Unit 1. Introduction to Intelligent Agents and Systems

Intelligent Systems

Escuela Superior de Informatica de Ciudad Real

Universidad de Castilla-La Mancha

Contents

Definitions I

1. **Systems** that **think** like **humans**:
    - ‘The exciting new effort to make computers think ... machines with minds, in the full and literal sense” (Haugeland, 1985)
    - “The automation of activities that we associate with human thinking, activities such as decision-making, problem solving, learning ...” (Bellman, 1978)
2. **Systems** that **think rationally**
3. **Systems** that **act** like **humans**
4. **Systems** that **act** rationally

Definitions II

1. **Systems** that **think** like **humans**
2. **Systems** that **think rationally**:
   - ‘The study of mental faculties through the use of computational models” (Charniak and McDermott, 1985)
   - “The study of the computations that make it possible to perceive, reason, and act” (Winston, 1992)
3. **Systems** that **act** like **humans**
4. **Systems** that **act** rationally

Definitions III

1. **Systems** that **think** like **humans**:
2. **Systems** that **think rationally**
3. **Systems** that **act** like **humans**:
    - "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990)
    - "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)
4. **Systems** that **act** rationally

Definitions IV

1. **Systems** that **think** like **humans**:
2. **Systems** that **think rationally**
3. **Systems** that **act** like **humans**
4. **Systems** that **act** rationally:
    - "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990)
    - "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

Acting humanly: The Turing test I

Turing (1950) "**Computing** machinery and **intelligence**":

- "*Can* **machines** *think*?" $\longrightarrow$ "*Can* **machines** *behave* **intelligently**?"
- It was **designed** to provide a **satisfactory** operational **definition** of intelligence.

Acting humanly: The Turing test II

- **Operational** test for **intelligent** behavior: the *Imitation Game*
- Turing defined **intelligent** behavior as the ability to achieve **human-level** performance in all **cognitive** tasks, sufficient to fool an **interrogator**.
- Roughly **speaking**, the test he proposed is that the **computer** should be interrogated by a **human** via a teletype, and **passes** the test if the **interrogator** cannot tell if there is a computer or a **human** at the other end.

Acting humanly: The Turing test III

- **Predicted** that by **2000**, a machine might have a **30 %** chance of **fooling** a **person** for 5 **minutes**
- **Anticipated** all major **arguments** against AI in **following** 50 years: https://en.wikipedia.org/wiki/Computing_Machinery_and_Intelligence#Nine_common_objections.

Acting humanly: The Turing test IV

Suggested major components of AI: **knowledge**, reasoning, language **understanding**, learning

- Represent **knowledge** and **reason** with it because this enables us to reach good **decisions** in a wide variety of situations.
- Generate **comprehensible** sentences in **natural language** because saying those sentences helps us get by in a complex **society**.

Thinking humanly: Cognitive Science I

- If we are going to say that a given **program** thinks like a **human**, we must have some way of **determining** how humans think. We need to get inside the actual **workings** of human minds.
- There are two **ways** to do this: through **introspection**–trying to **catch** our own thoughts as they go by–or through **psychological** experiments.

Thinking humanly: Cognitive Science II

- Once we have a **sufficiently** precise theory of the mind, it becomes **possible** to express the theory as a **computer** program
- The **interdisciplinary** field of cognitive **science** brings together **computer** models from AI and experimental techniques from **psychology** to try to construct precise and **testable** theories of the workings of the **human** mind

Thinking rationally: Laws of Thought I

- **Aristotle**: **first** to attempt to codify "**right thinking**", that is, **irrefutable** reasoning process.
- 'Socrates is a **man**; all men are **mortal**; therefore Socrates is mortal." These **laws of thought** were supposed to **govern** the operation of the **mind**, and initiated the field of **logic**
- Several **Greek** schools developed various forms of *logic*: *notation* and *rules of derivation* for thoughts; may or may not have proceeded to the idea of **mechanization**

Thinking rationally: Laws of Thought II

- Direct line through **mathematics** and **philosophy** to modern AI
- By **1965**, programs **existed** that could, given enough **time** and **memory**, take a description of a problem in **logical notation** and find the solution to the problem, if one exists

Thinking rationally: Laws of Thought III

Problems:

1. Not all **intelligent** behavior is **mediated** by logical **deliberation**
2. *What is the* **purpose** *of thinking*? What **thoughts** *should* **I** have out of all the thoughts (**logical** or otherwise) that **I** *could* have?

Acting rationally

- *Rational* **behavior**: doing the **right** thing
- The right thing: that which is **expected** to maximize goal **achievement**, given the available information
- Doesn't necessarily involve **thinking**—e.g., blinking reflex—but thinking should be in the **service** of rational **action**

Aristotle (Nicomachean Ethics)

Every art and every inquiry, and similarly every action and pursuit, is thought to aim at some good

Rational agents I

- An *agent* is an entity that perceives and acts
- This **course** is about **designing** *rational agents*
- **Abstractly**, an agent is a function from **percept** histories to **actions**:

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

- For any given class of **environments** and tasks, we seek the **agent** (or class of agents) with the best **performance**

Caveat

computational **limitations** *make perfect* **rationality** *unachievable*
(design best *program* for given machine **resources**)

Rational agents II

- In the "**laws of thought**" approach to AI, the whole emphasis was on **correct inferences**.
- Making correct inferences is **sometimes** part of being a **rational** agent, because one way to act **rationally** is to reason logically to the **conclusion** that a given action will achieve one's **goals**, and then to act on that **conclusion**.

Rational agents III

- On the other hand, correct **inference** is not all of **rationality**, because there are often **situations** where there is no probably correct thing to do, yet something must still be done.
- There are also **ways** of acting **rationally** that cannot be **reasonably** said to involve inference. For example pulling one's hand off of a hot stove is a reflex **action** that is more successful than a slower action taken after careful deliberation.
- This course will therefore **concentrate** on general principles of **rational agents**, and on components for **constructing** them.

Contents

AI prehistory I

Philosophy	**logic**, methods of reasoning
	mind as physical system
	foundations of **learning**, language, rationality
Mathematics	formal **representation** and proof
	algorithms, **computation**, (un)decidability, (in)tractability, **probability**
Psychology	adaptation
	phenomena of **perception** and motor control
	experimental techniques (**psychophysics**, etc.)

Economics	formal theory of rational **decisions**
Linguistics	**knowledge** representation
	grammar
Neuroscience	plastic **physical** substrate for mental activity
Control theory	**homeostatic** systems, stability
	simple optimal agent **designs**

Potted history of AI I

1943	McCulloch & Pitts: Boolean circuit model of **brain**
1950	**Turing**'s "Computing Machinery and Intelligence"
1952–69	Look, Ma, **no hands**!
1950s	Early AI programs, including **Samuel's checkers** program, Newell & **Simon's Logic** Theorist, Gelernter's Geometry Engine
1956	**Dartmouth** meeting: "Artificial Intelligence" adopted
1965	**Robinson**'s complete algorithm for **logical** reasoning
1966–74	AI discovers computational **complexity** **Neural** network research almost **disappears**

Potted history of AI II

1969–79	Early development of **knowledge-based systems**
1980–88	Expert systems **industry** booms
1988–93	Expert systems **industry** busts: “AI Winter”
1985–95	Neural **networks** return to **popularity**
1988–	Resurgence of **probability**; general increase in technical depth “**Nouvelle AI**”: ALife, GAs, soft computing
1995–	**Agents**, agents, everywhere . . .
2003–	Human-level **AI back on** the agenda

Potted history of AI III

2004	**DARPA** Grand Challenge: competition **autonomous** vehicles in the Mojave Desert (AI and autonomous driving)
2006	Hinton: Beginning of **Deep learning** with the architecture Deep Belief Network (DBN).
2011	IBM's **Watson** defeats human champions on the show "Jeopardy!"(**NLP** capabilities of machines.)
2012	**AlexNet**: significant **breakthrough** in deep learning occurs (victory in the **ImageNet** Visual Recognition Challenge)
2014	**Facebook** AI Research (FAIR) introduces **DeepFace**.

2015	Google **DeepMind** develops **AlphaGo**,
2016	**Chatbots** become popular (Siri, Alexa, etc.)
2017	Generative Adversarial Networks (**GANs**): realistic data
2018	**OpenAI** releases the **GPT**: revolutionizing **NLP** by generating coherent and **high-quality** text.

Potted history of AI IV

2020	Google **DeepMind** presents **AlphaFold**, predict the **3D** structures of **proteins**.
2020	**AI** used in the detection and tracking of **COVID-19**
2021	**OpenAI** releases **GPT-3**, the largest language model to date.
2022	**ChatGPT**, **Deepfakes** and actors with AI rights, **OpenAI** launches **DALL-E2** revolutionises imaging
2023	Other **LLMs** (Large Language Models) or **conversational** chatbots: **LLaMA2** (Large Language Model **Meta** AI) and **Bard** (Google).
2024	**Local** LLMs: **Privacity**, costs, optimization, specialization, ... Ethics and Regulation: **European AI Act.** Advances in **Natural Language Generation**.

Potted history of AI V

2025	Evolving from the generation of texts and images towards the creation of **autonomous agents** (systems that can reason, make complex decisions, etc. without **human intervention**).
	Regulation and **Governance** of AI: Regulatory Debates in the US and the EU; Copyright, Ethics and the Humanoid AI Deception.
	AI in Education: **Cognitive** Laziness - The **challenge** is to use AI for **learning** without eroding critical thinking and creativity.

State of the art I

Which of the following can be done at present?

- *Play a decent game of table tennis*
- *Drive safely along a curving mountain road*
- *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*
- *Play a decent game of bridge*

State of the art II

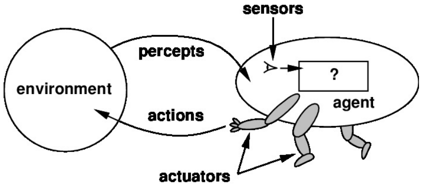
- *Discover and prove a new mathematical theorem*
- *Design and execute a research program in molecular biology*
- *Write an intentionally funny story*
- *Give competent legal advice in a specialized area of law*
- *Translate spoken English into spoken Swedish in real time*
- *Converse successfully with another person for an hour*
- *Perform a complex surgical operation*
- *Unload any dishwasher and put everything away*

Contents

Outline

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

Agents and environments I

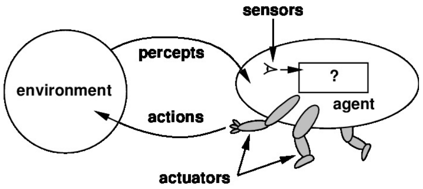


- *Agents* include **humans**, robots, **softbots**, thermostats, etc.
- The *agent function* maps from **percept** histories to **actions**:

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

- The *agent program* runs on the **physical** *architecture* to **produce** f

Agents and environments II



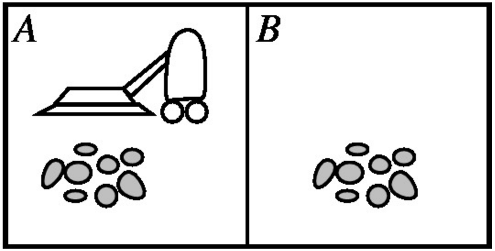
- **Control** Policy for the **Agent**
- **Perception** Action **Cycle** (loop, feedback)

Applications Of AI

Domain: environment, **sensors**, actuators.

- **Finance**: market, prices or world events, trades.
- **Robotics**: world, cameras/tactile sensors, motors.

Vacuum-cleaner world



- **Percepts**: location and **contents**, e.g., [*A*, *Dirty*]
- **Actions**: *Left*, *Right*, *Suck*, *NoOp*

A vacuum-cleaner agent I

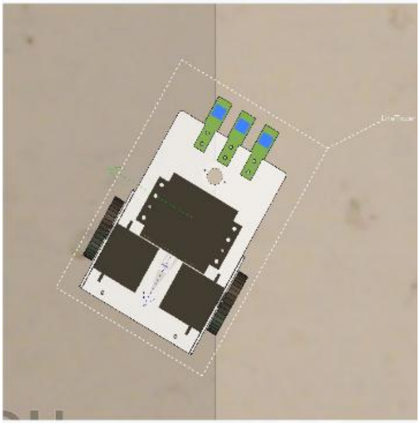
Percept sequence	Action
[A, *Clean*]	*Right*
[A, *Dirty*]	*Suck*
[B, *Clean*]	*Left*
[B, *Dirty*]	*Suck*
[A, *Clean*], [A, *Clean*]	*Right*
[A, *Clean*], [A, *Dirty*]	*Suck*
[A, *Clean*], [B, *Clean*]	*Left*
[A, *Clean*], [B, *Dirty*]	*Suck*
[A, *Dirty*], [A, *Clean*]	*Right*
[A, *Dirty*], [A, *Dirty*]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

A vacuum-cleaner agent II

```
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**? Every **entry** in the table for the **agent** function is filled **correctly**.
- What does it **mean** to do the **right** thing?
- What makes an agent **good** or bad, **intelligent** or stupid?

Lane Tracker Agent I



Lane Tracker Agent II

- **Lane Sensors**: Left Sensor, Middle Sensor and Right Sensor
- **Actuators**: MWL (Motor Wheel Left) and MWR (Motor Wheel Right).

Lane Tracker Agent III

- **Goal**: Track the lane (complete the path)
- **Environment Data**: Left Sensor $\{Y|B\}$, Middle Sensor $\{Y|B\}$, Right Sensor $\{Y|B\}$ where Y is Yellow (no line) and B is Black (line).
- **Actions**: MWL $\{0|1\}$, MWR $\{0|1\}$ where 0 is Reduce Velocity and 1 is Normal Velocity.

Lane Tracker Agent IV

- **Agent Program**:
  If Right Sensor==B then MWR=0; Trajectory deviation to the right.
  If Left Sensor==B then MWL=0; Trajectory deviation to the left.
  In other case MWL=MWR=1;

Rationality I

Fixed *performance measure* evaluates the *environment sequence*

- **amount** of dirt **cleaned**? (**dumping** the dirt on the **floor**).
- 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**?

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

Rationality II

What is **rational** at any given **time** depends on four things:

- The **performance measure** that defines the criterion of **success**.
- The agent's prior **knowledge** of the **environment**.
- The **actions** that the agent can **perform**.
- The agent's **percept** sequence to **date**.

Rational Agent

For each possible percept **sequence**, a rational agent should select an **action** that is expected to **maximize** its performance **measure**, given the evidence **provided** by the percept sequence and **whatever** built-in knowledge the agent has.

Rationality III

- 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 $\Longrightarrow$ 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** safety, destination, profits, legality, comfort, . . .
- **Environment** streets/freeways, traffic, pedestrians, weather, . . .
- **Actuators** steering, accelerator, brake, horn, speaker/display, . . .
- **Sensors** video, accelerometers, engine sensors, keyboard, GPS, . . .

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 I

Environments can have different **characteristics**.

- **Fully versus partially observable**. An environment is called fully observable if what your agent can sense at any point in time is completely sufficient to make the optimal decision i.e. its sensors can see the entire state of the environment. That is in contrast to some other environments where agents need memory to make the best decision.
- **Deterministic versus stochastic**. Deterministic environment is one where your agent's actions uniquely determine the outcome. In stochastic environment, there is certain amount of randomness.

Environment types II

- **Episodic vs. sequential**: In an episodic task environment, the agent's experience is divided into atomic episodes. In each episode the agent receives a percept and then performs a single action. Crucially, the next episode does not depend on the actions taken in previous episodes (an agent in an assembly line bases each decision on the current part, regardless of previous decisions); moreover, the current decision doesn't affect whether the next part is defective. In sequential environments, on the other hand, the current decision could affect all future decisions (Chess and taxi driving). Episodic environments are much simpler than sequential environments because the agent does not need to think ahead.

Environment types III

- **Static vs. dynamic**: If the environment can change while an agent is deliberating, then we say the environment is dynamic for that agent; otherwise, it is static. Static environments are easy to deal with because the agent need not keep looking at the world while it is deciding on an action, nor need it worry about the passage of time. Dynamic environments, on the other hand, are continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing. If the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is semidynamic. Taxi driving is clearly dynamic: the other cars and the taxi itself keep moving while the driving algorithm dithers about what to do next. Chess, when played with a clock, is semidynamic. Crossword puzzles are static.

Environment types IV

- **Discrete versus continuous**. A discrete environment is one where you have finitely many action choices, and finitely many things you can sense. For example, in chess there's finitely many board positions, and finitely many things you can do.
- **Single agent vs. multiagent**: The distinction between single-agent and multiagent environments may seem simple enough (crossword puzzle agent vs. agent playing chess is in a two-agent environment). But, Does an agent A (the taxi driver for example) have to treat an object B (another vehicle) as an agent, or can it be treated merely as an object behaving according to the laws of physics?.

Environment types V

- **Benign versus adversarial environments**. In benign environments, the environment might be random. It might be stochastic, but it has no objective on its own that would contradict your own objective. For example, weather is benign. Contrast this with adversarial environments, such as many games, like chess, where your opponent is really out there to get you.

Environment types VI

	Solitaire	Backgammon	Internet shopping	Taxi
Observable	Yes	Yes	No	No
Deterministic	Yes	No	Partly	No
Episodic	No	No	No	No
Static	Yes	Semi	Semi	No
Discrete	Yes	Yes	Yes	No
Single-agent	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

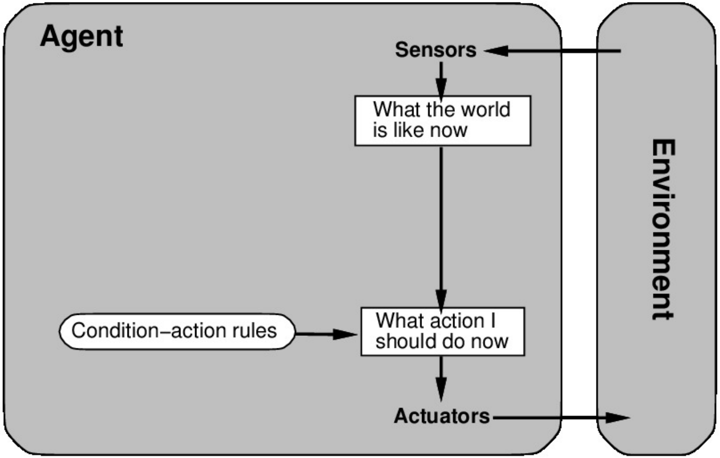
- agent = architecture + program
- **Program**: Table-driven-agent is the **simplest**.
- **Architecture**: computing **device** with **physical** sensors and actuators.

Four basic **types** of programs 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 I



Simple reflex agents II

agent program

```
**function** SIMPLE-REFLEX-AGENT(percept) **returns** an action
persistent: rules, a set of condition-action rules
        state ← INTERPRET-INPUT(percept)
        rule ← RULE-MATCH(state, rules)
        action ← rule.ACTION
        return action
```

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

    if status = Dirty then . . .
```

Model Based Agents (Internal State)

- The most **effective** way to handle partial **observability** is for the agent to keep track of the part of the **world** it can 't see now.
- That is, the agent should **maintain** some sort of internal state that **depends** on the percept history and thereby **reflects** at least some of the **unobserved** aspects of the current state (car: previous frame).
- The **current** percept is **combined** with the **old** internal state to generate the **updated** description of the current **state**, based on the agent's **model** of how the world **works**.

Agent Program with internal state

- This program is very **similar** to the reflex agent program.
- The **state** is a **description** of the world and the **rules** are of the class **condition/action**.

Program

```
1: function AGENT-WITH-STATE(e)
2:         perception ← percept(e)
3:         state ← next(state,perception)
3:         rule ← SelectAccion(state,rules) (predefined rules)
4:         action← ActionRule(rule)
5:         return action
6:end function
```

Vacuum Cleaner agent with state

Programa

```
function AGENT-STATE-VCLEANER(e=[location, state]) return an action
static last_A=$\infty$ ,last_B=$\infty$
    while(TRUE)
        last_A=++; last_B=++;
        if (state=DIRTY) then
            if (location=A) then last_A=0 else last_B=0;
                return SUCK
        else if location=A then
            if (last_B $>$ 3) return RIGHT else NoOP
        else if location=B then
            if (last_A $>$ 3) return LEFT else NoOP
    end while;
end function
```

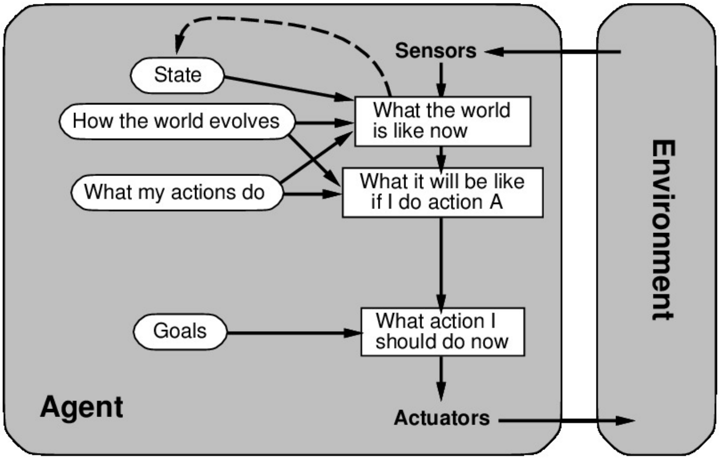
Goal-based agents I

- **Knowing** something about the current **state** of the environment is not always **enough** to decide what to do.
- For example, at a **road** junction, the taxi can turn **left**, turn right, or go **straight** on. The correct **decision** depends on where the taxi is **trying** to get to.

Goal-based agents II

- That is, as well as a current state **description**, the agent needs some sort of **goal** information that **describes** situations that are **desirable**, for example, being at the passenger's **destination**.
- The agent **program** can combine this with the **model** (the same information as was used in the model- based reflex agent) to choose **actions** that achieve the goal.

Goal-based agents III



Utility-based agents I

- Goals alone are not **enough** to generate **high-quality behavior** in most environments.
- For example, many **action sequences** will get the taxi to its **destination** (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others.
- **Goals** just provide a crude **binary** distinction **between** "happy" and "unhappy" states.

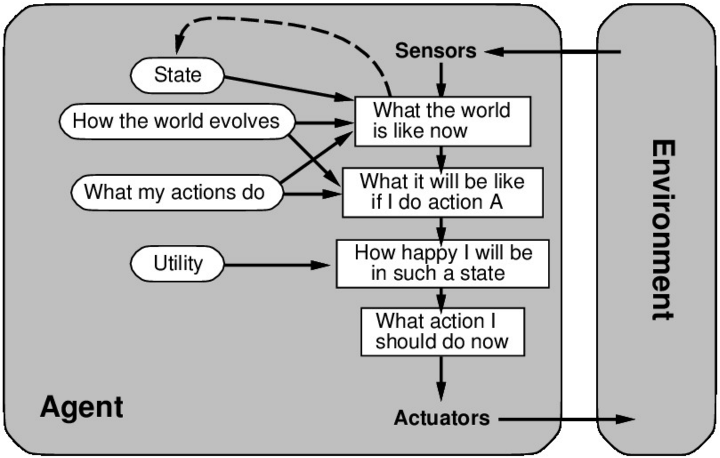
Utility-based agents II

- A more general **performance** measure should allow a **comparison** of different world states **according** to exactly how happy they would make the **agent**.
- Because "happy" does not sound very **scientific**, economists and utility computer **scientists** use the term utility instead.
- We have **already** seen that a **performance** measure **assigns** a score to any given sequence of **environment** states, so it can easily distinguish between more and less **desirable** ways of utility function getting to the taxi's **destination**.

Utility-based agents III

- An agent's utility function is essentially an **internalization** of the performance **measure**.
- If the **internal** utility function and the external **performance** measure are in **agreement**, then an agent that chooses actions to **maximize** its utility will be **rational** according to the external **performance** measure.

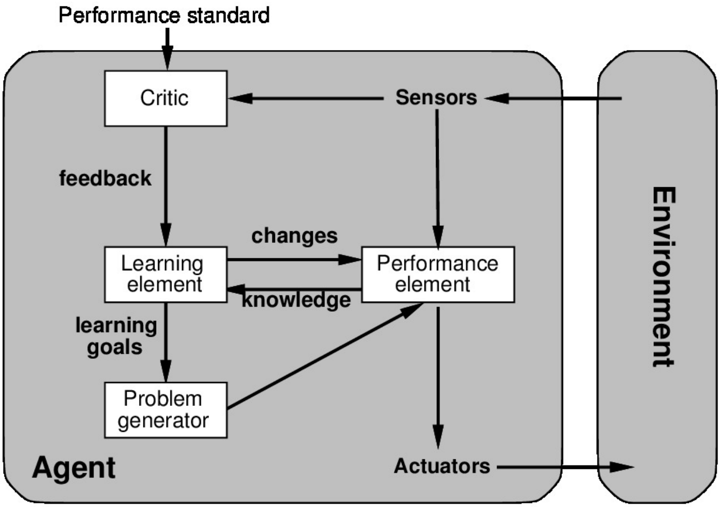
Utility-based agents IV


Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Learning agents I

- **Learning** allows the agent to **operate** in initially **unknown** environments and to become more **competent** than its **initial** knowledge alone might allow.
- A learning **agent** is divided **four** conceptual **components**.

Learning agents II



Summary I

- *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**

Summary II

- *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*

Unit 1. Introduction to Intelligent Agents and Systems

Intelligent Systems

Escuela Superior de Informatica de Ciudad Real

Universidad de Castilla-La Mancha

Unit 2. Problem solving and search

Intelligent Systems

Escuela Superior de Informatica de Ciudad Real

Universidad de Castilla-La Mancha

Contents

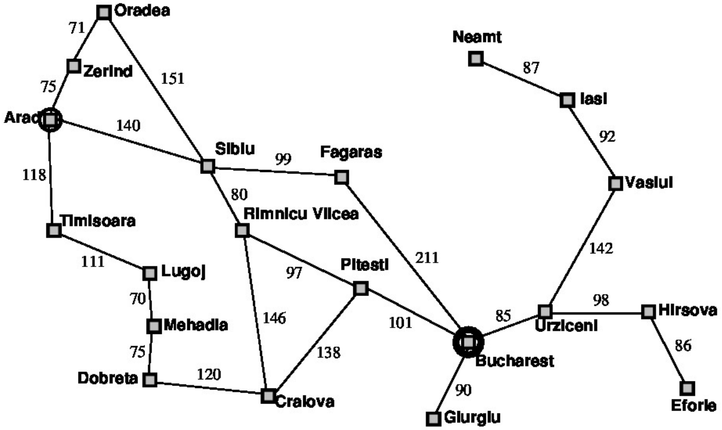
Introduction

- The main **objective** is to **build** agents that can **plan** ahead to solve **problems**.
- In this kind of **problems** there are many **states**.

Example: Romania

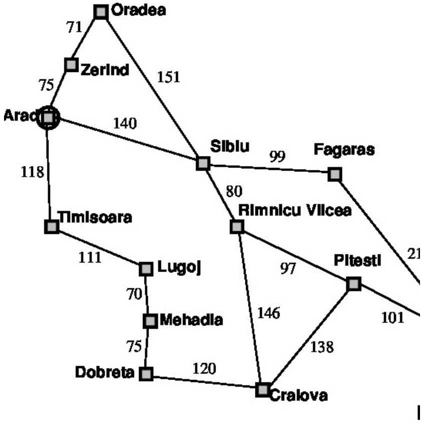
- On holiday in **Romania**; currently in **Arad**.
- **Flight** leaves tomorrow from **Bucharest**
- *Formulate goal*: be in **Bucharest**
- *Formulate problem*:
  - *states*: various **cities**
  - *actions*: **drive** between cities
- *Find solution*: **sequence** of **cities**, e.g., Arad, Sibiu, Fagaras, Bucharest

Example: Romania



Exercise

- Is there a **solution** for the problem of driving from **Arad** to **Bucharest**?



Problem-solving agents

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

Note: this is *offline* problem solving; solution executed “eyes closed”. *Online* problem solving involves acting without complete knowledge.

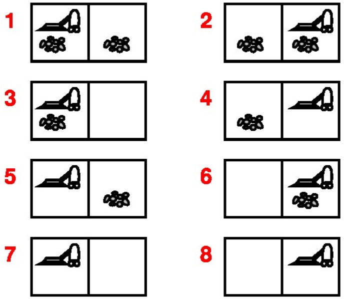
Contents

Problem types and State Spaces

- *Deterministic, fully observable*: *single-state problem*. **Agent** knows exactly which **state** it will be in; solution is a sequence
- *Non-observable*: *conformant problem*. Agent may have **no idea** where it is; solution (if any) is a **sequence**
- *Nondeterministic* and/or *partially observable*: *contingency problem*. **Percepts** provide *new* information about current **state** solution is a *contingent plan* or a *policy* often *interleave* search, execution
- *Unknown state space*: *exploration problem* ("**online**")

Example: vacuum world

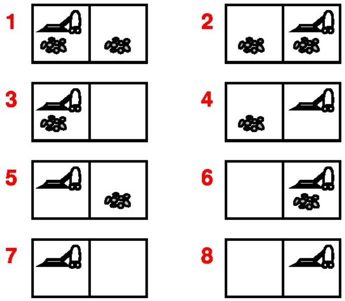
Single-state, start in #5. **Solution**



Example: vacuum world

Single-state, start in #5. **Solution** [*Right*, *Suck*]

Sensorless, start in $\{1,2,3,4,5,6,7,8\}$ e.g., *Right* goes to $\{2,4,6,8\}$. **Solution**

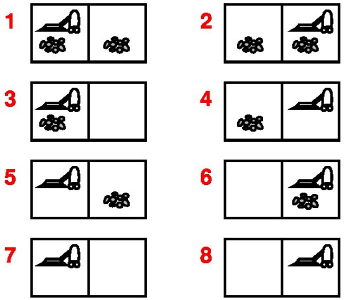


Example: vacuum world

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

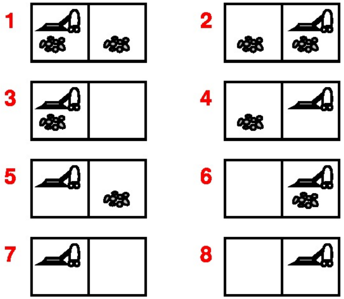


Example: vacuum world

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*]



Contents

Single-state problem formulation

- *initial state* e.g., "at **Arad**"
- *successor function* $S(x) =$ set of **action–state-cost** e.g., $S(Arad) = \{(Arad \rightarrow Zerind), Zerind, 75\ , \ldots\}$
- 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

Solution

A *solution* is a sequence of actions leading from the **initial** state to a **goal** state

Selecting a state space I

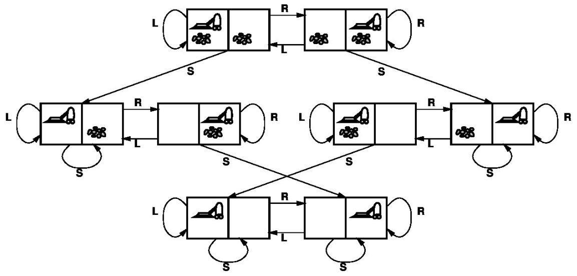
- **Real world is absurdly complex**: state space must be *abstracted* for problem **solving**
- **(Abstract) state**: set of real **states**
- **(Abstract) action**: complex combination of real actions e.g., "Arad $\rightarrow$ Zerind" represents a complex set of possible routes, detours, rest stops, 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

Selecting a state space II

- **abstract** action should be “**easier**” than the **original** problem!
- The choice of a good **abstraction** thus involves removing as much **detail** as possible while **retaining** validity and ensuring that the abstract **actions** are easy to **carry out**.
- Were it not for the **ability** to construct useful abstractions, intelligent **agents** would be completely **swamped** by the real world.

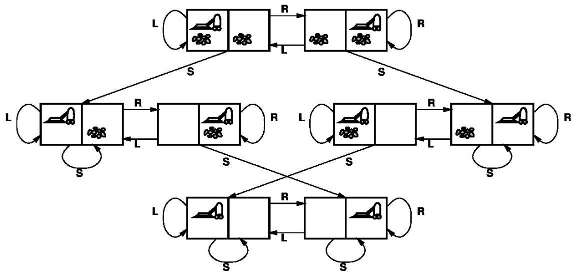
Contents

Example: vacuum world state space graph I



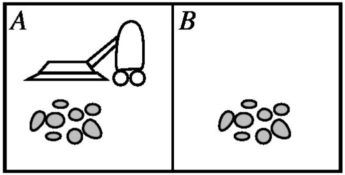
states: ?
actions: ?
goal test: ?
path cost: ?

Example: vacuum world state space graph II



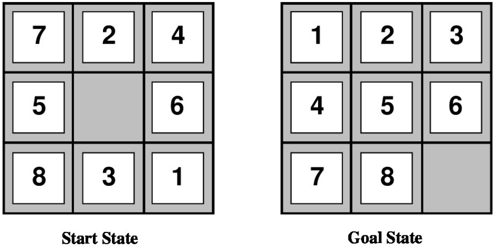
states: integer dirt and robot locations (ignore dirt *amounts* etc.)
actions: *Left*, *Right*, *Suck*, *NoOp*
goal test: no dirt
path cost: 1 per action (0 for *NoOp*)

Exercise I



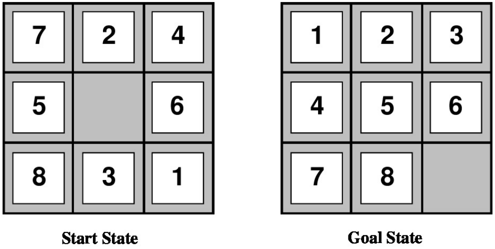
- **Power switch**: on/off/sleep
- **Dirt sensing camera**: on/off
- **Brushes height**: 1/2/3/4/5
- **Positions**: 10 (not only A and B)
- How many **states** in the **state space** are?

Example: The 8-puzzle I



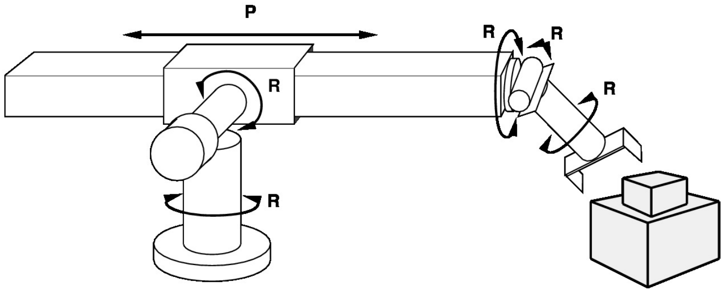
states: ?
actions: ?
goal test: ?
path cost: ?

Example: The 8-puzzle II



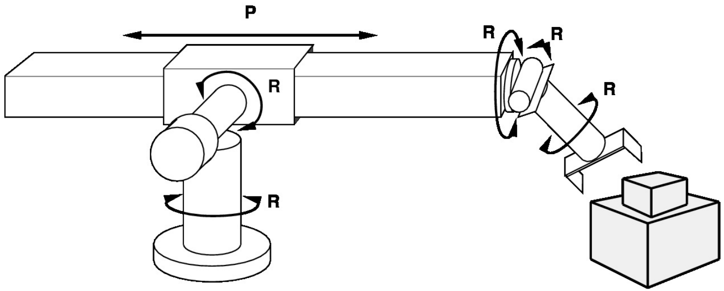
states: integer locations of tiles (ignore intermediate positions)
actions: move blank left, right, up, down (ignore unjamming etc.)
goal test: goal state (given)
path cost: 1 per move
[Note: optimal solution of *n*-Puzzle family is NP-hard]

Example: robotic assembly



states:
actions:
goal test:
path cost:

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

The missionaries and cannibals problem I

- Three **missionaries** and three **cannibals** are on one side of a **river**, along with a **boat**.
- The **boat** can hold one or two **people** (and obviously cannot be paddled to the other side of the river with zero people in it).
- The **goal** is to get **everyone** to the other **side**, without ever leaving a group of missionaries **outnumbered** by cannibals.
- Your task is to **formulate** this as a search **problem**.

The missionaries and cannibals problem II

1. Define a **state** representation.
2. Give the **initial** and goal **states** in this representation.
3. Define the **successor** function in this **representation**.
4. What is the **cost** function in your successor **function**?
5. What is the total **number** of reachable **states**?

Explorers problem

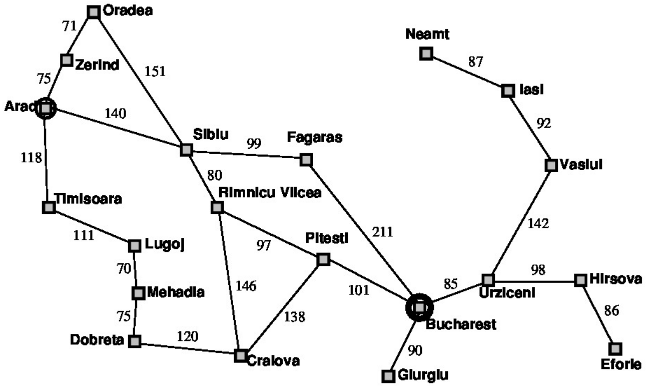
Four explorers: Alex, Brook Chris and Dusty need to cross a river in a small boat that can only support a weight of 100kg. If Alex weighs 90kg, Brook weighs 80kg, Chirs weighs 60kg, Dusty weighs 40kg and they carry 20kg of food and material. How could they cross the river?

1. The student must define the state space for this case.
2. The student must define the problem.

Contents

Example route finding

- Elements of the problem?
- Frontier, explored states and unexplored?



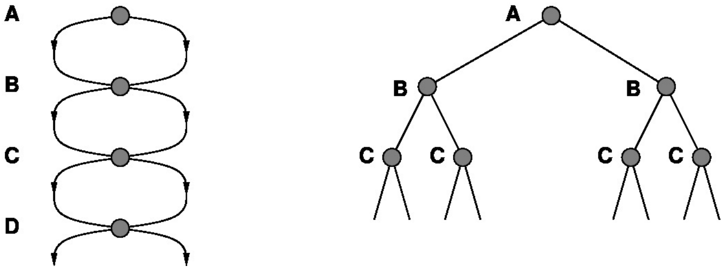
Tree search algorithms

- 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
```

Repeated states

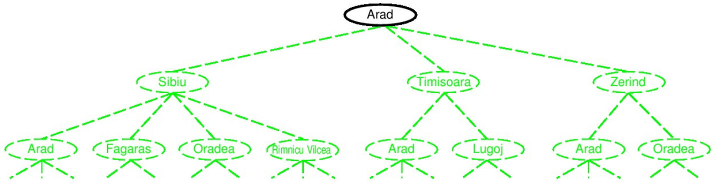
- Failure to **detect** repeated **states** can turn a linear **problem** into an exponential one!



Graph search

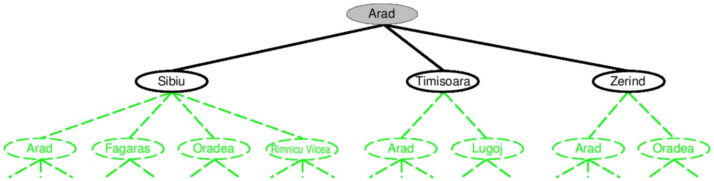
```
**function** GRAPH-SEARCH( *problem*, *fringe*) **returns** a solution, or failure

*closed* ← an empty set
*fringe* ← INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), *fringe*)
**loop do**
if *fringe* is empty **then return** failure
*node* ← 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* ← INSERTALL(EXPAND(*node*, *problem*), *fringe*)
**end**
```

Tree search example I



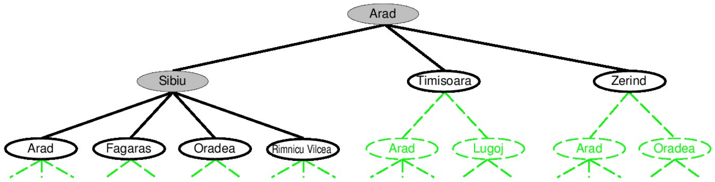
Frontier	Arad

Tree search example II



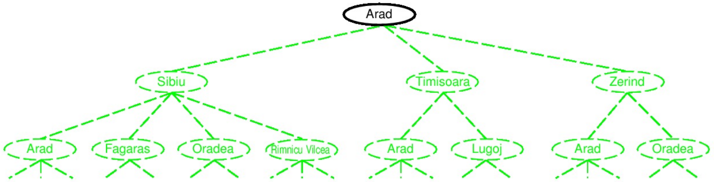
Frontier	Sibiu	Timisoara	Zerind

Tree search example III



Frontier	Timisoara	Zerind	Arad	Fagaras	Oradea	Rimn. V.

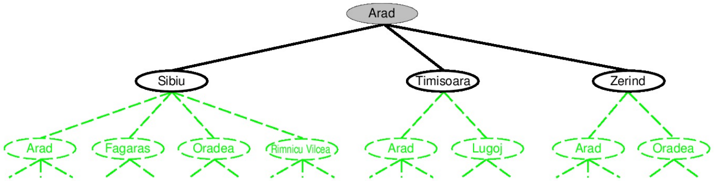
Graph search (Erroneous) I



Frontier	Arad

Closed	

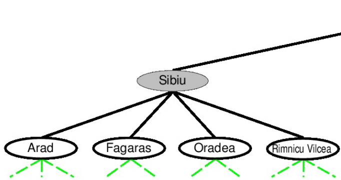
Graph search (Erroneous) II



Frontier	Sibiu	Timisoara	Zerind

Closed	Arad

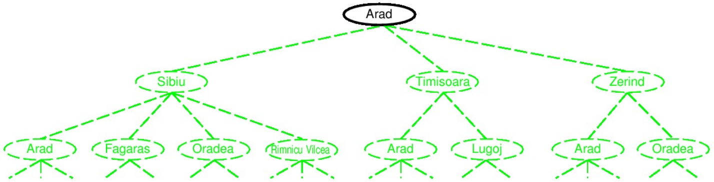
Graph search (Erroneous) III



Frontier	Timisoara	Zerind	Fagaras	Oradea	Rimnicu Vlicea

Closed	Arad	Sibiu

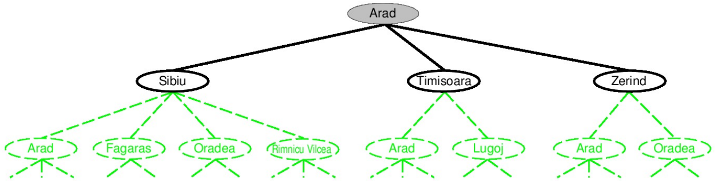
Graph Search Example I



Frontier	Arad

Closed	

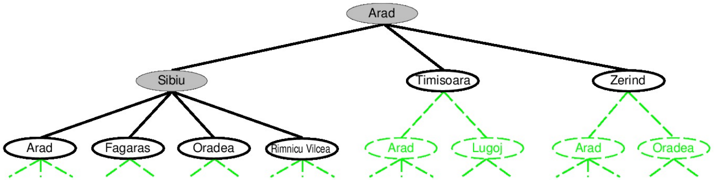
Graph Search Example II



Frontier	Sibiu	Timisoara	Zerind

Closed	Arad

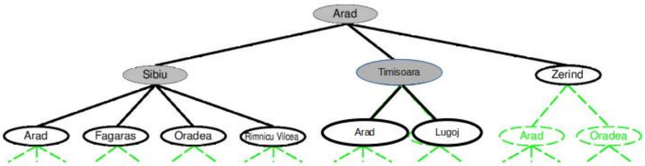
Graph Search Example III



Frontier	Timisoara	Zerind	Arad	Fagaras	Oradea	R. Vlicea

Closed	Arad	Sibiu

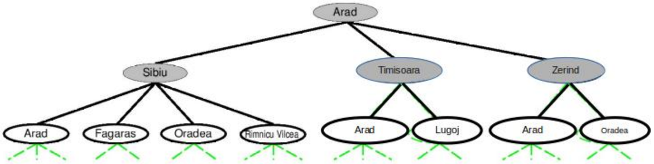
Graph Search Example IV



Zerind	Arad	Fagaras	Oradea	R. Vlicea	Arad	Lugoj

Closed	Arad	Sibiu	Timisoara

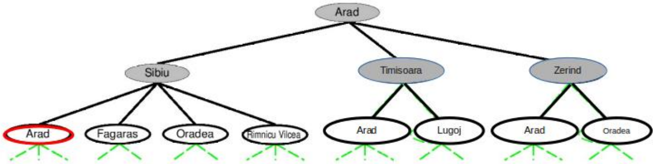
Graph Search Example V



Arad	Fagaras	Oradea	R. Vlicea	Arad	Lugoj	Arad	Oradea

Closed	Arad	Sibiu	Timisoara	Zerind

Graph Search Example VI



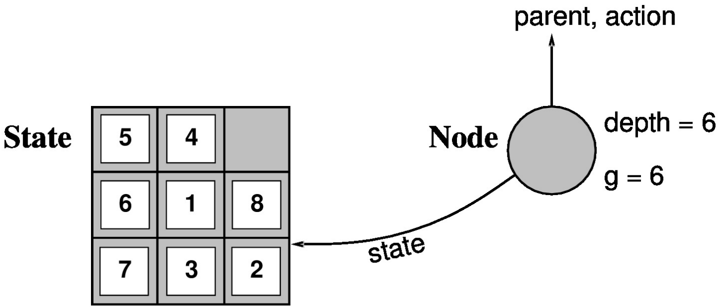
Fagaras	Oradea	R. Vlicea	Arad	Lugoj	Arad	Oradea

Closed	Arad	Sibiu	Timisoara	Zerind

Implementation: states vs. nodes I

- A *state* is a (representation of) a **physical configuration**
- A *node* is a data structure **constituting** part of a search tree includes:
    - *parent* and action that generates this new state executed by the parent.
    - *depth*
    - *path cost* $g(x)$
- States do **not** have **parents**, children, **depth**, or path cost!

Implementation: states vs. nodes II



- The **Expand function** creates new nodes, **filling** in the various fields and using the **SuccessorFn** of the **problem** to create the corresponding **states**.

Algorithm to be implemented I

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

closed ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
loop do
if fringe is empty then return failure
node ← 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 ← INSERTALL(EXPAND(node, problem), fringe)
end
```

Algorithm to be implemented II

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

Issues to be addressed I

A lot of new concepts, **structures**, ideas, etc. have been **introduced** and can generate a series of doubts or questions for the student.

- Why does the **algorithm** allow to add states already visited at the frontier? Wouldn't it be more efficient to **check** if they are in the list of **visited** states before adding them?
- If I really have my tree structure with **nodes** that have a **pointer** or **reference** to the **parent**? Why do I need the frontier?

Issues to be addressed II

- Why does the **algorithm** specify that the first element of the frontier is taken (and deleted) but does not specify where **new elements** should be **inserted**?
- Why does the **algorithm** test whether a **node** contains a state that satisfies the **objective function** only when it is **extracted** from the **frontier** and not when it is inserted? Wouldn't this be much more **efficient**?

Issues to be addressed III

- Which of the two **algorithms** are we going to use finally?, **tree** search or **graph** search?
- Why do we **speak** of **search algorithms** instead of **search algorithm** if we are **only** going to **use** the graph search algorithm?
- If I have to **solve** a search problem, Do I have to **specify** all the fields of every node in the tree? In this case, which **search** node format or which fields should **appear**?

Issues to be addressed IV

At present only these issues can be answered:

- Why does the **algorithm** allow to add states already visited at the frontier? Wouldn't it be more efficient to **check** if they are in the list of **visited** states before adding them?
  If we check when **introducing** in the frontier it must be done **for all successors**. This query, if the list of visitors has many elements (as happens in this type of problem) can be very **temporarily expensive**. If we make it when **extracting** from the frontier, only a single query is made. Moreover, for some strategies (what is a strategy?), nodes with repeated states added to the frontier üsually remain in final positions", so it is **difficult** to get them to leave the fringe and this single consultation is not really carried out.

Issues to be addressed V

- If I really have my tree structure with **nodes** that have a **pointer** or **reference** to the **parent**? Why do I need the frontier? We lack the basic **primitives** to **generate** and browse the tree, so we need an auxiliary structure that allows not only access to the nodes but also to **establish** different order criteria for the generation and/or **exploration** of the tree.

Issues to be addressed VI

- Which of the two **algorithms** are we going to use finally?, **tree** search or **graph** search?
  The one that incorporates **control of visited or explored states**. That is to say, the search in Graph. Otherwise, trivial problems can become very complex to solve.

Search strategies I

Strategy

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**?

Search strategies II

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

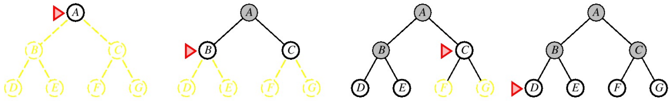
Use only the information available in the problem definition

- **Breadth**-first search
- **Uniform**-cost search
- **Depth**-first search
- **Depth**-limited search
- **Iterative** deepening search

Breadth-first search I

- Expand **shallowest** (shortest path) unexpanded **node**
- *Implementation*: fringe is a **FIFO** queue, i.e., new **successors** go at end

Breadth-first search II



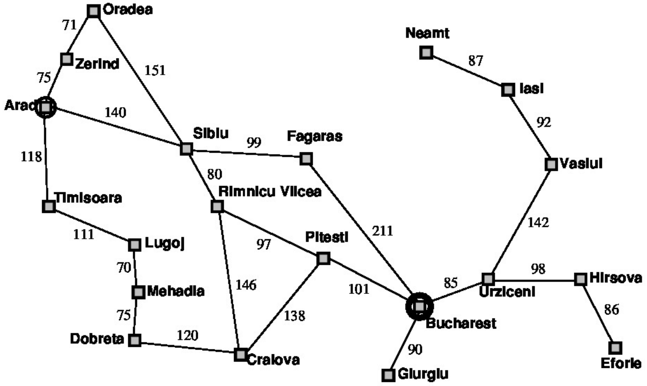
Frontier | A

Frontier | B | C

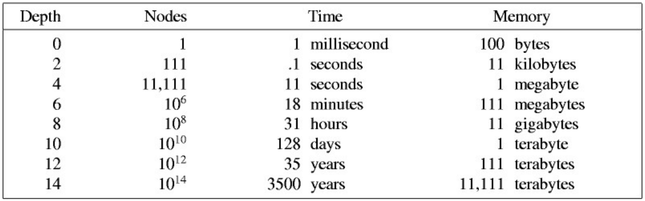
Frontier | C | D | E

Frontier | D | E | F | G

Example: Breadth-first search



Properties of breadth-first search I

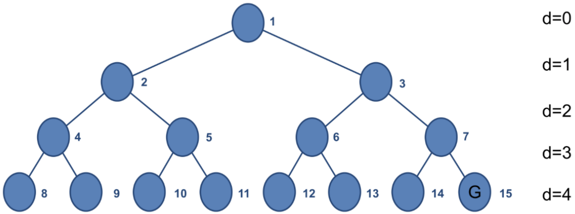


Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

- **Complete**: Yes (if b is finite)
- **Time**: $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, 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.

Properties of breadth-first search II

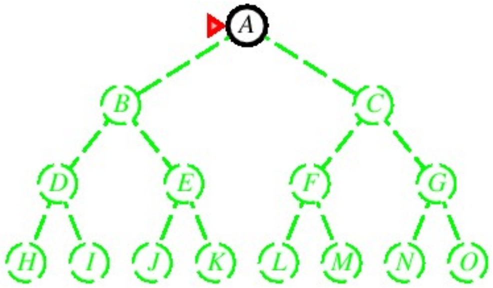
- In the previous analysis it is assumed the worst possible situation that is be the one represented in this figure (**goal** at far right leaf)



Depth-first search I

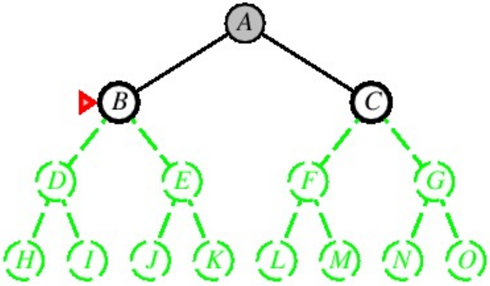
- Expand **deepest** unexpanded node
- *Implementation*: *fringe* = **LIFO** queue, i.e., put **successors** at front

Depth-first search II



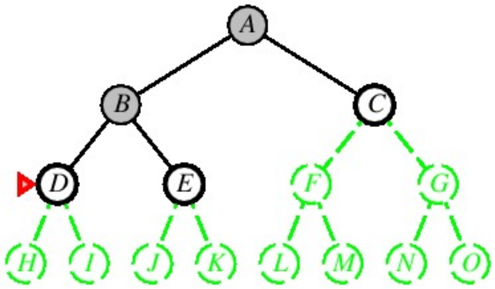
Frontier	A

Depth-first search III



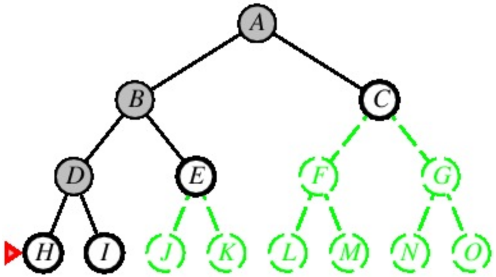
Frontier	B	C

Depth-first search IV



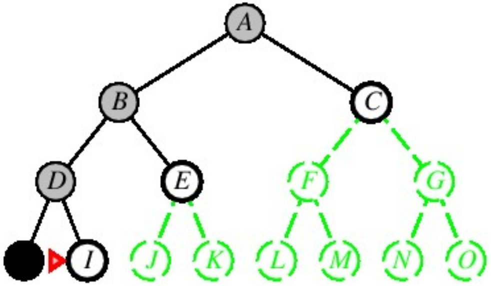
Frontier	D	E	C

Depth-first search V



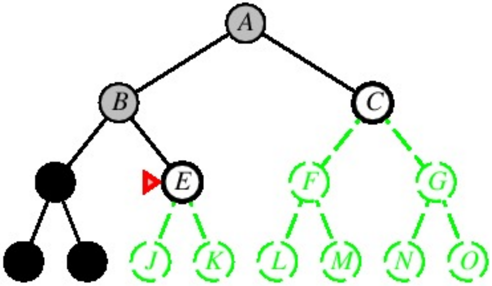
Frontier	H	I	E	C

Depth-first search VI



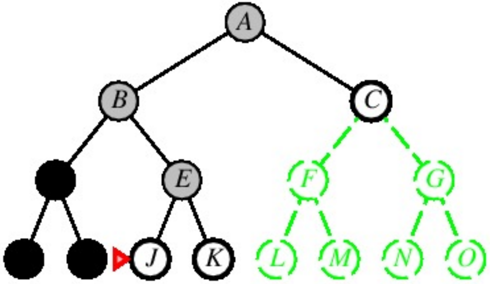
Frontier	I	E	C

Depth-first search VII



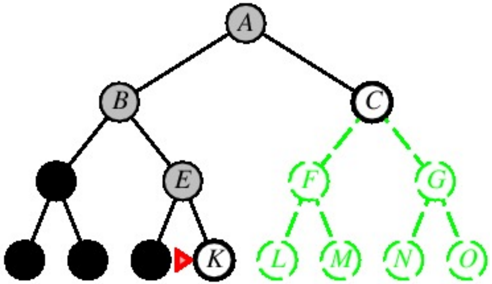
Frontier	E	C

Depth-first search VIII

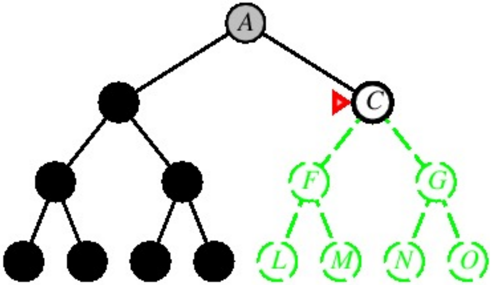


Exercise: Simulate the frontier during the rest of the search.

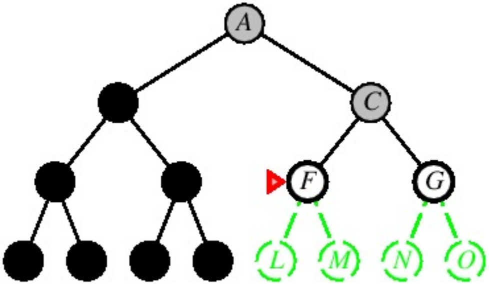
Depth-first search IX



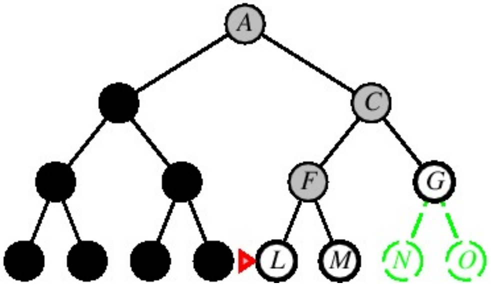
Depth-first search X



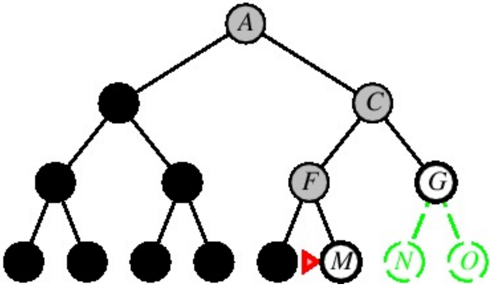
Depth-first search XI



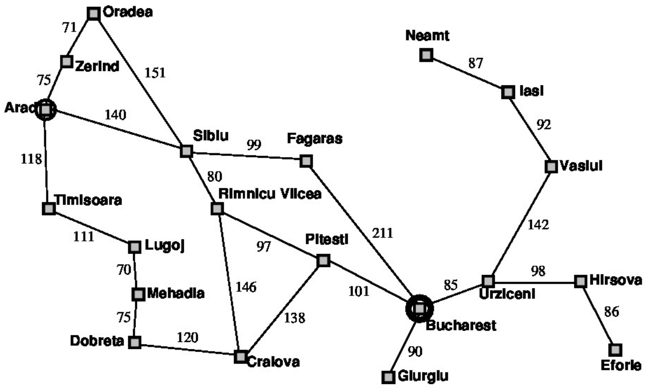
Depth-first search XII



Depth-first search XIII

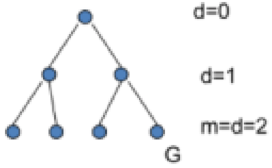


Example: Depth-first search



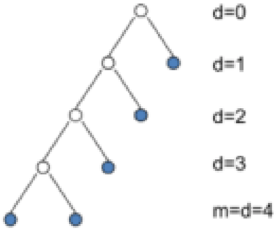
Properties of depth-first search I

- **Complete** No: **fails** in **infinite-depth** spaces, spaces with loops. Modify to avoid repeated states along path $\Rightarrow$ 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



Properties of depth-first search II

- **Space**: $O(bm)$, i.e., linear **space**!



- **Optimal**: **No**

Depth-limited search

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

DLS: Problems with "Boolean" pruning I

- The maximum depth **limits** the **maximum number of actions** an **agent** can execute to reach a state satisfying the **goal test**.
- For instance, if l=4 with the initial state being Arad and the goal state being Bucharest, there are two possible solutions: (**Go to Sibiu, Go to Fagaras, Go to Bucharest**) and (**Go to Sibiu, Go to RV, Go to Pitesti, Go to Bucharest**) with depth=3 and depth=4, respectively.

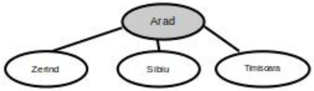
DLS: Problems with "Boolean" pruning II

- So, if we execute the search algorithm with **DLS and l=4**, one of the two **solutions** must be reached.
- But what happens if the **exploration order of the successors** from Arad is **Zerind, Sibiu and Timisoara**?
- **Simulate** the search algorithm with visited control.

Frontier	Arad (0)

Closed	

DLS: Problems with "Boolean" pruning III



Frontier	Zerind(1)	Sibiu(1)	Timisoara(1)

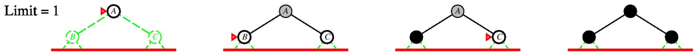
Closed	Arad

Iterative deepening search

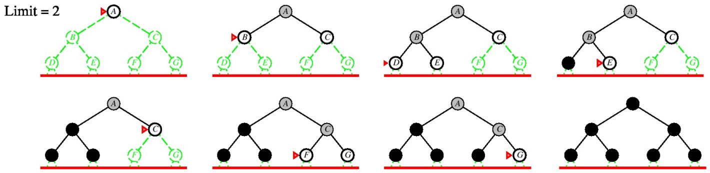
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem

    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
end
```

Iterative deepening search $l = 0$

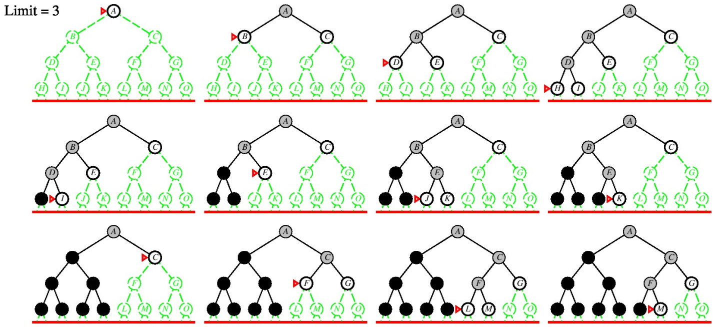
Iterative deepening search $l = 1$



Iterative deepening search $l = 2$



Iterative deepening search $l = 3$



Properties of iterative deepening search I

- It takes the advantages of **BFS** and **DFS**. The **temporal** complexity is related to d because it is always going to find the less deep **solution** and with respect to the memory needed it can remove parts of the tree.
- **Complete**: Yes
- **Time**: $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$. Here it is d and not $d+1$ because this strategy **does not generate successors** with a depth greater than the limit d. So, the don't need to be stored and even generated.
- **Space** $O(bd)$
- **Optimal** Yes, if step cost $= 1$. Can be modified to explore uniform-cost tree

Properties of iterative deepening search II

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

$$\begin{aligned} N(IDS) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(BFS) &= 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100 \end{aligned}$$

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

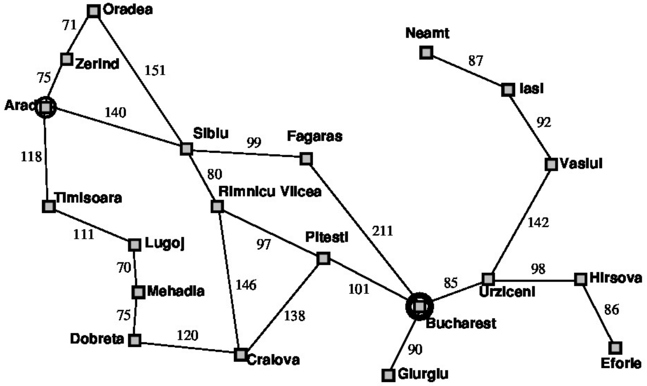
Uniform-cost search I

- Expand **least-cost** unexpanded node
- *Implementation*: *fringe* = queue **ordered** by path cost, **lowest** first
- **Equivalent** to breadth-first if step costs all **equal**

Uniform-cost search II

- **Complete Yes**, if step cost $\geq \epsilon$
- **Time** of **nodes** with $g \leq$ cost of **optimal solution**, $O(b^{C^*/\epsilon})$ where C^* is the cost of the optimal solution.
- **Time** $O(log(b^d)b^d) \rightarrow O(b^d)$
- **Space** of **nodes** with $g \leq$ cost of **optimal** solution, $O(b^{C^*/\epsilon})$
- **Space**: $O(b^d)$ worst case if like **breadth** first search (cost=1)
- **Optimal** Yes: nodes **expanded** in **increasing** order of $g(n)$

Example: Uniform-cost search



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}	$b^{C^*/\epsilon}$	b^m	b^l	b^d
Space	b^{d+1}	$b^{C^*/\epsilon}$	bm	bl	bd
Optimal?	Yes*	Yes	No	No	Yes*

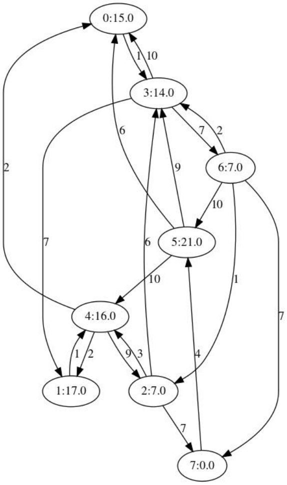
Algorithmic unification I

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure

fringe ← an empty list ordered by f (value) in ascendent order
closed (visitados) ← an empty set
fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
loop do
if fringe is empty then return failure
node ← 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 ← INSERTALL(EXPAND(node, problem), fringe)
end
```

Algorithmic unification II

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        ID[s] ← LastID+1;
        PARENT-NODE[s] ← node;
        ACTION[s] ← action;
        STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action, result)
        DEPTH[s] ← DEPTH[node] + 1
        VALUE[s] ← DEPTH[s](anchura)
        VALUE[s] ← -DEPTH[s](profundidad)
        VALUE[s] ← PATH-COST[s](costo uniforme)
        add s to successors
    end for
return successors
```

Resolution of Exercises I



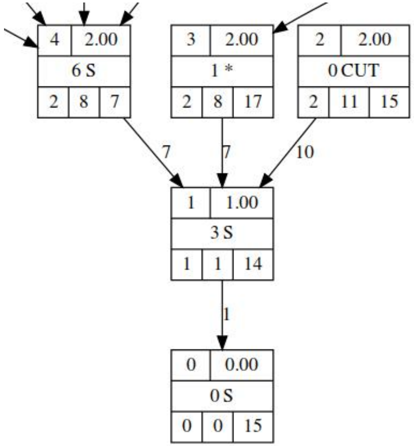
Resolution of Exercises II

- **Initial state**: 0
- **Final state**: 7
- **Cost of the action**: Label of the arcs.
- **Goal of the exercise**: Create the tree search for a concrete strategy.

Resolution of Exercises III

A node has the next **attributes**. With respect to the **theoretical** definition of node the action is removed in order to simplify this graphical **representation**.
First row: Id; Value
Second row: State
Third row: Depth; Cost; Heuristic.
Labels that during the execution are added:
CUT: Node extracted from the frontier and that was not expanded because it belongs to Closed.
*****: Expanded node.
S: Node that is part of the solution path. This is the last label added once the algorithm has finished and this label overwrites the *.

Resolution of Exercises IV



Pending issues I

We now try to give answer to some pending questions:

- Why do we **speak** of **search algorithms** and not of **search algorithm** if we are **only** going to **use** the graph search algorithm?
  In the **literature** each **strategy** has its own **algorithm**. We in this course **unify them all in one** making small **modifications** being the main one the management of the frontier.

Pending issues II

- If I have to **solve** a search problem, Do I have to **specify** all the fields of every node in the tree? In this case, which **search** node format or which fields should **appear**?
  Indeed, all fields in the tree node must be specified. Apart from the theoretical ones at the implementation level, a unique **ID** is added for each node, as well as the **value or f** which is the **ordering criterion at the frontier**.

Pending issues III

- Why does the **algorithm** specify that the first element of the frontier is taken (and deleted) but does not specify where **new elements** should be **inserted**?
  It is a **general algorithm** that works for all strategies. It has already been studied that depending on the **strategy** the new nodes are added at a **specific** position in the frontier.

Pending issues IV

- Why does the **algorithm** test whether a **node** contains a state that satisfies the **objective function** only when it is **extracted** from the **frontier** and not when it is inserted? Wouldn't this be much more **efficient**?
  For DFS and BFS it is valid to stop the algorithm when a successor that satisfies the goal function is obtained. But for UCS it is not so since we can add a node that satisfies the objective function but it represents a path of higher cost that another one that appears later. As our objective is by means of a single algorithm to implement all the strategies and although it is a little more inefficient for BFS and DFS we only stop the algorithm if a node extracted from the frontier satisfies the goal function

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

Unit 3. Informed Search Algorithms

Intelligent Systems

Escuela Superior de Informatica de Ciudad Real

Universidad de Castilla-La Mancha

Contents

Uninformed Search?

Previous **strategies** are called **uniformed** because:

- They don't consider any **information** about the states and the **goals** in order to decide which **path** to expand first on the **frontier**.
- They are **general** and do not consider specific **characteristics** of the problem.
- Blind search **algorithms** do not take into account the goal until they are in the **goal** node.

Informed Strategies

In some **problems** there exist **extra knowledge** that can be used to **guide** the **search**.

Contents

Review: Tree search

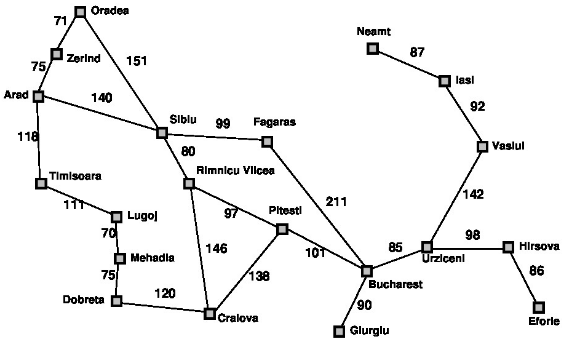
```
function TREE-SEARCH(problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST[problem] applied to STATE(node) succeeds return
node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

- 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



Straight–line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

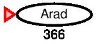
Greedy search

- **Evaluation** function $h(n)$ ($h$euristic): **estimate** of cost from n to the closest **goal**
- E.g., $h_{\mathrm{SLD}}(n) =$ **straight-line** distance from n to **Bucharest**
- **Greedy** search expands the node that *appears* to be **closest** to goal

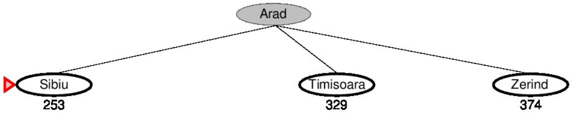
Heuristics

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

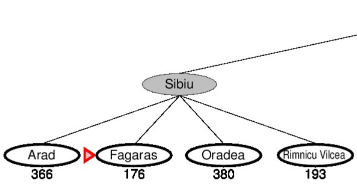
Greedy search example



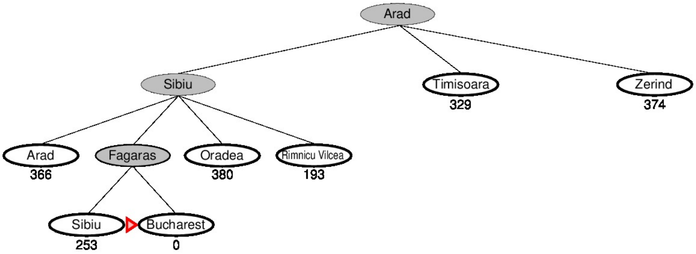
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

- **Complete** No and it can get **stuck** in loops, e.g.,
  Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$
  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 I

- *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 II

- $h(n)$ needs to be **efficient** to **compute**.
- A* search uses an *admissible* **heuristic**:
  $h(n) \leq h^*(n)$ where $h^*(n)$ is the *true* cost from n.
  (Also require $h(n) \geq 0$, so $h(G) = 0$ for any goal G.)

Admissible

An admissible **heuristic** is a **non negative** heuristic function that is an **underestimate** of the actual cost of a path to a goal.

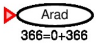
A* search III

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

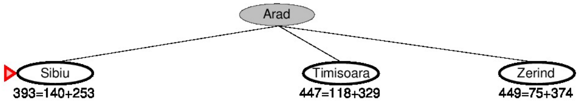
Underestimate

$h(n)$ is **underestimate** if there is no **path** from n to a goal with **cost** less than h(n).

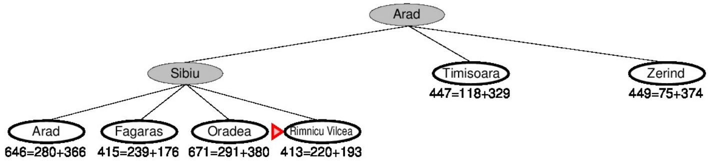
A* search example



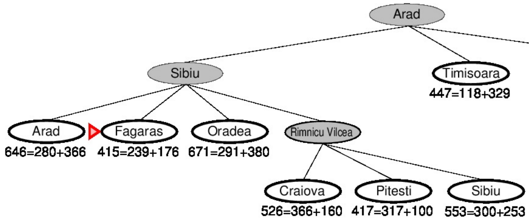
A* search example



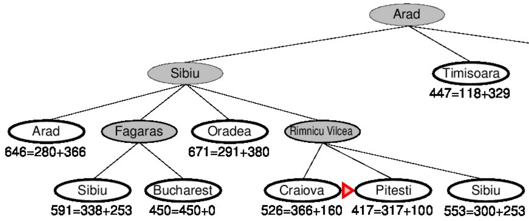
A* search example



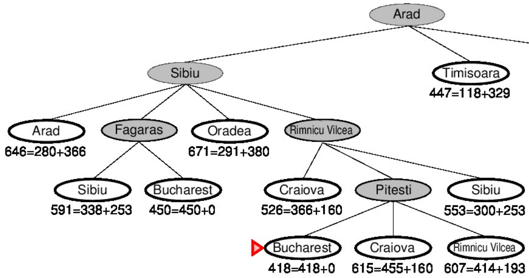
A* search example



A* search example



A* search example

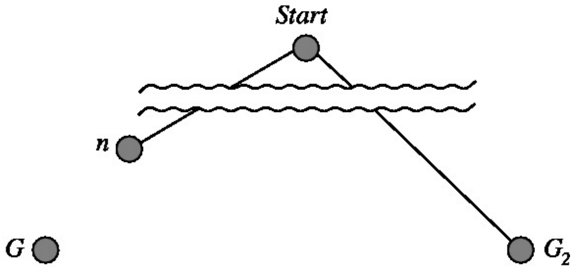


Admissibility of A*

- **A*** is **complete** (finds a solution, if one exists) and **optimal** (finds the optimal path to a goal) if:
    1. the **branching** factor is **finite**
    2. arc **costs** are > 0
    3. h(n) is **admissible**: an **underestimate** of the length of the **shortest** path from n to a goal node.

Optimality of A* (standard proof) I

- 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.



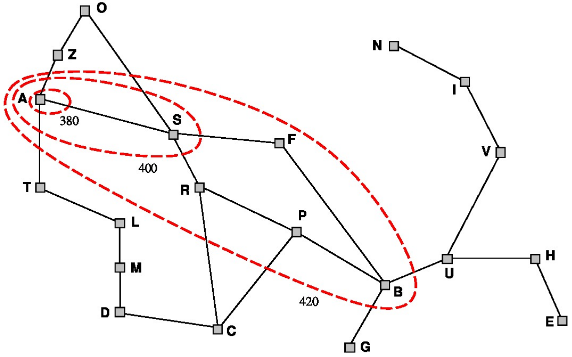
Optimality of A* (standard proof) II

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

- 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 $f_i < f_{i+1}$



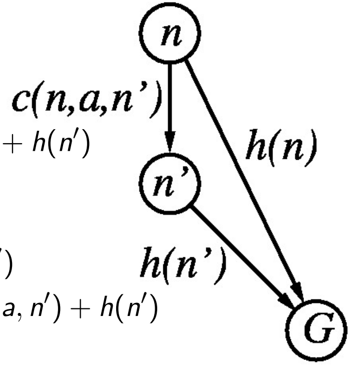
Properties of A*

- **Complete** 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 I

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**.

Proof of lemma: Consistency II

A consistent (or monotone) heuristic function is a function that estimates the distance of a given state to a goal state, and that is always at most equal to the estimated distance from any neighbouring vertex plus the step cost of reaching that neighbor.

Formally, for every node N and every successor P of N generated by any action a, the estimated cost of reaching the goal from N is no greater than the step cost of getting to P plus the estimated cost of reaching the goal from P

Lemma

While all consistent heuristics are admissible, not all admissible heuristics are consistent.

Proof of lemma: Consistency III

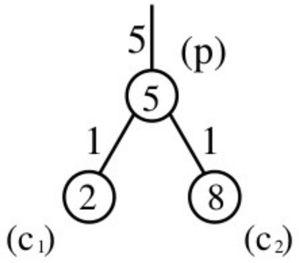
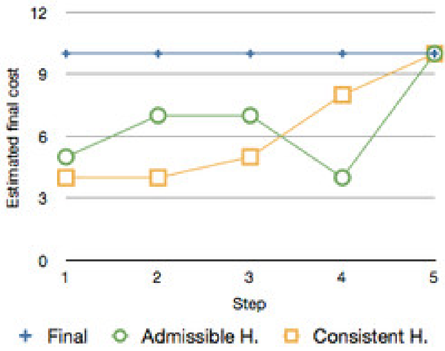


Figure 9: Inconsistent heuristic

Proof of lemma: Consistency IV

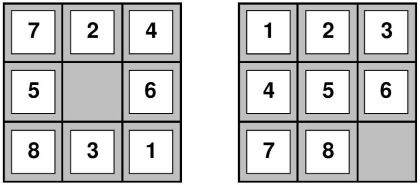


Contents

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**

$h_1(S) = 6$
$h_2(S) = 4+0+3+3+1+0+2+1 = 14$

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:

$$\begin{aligned} d = 14 \quad & \text{IDS} = 3{,}473{,}941 \text{ nodes} \\ & \text{A}^*(h_1) = 539 \text{ nodes} \\ & \text{A}^*(h_2) = 113 \text{ nodes} \\ d = 24 \quad & \text{IDS} \approx 54{,}000{,}000{,}000 \text{ nodes} \\ & \text{A}^*(h_1) = 39{,}135 \text{ nodes} \\ & \text{A}^*(h_2) = 1{,}641 \text{ nodes} \end{aligned}$$

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

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

Contents

Iterative improvement algorithms I

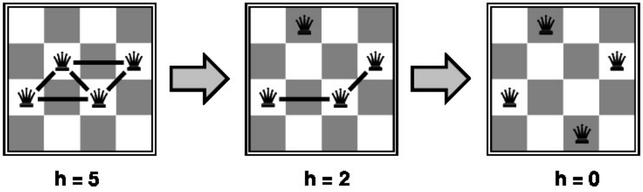
- Local search **algorithms** operate using a **single current node** (not multiple paths) and generally move only to **neighbours** of that node.
- They have two major **advantages**:
  1. they use very little memory (usually a constant amount).
  2. they can often find **reasonable** solutions in large or **infinite** (continuous) state spaces for which **classical** search algorithms are not **suitable**.

Iterative improvement algorithms II

- They are useful in many **optimization** problems in which the **aim** is to find the best **state** according to an **objective function**.
- In such **cases**, can use *iterative improvement* **algorithms**; keep a single "**current**" state, try to improve it:
  - **Heuristics** or **Cost**: **Minimization** problems.
  - **Objective** function: **Maximization** problems.

Example: *n*-queens

- Put n **queens** on an $n \times n$ **board** with no two **queens** on the same row, **column**, or diagonal
- Move a **queen** to reduce number of **conflicts**. What is h? What is the successor function?



- **Almost** always solves *n*-queens problems almost **instantaneously** for very **large** n, e.g., $n = 1 million$

Hill-climbing (or gradient ascent/descent) I

```
function HILL-CLIMBING( problem) returns a state that is a local maximum
    inputs: problem, a problem
    local variables: current, a node
                     neighbor, a node

    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] $\leq$ VALUE[current] then return STATE[current]
        current ← neighbor
end
```

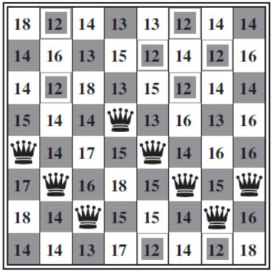
Descent

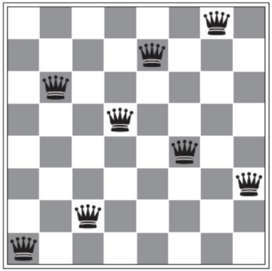
What options do we have if the VALUE function must be minimized?

Hill-climbing (or gradient ascent/descent) II

Question

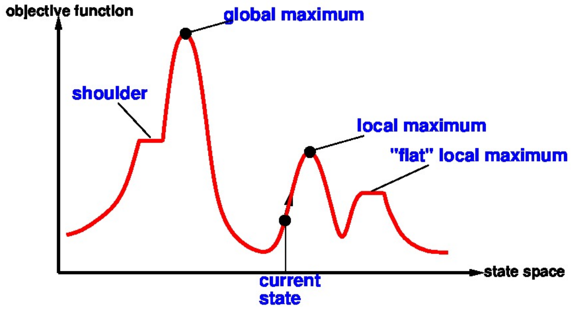
- Which is the **value** for h in the two **situations** described below?
- For the right picture look for a **neighbour** that improves de **evaluation** function?





Hill-climbing (or gradient ascent/descent) III

- **Useful** to **consider** *state space landscape*



Hill-climbing (or gradient ascent/descent) IV

Random-restart hill climbing **overcomes** local **maxima**—trivially **complete**. To put a limit on a number of consecutive sideways move allowed.

- In 8-queens, 14% the problem is **solved**. The rest it gets **stuck**. It needs only 4 steps on average to find the optimal **solution**.
- If we limit the number possible of **movements** to 100, then the **percentage** of problems instances solved **raises** to 94%, but the algorithm **averages** needs 21 steps for each **successful** instance.

Another **possibility** is *random sideways moves* **escape** from **shoulders** loop on flat maxima.

Simulated annealing I

- **Annealing** is a process in **metallurgy** where metals are slowly cooled to make them reach a state of low energy where they are very **strong**.
- Simulated **annealing** is an analogous method for **optimization**.
- It is **typically** described in terms of **thermodynamics**.
- The random movement **corresponds** to high temperature; at low temperature, there is little **randomness**.

Simulated annealing II

- Simulated **annealing** is a process where the temperature is **reduced** slowly, starting from a random **search** at high temperature eventually **becoming** pure greedy descent as it approaches zero **temperature**.
- The **randomness** should tend to jump out of local **minima** and find regions that have a low **heuristic** value; greedy descent will **lead** to local minima.
- At high **temperatures**, worsening steps are more likely than at lower **temperatures**
- Idea: **escape** local **maxima** by **allowing** some "**bad**" moves *but gradually decrease their size and frequency*.

Simulated annealing III

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     next, a node
                     $T$, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for $t \leftarrow 1$ to $\infty$ do
        $T \leftarrow$ schedule[$t$]
        if $T = 0$ then return current
        next ← a randomly selected successor of current
        $\Delta E \leftarrow$ VALUE[next] − VALUE[current]
        if $\Delta E > 0$ then current ← next
        else current ← next only with probability $e^{\Delta E / T}$
```

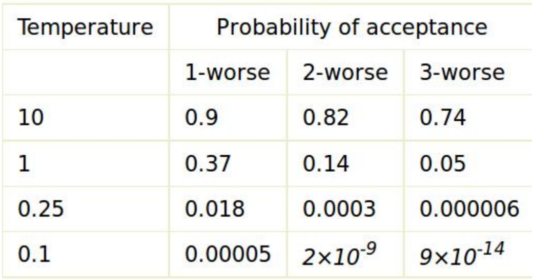
Properties of simulated annealing I

$$p(x) = e^{\frac{h(A)-h(A')}{T}}$$

- Thus, if **h(A')** is close to **h(A)**, the **assignment** is more likely to be **accepted**.
- If the **temperature** is high, the **exponent** will be close to zero, and so the **probability** will be close to 1.
- As the temperature **approaches** zero, the **exponent** approaches $-\infty$, and the probability approaches **zero**.

Properties of simulated annealing II

Source: `http://artint.info/html/ArtInt_89.html#sim-ann-prob-fig`



Temperature	Probability of acceptance		
	1-worse	2-worse	3-worse
10	0.9	0.82	0.74
1	0.37	0.14	0.05
0.25	0.018	0.0003	0.000006
0.1	0.00005	2×10^{-9}	9×10^{-14}

Figure: Probability of simulated annealing accepting worsening steps

Properties of simulated annealing III

- Simulated annealing requires an **annealing schedule**, which specifies how the temperature is **reduced** as the search **progresses**.
- Geometric **cooling** is one of the most widely used **schedules**.
- An example of a **geometric** cooling schedule is to start with a **temperature** of 10 and **multiply** by 0.97 after each step; this will have a **temperature** of 0.48 after 100 steps.
- Finding a good **annealing** schedule is an **art**.

Local beam search

- *Idea*: keep k states **instead** of 1; choose top k of all their **successors**
- Not the **same** as k searches run in **parallel**! Searches that **find** good **states** recruit other searches to join them
- *Problem*: quite **often**, all k states end up on **same local hill**
- *Idea*: choose k successors **randomly**, biased towards good **ones**
- Observe the close **analogy** to natural **selection**!

Genetic algorithms I

- Similar to **stochastic** local **beam** search but generate **successors** from *pairs* of **states**.
- **States** are like **individuals** of the **population**.
- Original **population** is randomly **generated**.
- Each **individual** is represented as a **string** (**chromosome**).

Genetic algorithms II

- Each individual is **rated** by an objective function (**fitness function**).
- Probability of **being** chosen for **reproduction** directly proportional to **fitness**.
- Two **parents** produce **offspring** by **crossover** then with some small **probability** occurs **mutation**: bits of the string are changed.

Crossover I

- Given two individuals:

$$X_1 = a_1, X_2 = a_2, \ldots, X_m = a_m$$

$$X_1 = b_1, X_2 = b_2, \ldots, X_m = b_m$$

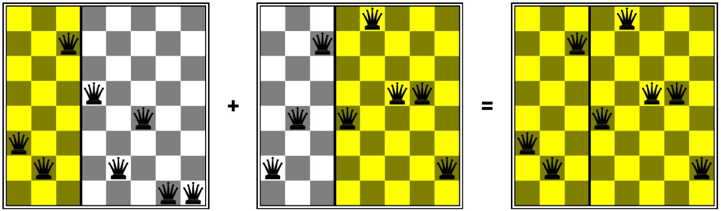
- Select *i* at random.
- Form two offspring:

$$X_1 = a_1, \ldots, X_i = a_i, X_{i+1} = b_{i+1}, \ldots, X_m = b_m$$

$$X_1 = b_1, \ldots, X_i = b_i, X_{i+1} = a_{i+1}, \ldots, X_m = a_m$$

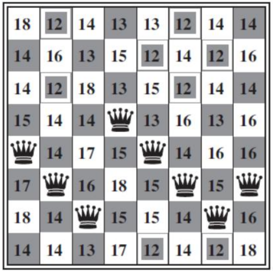
Crossover II

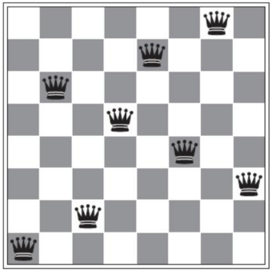
- **Crossover** helps *iff substrings are meaningful components*
- The **effectiveness** depends on the **ordering** of the variables.
- **Many** variations are **possible**.
- GAs require **states** encoded as **strings** (*GPs* use *programs*)



n-queens problem I

8-queens problem **states**: assume each queen has its own column, represent a state by listing a row **where** the queen is in each column (digits 1 to 8):





n-queens problem II

- **Fitness function**: instead of -h as before, use the number of **nonattacking** pairs of queens.
- There are 28 **pairs** of different queens, smaller column first, all together, so solutions have fitness 28. (Basically, fitness function is 28-h.)
- For example, **fitness** of the state above is 27 (queens in **columns** 4 and 7 attack each other)

Algorithm Operation I

```
function GENETIC-ALGORITHM(population,FITNESS-FN) returns an individual
    inputs: population, a set of individuals
            FITNESS-FN, a function that measures the fitness of an individual
    repeat
        new-population ← empty set
        for i=1 to SIZE(population)  do
            x ← RANDOM-SELECTION(population,FITNESS-FN)
            y ← RANDOM-SELECTION(population,FITNESS-FN)
            child ← REPRODUCE(x, y)
            if (small random probability) then child ← MUTATE(child)
            add child to new-population
    until some individual is fit enough or enough time has elapsed
    return the best individual in population, according to FITNESS-FN
```

Algorithm Operation II

```
function REPRODUCE( $x, y$) returns an individual
    inputs: $x, y$, parent individuals
    $n \leftarrow$ LENGTH($x$); $c \leftarrow$ random number from 1 to n
    return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c+1, n))
```

Algorithm Operation III

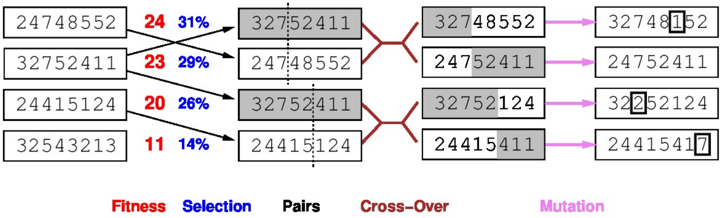


Figure: Example 8-queens

Algorithm Operation IV

- Like **stochastic** beam search, but pairs of individuals are **combined** to create the offspring:
- For each **generation**:
  - **Randomly** choose pairs of individuals where the **fittest** individuals are more likely to be chosen.
  - For each **pair**, perform a cross-over: form two **offspring** each taking different parts of their parents:
  - Mutate **some** values.
- Stop when a **solution** is found.

Algorithm Operation V

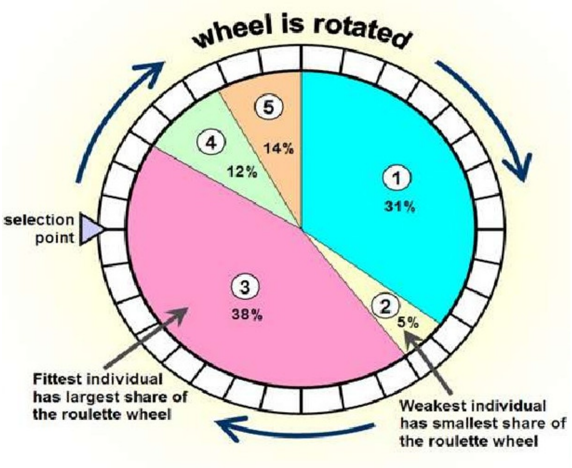

wheel is rotated
5
14%
4
12%
1
31%
selection point
3
38%
2
5%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

Figure: Roulette Wheel Selection

Unit 4. Constraint Satisfaction Problems

Intelligent Systems

Escuela Superior de Informatica de Ciudad Real

Universidad de Castilla-La Mancha

Learning Objectives

At the end of this **unit** you should be able to:

- **recognize** and **represent** constraint satisfaction **problems**
- show how **constraint** satisfaction **problems** can be solved with **search**
- **implement** and trace arc-consistency of a **constraint** graph
- show how domain **splitting** can solve constraint **problems**

Contents

Definition I

- **Standard search problem**: *state* is a "black box"—any old data structure that supports goal test, eval, successor
- **CSP**: *state* is **defined** by **variables** X_i with *values* from **domain** D_i
- **goal test** is a set of **constraints** specifying allowable **combinations** of values for subsets of variables

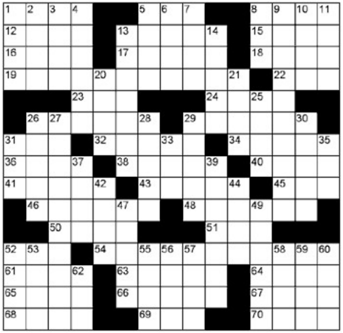
Definition II

Then a CSP is **characterized** by

- A set of **variables** $V_1, V_2, \ldots, V_n$.
- Each **variable** V_i has an associated **domain** D_{V_i} of possible **values**.
- There are **hard constraints** on various subsets of the **variables** which specify legal combinations of **values** for these variables.
- A solution to the CSP is an **assignment** of a value to each **variable** that satisfies all the **constraints**.

Possible Worlds I

- A **possible world** is a complete **assignment** of values to each **variable**.
- Crossword Puzzle:



Possible Worlds II

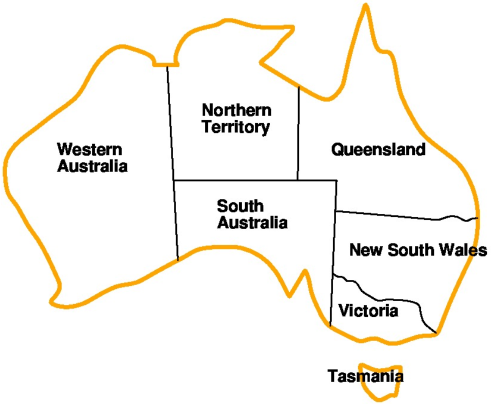
Crossword Puzzle:

- Variables are **words** that have to be **filled in**.
- **Domains** are **English** words of correct **length**.
- Possible **worlds** are all the ways of **assigning** words.
- With a number of **English** words of **15,000** for this length and a **number** of **words** of 70. How many possible worlds?
- 70^{15000}; $70 * 15000$; 15000^{70}

Models

- A **model** of a CSP is an **assignment** of values to all its **variables** that satisfies all of its **constraints**.
- Determine **whether** or not a **model** exists.
- Find a **model** or find all the **models**.
- Find the best model **given** a quality **measure** (optimization **problem**).

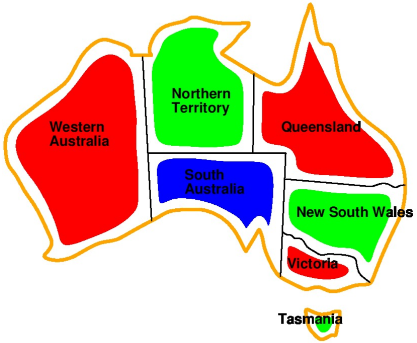
Example: Map-Coloring I



Example: Map-Coloring II

- $Variables\ WA,\ NT,\ Q,\ NSW,\ V,\ SA,\ T$
- $Domains\ D_i = \{red, green, blue\}$
- $Constraints$: adjacent **regions** must have **different colors**
  1. $WA \neq NT$ (if the language allows this)
  2. $(WA, NT) \in$
     $\{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Example: Map-Coloring III



A **model** is:, e.g.,
$\{WA = red, NT = green, Q = red, NSW = green, V = red,$
$SA = blue, T = green\}$

Varieties of constraints I

- **Unary** constraints involve a **single variable**, e.g., $SA \neq green$
- **Binary** constraints involve **pairs** of **variables**, e.g., $SA \neq WA$; Crossword: two **words** have the **same** point where they **intersect**.
- **Higher-order** constraints **involve** 3 or more variables, e.g., **AllDiff**.

Varieties of constraints II

- The scope of a **constraint** is the set of variables that are **involved** in that **constraint**.
- **Preferences** (soft constraints), e.g., *red* is **better** than *green* often **representable** by a **cost** for each variable **assignment** $\rightarrow$ constrained optimization problems

Real-world CSPs

- **Assignment** problems: e.g., who **teaches** what class
- **Timetabling** problems: e.g., which class is **offered** when and where?
- **Hardware** configuration
- Transportation **scheduling**
- **Factory** scheduling
- **Floorplanning**

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

Contents

Introduction I

- Even the **simplest** problem of **determining** whether or not a model exists in a general CSP with **finite domains** is **NP-hard**.
- There is no known **algorithm** with worst case **polynomial** run-time.
- **Anyway**, if is **possible** to find efficient (polynomial) **consistency** algorithms that reduce the size of search **space**.

Introduction II

- Identify **special** cases for which **algorithms** are **efficient**.
- Work on **approximation** algorithms that can find good **solutions** quickly, even though they may offer no **theoretical** guarantees.
- Find **algorithms** that are fast on **typical** (not worst case) cases.

Generate-and-Test Algorithm I

Example: **scheduling** activities

- **Variables:** A, B, C, D, E that represent the **starting times** of various activities.
- **Domains:** $D_A = \{1, 2, 3, 4\}$, $D_B = \{1, 2, 3, 4\}$, $D_C = \{1, 2, 3, 4\}$, $D_D = \{1, 2, 3, 4\}$, $D_E = \{1, 2, 3, 4\}$
- **Constraints:**
  $(B \neq 3) \wedge (C \neq 2) \wedge (A \neq B) \wedge (B \neq C) \wedge (C < D)$
  $\wedge (A = D) \wedge (E < A) \wedge (E < B) \wedge (E < C) \wedge (E < D) \wedge (B \neq D)$.

Generate-and-Test Algorithm II

- Generate the **assignment** space $D = D_{V_1} \times D_{V_2} \times \ldots \times D_{V_n}$.
- **Test** each assignment with the **constraints**.
- **Example:**

$$\begin{aligned} D &= D_A \times D_B \times D_C \times D_D \times D_E \\ &= \{1,2,3,4\} \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &\quad \times \{1,2,3,4\} \times \{1,2,3,4\} \\ &= \{(1,1,1,1,1),(1,1,1,1,2),...,(4,4,4,4,4)\}. \end{aligned}$$

- How many **assignments** need to be **tested** for k variables each with **domain** size d and c constraints?

CSP as a search problem I

- **State**: is defined by an **assignment** of variables Xi with **values** from **domain** Di (partial assignment of values to variables).
- **Initial state**: the empty **assignment** {}, in which all variables are **unassigned**.
- **Successor function**: a value can be **assigned** to any unassigned **variable**, provided that it does not **conflict** with previously **assigned** variables.

CSP as a search problem II

- **Goal test**: the current **assignment** is complete. i.e. all **variables** are assigned values complying to the set of **constraints**
- **Path cost**: a **constant** cost (e.g., 1) for **every** step
- Every **solution** must be a complete assignment and **therefore** appears at **depth n** if there are n variables.

CSP as a search problem III

- **Furthermore**, the search tree **extends** only to **depth n**.
- It is also the **case** that the **path** by which a **solution** is reached is **irrelevant**.
- Which search **algorithm** would be the most **appropriate** for this formulation of **CSP**?

Backtracking search

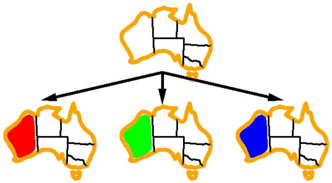
- Variable **assignments** are *commutative*, i.e., $[WA = red$ then $NT = green]$ **same** as $[NT = green$ then $WA = red]$
- Only **need** to consider **assignments** to a single **variable** at each node
- Depth-first **search** for CSPs with **single-variable** assignments is called **backtracking** search
- Backtracking **search** is the **basic** uninformed **algorithm** for CSPs
- Can **solve** n-queens for $n \approx 25$

Backtracking search

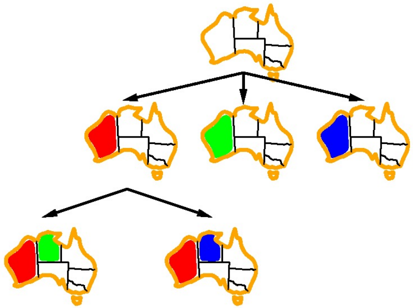
```
function BACKTRACKING-SEARCH(csp) returns solution/failure return
RECURSIVE-BACKTRACKING({ }, csp) function

RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
  var ← 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 ← RECURSIVE-BACKTRACKING(assignment, csp)
       if result ≠ failure then return result
     remove {var = value} from assignment
   return failure
```

Backtracking example

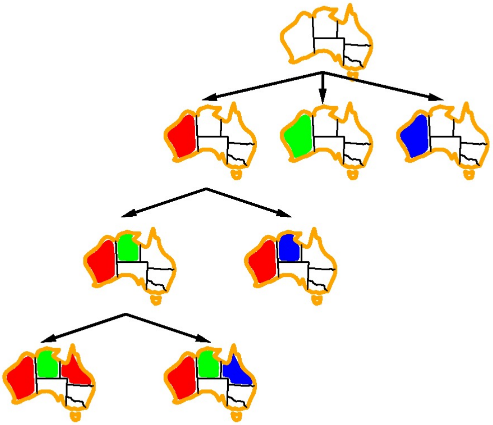
Backtracking example



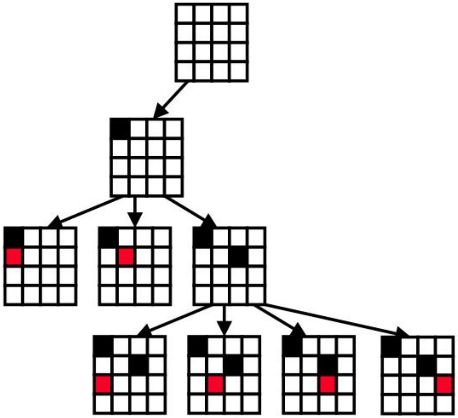
Backtracking example



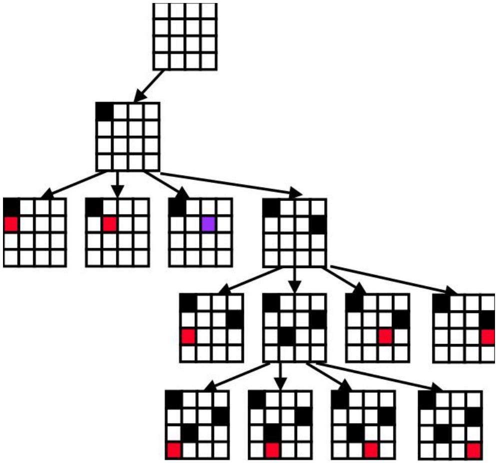
Backtracking example



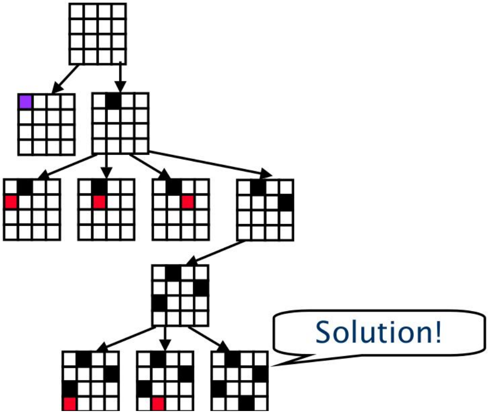
Example 4-queens



Example 4-queens

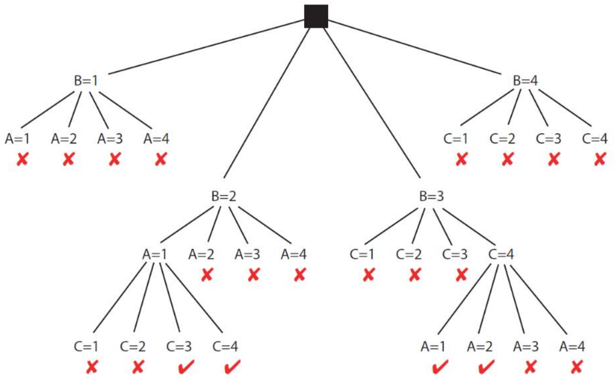


Example 4-queens



Backtracking example

- 3 **variables**: A, B, C with **domains** = {1,2,3,4}.
- **Constraints**: $A < B$, $B < C$.



Improving backtracking efficiency I

General-purpose methods can give huge **gains** in **speed**:

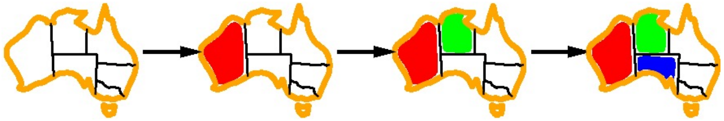
1. Which **variable** should be assigned **next**?
2. In what order **should** its values be **tried**? (Example: $V_n \neq V_{n-1}$ and $V_n = V_{n-1}$)
3. Can we **detect** inevitable **failure** early?
4. Can we take **advantage** of problem **structure**?

Improving backtracking efficiency II

- It can be used **one** or more **heuristics**. e.g, **variable** in the largest **number** of **constraints**: "**If you are going to fail on this branch, fail early**!"
- Can also be **smart** about which **values** to consider **first**
- This is a **different** use of the word **heuristic**:
  - Still **true** in this context: can be **computed** cheaply during the search and **provides** guidance to the search **algorithm**
  - But **not true** any more in this **context**: estimate of the **distance** to the goal

Minimum remaining values

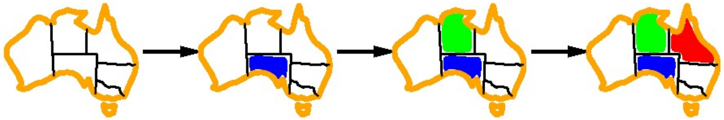
Minimum remaining **values** (MRV): choose the **variable** with the **fewest** legal **values**.



Degree heuristic

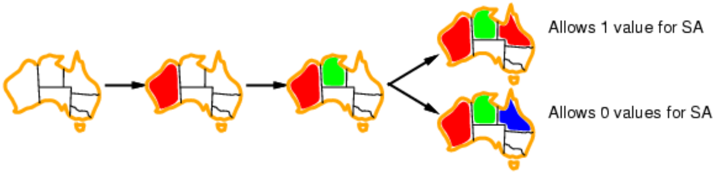
Tie-breaker among MRV variables

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**

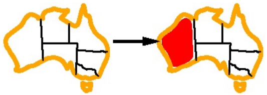
Terminate search when any **variable** has **no legal values**

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**

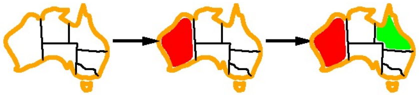


WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦

Forward checking

Idea: Keep track of **remaining** legal **values** for unassigned **variables**

Terminate search when any **variable** has **no legal values**

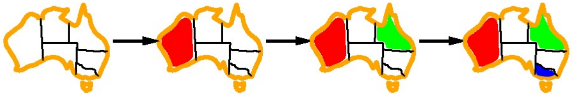


WA	NT	Q	NSW	V	SA	T
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦

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
🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦
🟥	🟩🟦	🟥🟩🟦	🟥🟩🟦	🟥🟩🟦	🟩🟦	🟥🟩🟦
🟥	🟦	🟩	🟥🟦	🟥🟩🟦	🟦	🟥🟩🟦
🟥	🟦	🟩	🟥	🟦		🟥🟩🟦

Constraint propagation

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

WA	NT	Q	NSW	V	SA	T
🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦
🟥	🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟥 🟩 🟦	🟩 🟦	🟥 🟩 🟦
🟥	🟦	🟩	🟥 🟦	🟥 🟩 🟦	🟦	🟥 🟩 🟦

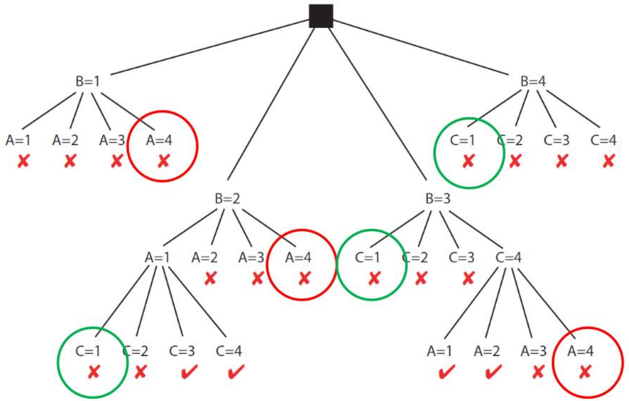
NT and *SA* **cannot** both be **blue**!: *Constraint propagation* repeatedly **enforces** constraints **locally**

Contents

Consistency Algorithms I

- **Idea**: **prune** the **domains** as much as **possible** before **selecting** values from them.
- A variable is **domain consistent** if no value of the **domain** of the node is **ruled** impossible by any of the **constraints**.
- **Easy** to **identify** in **unary** constraints but very **difficult** in k-ary **constraints**.
- **Example:** Is the **scheduling** example domain **consistent**? $D_B = \{1,2,3,4\}$ isn't domain consistent as $B = 3$ violates the constraint $B \neq 3$.

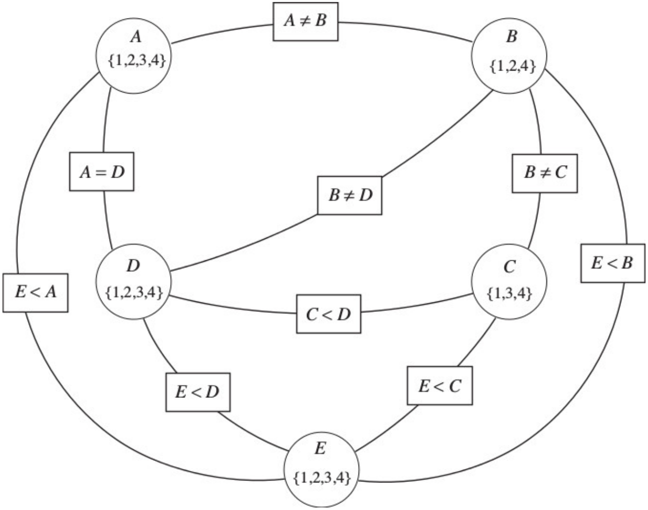
Consistency Algorithms II



Constraint Network I

- There is a **oval-shaped** node for each **variable**.
- There is a **rectangular** node for each **constraint**.
- There is a domain of **values** associated with each **variable** node.
- There is an arc from **variable** X to each **constraint** that involves X.

Constraint Network II



Constraint Network III

- An arc $< X, r(X, \overline{Y}) >$ is arc **consistent** if, for **each value** $x \in dom(X)$, there is some value $\overline{y} \in dom(\overline{Y})$ such that $r(x, \overline{y})$ is **satisfied**.
- A **network** is arc **consistent** if all its **arcs** are arc consistent.
- **What** if arc $< X, r(X, \overline{Y}) >$ is *not* arc **consistent**? All **values** of X in $dom(X)$ for which there is no **corresponding** value in $dom(\overline{Y})$ can be deleted from $dom(X)$ to **make** the arc $< X, r(X, \overline{Y}) >$ **consistent**.

Constraint Network IV

- The **arcs** can be **considered** in turn **making** each arc **consistent**.
- When an arc has **been** made arc **consistent**, does it ever need to be **checked** again?
- An arc $< X, r(X, \overline{Y}) >$ **needs** to be **revisited** if the domain of one of the Y's is **reduced**.
- Three possible **outcomes** when all arcs are **made** arc consistent: (Is there a **solution**?)
  - One **domain** is empty $\rightarrow$ no **solution**
  - Each domain has a **single** value $\rightarrow$ **unique** solution
  - Some **domains** have more than **one** value $\rightarrow$ there may or may not be a **solution** (SEARCH)

Arc consistency algorithm I

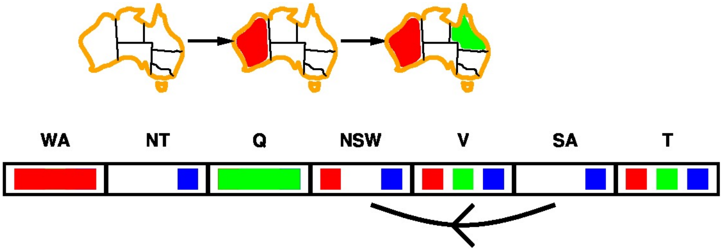
```
function AC-3( csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(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
```

Arc consistency algorithm II

```
function REMOVE-INCONSISTENT-VALUES( $X_i, X_j$ ) returns true iff
succeeds
removed ← 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 ← true
return removed
```

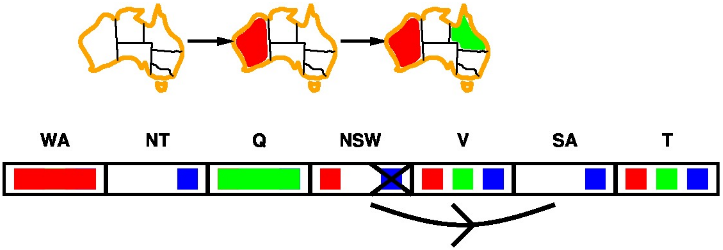
Arc consistency

- **Simplest** form of **propagation** makes each arc *consistent*
- $X \to Y$ is **consistent** iff for *every* value x of X there is *some* **allowed** y



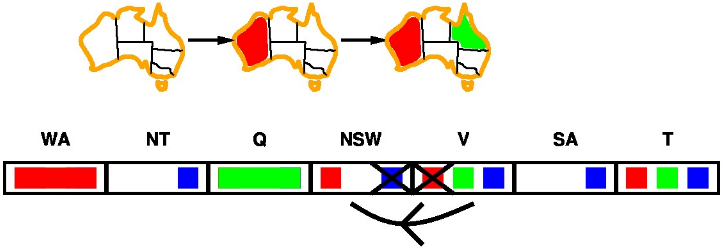
Arc consistency

- **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

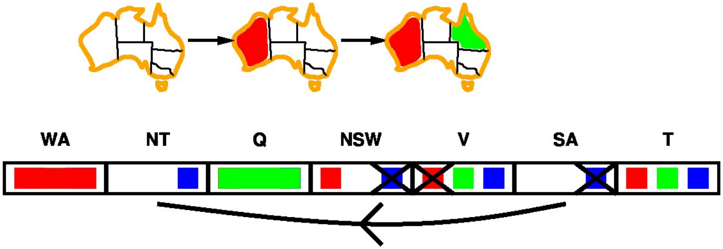
- **Simplest** form of **propagation** makes each arc *consistent*
- $X \to Y$ is **consistent** iff for *every* value x of X there is *some* **allowed** y



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

Arc consistency

- **Simplest** form of **propagation** makes each arc *consistent*
- $X \to Y$ 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**.

n-queens example I

- X_i **represents** the row where the **Queen** in column i is situated.
- $X_i = \{1,2,3,4]\}$ represents the **domain**.
- R(X_i, X_j) represents the constraint "**no attack**" between Queens in column i and j.

Is the assignment X_1=1 arc-consistent ?

n-queens example II

Step 0

- $X_1 = [1]$ $X_2 = [1,2,3,4]$ $X_3 = [1,2,3,4]$ $X_4 = [1,2,3,4]$
- QUEUE: R(X_1, X_2), R(X_2, X_1), R(X_1, X_3), R(X_3, X_1), R(X_1, X_4), R(X_4, X_1)

Step 1

- Constraint R(X_1, X_2) X_1=[1],X_2=[1,2,3,4] **OK**
- QUEUE: R(X_2, X_1), R(X_1, X_3), R(X_3, X_1), R(X_1, X_4), R(X_4, X_1)

Step 2

- Constraint R(X_2, X_1) X_2=[1,2,3,4] X_1=[1] **FAIL** X_2=[3,4] X_1=[1] **OK**
- QUEUE: R(X_1, X_3), R(X_3, X_1), R(X_1, X_4), R(X_4, X_1), R(X_3, X_2), R(X_4, X_2)

n-queens example III

Step 3

- Constraint R(X_1, X_3) X_1=[1] X_3=[1,2,3,4] **OK**
- QUEUE: R(X_3, X_1), R(X_1, X_4), R(X_4, X_1), R(X_3, X_2), R(X_4, X_2)

Step 4

- Constraint R(X_3, X_1) X_3=[1,2,3,4] X_1=[1] **FAIL** X_3=[2,4] X_1=[1] **OK**
- QUEUE: R(X_1, X_4), R(X_4, X_1), R(X_3, X_2), R(X_4, X_2), R(X_2, X_3), R(X_4, X_3),

Step 5

- Constraint R(X_1, X_4), X_1=[1] X_4 = [1,2,3,4] **OK**
- QUEUE: R(X_4, X_1), R(X_3, X_2), R(X_4, X_2), R(X_2, X_3), R(X_4, X_3),

n-queens example IV

Step 6

- Constraint R(X_4, X_1), X_4 = [1,2,3,4] X_1=[1] **FAIL** X_4 = [2,3] X_1=[1] **OK**
- QUEUE: R(X_3, X_2), R(X_4, X_2), R(X_2, X_3), R(X_4, X_3), R(X_2, X_4), R(X_3, X_4)

Step 7

- Constraint R(X_3, X_2), X_3=[2,4] X_2=[3,4] **FAIL** X_3 = [2] X_2=[3,4] **OK**
- QUEUE: R(X_4, X_2), R(X_2, X_3), R(X_4, X_3), R(X_2, X_4), R(X_3, X_4), R(X_1, X_3), R(X_4, X_3),

Step 8

- Constraint R(X_4, X_2), X_4 = [2,3] X_2=[3,4] **OK**

n-queens example V

- QUEUE: R(X_2, X_3), R(X_4, X_3), R(X_2, X_4), R(X_3, X_4), R(X_1, X_3), R(X_4, X_3),

Step 9

- Constraint R(X_2, X_3), X_2=[3,4] X_3 = [2] **FAIL** X_2=[4] X_3 = [2]
- QUEUE: R(X_4, X_3), R(X_2, X_4), R(X_3, X_4), R(X_1, X_3), R(X_4, X_3), R(X_1, X_2), R(X_4, X_2),

Step 10

- Constraint R(X_4, X_3), X_4=[2,3] X_3 = [2] **FAIL** X_4=[] X_3 = [2] **EMPTY DOMAIN** X_4, NOT POSSIBLE ARC-CONSISTENT
- QUEUE: R(X_2, X_4), R(X_3, X_4), R(X_1, X_3), R(X_4, X_3), R(X_1, X_2), R(X_4, X_2),

n-queens example VI

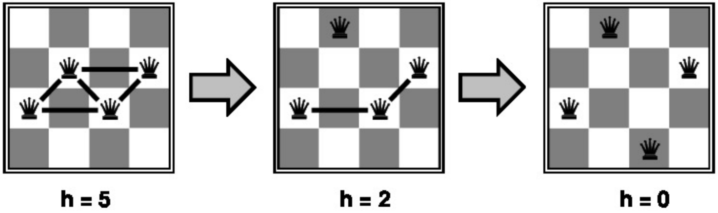
- In ten steps it is determined that the ASSIGNMENT X_1=[1] is erroneous
- The AC3 algorithm returns the domains X_2=[4],X_3=[2],X_4=[] (X_4 is empty)

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



Summary I

- 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**

Summary II

- Constraint **propagation** (e.g., arc consistency) does **additional** work to **constrain** values and detect **inconsistencies**
- The CSP **representation** allows analysis of problem **structure**
- Iterative min-**conflicts** is usually effective in **practice**

Unit 5. Adversarial Search

Intelligent Systems

Escuela Superior de Informatica de Ciudad Real

Universidad de Castilla-La Mancha

Contents

Games vs. search problems I

- "**Unpredictable**" opponent ⇒ solution is a **strategy** specifying a move for every possible **opponent** reply.
- Time **limits** ⇒ unlikely to **find** goal, must **approximate**.

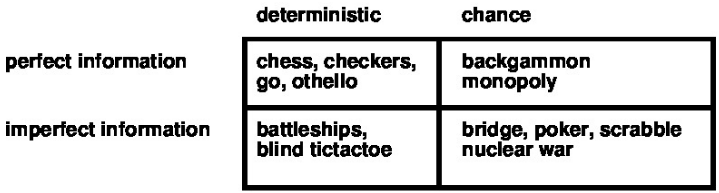
Plan of attack:

- 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)

Games vs. search problems II

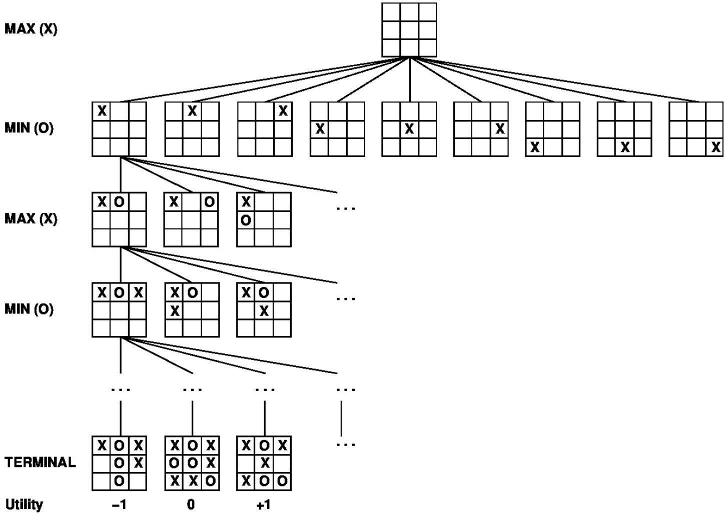
- **First chess** program (Turing, 1951)
- **Machine** learning to **improve** evaluation **accuracy** (Samuel, 1952–57)
- **Pruning** to allow deeper **search** (McCarthy, 1956)

Types of games



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

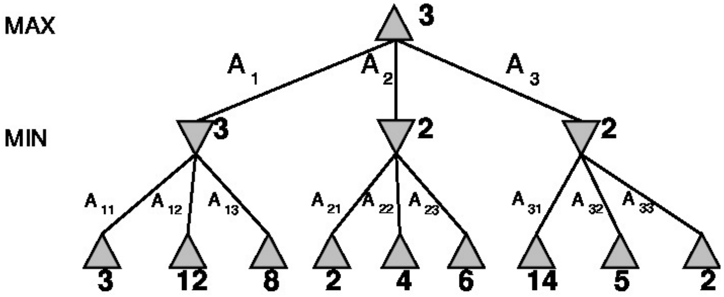
Game tree (2-player, deterministic, turns)



Contents

Minimax

- **Perfect** play for **deterministic** with perfect-**information** games.
- **Idea**: choose **move** to position with **highest** *minimax value*.
- **minimax value** = Best **achievable** payoff against best **play**.



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 $v \leftarrow$ MAX(v, MIN-VALUE(s))
   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 $v \leftarrow$ MIN(v, MAX-VALUE(s))
   return v
```

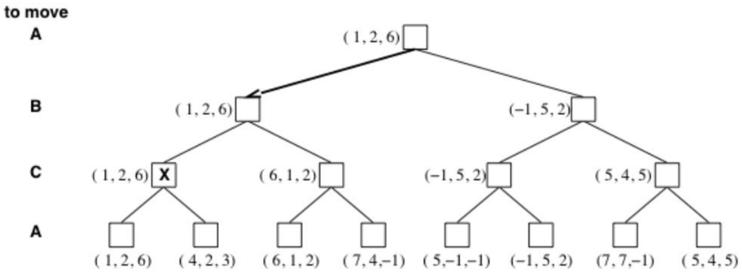
Properties of minimax

- **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 $\Rightarrow$ exact solution **completely** infeasible
- But do we need to **explore every path**?

Minimax for multiple players I

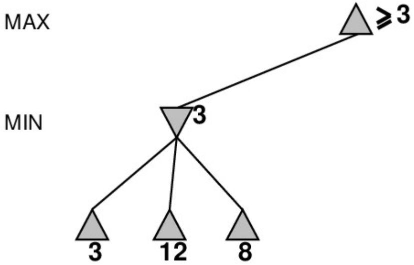
- All **players** are **Max**.
- The evaluation **function** is given by a **vector**.
- Each layer is **assigned** to one **player**.
- Turn: every n **layers** for n players.

Minimax for multiple players II

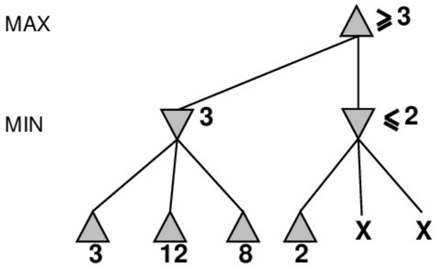


Contents

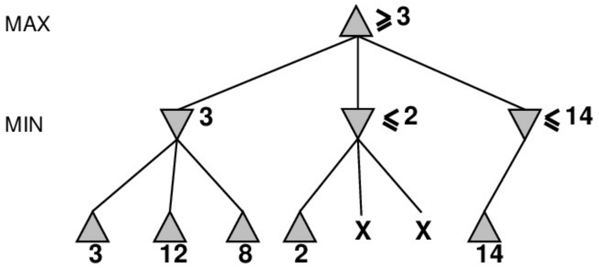
α–β pruning example I



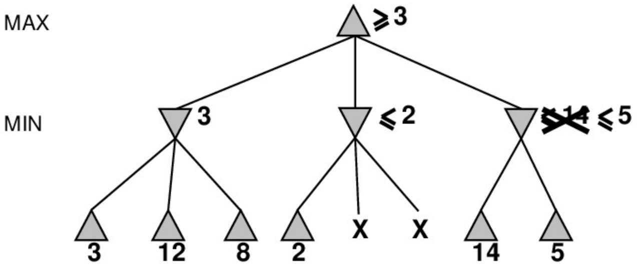
α–β pruning example II



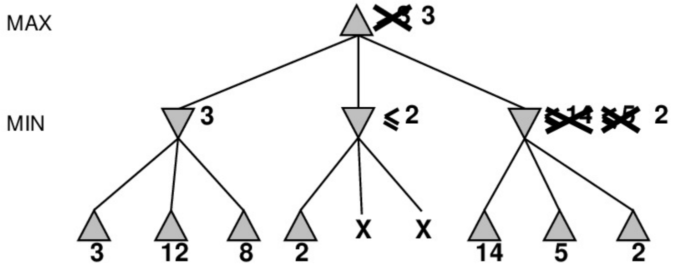
α–β pruning example III



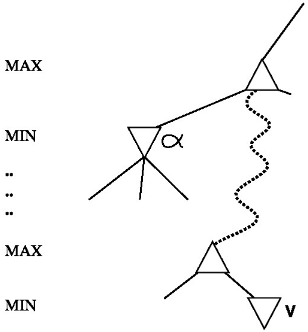
α–β pruning example IV



α–β pruning example V



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 I

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

```
**function** MIN-VALUE(*state*, $\alpha$, $\beta$) **returns** *an utility value*
same as MAX-VALUE but with roles of $\alpha$, $\beta$ reversed
```

The α–β algorithm II

```
**function** MAX-VALUE($state, \alpha, \beta$) **returns** *an utility value*
**inputs**: *state*, current state in game
$\alpha$ the value of the best alternative for MAX along the path to *state*
$\beta$ 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**
    $v \leftarrow$ MAX($v$, MIN-VALUE($s, \alpha, \beta$))
    **if** $v \geq \beta$ **then return** $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
**return** $v$
```

Properties of α–β

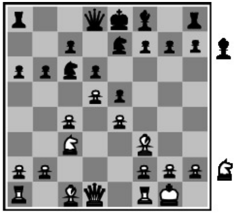
- **Pruning** does not affect **final** result
- Good move **ordering** improves effectiveness of **pruning**
- With "perfect ordering," time **complexity** $= O(b^{m/2}) \Rightarrow$ *doubles* solvable **depth**
- A simple **example** of the value of **reasoning** about which **computations** are relevant (a form of)
- **Unfortunately**, 35^{50} is still **impossible**!

Resource limits

Standard approach:

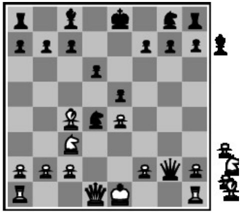
- **Use** CUTOFF-TEST instead of TERMINAL-TEST e.g., depth **limit** (perhaps add *quiescence search*)
- **Use** EVAL instead of UTILITY i.e., *evaluation function* that estimates **desirability** of position
- **Suppose** we have 100 **seconds**, explore 10^4 **nodes/second** $\Rightarrow$ 10^6 nodes per move $\approx 35^{8/2}$ $\Rightarrow$ α–β reaches depth 8 $\Rightarrow$ **pretty** good chess **program**

Evaluation functions



Black to move

White slightly better



White to move

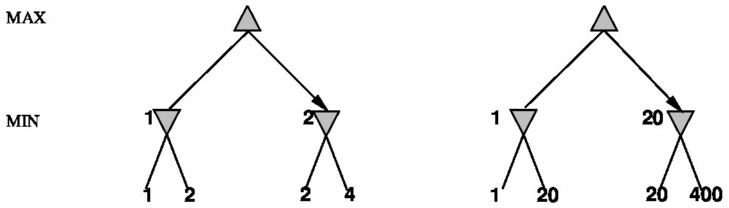
Black winning

For **chess**, typically **weighted sum** of *features*

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

e.g., $w_1 = 9$ with
$f_1(s) =$ (number of white queens) – (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 I

- **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

Deterministic games in practice II

.

- **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.

Unit 6. Reinforcement Learning

Intelligent Systems

Universidad de Castilla-La Mancha

Escuela Superior de Informatica de Ciudad Real

Introduction

Learning

Learning is the ability to improve one's behavior based on **experience**.

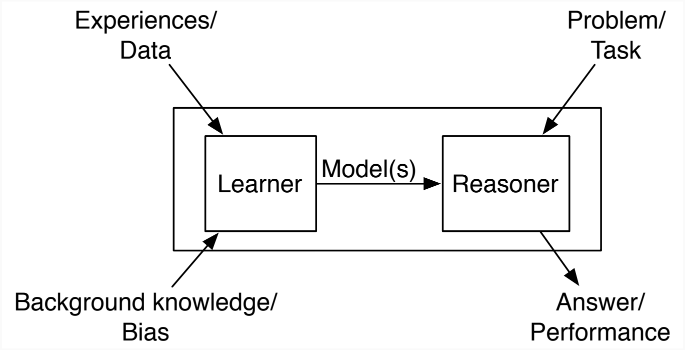
- The range of **behaviors** is **expanded**: the agent can do more.
- The accuracy on tasks is **improved**: the agent can do things **better**.
- The **speed** is improved: the agent can do things **faster**.

Components of a learning problem i

The following components are part of any learning problem:

- **Task**: The **behavior** or task that's being **improved**. For example: classification, acting in an environment, etc.
- **Data**: The **experiences** that are being used to improve **performance** in the task.
- **Measure of improvement How** can the improvement be **measured**? For example: increasing accuracy in **prediction**, new skills that were not present initially, **improved** speed, etc.

Components of a learning problem ii



Feedback

Learning tasks can be characterized by the **feedback** given to the learner.

- **Supervised learning** What has to be learned is **specified** for each example.
- **Unsupervised learning** No **classifications** are given; the learner has to discover **categories** and regularities in the data.
- **Reinforcement learning Feedback** occurs after a **sequence** of actions.

Example Supervised Learning i

Training Examples:

	Action	Author	Thread	Length	Where
e1	skips	known	new	long	home
e2	reads	unknown	new	short	work
e3	skips	unknown	old	long	work
e4	skips	known	old	long	home
e5	reads	known	new	short	home
e6	skips	known	old	long	work

New Examples:

e7	???	known	new	short	work
e8	???	unknown	new	short	work

We want to **classify** new examples on **feature** *Action* based on the **examples**' *Author*, *Thread*, *Length*, and *Where*.

Example Supervised Learning ii

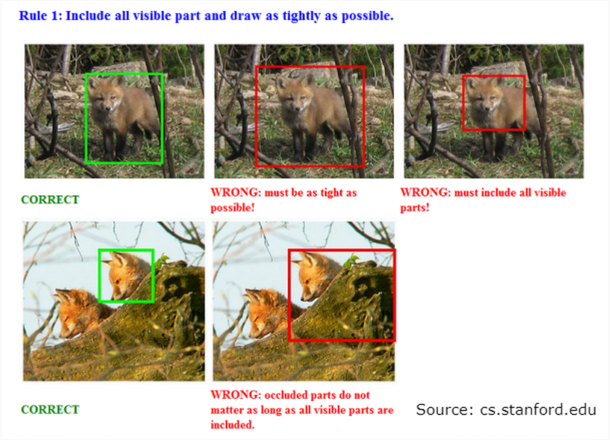

Rule 1: Include all visible part and draw as tightly as possible.
CORRECT
WRONG: must be as tight as possible!
WRONG: must include all visible parts!
CORRECT
WRONG: occluded parts do not matter as long as all visible parts are included.
Source: cs.stanford.edu

Figure 1: Labelling Pictures

Example Unupervised Learning

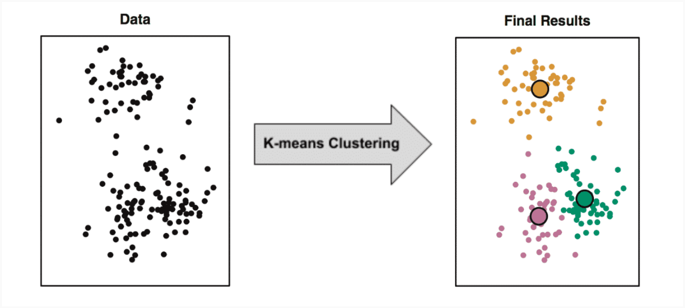


Figure 2: Clustering Algorithm

Data

- Data isn't perfect:
    - the features given are inadequate to predict the classification
    - there are examples with missing features
    - some of the features are assigned the wrong value
- overfitting occurs when distinctions appear in the training data, but not in the unseen examples.

Main concepts of Reinforcement learning

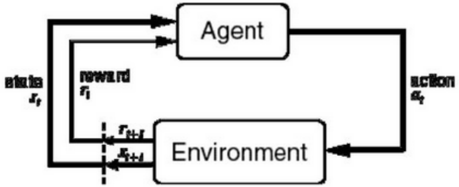
Examples Reinforcement Learning

- Goalkeeper: `https://www.youtube.com/watch?v=CIF2SBVY-J0&list=PL5nBAYUyJTrM48dViibyi68urttMlUv7e&index=9`
- Lane Tracker: `https://www.youtube.com/watch?v=jaTOSLd56hI&list=PL5nBAYUyJTrM48dViibyi68urttMlUv7e&index=18`
- Atari: `https://www.youtube.com/watch?v=V1eYniJ0Rnk&list=PL5nBAYUyJTrM48dViibyi68urttMlUv7e&index=32`
- Hide and Seek: `https://www.youtube.com/watch?v=kopoLzvh5jY`

RL Basics i

- **Reinforcement learning** is an area **concerned** with how an agent **ought** to take actions in an **environment** so as to **maximise** some notion of reward.
- “**A way of programming agents by reward and punishment without needing to specify how the task is to be achieved**.”
- Specify what to do, but not how to do it.
  - Only **formulate** the **reward** function.
  - **Learning** “fills in the details”.
- Compute better **final solutions** for a task.
  - Based on actual **experiences**, not on **programmer** assumptions.
- Less (human) time **needed** to find a **good solution**.

RL Basics ii



Agent and environment interact at discrete time steps: $t = 0, 1, 2, \ldots$

Agent observes state at step t: $s_t \in S$

produces action at step t: $a_t \in A(s_t)$

gets resulting reward: $r_{t+1} \in \Re$

and resulting next state: s_{t+1}

Figure 3: Agent environment interaction in RL

RL Basics iii

- In **reinforcement learning** problems the **feedback** is simply a scalar value which may be **delayed** in time.
- This **reinforcement signal** reflects the **success** or **failure** of the entire system after it has **performed** some sequence of actions.
- Hence the **reinforcement signal** does not assign **credit** or **blame** to any one action (the **temporal credit assignment problem**), or to any particular node or system element (**the structural credit assignment problem**).

RL Basics iv

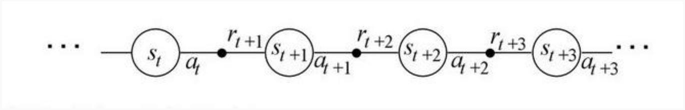


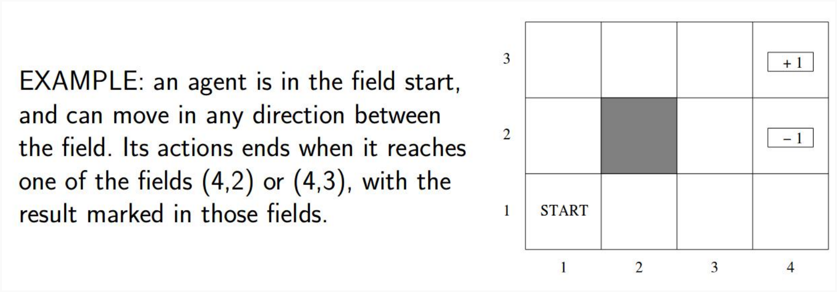
Figure 4: Agent environment interaction in RL

Sequential Decision Problems. Markov Decision Processes

Definition

- In **sequential decision problems** the utility of agent's actions do not **depend on** single **decisions**, **expressed** with the state, which the agent would have gotten into, as the result of this decision, but rather on the **whole sequence** of agent's action.

Tne World i



EXAMPLE: an agent is in the field start, and can move in any direction between the field. Its actions ends when it reaches one of the fields (4,2) or (4,3), with the result marked in those fields.

Figure 5: Agent environment interaction in RL

- **Reward** for (4,3) is +1, for (4,2) is -1, for **any other state** is -0.04.

Tne World ii

- If the problem was fully deterministic — and the agent's knowledge of its position was complete — then the problem would be reduced to action planning. For example, for the above example the correct solution would be the action plan: U-U-R-R-R. Equally good would be the plan: R-R-U-U-R. If the single actions did not cost anything (ie. only the final state did matter), then equally good would also be the plan: R-R-R-L-L-L-U-U-R-R-R, and many others

Tne World iii

- Stochastic:

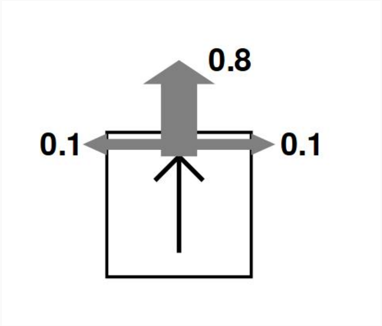


Figure 6: Agent environment interaction in RL

Tne World iv

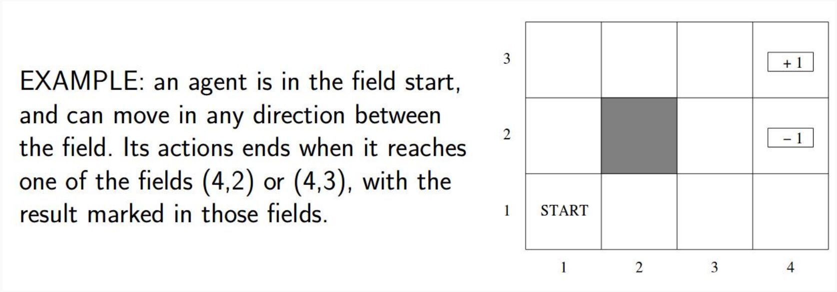

EXAMPLE: an agent is in the field start, and can move in any direction between the field. Its actions ends when it reaches one of the fields (4,2) or (4,3), with the result marked in those fields.

Figure 7: Agent environment interaction in RL

- What is the reliability of our sequence U-U-R-R-R?

Tne World v

EXAMPLE: an agent is in the field start, and can move in any direction between the field. Its actions ends when it reaches one of the fields (4,2) or (4,3), with the result marked in those fields.

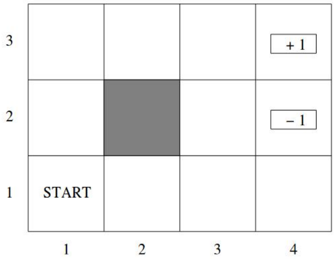


Figure 8: Agent environment interaction in RL

- What is the reliability of our sequence U-U-R-R-R?
- The solution is $(0.8^5) + (0.1^4 * 0.8) = 0.32768 + 0.00008 = 0.32776$

Definition MDP i

- A Markov Decision Process (MDP) is an **extension** of the **standard** (unhidden) Markov model
- Each **state** has a **collection** of actions that can be **performed** in that particular state.
- These **actions** serve to **move** the system into a **new state**.
- More formally, the **MDP's state transitions** can be **described** by the **transition function** $T(s, a, s')$, where a is an action moving performable during the current state s, and s' is some **new state**.

Definition MDP ii

- MDPs holds that the **probability of finding the system** in a given state is dependent only on the previous state.

$$P(S_t = s' | S_{t-1} = s,\ a_t = a) = T(s, a, s')$$

- Each **MDP also has a reward function** $R : S \mapsto \mathbb{R}$. This reward function **assigns** some value $R(s)$ to being in the state $s \in S$.

Definition MDP iii

- The **goal** of a Markov Decision Process is to **move** from the **current state** s to some **final state** in a way that a) **maximizes** $R(s)$ and b) **maximizes** R's potential value in the future.
- Given a Markov Decision Process **we wish to find a** *policy* – a **mapping** from states to actions.
- The **policy** function $\Pi : S \mapsto A$ selects the **appropriate** action $a \in A$ given the **current state** $s \in S$.
- The **optimal policy function** Π^* is the policy that **maximizes** the expected utility.

Optimal Policy i

We can determine the optimal policy for the previous example problem. Note that at point (3,2) the policy makes the agent move left, which may seem wrong, but allows the agent to avoid ending up in state (4,2). Similarly in state (4,1).

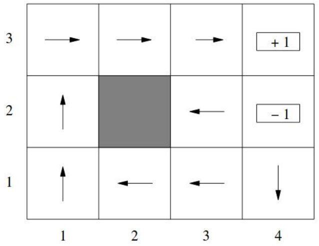


Figure 9: Agent environment interaction in RL

Optimal Policy ii

- **Previous** policy assumes **zero cost of the moves**. Considering the agent's **outcome** not only the final state but the **number of moves**, maybe is not **optimal**.
- **Question**: Justify the **optimal policies** for (1,1), (1,3), (4,1) and (3,2).

Optimal Policy iii

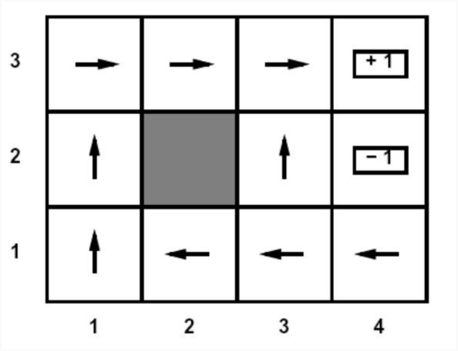


Figure 10: Optimal policy with R(S)=-0.04

Optimal Policy iv

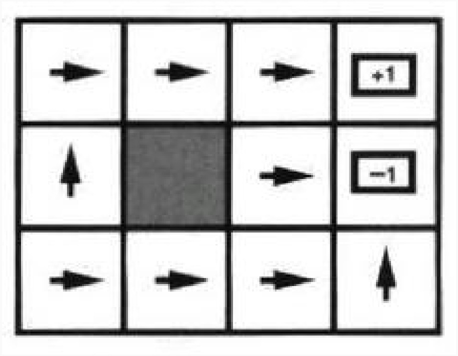


Figure 11: Optimal policy with R(S) ?

Optimal Policy v

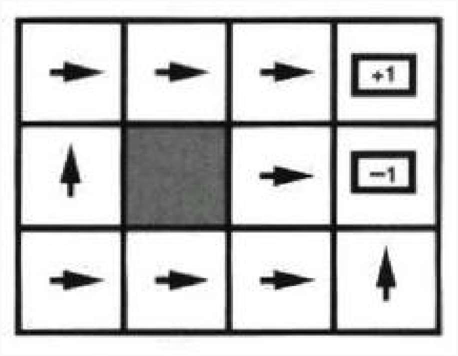


Figure 12: Optimal policy with R(S)<-1.6284

Optimal Policy vi

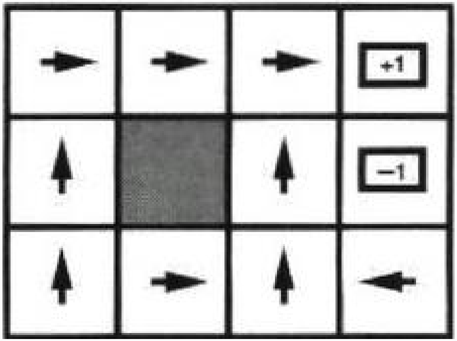


Figure 13: Optimal policy with R(S) ?

Optimal Policy vii

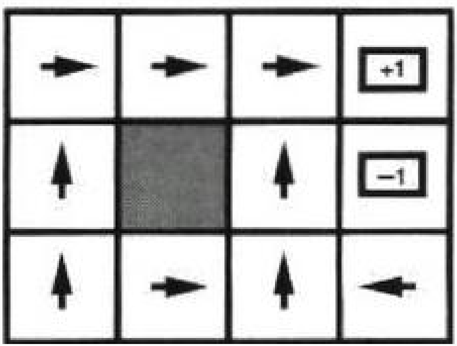


Figure 14: Optimal policy with $-0,4278 < R(S) < -0,0850$

Optimal Policy viii

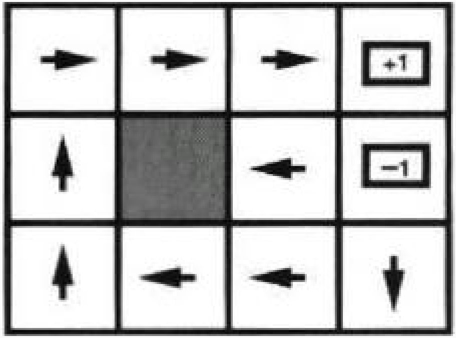


Figure 15: Optimal policy with R(S) ?

Optimal Policy ix

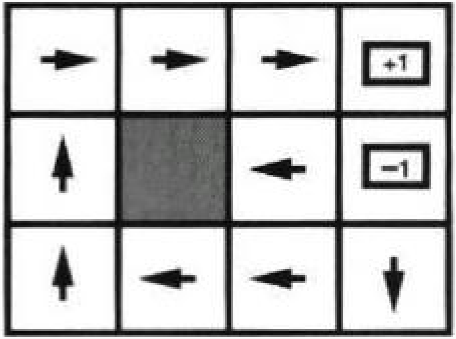


Figure 16: Optimal policy with $-0,0221 < R(S) < 0$

Optimal Policy x

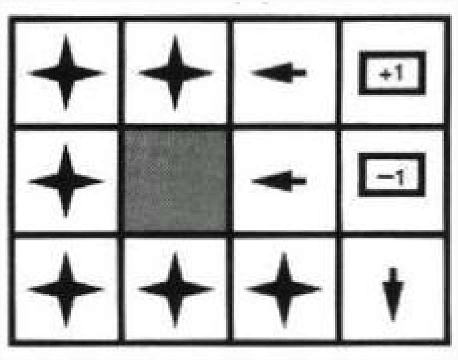


Figure 17: Optimal policy with R(S) ?

Optimal Policy xi

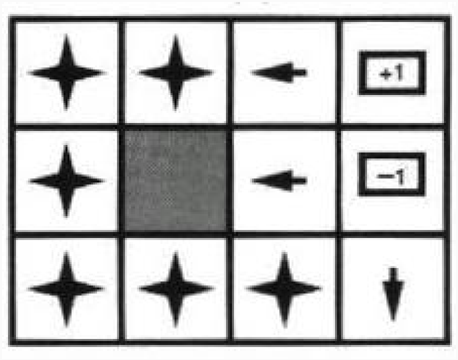


Figure 18: Optimal policy with R(S)>0

Main Notions: Policies

- *Policy*: The **function** that allows us to **compute** the next action for a **particular** state.
- An *optimal Policy* is a policy that **maximizes** the expected reward/**reinforcement**/feedback of a **state**.
- Thus, the task of RL is to use **observed rewards** to find an **optimal policy** for the environment.

Main Notions: Modes of Learning

- *Passive Learning*: Agents **policy** is fixed and our task is to **learn** how good the policy is.
- *Active Learning*: Agents **update** policies as they **learn**.
- *Model based learning*: Learn **transition** and **reward** value, use it to get the **optimal policy**.
- *Model free learning*: derive the **optimal policy** without **learning** the model.

Main Notions: Exploration vs. Exploitation

- *Exploitation* Use the **knowledge** already learned on what the next best **action** is in the **current state**.
- *Exploration* In order to improve policies the agent must explore a **number of states**. I.e., select an action **different** of the one that it currently thinks is best.

Difficulties of Reinforcement learning i

- *Blame attribution problem*: The problem of **determining** which action was **responsible** for a **reward** or **punishment**.
    - Responsible action may have occurred a **long time before** the reward was **received**.
    - A **combination** of **actions** might have **lead** to a **reward**.

Difficulties of Reinforcement learning ii

- *Recognising delayed rewards*: What **seem** to be **poor** actions now might lead to much **greater** rewards in the **future** than what **appears** to be good **actions**.
  - Future **rewards** need to be **recognised** and **back-propagated**.
  - Problem **complexity** increases if the **world** is **dynamic**.

Difficulties of Reinforcement learning iii

- *Explore-exploit dilemma*: If the agent has worked out a good course of actions, should it continue to follow these actions or should it explore to find better actions?
    - Agent that never explores can not improve its policy.
    - Agent that only explores never uses what it has learned.

Value Iteration

Introduction

- The **consequences** of **actions** (i.e., rewards) and the effects of **policies** are not always known **immediately**. As such, we need some **mechanisms** to **control** and **adjust** policy when the **rewards** of the current state space are **uncertain**. These **mechanisms** are collectively referred to as **reinforcement learning**

Discount factor

- One **common** way of trading off **present** reward against **future** reward is by introducing a *discount rate* γ. The discount rate is **between** 0 and 1, and we can use it to **construct** a **weighted** estimate of **future** rewards:

$$\sum_{t=0}^{\infty} \gamma^t r_t$$

- Here, we **assume** that $t = 0$ is the current time. Since $0 < \gamma < 1$, **greater** values of t (indicating rewards farther in the future) are given **smaller weight** than rewards in the nearer future.

Value Function i

- Let $V^{\Pi}(s)$ be the **value function** for the policy Π. This function $V^{\Pi} : S \mapsto \mathbb{R}$ maps the **application** of Π to some state $s \in S$ to some reward value. Assuming the system starts in state s_0, we would **expect** the system to have the **value**

$$V^{\Pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) \mid s_0 = s, \Pi\right]$$

Value Function ii

- Since the **probability** of the system being in a **given state** $s' \in S$ is **determined** by the **transition** function $T(s, a, s')$, we can rewrite the **formula** above for some **arbitrary** state $s \in S$ as

  $$V^{\Pi}(s) = R(s) + \sum_{s'} T(s, a, s')\gamma V^{\Pi}(s')$$

  where $a = \Pi(s)$ is the **action selected** by the **policy** for the given **state** s.

Value Function iii

- Our goal here is to **determine the optimal policy Π*(s)**. Examining the formula above, we see that $R(s)$ is **unaffected** by choice of policy. This makes **sense** because at any given state s, local reward term $R(s)$ is **determined** simply by **virtue** of the fact that the system is in state s.
- Thus, if we wish to find the **maximum policy** value function (and therefore find the optimum policy) we must find the action a that **maximises** the summation term above:

$$V^{\Pi^*}(s) = R(s) + \max_a \sum_{s'} T(s, a, s')\gamma V^{\Pi^*}(s')$$

  Note that this **formulation** assumes that the number of **states** is **finite**.

Value Iteration Algorithm i

- The formula above forms the **basis** of the *value iteration* algorithm. This operation **starts with some initial policy** value function guess and **iteratively** refines $V(s)$ until some acceptable **convergence** is reached:

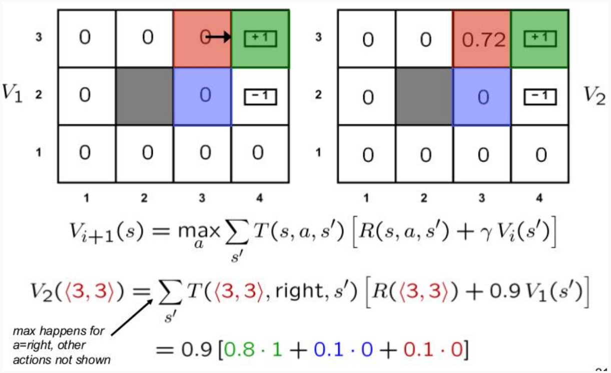
Value Iteration Algorithm ii

```
1. Initialize $V(s)$.
2. Repeat until converged:
    2.1 for all $s \in S$:
        2.1.1 $R(s) = R(s) + \max_a \sum_{s'} T(s, a, s')\gamma V(s')$
        2.1.2 $\Pi(s) = \arg\max_a Q(s, a)$
```

Value Iteration Algorithm iii

- Each **pass of the value** iteration maximises $V(s)$ and assigns to $\Pi^*(s)$ the action a that **maximises** $V(s)$. The function $Q(s, a)$ represents the **potential** value for $V(s)$ **produced** by the action $a \in A$.

Value Iteration Algorithm iv



$$V_{i+1}(s) = \max_a \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V_i(s')\right]$$

$$V_2(\langle 3,3\rangle) = \sum_{s'} T(\langle 3,3\rangle, \text{right}, s') \left[R(\langle 3,3\rangle) + 0.9\, V_1(s')\right]$$

$$= 0.9\,[0.8 \cdot 1 + 0.1 \cdot 0 + 0.1 \cdot 0]$$

Figure 19: Example Bellmann Updates

Value Iteration Algorithm v

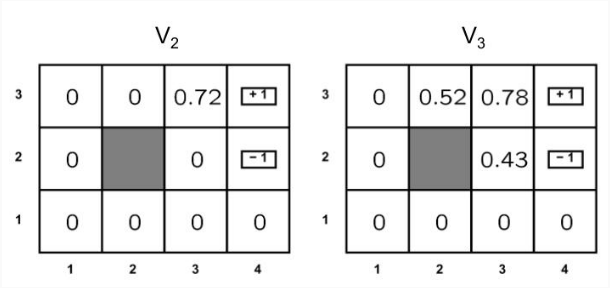


Figure 20: Example Bellman Updates

Value Iteration Algorithm vi

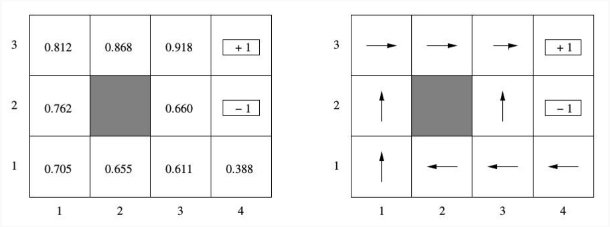


Figure 21: Final Values

Value Iteration Algorithm vii

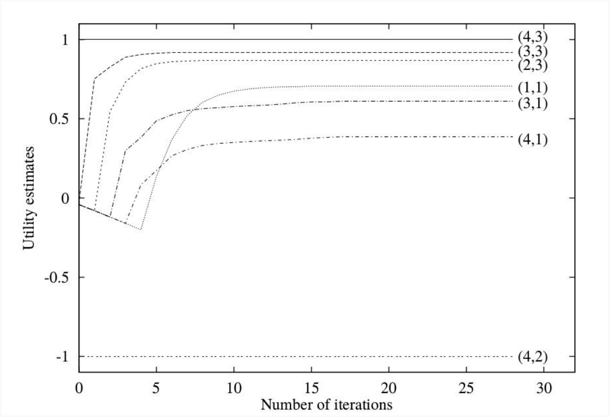


Figure 22: Convergence

Q-learning

Q-function

- In addition to the **state** value-function, for **convenience** RL **algorithms** introduce another function which is the **state-action** pair Q function. $Q : S \times A \rightarrow R$
- Q-Learning poses an idea of **assessing** the **quality** of an **action** that is taken to move to a **state** rather than determining the possible **value** of the state being moved to
- The optimal **Q-function Q*(s, a)** means the **expected** total **reward** received by an agent starting in s and picks action a, then will behave optimally afterwards. There, Q*(s, a) is an **indication** for how good it is for an agent to **pick action** a while being in **state s**.

Basic Idea

- **Q-Learning** is an example of **model-free learning** algorithm. It does not assume that agent knows anything about the **state-transition** and reward models. However, the agent will **discover** what are the good and **bad actions** by trial and error.
- The **basic** idea of **Q-Learning** is to approximate the **state-action pair**s Q-function from the samples of Q(s, a) that we observe during **interaction** with the environment. This approach is known as **Time-Difference Learning**.

Q-Learning

- Learn **quality** of state-action combinations:: $Q : S \times A \rightarrow R$
- We **learn** Q over a (possibly infinite) **sequence** of discrete time events.

$$\langle s0, a0, r1, s1, a1, r2, s2, a2, r3, s3, a3, r4, s4 \ldots \rangle$$

  where s_i are states, a_i actions, y r_i rewards.
- Learn **quality** of a **single experience** that, i.e., for $\langle s, a, r, s' \rangle$.

Q-Learning Algorithm i

- **Maintain** a **table** for Q with an **entry** for each valid **state-action pair** (s, a).
- **Initialise** the table with **some uniform** value.
- **Update** the **values** over **time points** $t \geq 0$ according to the following formula:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \times \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

where α is the **learning rate** and γ is the **discount factor**.

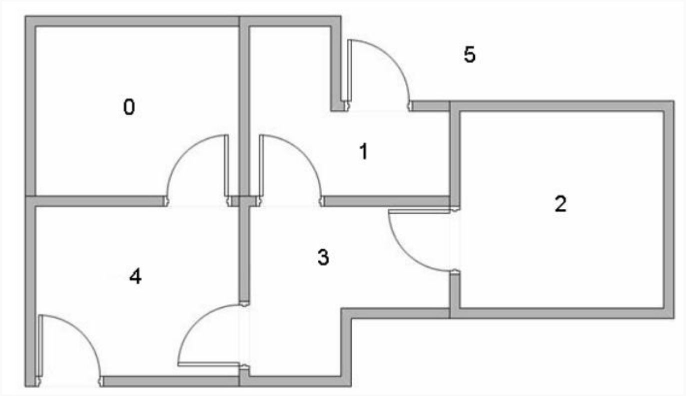
Q-Learning Algorithm ii

1. Set the γ parameter and environment rewards in matrix R
2. Initialize matriz Q to zero.
3. For each episode:
    - Select a random initial state.
    - Do While the goal state hasn't been reached:
        3.1 Select one among all possible actions for the current state.
        3.2 Using this possible action, consider going to the next state.
        3.3 Get maximum Q value for this next state based on all possible actions.
        3.4 Compute $Q(state, action)$
        3.5 Set the next state as current state.
    - End Do While
4. End For

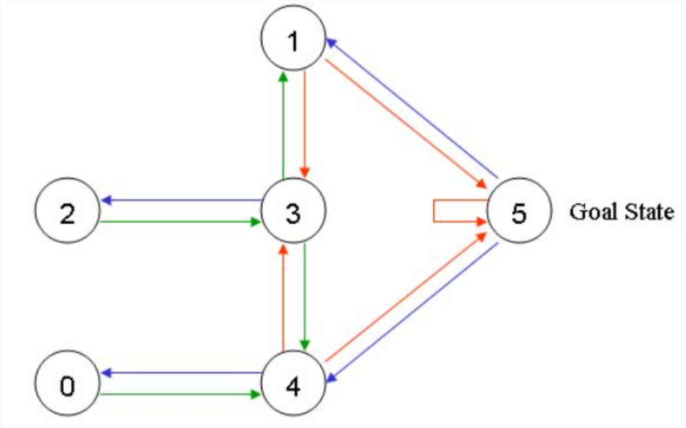
Learning rate

- Models how **forgetful** an agent is
- Learning rate α (small Greek letter alpha) determines to what **extend the newly acquired information** will override the old information.
- $\alpha = 0$ will the agent **not learn anything**.
- $\alpha = 1$ will make the agent **consider only the most recent information**.

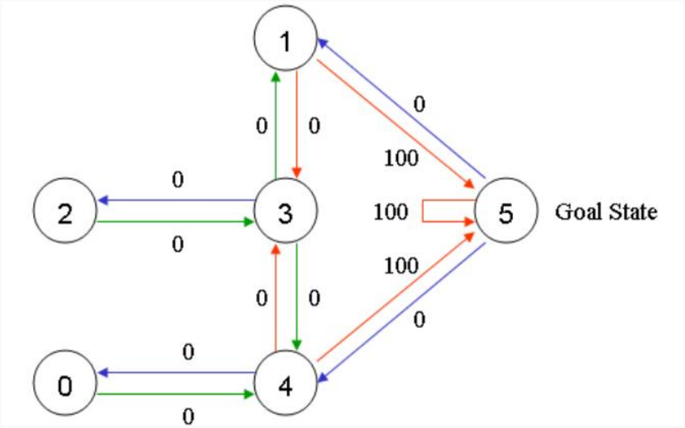
Path finding problem i



Path finding problem ii

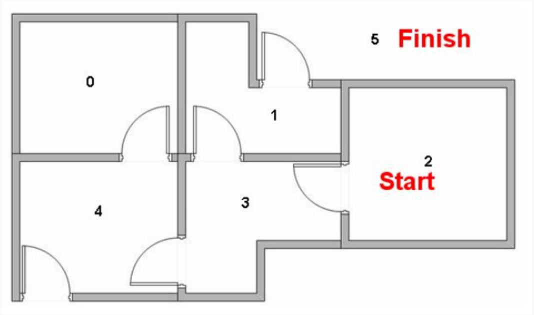


Path finding problem iii



Path finding problem iv

- Suppose we want to model some kind of **simple evacuation of an agent from any room in the buildin**g. Now suppose we have an agent in Room 2 and we want the agent to learn to reach outside the house (5).



Path finding problem v

- We can put the **state diagram and the instant reward values** into the following reward table, ”matrix R”

$$
\begin{array}{ccc}
 & \textbf{State} & \begin{array}{cccccc} & & \textbf{Action} & & & \\ 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
R= & \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} & \begin{bmatrix} -1 & -1 & -1 & -1 & 0 & -1 \\ -1 & -1 & -1 & 0 & -1 & 100 \\ -1 & -1 & -1 & 0 & -1 & -1 \\ -1 & 0 & 0 & -1 & 0 & -1 \\ 0 & -1 & -1 & 0 & -1 & 100 \\ -1 & 0 & -1 & -1 & 0 & 100 \end{bmatrix}
\end{array}
$$

Path finding problem vi

- Now we'll add a similar matrix, "Q", to the **brain of our agent**, representing the **memory of what the agent has learned** through experience.

$$Q = \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

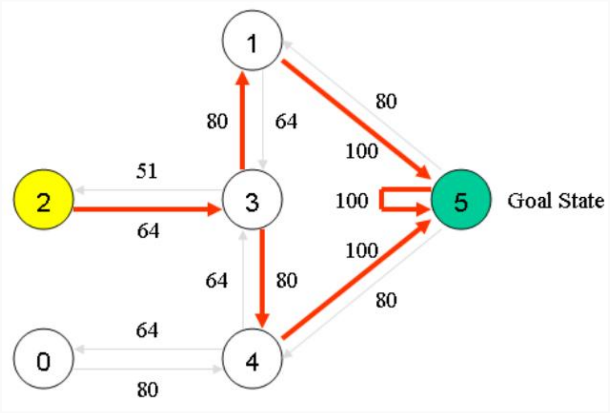
Path finding problem vii

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 400 & 0 \\ 0 & 0 & 0 & 320 & 0 & 500 \\ 0 & 0 & 0 & 320 & 0 & 0 \\ 0 & 400 & 256 & 0 & 400 & 0 \\ 320 & 0 & 0 & 320 & 0 & 500 \\ 0 & 400 & 0 & 0 & 400 & 500 \end{bmatrix} \end{array}$$

Path finding problem viii

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{bmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 51 & 0 & 80 & 0 \\ 64 & 0 & 0 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{bmatrix} \end{array}$$

Path finding problem ix



Path finding problem x

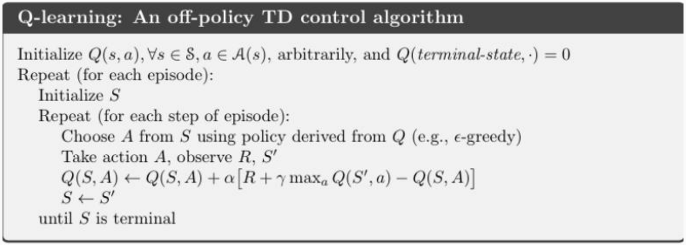
For example, **from initial State 2**, the agent can use the matrix Q as a guide:

- From **State 2** the **maximum** Q values suggests the action to go to **state 3**.
- From **State 3** the maximum Q values **suggest** two alternatives: go to state 1 or 4. Suppose we **arbitrarily** choose to go to 1.
- From State 1 the **maximum Q** values **suggests** the action to go to state 5.
- **Thus the sequence is 2-3-1-5.**

Grid World Example i

- Use a **reinforcement learning algorithm to compute the best policy** for finding the **gold** with as few steps as possible while avoiding the **bomb**.
- Reward function: −1 for each **navigation action**, an additional +10 for finding the **gold**, and an additional −10 for hitting the **bomb**..
- Do **not use discounting** (that is, set γ=1) and the learning rate is α=0.1.

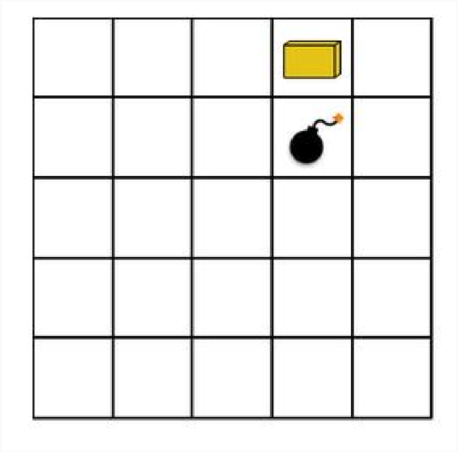
Grid World Example ii



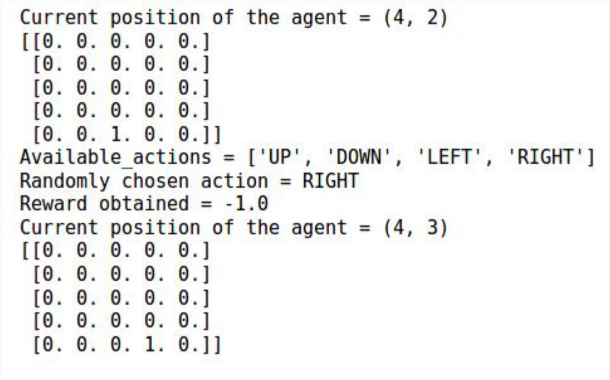
Q-learning: An off-policy TD control algorithm

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Repeat (for each step of episode):
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Grid World Example iii



Grid World Example iv



```
Current position of the agent = (4, 2)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
Available_actions = ['UP', 'DOWN', 'LEFT', 'RIGHT']
Randomly chosen action = RIGHT
Reward obtained = -1.0
Current position of the agent = (4, 3)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0.]]
```

Grid World Example v

- current_q_value + self.alpha * (reward + self.gamma * max_q_value_in_new_state - current_q_value)
- New Q-value for (0, 1) UP is -0.1
- 0 + 0.1 * (-1.0 + 1 * 0 - 0)
- Reward obtained = -1.0
- New position of the agent = (0, 1)

Grid World Example vi

- current_q_value + self.alpha * (reward + self.gamma * max_q_value_in_new_state - current_q_value)
- New Q-value for (1, 2) RIGHT is -1.0
- 0 + 0.1 * (-10.0 + 1 * 0 - 0)
- Reward obtained = -10.0
- New position of the agent = (1, 3)

Grid World Example vii

- current_q_value + self.alpha * (reward + self.gamma * max_q_value_in_new_state - current_q_value)
- New Q-value for (0, 2) RIGHT is 1.9
- 1.0 + 0.1 * (10.0 + 1 * 0 - 1.0)
- Reward obtained = -0.0
- New position of the agent = (0, 3)

Grid World Example viii

- current_q_value + self.alpha * (reward + self.gamma * max_q_value_in_new_state - current_q_value)
- New Q-value for (0, 2) RIGHT is 2.71
- 1.9 + 0.1 * (10.0 + 1 * 0 - 1.9)
- Reward obtained = 10.0
- New position of the agent = (0, 3)

Grid World Example ix

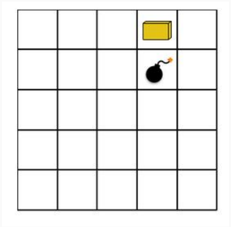
- current_q_value + self.alpha * (reward + self.gamma * max_q_value_in_new_state - current_q_value)
- New Q-value for (1, 2) UP is 2.3751096381
- 2.026431198 + 0.1 * (-1.0 + 1 * 6.513215599 - 2.026431198)
- Reward obtained = -1.0
- New position of the agent = (0, 2)

Grid World Example x

- current_q_value + self.alpha * (reward + self.gamma * max_q_value_in_new_state - current_q_value)
- New Q-value for (0, 2) RIGHT is 6.8618940391
- 6.513215599 + 0.1 * (10.0 + 1 * 0 - 6.513215599)
- Reward obtained = 10.0
- New position of the agent = (0, 3)

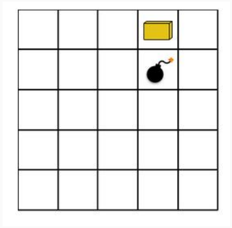
Grid World Example xi

- 100 **Episodes**, Final Values for (1, 2):
- LEFT: 1.41817474422
- RIGHT: -4.68559
- DOWN: 3.14201650717
- UP: 9.0



Grid World Example xii

- 100 **Episodes**, final values for (3, 0):
- LEFT: -1.39
- RIGHT: -0.915458555225
- DOWN: -1.40497195927
- UP: -1.2965333916



Grid World Example xiii

- 100 **Episodes**, Final values for (2, 3):
- LEFT: 3.05343350461
- RIGHT: -0.6653219
- DOWN: -0.6721544332
- UP: -1.0