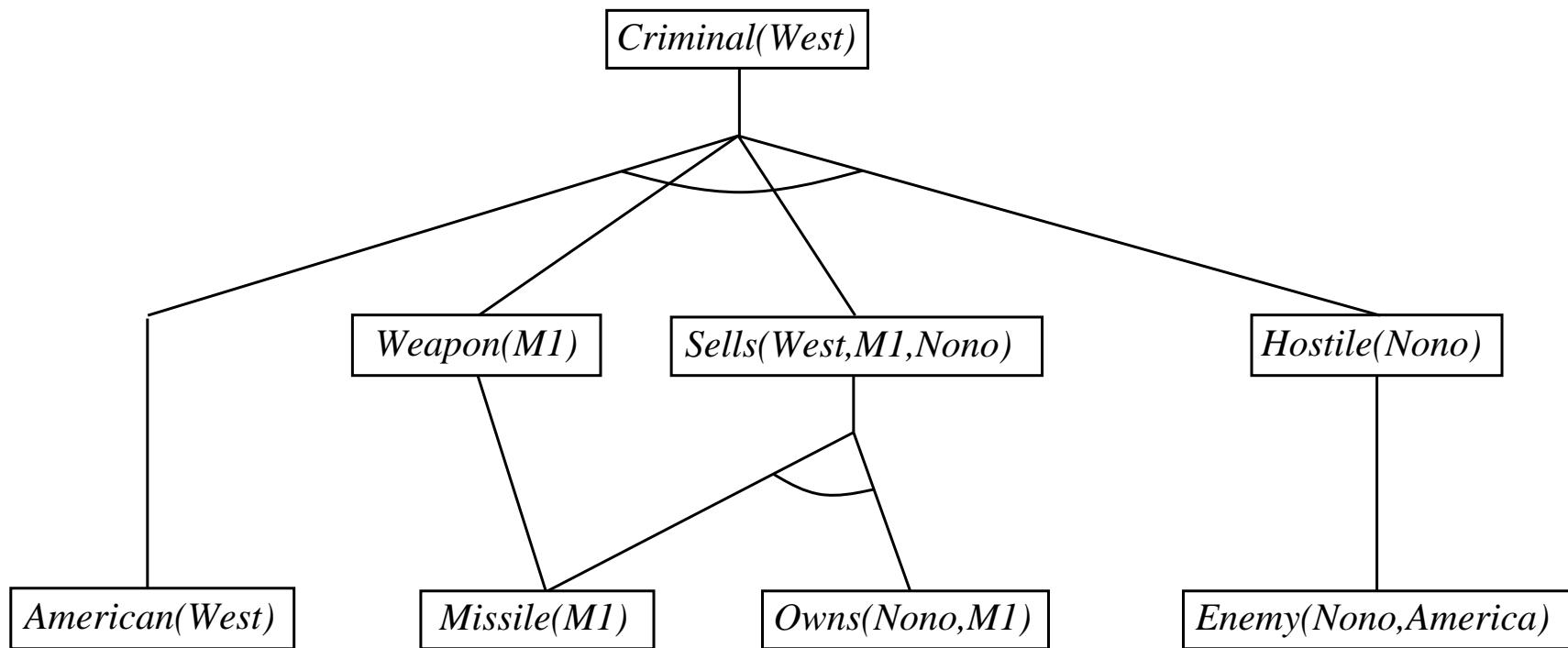
Owns(Nono,M1)

Enemy(Nono,America)

Forward chaining proof



Forward chaining proof



Properties of forward chaining

Sound and complete for first-order definite clauses
(proof similar to propositional proof)

Datalog = first-order definite clauses + **no functions** (e.g., crime KB)
FC terminates for Datalog in poly iterations: at most $p \cdot n^k$ literals

May not terminate in general if α is not entailed

This is unavoidable: entailment with definite clauses is semidecidable

Efficiency of forward chaining

Simple observation: no need to match a rule on iteration k
if a premise wasn't added on iteration $k - 1$

⇒ match each rule whose premise contains a newly added literal

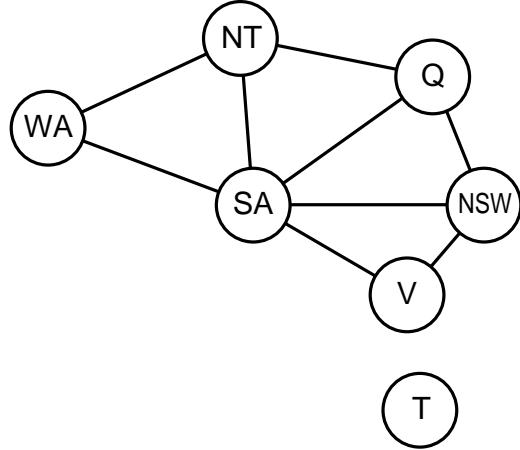
Matching itself can be expensive

Database indexing allows $O(1)$ retrieval of known facts
e.g., query $\text{Missile}(x)$ retrieves $\text{Missile}(M_1)$

Matching conjunctive premises against known facts is NP-hard

Forward chaining is widely used in deductive databases

Hard matching example


$$\begin{aligned} & \text{Diff}(wa, nt) \wedge \text{Diff}(wa, sa) \wedge \\ & \text{Diff}(nt, q) \text{Diff}(nt, sa) \wedge \\ & \text{Diff}(q, nsw) \wedge \text{Diff}(q, sa) \wedge \\ & \text{Diff}(nsw, v) \wedge \text{Diff}(nsw, sa) \wedge \\ & \text{Diff}(v, sa) \Rightarrow \text{Colorable}() \\ & \text{Diff}(Red, Blue) \quad \text{Diff}(Red, Green) \\ & \text{Diff}(Green, Red) \quad \text{Diff}(Green, Blue) \\ & \text{Diff}(Blue, Red) \quad \text{Diff}(Blue, Green) \end{aligned}$$

Colorable() is inferred iff the CSP has a solution

CSPs include 3SAT as a special case, hence matching is NP-hard

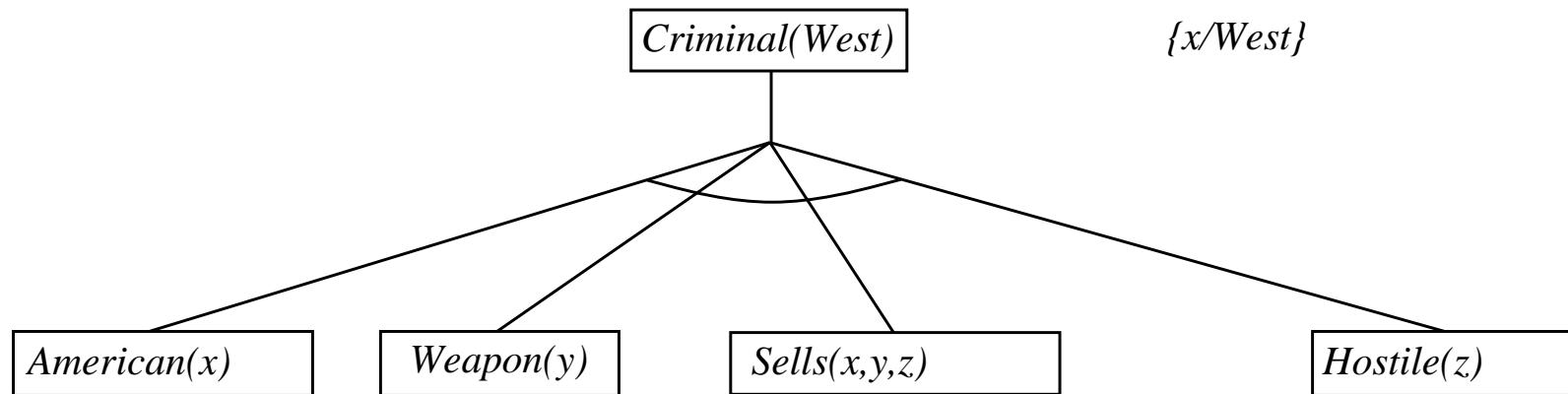
Backward chaining algorithm

```
function FOL-BC-ASK(KB, goals, θ) returns a set of substitutions
  inputs: KB, a knowledge base
          goals, a list of conjuncts forming a query (θ already applied)
          θ, the current substitution, initially the empty substitution { }
  local variables: answers, a set of substitutions, initially empty
  if goals is empty then return {θ}
  q' ← SUBST(θ, FIRST(goals))
  for each sentence r in KB
    where STANDARDIZE-APART(r) = ( p1 ∧ ... ∧ pn ⇒ q)
    and θ' ← UNIFY(q, q') succeeds
    new_goals ← [p1, ..., pn | REST(goals)]
    answers ← FOL-BC-ASK(KB, new_goals, COMPOSE(θ', θ)) ∪ answers
  return answers
```

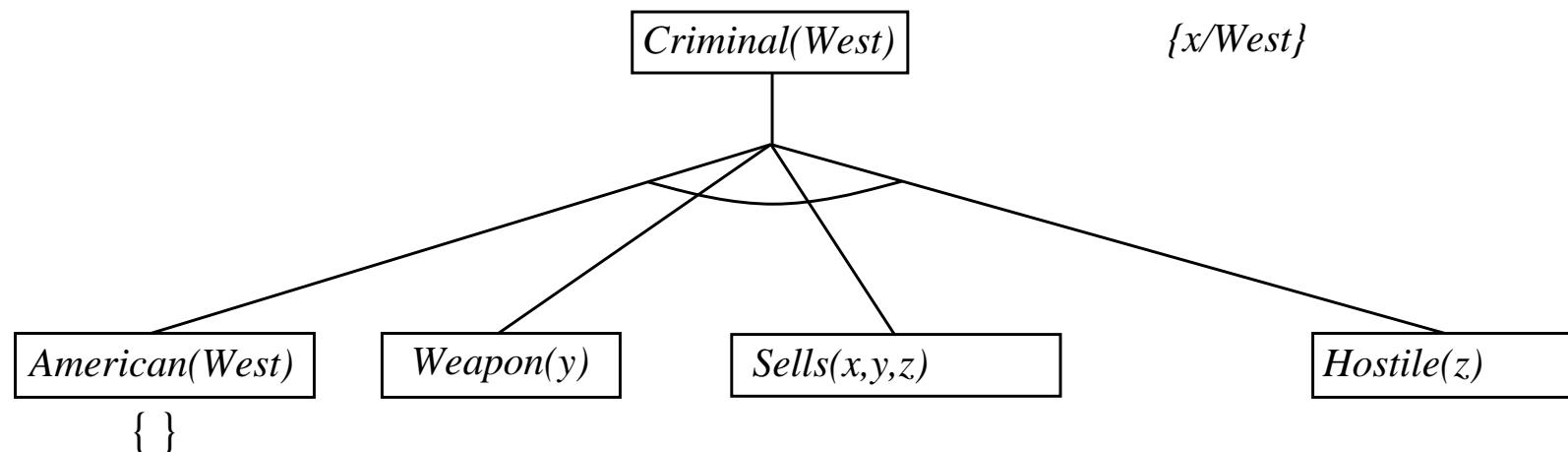
Backward chaining example

Criminal(West)

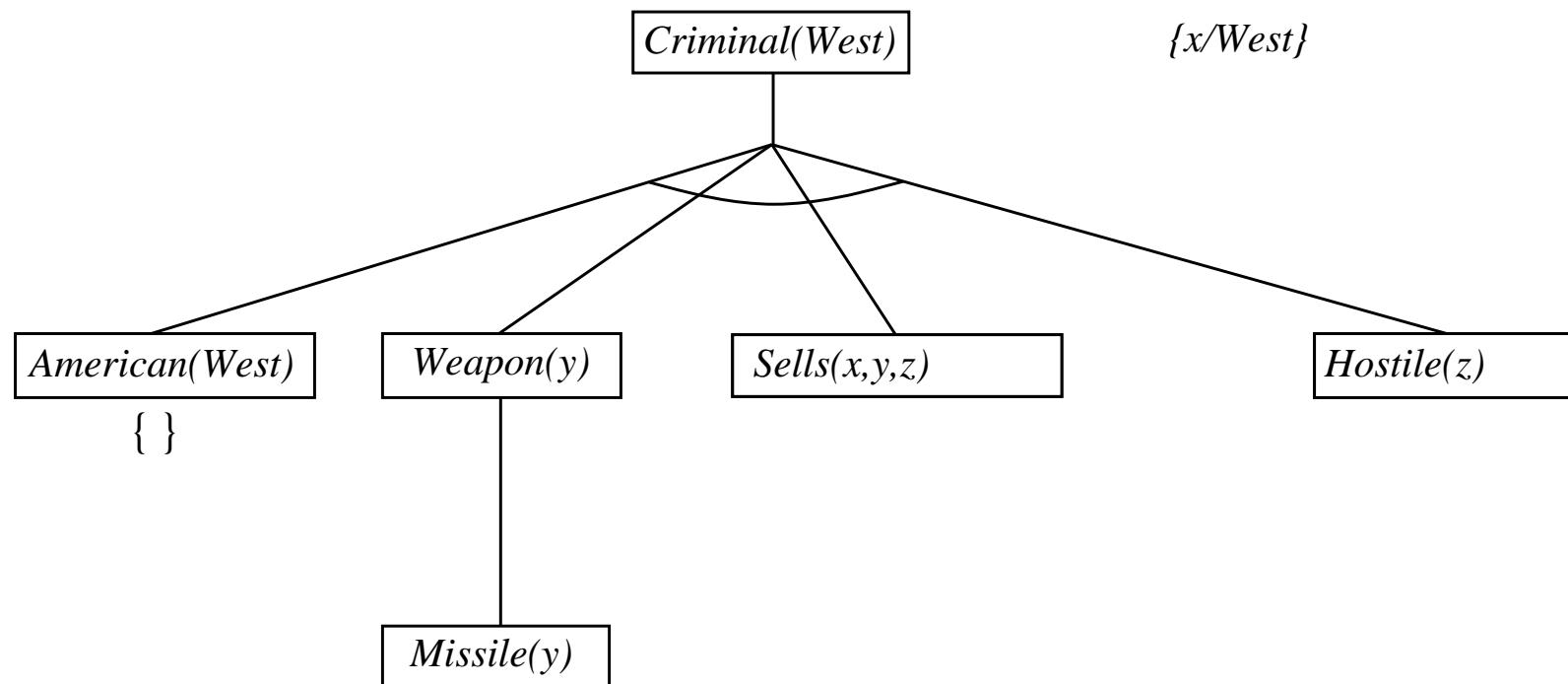
Backward chaining example



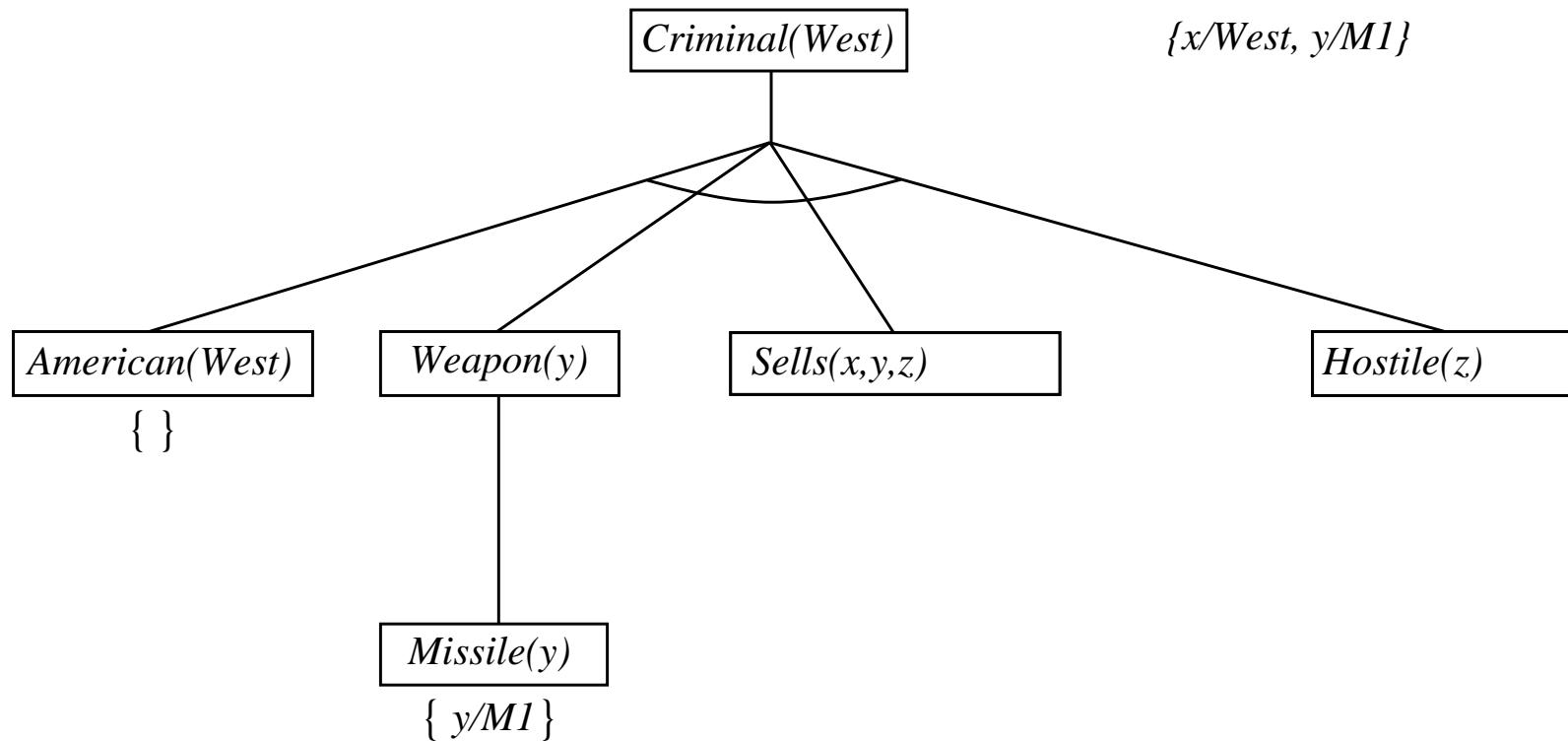
Backward chaining example



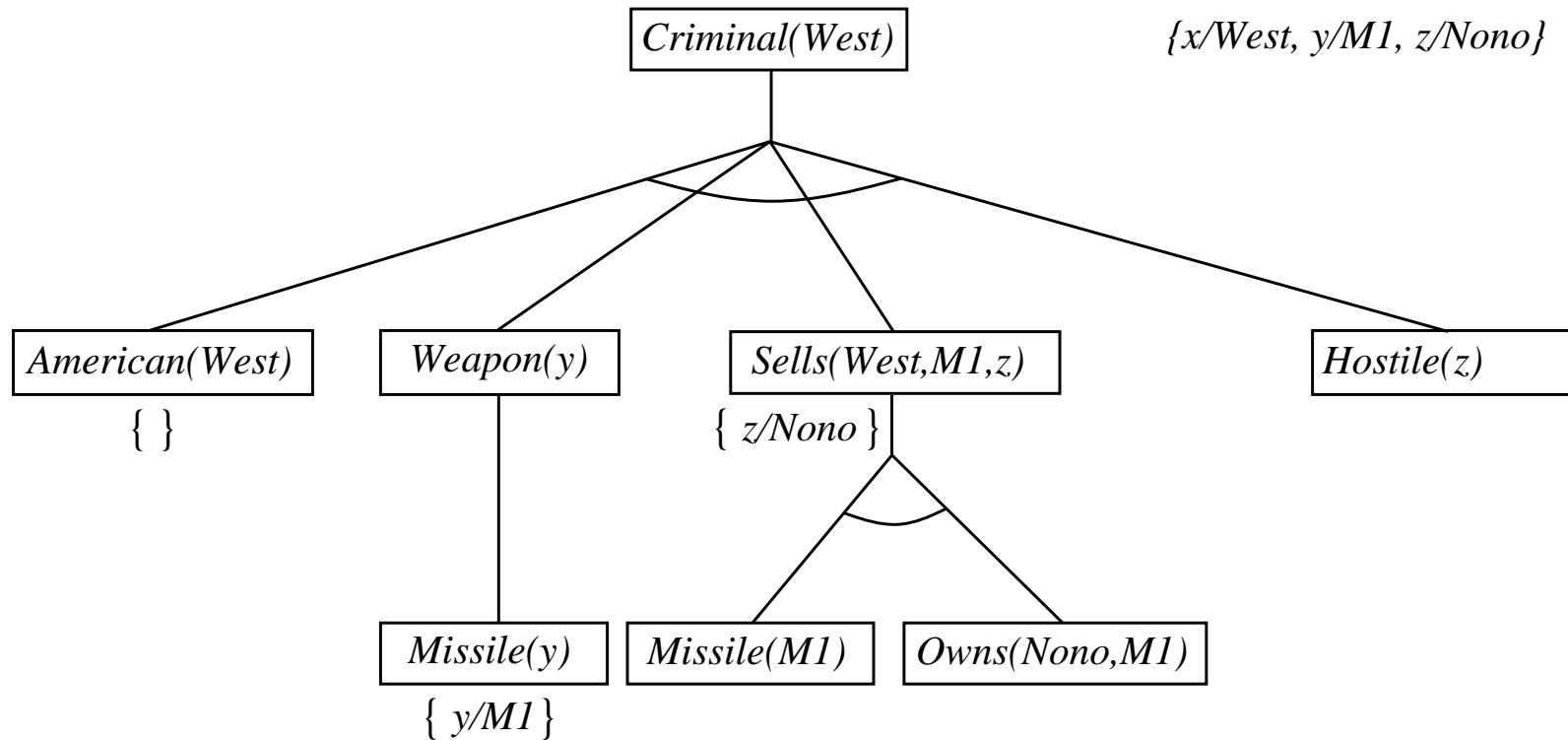
Backward chaining example



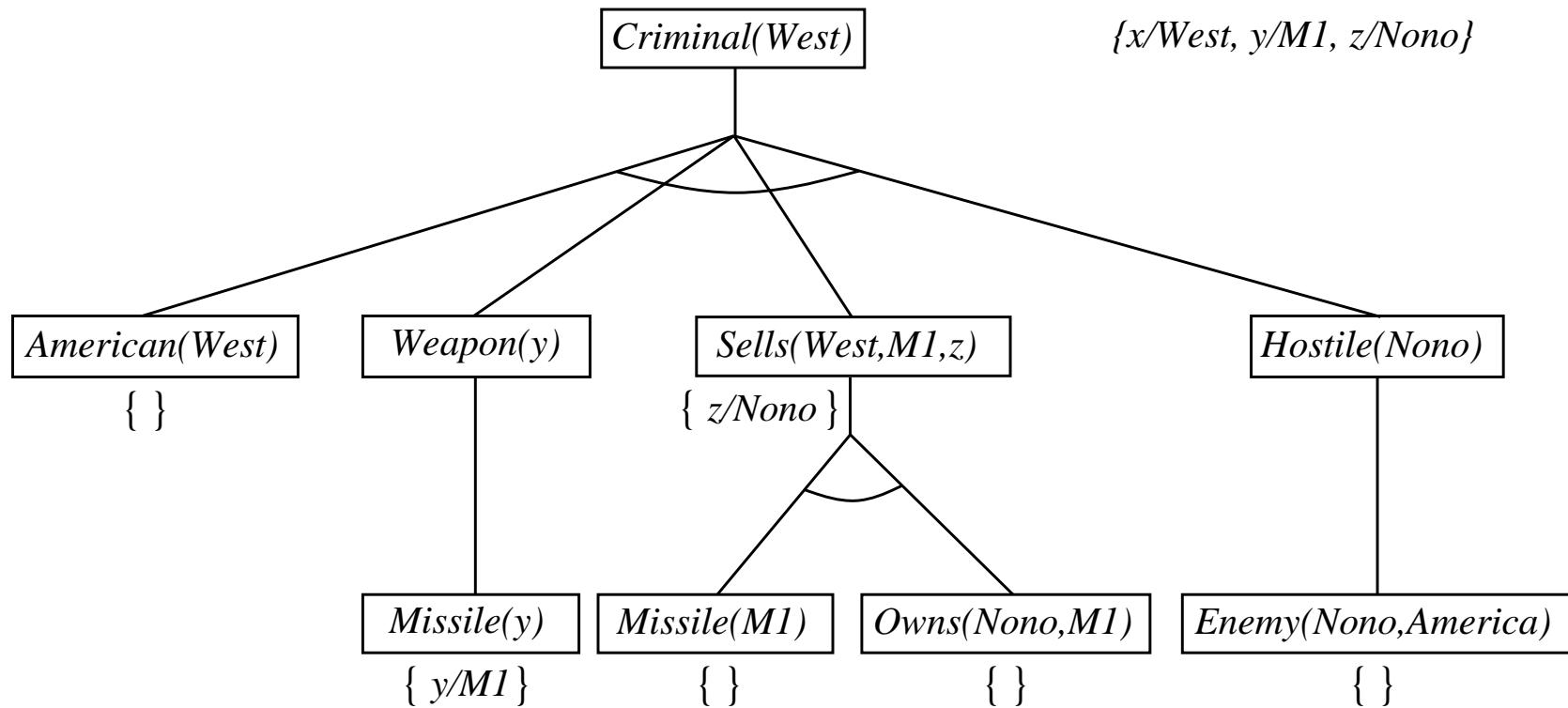
Backward chaining example



Backward chaining example



Backward chaining example



Properties of backward chaining

Depth-first recursive proof search: space is linear in size of proof

Incomplete due to infinite loops

⇒ fix by checking current goal against every goal on stack

Inefficient due to repeated subgoals (both success and failure)

⇒ fix using caching of previous results (extra space!)

Widely used (without improvements!) for logic programming

Logic programming

Sound bite: computation as inference on logical KBs

Logic programming

1. Identify problem
2. Assemble information
3. Tea break
4. Encode information in KB
5. Encode problem instance as facts
6. Ask queries
7. Find false facts

Ordinary programming

- Identify problem
- Assemble information
- Figure out solution
- Program solution
- Encode problem instance as data
- Apply program to data
- Debug procedural errors

Should be easier to debug $\text{Capital}(\text{NewYork}, \text{US})$ than $x := x + 2$!

Prolog systems

Basis: backward chaining with Horn clauses + bells & whistles

Widely used in Europe, Japan (basis of 5th Generation project)

Compilation techniques \Rightarrow approaching a billion LIPS

Program = set of clauses = head :- literal₁, ... literal_n.

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
```

Efficient unification by [open coding](#)

Efficient retrieval of matching clauses by direct linking

Depth-first, left-to-right backward chaining

Built-in predicates for arithmetic etc., e.g., X is Y*Z+3

Closed-world assumption (“negation as failure”)

e.g., given alive(X) :- not dead(X).

alive(joe) succeeds if dead(joe) fails

Prolog examples

Depth-first search from a start state X:

```
dfs(X) :- goal(X).  
dfs(X) :- successor(X,S),dfs(S).
```

No need to loop over S: successor succeeds for each

Appending two lists to produce a third:

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

query: append(A,B,[1,2]) ?

answers: A=[] B=[1,2]
 A=[1] B=[2]
 A=[1,2] B=[]

Resolution: brief summary

Full first-order version:

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

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$.

For example,

$$\frac{\neg Rich(x) \vee Unhappy(x) \\ Rich(Ken)}{Unhappy(Ken)}$$

with $\theta = \{x/Ken\}$

Apply resolution steps to $CNF(KB \wedge \neg \alpha)$; complete for FOL

Conversion to CNF

Everyone who loves all animals is loved by someone:

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

1. Eliminate biconditionals and implications

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

2. Move \neg inwards: $\neg\forall x, p \equiv \exists x \ \neg p$, $\neg\exists x, p \equiv \forall x \ \neg p$:

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

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

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

Conversion to CNF contd.

3. Standardize variables: each quantifier should use a different one

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

4. Skolemize: a more general form of existential instantiation.

Each existential variable is replaced by a **Skolem function** of the enclosing universally quantified variables:

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

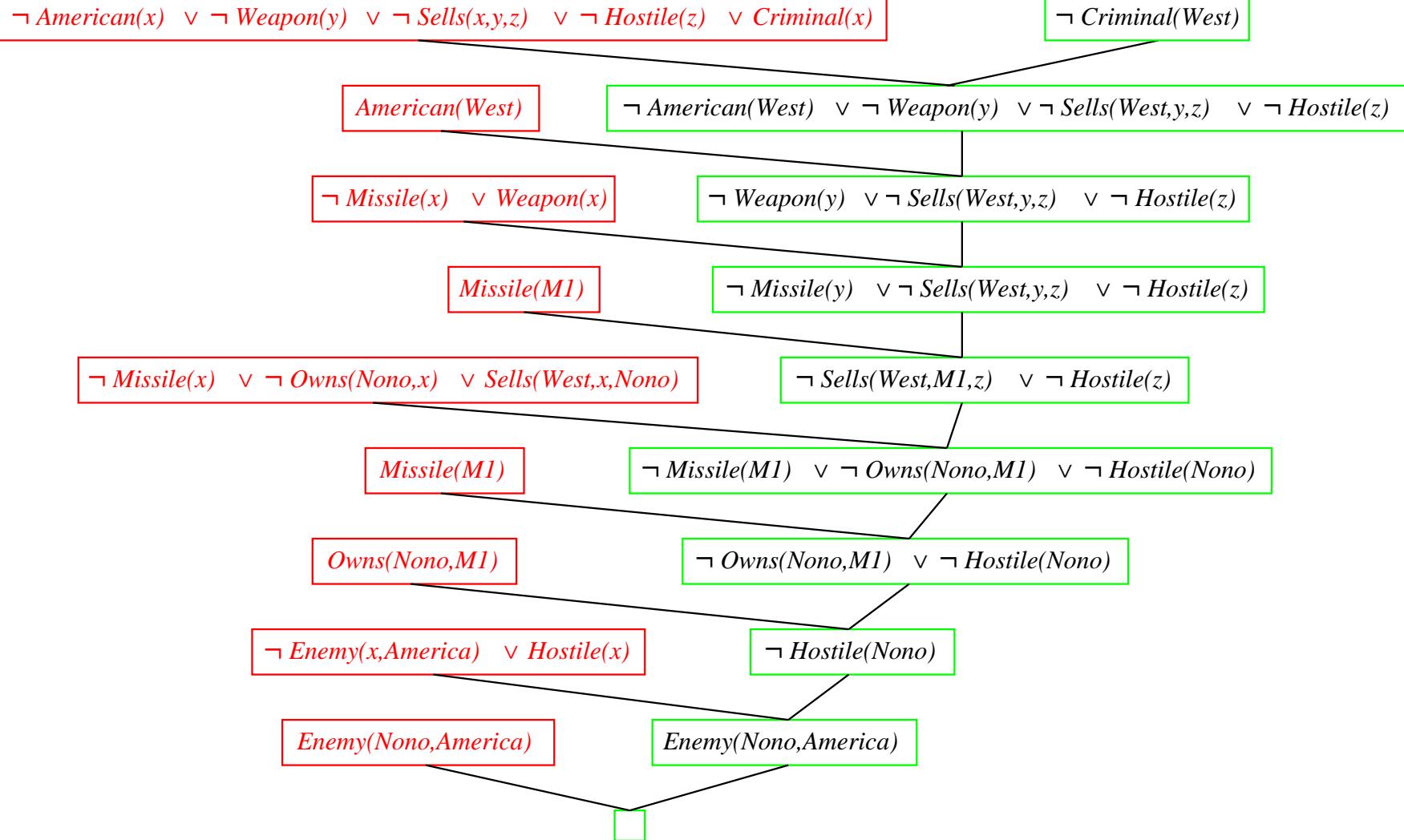
5. Drop universal quantifiers:

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

6. Distribute \wedge over \vee :

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

Resolution proof: definite clauses



Example Sentences

1. Consider the following axioms:

1. Every child loves Santa.
 $\forall x (CHILD(x) \rightarrow LOVES(x, Santa))$
2. Everyone who loves Santa loves any reindeer.
 $\forall x (LOVES(x, Santa) \rightarrow \forall y (REINDEER(y) \rightarrow LOVES(x, y)))$
3. Rudolph is a reindeer, and Rudolph has a red nose.
 $REINDEER(Rudolph) \wedge REDNOSE(Rudolph)$
4. Anything which has a red nose is weird or is a clown.
 $\forall x (REDNOSE(x) \rightarrow WEIRD(x) \vee CLOWN(x))$
5. No reindeer is a clown.
 $\neg \exists x (REINDEER(x) \wedge CLOWN(x))$
6. Scrooge does not love anything which is weird.
 $\forall x (WEIRD(x) \rightarrow \neg LOVES(Scrooge, x))$
7. (Conclusion) Scrooge is not a child.
 $\neg CHILD(Scrooge)$

2. Consider the following axioms:

1. Anyone who buys carrots by the bushel owns either a rabbit or a grocery store.
 $\forall x (BUY(x) \rightarrow \exists y (OWNS(x, y) \wedge (RABBIT(y) \vee GROCERY(y))))$
2. Every dog chases some rabbit.
 $\forall x (DOG(x) \rightarrow \exists y (RABBIT(y) \wedge CHASE(x, y)))$
3. Mary buys carrots by the bushel.
 $BUY(Mary)$
4. Anyone who owns a rabbit hates anything that chases any rabbit.
 $\forall x \forall y (OWNS(x, y) \wedge RABBIT(y) \rightarrow \forall z \forall w (RABBIT(w) \wedge CHASE(z, w) \rightarrow HATES(x, z)))$
5. John owns a dog.
 $\exists x (DOG(x) \wedge OWNS(John, x))$
6. Someone who hates something owned by another person will not date that person.
 $\forall x \forall y \forall z (OWNS(y, z) \wedge HATES(x, z) \rightarrow \neg DATE(x, y))$
7. (Conclusion) If Mary does not own a grocery store, she will not date John.
 $((\neg \exists x (GROCERY(x) \wedge OWN(Mary, x))) \rightarrow \neg DATE(Mary, John))$

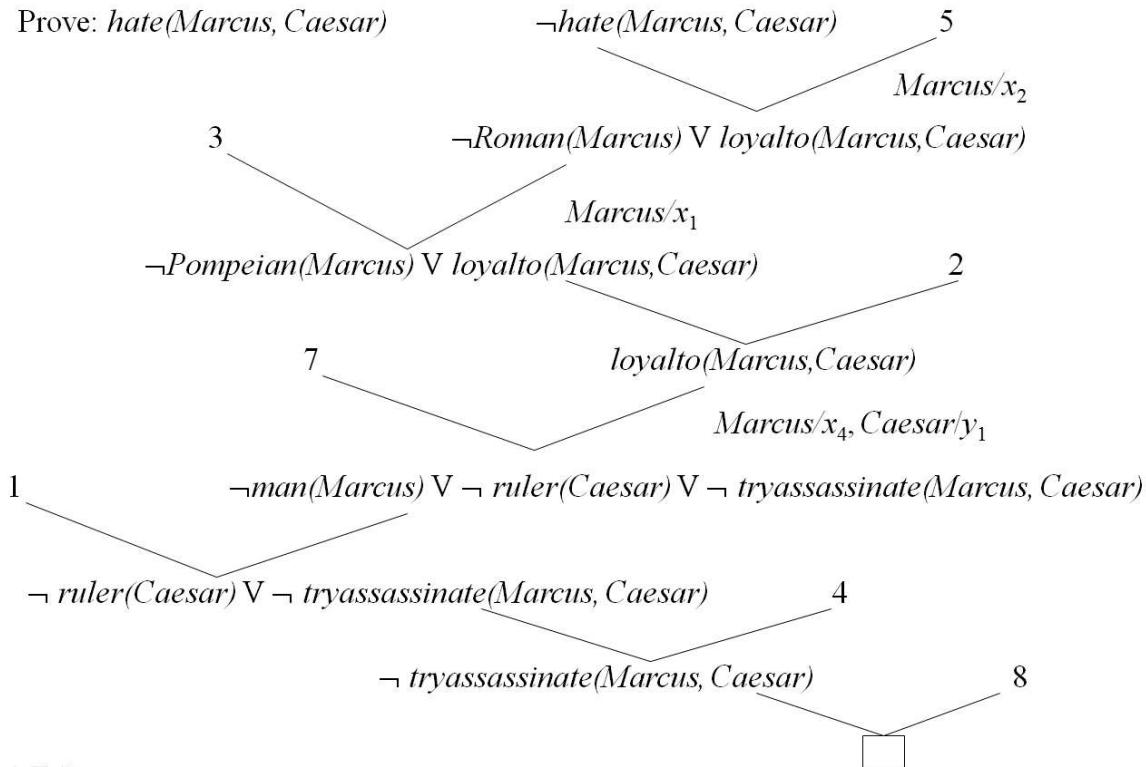
- A Predicate Logic Example

1. Marcus was a man
 $\text{Man}(\text{Marcus})$
2. Marcus was a Pompeian
 $\text{Pompeian}(\text{Marcus})$
3. All Pompeians were Romans
 $\forall x: \text{Pompeian}(x) \rightarrow \text{Roman}(x)$
4. Caesar was a ruler
 $\text{Ruler}(\text{Caesar})$
5. All Romans were either loyal to Caesar or hated him
 $\forall x: \text{Roman}(x) \rightarrow \text{Loyalto}(x, \text{Caesar}) \vee \text{Hate}(x, \text{Caesar})$
6. Everyone is loyal to someone
 $\forall x: \exists y: \text{Loyalto}(x, y)$
7. People only try to assassinate rulers they aren't loyal to
 $\forall x: \forall y: \text{Person}(x) \wedge \text{Ruler}(y) \wedge \text{Tryassassinate}(x, y) \rightarrow \neg \text{Loyalto}(x, y)$
8. Marcus tried to assassinate Caesar
 $\text{Tryassassinate}(\text{Marcus}, \text{Caesar})$
9. All men are people
 $\forall x: \text{Man}(x) \rightarrow \text{Person}(x)$

- A Resolution Proof

1. $\text{Man}(\text{Marcus})$
2. $\text{Pompeian}(\text{Marcus})$
3. $\neg \text{Pompeian}(x_1) \vee \text{Roman}(x_1)$
4. $\text{Ruler}(\text{Caesar})$
5. $\neg \text{Roman}(x_2) \vee \text{Loyalto}(x_2, \text{Caesar}) \vee \text{Hate}(x_2, \text{Caesar})$
6. $\text{Loyalto}(x_3, y_2)$
7. $\neg \text{Man}(x_4) \vee \neg \text{Ruler}(y_1) \vee \neg \text{Tryassassinate}(x_4, y_1) \vee \neg \text{Loyalto}(x_4, y_1)$
8. $\text{Tryassassinate}(\text{Marcus}, \text{Caesar})$

Prove: $\text{hate}(\text{Marcus}, \text{Caesar})$



- A Resolution Proof

5, 3, 2, 7, 1, 4, 8