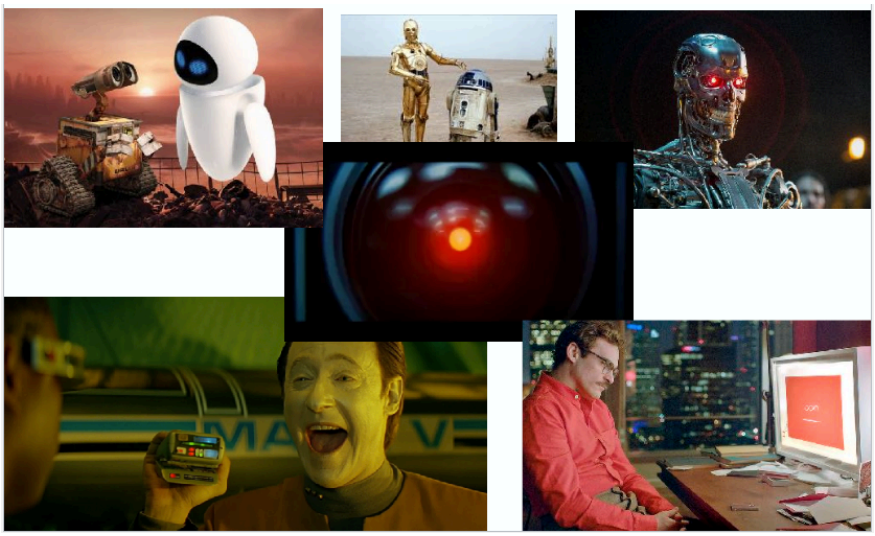
BIM309 Artificial Intelligence

Week 2 – Introduction to AI

Outline

- Overview of AI
- Applications of AI
- AI Foundations and History

AI in the Movies



Definition of AI

"**Intelligence**: The ability to **learn** and **solve problems**"

Webster's Dictionary

"**Artificial Intelligence (AI)** is the intelligence exhibited by machines or software"

Wikipedia

"The science and engineering of making **intelligent machines**"

McCarthy

"The study and design of **intelligent agents**, where an intelligent agent is a system that **perceives its environment** and **takes actions** that maximize its **chances of success**"

Russel and Norvig AI book

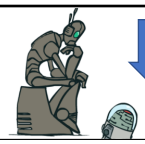
Why AI?

"If we can make computers more intelligent and understand the world and the environment better, it can make life so much better for many of us. *Just as the Industrial Revolution freed up a lot of humanity from physical drudgery, I think AI has the potential to free up humanity from a lot of the mental drudgery.*"

Andrew Ng

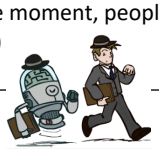
What is AI?

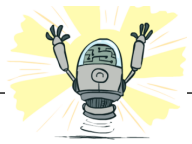
Measure success in terms of fidelity to ***human*** performance



Measure success against an ***ideal*** concept of intelligence, ***rationality***

	Thinking Humanly (1)	**Thinking Rationally** (3)
concerned with ***Thought processes*** and ***reasoning*** →	"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)	"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)
	Acting Humanly (2)	**Acting Rationally** (4)
address ***Behaviour*** →	"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 which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998) "AI … is concerned with intelligent behavior in artifacts." (Nilsson, 1998)





Four schools of thoughts (Russel & Norvig)

What is AI?

1. Thinking Humanly: Cognitive Approach

machines with minds 

- Requires to determine how humans think!
- 1960's "cognitive revolution"
- Requires scientific theories of internal activities of the brain
  - What level of abstraction?
  - Is it "Knowledge" or "circuits"?
  - How to validate the knowledge?

Today, Cognitive Science and Artificial Intelligence are distinct disciplines.

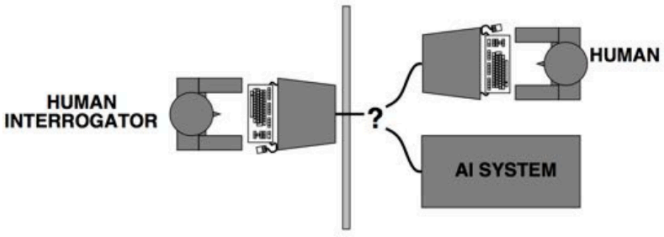
What is AI?

2. Acting Humanly:

machines do things just like humans 

- ***Turing test (Alan Turing 1950):*** A computer passes the test of intelligence, if it can fool a human interrogator

Can machines think?
Imitation Game



Problem:
Turing test is not reproducible or amenable to mathematical analysis

- ***Major components of AI:*** knowledge, reasoning, language, understanding, learning

Turing, Alan M. "Computing machinery and intelligence." Mind 59.236 (1950): 433-460.

What is AI?

3. Thinking Rationally: Law of thoughts

involves using math and logic to do AI

- Codify "right thinking" with **logic**
- Several Greek schools developed various forms of logic: *notation* and *rules of derivation* for thoughts
- Problems:
  1. Not all knowledge can be expressed with logical notations
  2. Computational blow up
  3. Logical systems tend to do the wrong thing in the presence of **uncertainty**

What is AI?

4. Acting Rationally: Rational Agents

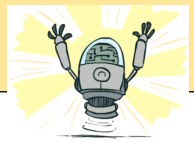
machines to act rationally rather than humanly

- **Rational behavior: doing the "right thing"**
  - The ***right thing***: which is expected to maximize goal achievement, given the available information about the environment
  - Doesn't necessarily involve thinking, e.g., blinking reflex
  - Thinking should be in the service of rational action
  - Entirely dependent on goals!
  - Rational ≠ successful OR Irrational ≠ insane, irrationality is sub-optimal action
- **Our focus here: Rational Agents**
  - A **rational agent** is one that acts so as to achieve the best outcome, or when there is uncertainty, the best expected outcome
  - Systems which make the best possible decisions given goals, evidence, and constraints
  - In the real world, usually lots of uncertainty … and lots of complexity
  - Usually, we're just approximating rationality
- Aristotle (Nicomachean Ethics):

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

What is AI?

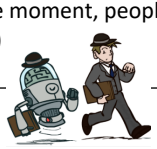
Thinking Humanly (1)	Thinking Rationally (3)
"The exciting new effort to make computers think … machines with minds, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"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)
Acting Humanly (2)	**Acting Rationally → OUR APPROACH (4)**
"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 which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998) "AI … is concerned with intelligent behavior in artifacts." (Nilsson, 1998)



Four schools of thoughts (Russel & Norvig)

What is AI?

Thinking Humanly (1)	Thinking Rationally (3)
"The exciting new effort to make computers think … machines with minds, in the full and literal sense." (Haugeland, 1985) **machines with minds** **Cognitive Approach** "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) **involves using math and logic to do AI** **Law of thoughts** "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992) **codifying the right thinking with logic**
Acting Humanly (2)	**Acting Rationally → OUR APPROACH (4)**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) **machines do things just like humans** "The study of how to make computers do things which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998) **machines to act rationally rather than humanly** "AI … is concerned with intelligent behavior in artifacts." (Nilsson, 1998)



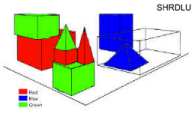
Four schools of thoughts (Russel & Norvig)

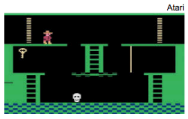
AI Today

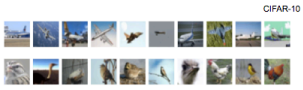
- Prior to 2012, AI results closely tracked Moore's Law, with compute doubling every two years
- Post-2012, compute has been **doubling every 3.4 months**
- In 2019, the largest AI conference, **NeurIPS**, expects 13,500 attendees, up 41% over 2018 and over **800% relative to 2012**
- In the US, the share of **jobs** in AI-related topics increased from **0.26% of total jobs posted in 2010 to 1.32% in October 2019**

AI Today: In Vitro vs. Real-World

- One of the biggest changes in AI is the transition from the lab to the real-world
  - AI was limited to artificial enviroments and datasets
  - Spur the development of new methods
- Now much more real-world deployment of AI in ways that have a direct impact on people's lives



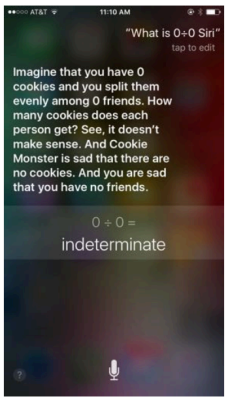




Applications of AI

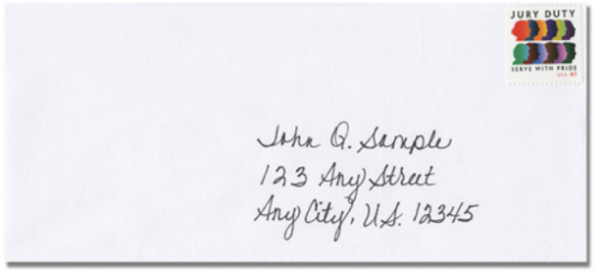
Speech Recognition & Question Answering

- ***Virtual assistants***: Siri (Apple), Alexa (Amazon), Google Assistant, Cortana (Microsoft)
- "They" helps get things done: send an email, make an appointment, find a restaurant, tell you the weather and more
- Leverage deep neural networks to handle **speech recognition** and **natural language understanding**



Applications of AI

Handwriting recognition (check, zipcode)



Applications of AI

Machine Translation

- Historical motivation: Translate Russian to English
- First systems using **mechanical translation** (one-to-one correspondence) failed?
- **"Out of sight, out of mind" → "Invisible, imbecile".**

Oops!

- MT has gone through ups and downs
- Today, **Statistical Machine Translation** leverages the vast amounts of **available translated corpuses**
- While there is room for improvement, machine translation has made significant progress

Applications of AI

Machine Translation

Voice & Text Translator
https://www.skype.com/en/features/skype-translator/

Applications of AI

Machine Translation

Applications of AI

https://www.youtube.com/watch?v=HKFHUrx23ts robot nao
https://www.youtube.com/watch?v=SARB9OI_Wz4 robot asimo

Robotics

Awesome robots today! NAO, ASIMO, and more!

Applications of AI

Recommendation Systems (Collaborative Filtering)

Applications of AI

Sentiment Analysis

Input: movie review

Shows moments of promise but ultimately succumbs to clichés and pat storytelling.

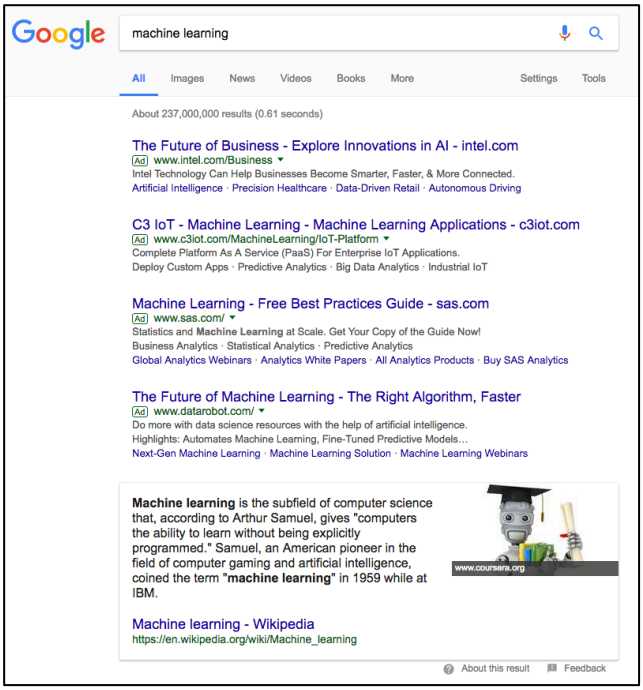
Output: sentiment

POSITIVE or **NEGATIVE**

Applications of AI

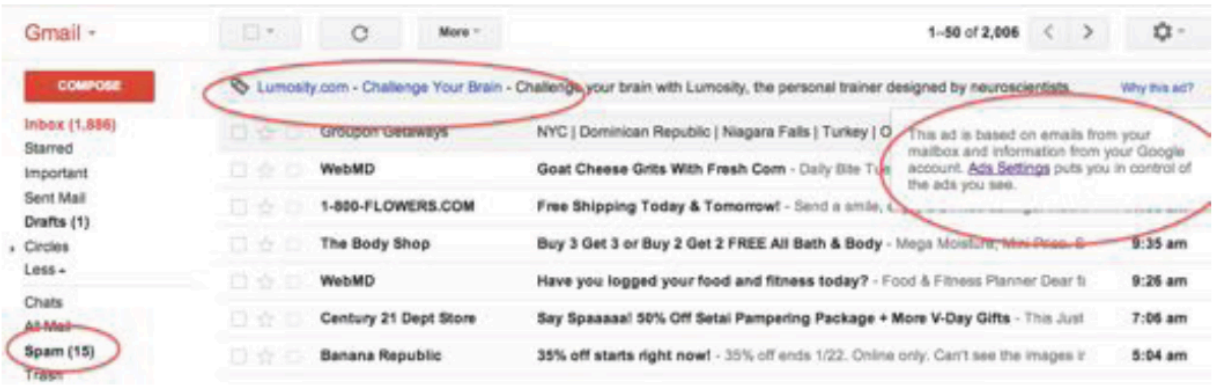
Search Engines

Information Retrieval



Applications of AI

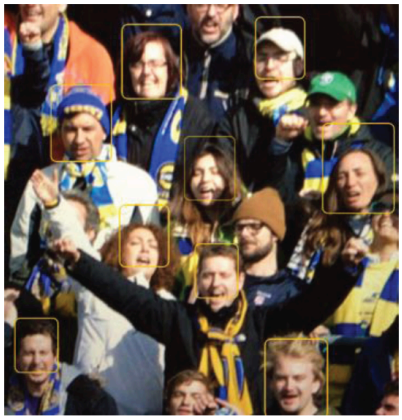
Email



Applications of AI

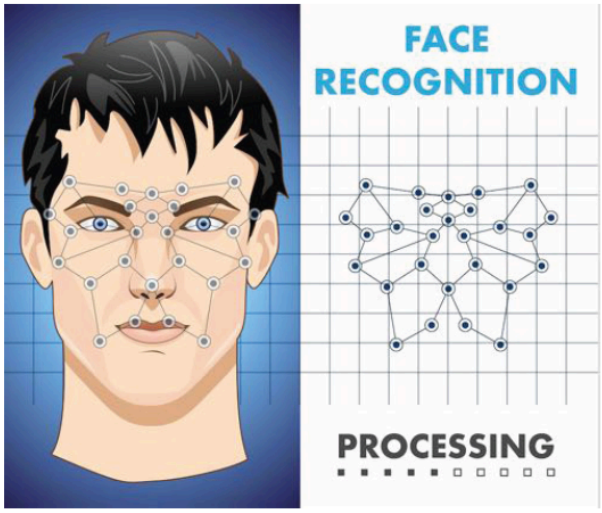
Face Detection

Viola-Jones method



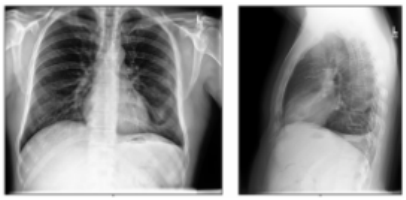
Applications of AI

Face Recognition

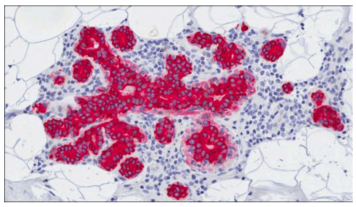


Applications of AI

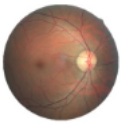
Chest Radiology



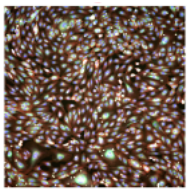
Detection of breast cancer in mammography images



Diabetic Retinopathy



Drug Screening for COVID-19



Healthcare

Applications of AI

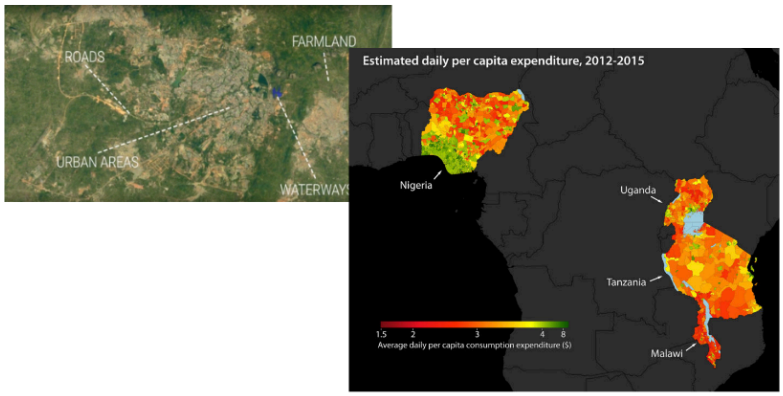
Visual Assistive Technology

- Seeing AI, the free Microsoft app for iOS for those who are blind or who have low vision
- Read printed text, currency, and describe physical objects, product labels and colors, among other things
- Supports English, Dutch, French, German, Japanese and Spanish

https://www.microsoft.com/en-us/ai/seeing-ai

Applications of AI

Poverty Mapping



Applications of AI

Chess (1997): Kasparov vs. IBM Deep Blue

Powerful search algorithms!

Applications of AI

Jeopardy! (2011): Humans vs. IBM Watson

Natural Language Understanding and Information Extraction!

Applications of AI

Go (2016): Lee Sedol vs. Google AlphaGo

AlphaGo versus Lee Sedol
4–1

Seoul, South Korea, 9–15 March 2016	
Game one	AlphaGo won.
Game two	AlphaGo won.
Game three	AlphaGo won.
Game four	Lee Sedol won.
Game five	AlphaGo won.

"I misjudged the capabilities of AlphaGo and felt powerless.", quote after game 3.

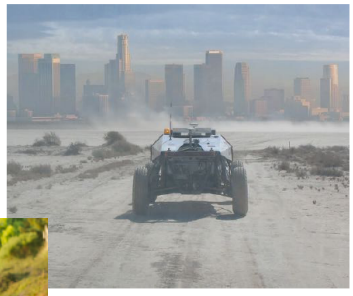
AlphaGo

Deep Learning, Reinforcement Learning and Search Algorithms!

Applications of AI

Autonomous Driving

- DARPA Grand Challenge
  - 2005: 132 miles
  - 2007: Urban challenge
  - 2009: Google self-driving car
  - 2015: Uber started self-driving car



State-of-the-art AI Applications

- Speech recognition
- Autonomous planning and scheduling
- Financial forecasting
- Game playing, video games
- Spam fighting
- Logistics planning
- Robotics (household, surgery, navigation)
- Machine translation
- Information extraction
- VLSI layout
- Automatic assembly
- Sentiment analysis
- Computer animation
- Fraud detection
- Recommendation systems
- Web search engines
- Autonomous cars
- Energy optimization
- Question answering systems
- Social network analysis
- Medical diagnosis, imaging
- Route finding
- Traveling salesperson
- Protein design
- Document summarization
- Transportation/scheduling

Many more!!!

Can you tell the difference between a real face and an AI-generated fake?

https://www.theverge.com/2019/3/3/18244984/ai-generated-fake-which-face-is-real-test-stylegan

http://whichfaceisreal.com/

Let's try Lyrebird

https://lyrebird.ai/

NVIDIA's GauGAN AI has learned to conversion the drawings into a chart!

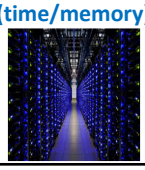
www.nvidia.com/en-us/research/ai-playground

Characteristics of AI Tasks

What's in common with all of these AI applications?

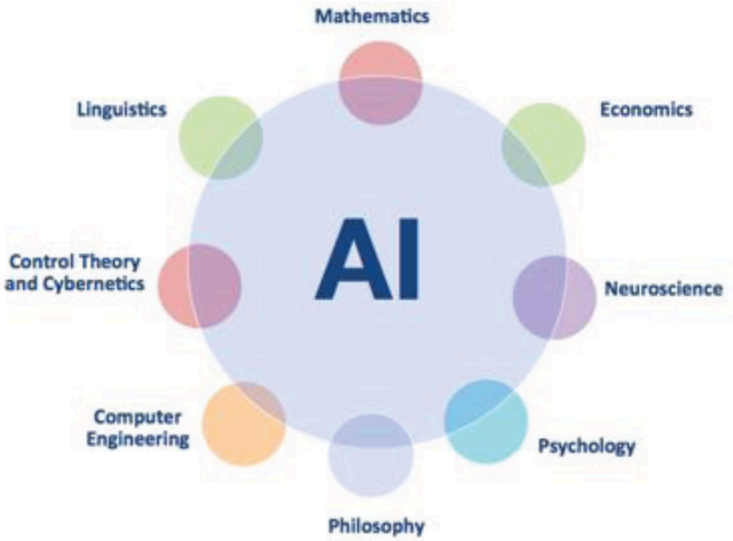
- **High societal impact** (affect billions of people)
  - It's clear that AI applications tend to be very **high impact**
- **Diverse** (language, games, robotics)
  - They are incredibly **diverse**, operating in very different domains, and requiring integration with many different modalities (natural language, vision, robotics)
- **Complex** (really hard)
  - These applications are also mind-bogglingly **complex** to the point where we shouldn't expect to find solutions that solve these problems perfectly
  - Two sources of complexity in AI tasks
    - ***Computational complexity***: exponential explosion
    - ***Information complexity***: need to acquire knowledge, cope with uncertainty

Computation (time/memory)



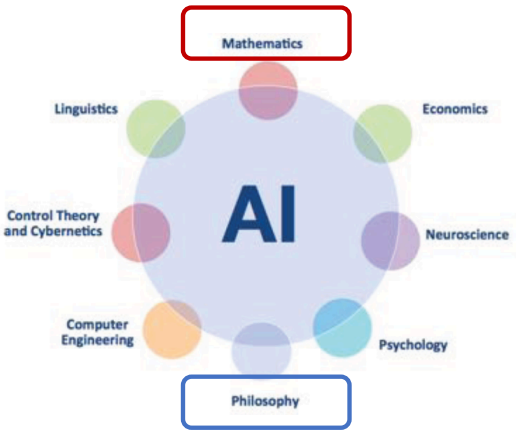
Information (data)

Foundations of AI



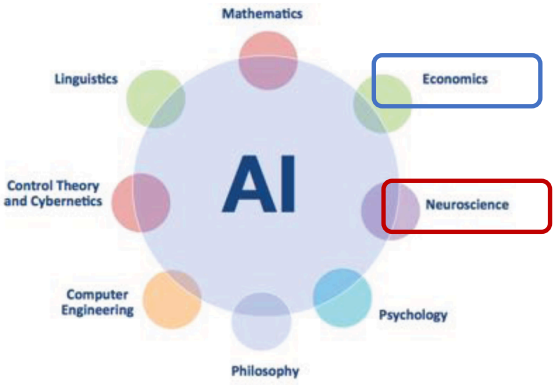
Foundations of AI

- **Philosophy**
  - Logic, methods of reasoning
  - Mind as physical system that operates as a set of rules
  - Foundations of learning, language, rationality
- **Mathematics**
  - *Logic:* Formal representation and proof
  - Computation, algorithms
  - Probability



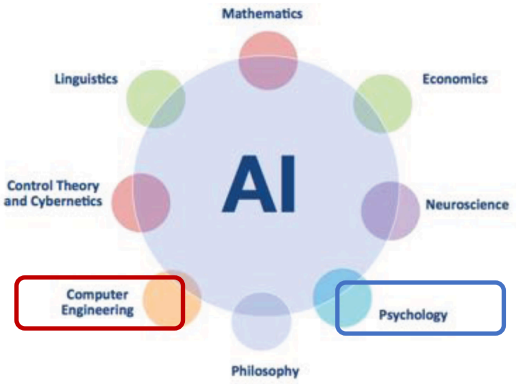
Foundations of AI

- **Economics**
  - Formal theory of rational decisions
  - Combined decision theory and probability theory for decision making under uncertainty
  - Game theory, Markov decision processes
- **Neuroscience**
  - Study of brain functioning
  - How brains and computers are (dis)similar



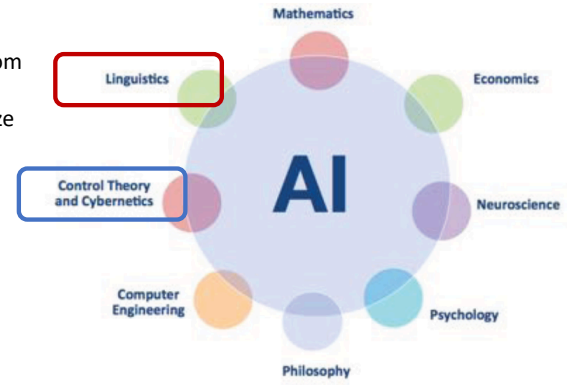
Foundations of AI

- **Psychology**
  - How do we think and act?
  - Cognitive psychology perceives the brain as an information processing machine
  - Led to the development of the field cognitive science: how could computer models be used to study language, memory, and thinking from a psychological perspective
- **Computer Engineering**
  - Cares about how to build powerful machines to make AI possible
  - E.g., Self-driving cars are possible today thanks to advances in computer engineering



Foundations of AI

- **Control Theory and Cybernetics**
  - Design simple optimal agents receiving feedback from the environment
  - Modern control theory design systems that maximize an objective function over time
- **Linguistics**
  - How are language and thinking related
  - Modern linguistics + AI = Computational linguistics (Natural Language Processing - NLP)

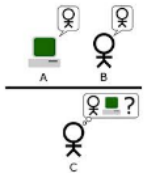


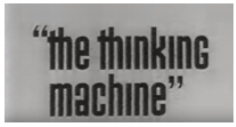
AI Founders

- Aristotle
- Alan Turing
- John Mc Carthy
- Warren McCulloh
- Walter Pitts
- Claude Shannon
- Marvin Minsky
- Dean Edmonds
- Herbert Simon
- Allen Newell
- David Waltz
- Tom Mitchell
- Stuart J. Russell
- Peter Norvig
- etc.

History of AI

- **1940-1950:** Gestation of AI
  - McCulloch & Pitts: Boolean circuit to model of brain
  - Turing's Computing Machinery and Intelligence
    http://www.turingarchive.org/browse.php/B/9
- **1950-1970:** Early enthusiasm, great expectations
  - Early AI programs, Samuel's checkers program
  - Birth of AI @ Dartmouth meeting 1956, John McCarthy coined "AI"
  - Check out the MIT video "The thinking Machine" on YouTube
    https://www.youtube.com/watch?v=aygSMgK3BEM
- **1970-1990:** Knowledge-based AI
  - Expert systems (XCON, MYCIN), AI becomes an industry
  - AI winter






```
if [premises] then [conclusion]
```

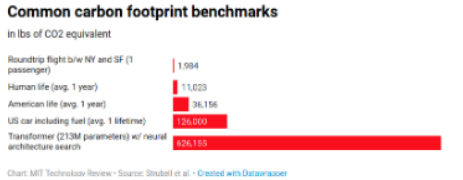
History of AI

https://www.youtube.com/watch?v=hbTXsHZBWdI

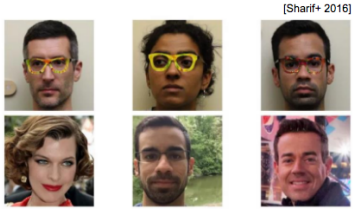
- **1990-present:** Scientific Approaches
  - Rise of Machine Learning in 1990s
  - Neural Networks: le retour
  - The emergence of intelligent agents
  - AI becomes "scientific", use of probability to model uncertainty
  - AI Spring!
    - ❖ Latest rebirth → ***new machine learning techniques, tons of data, and tons of computation***
  - The availability of very large datasets
    - ❖ Data will drive future discoveries and alleviate the complexity in AI

Risks of AI

- Energy Consumption
- Privacy
- Security
- Fairnes



[Sharif+ 2016]



[Evtimov+ 2017]

Summary

- AI is a hard (computational complexity, language, vision, etc) and a broad field with high impact on humanity and society
- What can AI do for us is already amazing!
- AI systems do not have to model human/nature but can act like or be inspired by human/nature
- How human think is beyond the scope of this course
- Rational (do the right thing) agents are central to our approach of AI
- Note that rationality is not always possible in complicated environment but we will still aim to build rational agents

Summary

- AI may be perceived as a scary area!
  Is AI a threat to our humankind?
- Professor Stephen Hawking, eminent scientist told BBC:

"The development of full artificial intelligence could spell the end of the human race."

Next Week: Rational Intelligent Agents

- This course is about designing **intelligent agents**
- An agent perceives the environment and act upon that environment to achieve some task
- An agent is function from percepts to actions
- We care specifically about **rational agents**
- Rationality is relative to how to act to maximize a **performance measure**
- AI aims to design the best agents (programs) that achieve the best performance given the computational limitations

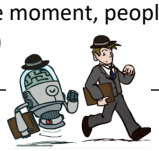
Agent = Architecture + Program

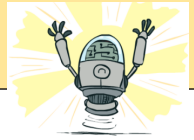
BIM309 Artificial Intelligence

Week 3 – Intelligent Agents

Review: What is AI?

Thinking Humanly (1)	Thinking Rationally (3)
"The exciting new effort to make computers think … machines with minds, in the full and literal sense." (Haugeland, 1985)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985)
machines with minds Cognitive Approach	involves using math and logic to do AI Law of thoughts
"[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
	codifying the right thinking with logic
Acting Humanly (2)	**Acting Rationally → OUR APPROACH (4)**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990)	"Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998)
machines do things just like humans	machines to act rationally rather than humanly
"The study of how to make computers do things which, at the moment, people are better." (Rich and Knight, 1991)	"AI … is concerned with intelligent behavior in artifacts." (Nilsson, 1998)





Four schools of thoughts (Russel & Norvig)

Review: 4. Acting Rationally – Rational Agents

machines to act rationally rather than humanly

- **Rational behavior: doing the "right thing"**
  - The ***right thing***: which is expected to maximize goal achievement, given the available information about the environment
  - Doesn't necessarily involve thinking, e.g., blinking reflex
  - Thinking should be in the service of rational action
  - Entirely dependent on goals!
  - Rational ≠ successful OR Irrational ≠ insane, irrationality is sub-optimal action
- **Our focus here: Rational Agents**
  - A **rational agent** is one that acts so as to achieve the best outcome, or when there is uncertainty, the best expected outcome
  - Systems which make the best possible decisions given goals, evidence, and constraints
  - In the real world, usually lots of uncertainty … and lots of complexity
  - Usually, we're just approximating rationality
- Aristotle (Nicomachean Ethics):

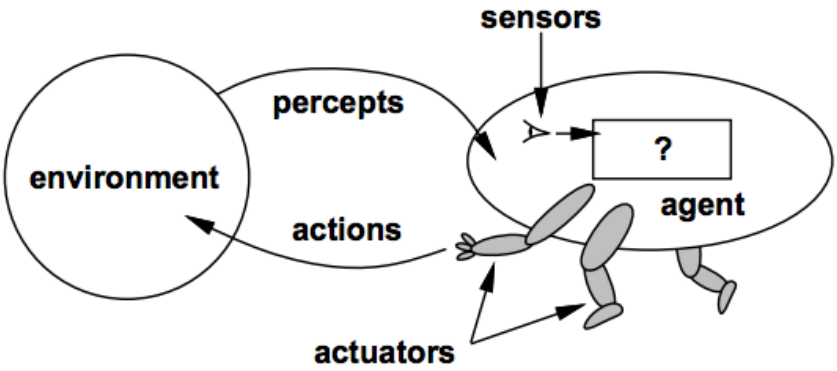
  "Every art and every inquiry, and similarly every action and pursuit, is thought to aim at some good."

Outline

- Agents and Environments
- *Good Behavior:* The Concept of Rationality
- The Nature of Environments
  - PEAS
  - Environment Types
- The Structure of Agents
  - Agent Types
  - How the components of agent programs work?

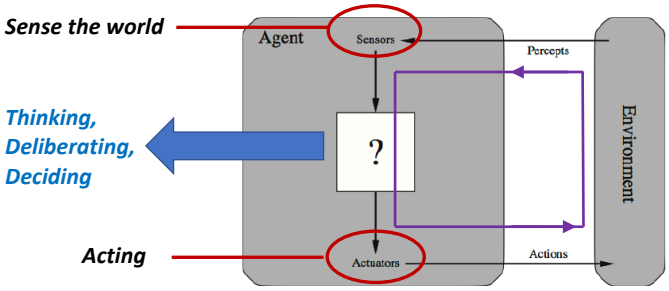
Agents and Environments

- **Agent:** An **agent** is anything that can be viewed as
  - **perceiving** its **environment** through **sensors** and
  - **acting** upon that **environment** through **actuators**



Agents and Environments

- **Agent:** An **agent** is anything that can be viewed as
  - **perceiving** its **environment** through **sensors** and
  - **acting** upon that **environment** through **actuators**



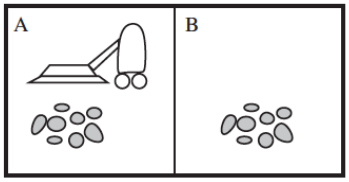
- An agent program runs in cycles of: **(1)perceive**, **(2)think**, and **(3)act**
- **Agent = Architecture + Program**

Agents and Environments

- **Human Agents**
  - Sensors: eyes, ears, and other organs
  - Actuators: hands, legs, mouth, and other body parts
- **Robotic Agents**
  - Sensors: cameras and infrared range finders
  - Actuators: various motors
- **Software Agents**
  - Sensors: keystrokes, file contents, network packages
  - Actuators: displaying on the screen, writing files, sending network packages

- ***Agents are everywhere!***
  - Thermostat
  - Cell phone
  - Vacuum cleaner
  - Robot
  - Alexa Echo
  - Self-driving car
  - Human
  - etc.

Vacuum Cleaner



- Percepts: location and contents, e.g., [A, Dirty]
- Actions: *Left*, *Right*, *Suck*, *NoOp*
- Agent function: mapping from percepts to actions $\boldsymbol{f} : P^* \rightarrow A$

Percept	Action
[A, clean]	*Right*
[A, dirty]	*Suck*
[B, clean]	*Left*
[B, dirty]	*Suck*

What is the right way to fill out the table?

What makes an agent good or bad, intelligent or stupid?

[A, clean], [A, clean]	*Right*
[A, clean], [A, dirty]	*Suck*

Good Behavior

- **Recall:** A **rational agent** is one that **does the right thing**
  - every entry in the table for the agent function is filled out correctly
  - doing the right thing is better than doing the wrong thing …
- … But what does it mean to do the right thing?

  Consider the *consequences* of the agent's behavior

  When an agent is plunked down in an environment, it generates a **sequence of actions** according to the percepts it receives. This sequence of actions causes the environment to go through a **sequence of states**.

  If the sequence is desirable, then the agent has **performed well** → This notion of desirability is captured by a **performance measure** that evaluates any given sequence of **environment states**

Good Behavior

- What are the ***consequences*** of the agent's actions on the ***state*** of its ***environment***?
  - Sequence of agent actions → sequence of environmental states
- How to control agent?
  - Introduce **performance measure**
  - So, rationality is relative to a **performance measure** - ***an objective criterion for success of an agent's behavior***
- Evaluate performance of environment or that of agent?
  - Notice that we said ***environment* states**, not ***agent* states**
  - If we define success in terms of agent's opinion of its own performance, an agent could achieve perfect rationality simply by deluding itself that its performance was perfect

Example: Performance Measure

- Possible performance measures for vacuum world?
  - Amount of dirt cleaned up in a 8-hour shift
  - +1 point for each clean square in time T
  - +1 point for clean square, -1 for each move
  - The time required for clean-up
  - The degree of sounds by the motor of the cleaner
  - Penalty for electricity consumed
- No one fixed performance measure for all tasks and agents

As a general rule, it is better to design performance measures according to what one actually wants in the environment, rather than according to how one thinks the agent should behave

Definition of a Rational Agent

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."*

The Concept of Rationality

- Define rationality relative to a **performance measure**
- Judge rationality based on:
  - The performance measure that defines the criterion of success
  - The agent prior knowledge of the environment
  - The possible actions that the agent can perform
  - The agent's percept sequence to date

Task Environment: PEAS

- When we define a rational agent, we group these properties under **PEAS**, which is actually ***the problem specification for the task environment that the rational agent is meant to solve***
- Task environments are essentially the "problems" to which rational agent that we want to design are the "solutions" (for this task environment)
- PEAS stands for:
  - ✓ **P**erformance
  - ✓ **E**nvironment
  - ✓ **A**ctuators
  - ✓ **S**ensors

In designing an intelligent agent, the first step must always be to specify the task environment as fully as possible

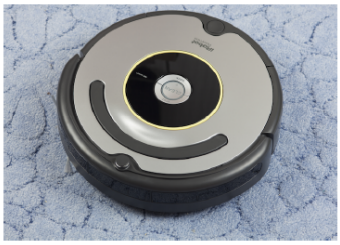
Example: PEAS – self-driving car

What is PEAS for a **self-driving car**?

- **P**erformance: Safety, time, legal drive, comfort
- **E**nvironment: Roads, other cars, pedestrians, road signs
- **A**ctuators: Steering, accelerator, brake, signal, horn
- **S**ensors: Camera, sonar, GPS, Speedometer, odometer, accelerometer, engine sensors, keyboard

Example: PEAS – vacuum cleaner

What is PEAS for a **vacuum cleaner**?



iRobot Roomba Series

- **P**erformance: Cleanness, efficiency (distance traveled to clean), battery life, security
- **E**nvironment: Room, table, wood floor, carpet, different obstacles
- **A**ctuators: Wheels, different brushes, vacuum extractor
- **S**ensors: Camera, dirt detection sensor, cliff sensor, bump sensors, infrared wall sensors

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display of scene categorization	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Figure 2.5 Examples of agent types and their PEAS descriptions.

Properties of Task Environments

- The nature of environment can dramatically impact the complexity of the required agent program as well as the difficulty of achieving goals in it

- Types of Task Environments
  - ✓ Fully Observable vs. Partially Observable
  - ✓ Single-agent vs. Multi-agent
  - ✓ Deterministic vs. Stochastic
  - ✓ Episodic vs. Sequential
  - ✓ Static vs. Dynamic
  - ✓ Discrete vs. Continuous
  - ✓ Known vs. Unknown

Fully Observable vs. Partially Observable

Can the agent observe the complete state of the environment?

- **Fully Observable:** An agent's sensors give it access to the complete state of the environment at all times (the full picture)
  - Practical, because the agent need not maintain any internal state to keep track of the world
- **Partially Observable:** An environment might be partially observable due to noisy and inaccurate sensors, or because parts of the state are simply missing from the sensor data
- *Example*: Vacuum Cleaner (room) vs. Self-driving Car (street)
  Perfect GPS vs. noisy pose estimation

Almost everything in the real world is partially observable

Single-agent vs. Multi-agent

- **Single-agent:** An agent operating by itself in an environment
  - *Example*: an agent solving a crossword puzzle by itself is a single-agent
- **Multi-agent:**
  - *Example*: an agent playing chess is in a two-agent environment
  - **Competitive Multi-agent**: If the agents are maximizing a performance metric that depends on other agents' behavior
    - *Example*: In chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent A's performance measure. Thus, chess is a **competitive** multi-agent environment

Deterministic vs. Stochastic

Is there uncertainty in how the world works?

- **Deterministic:** The next state of the environment is completely determined by the current state, and the action executed by the agent
  - If the environment is deterministic except for the actions of the other agents, then the environment is called **strategic**
  - In principle, an agent need not worry about uncertainty in a fully observable, deterministic environment
  - *Example*: The vacuum cleaner world is deterministic
- **Stochastic:** Opposite of deterministic is non-deterministic, or stochastic, in which uncertainty about outcomes is modeled or quantified in terms of probabilities
  - Actions have probabilistic outcomes - we have this possibility of making this action for the agent
  - If the environment is partially observable, then it could ***appear*** to be stochastic – Strongly related to ***partial observability***
  - *Example:* Basically, a car can't predict the behavior of other cars, or the traffic conditions, moreover, one's tires blow out and one's engine seizes up without warning

Episodic vs. Sequential

- **Episodic:** The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself (in other words, the next episode does not depend on the actions taken in previous ones)
  - *Example:* Many classification tasks are episodic
- **Sequential:** current decision could affect all future decisions
  - *Example:* Chess (short-term actions can have long-term consequences)
- Episodic environments are much simpler than sequential environments because the agent does not need to think ahead

Static vs. Dynamic

- **Static:** The environment is unchanged while an agent is deliberating or thinking
  - 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
  - *Example:* Crossword puzzles
- **Dynamic:** The environment is continuously asking the agent what it wants to do; if it hasn't decided yet, that counts as deciding to do nothing
  - *Example:* Driving car
  - The environment is **semi-dynamic** if the environment itself does not change with the passage of time but the agent's performance score does
    - *Example:* Chess, when played with a clock is semi-dynamic

Discrete vs. Continuous

- The discrete/continuous distinction applies to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent
- **Discrete:** A limited number of distinct, clearly defined percepts and actions
  - E.g., chess is an example of a discrete environment
    - the chess environment has a finite number of distinct states (excluding the clock)
    - Chess also has a discrete set of percepts and actions
  - E.g., while self-driving car evolves in a continuous one, in terms of both the actions and the time
    - Self-driving car is a continuous-state and continuous-time problem: the speed and location of the car and of the other vehicles sweep through a range of continuous values and do so smoothly over time

Known vs. Unknown

- Known vs. unknown distinction refers not to the environment itself, but to the agent's (or designer's) state of knowledge about the "rules of the game / laws of physics" of the environment
  - The designer of the agent may or may not have knowledge about the environment makeup
- In a **known** environment, the outcomes for all actions are given
- If the environment is **unknown**, the agent will have to learn how it works in order to make good decisions
- ***Note***: the distinction between known and unknown environments is not the same as the one between fully and partially observable environments
  - It is quite possible for a *known* environment to be *partially* observable—for example, in solitaire card games, I know the rules but I am still unable to see the cards that have not yet been turned over
  - Conversely, an *unknown* environment can be *fully* observable—in a new video game, the screen may show the entire game state but I still don't know what the buttons do until I try them

Example: Environment Types

Environment	Observable	Agents	Deterministic	Static	Discrete
8-puzzle	Fully	Single	Deterministic	Static	Discrete
Chess	Fully	Multi	Deterministic	(Semi)Static	Discrete
Pocker	Partially	Multi	Stochastic	Static	Discrete
Backgammon	Fully	Multi	Stochastic	Static	Discrete
Car	Partially	Multi	Stochastic	Dynamic	Continuous
Roomba	Partially	Single	Stochastic	Dynamic	Continuous

The environment type largely determines the agent design

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

Agent Functions and Agent Programs

- An agent is completely specified by the **agent function**
  - The **agent function** maps from ***percept sequences*** to ***actions***
  - So, describes agent's behavior

$$f : P^* \rightarrow A$$

abstract mathematical description

  - **Percept:** refers to the agent's perceptual inputs at any given instant
    - An agent's **percept sequence** is the complete history of everything the agent has ever perceived – or, **percept history**
    - In general, *an agent's choice of action at any given instant can depend on the entire percept sequence observed to date, but not on anything it hasn't perceived*

Agent Functions and Agent Programs

- An agent is completely specified by the **agent function**
  - The **agent function** maps from ***percept sequences*** to ***actions***

$$f : P^* \rightarrow A$$

abstract mathematical description

- ***Job of AI***: Design an **agent program** which implements the **agent function**
  - The **agent program** runs on the physical **architecture** to produce the **agent function**, f

concrete implementation, running within some physical system

  - Describes how the agent works
  - Agent function: Mathematical abstraction
  - Agent program: Concrete implementation

Agent = Architecture + Program

- Program must suit architecture
- ***Aim:*** Find a way to implement the rational agent function concisely

Table Lookup Agent

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*
…	…

- Drawbacks:
  - Huge table
  - Take a long time to build the table
  - No autonomy
  - Even with learning, need a long time to learn the table entries

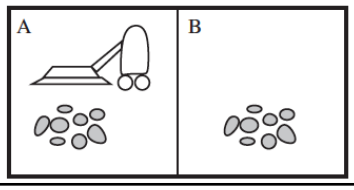


Table Lookup Agent

- Why a chess table agent is a bad idea:
  - Not enough space in universe to store table
  - Would take longer than your life to build
  - Too massive to be learned through observation
  - Where do you start?

- Reminder: The fact that a program can find a solution in principle does not mean that the program contains any of the mechanisms needed to find it in practice

Agent Types

- There are 4 types of agents in general, varying in ***the level of intelligence*** or the ***complexity of the tasks they are able to perform***

- Four basic types in order of increasing generality
  1. **Simple reflex** agents
  2. **Model-based** reflex agents
  3. **Goal-based** agents
  4. **Utility-based** agents

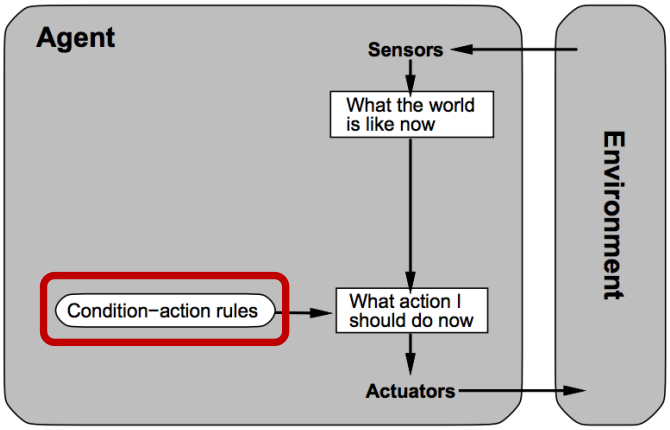
- All of which can be generalized into **learning agents** that can improve their performance and generate better actions

1. Simple Reflex Agents

- Forget the percept history!
- Can't we just act on the current state?
  - Select actions based on the current percept ignoring the rest of the percept history

1. Simple Reflex Agents

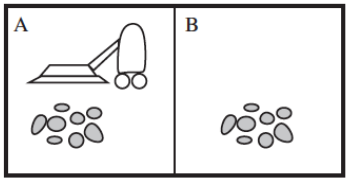
- Simple reflex agents select an action based on **the current state only**, ignoring the percept history
- Simple, but turn out to be of very limited intelligence
- Can only work if the environment is **fully observable**, that is **the correct action is based on what is perceived currently** – **based on the current percept only**



Condition-action rule

if *car-in-front-is-braking* **then** *initiate-braking*

Example: Vacuum (Reflex) Agent



What if the vacuum agent is deprived from its location sensor?

- Let's write the algorithm for the Vacuum cleaner …
- Percepts: location and contents (location sensor, dirt sensor)
- Actions: Left, Right, Suck, NoOp

Percept	Action
[A, clean]	Right
[A, dirty]	Suck
[B, clean]	Left
[B, dirty]	Suck

```
function VacuumAgent (location, state)
return action

    if status = Dirty then return Suck
    else if location = A then return Right
            else return Left
```

1. Simple Reflex Agents

- Pros:
  - Simple
- Cons:
  - For this to produce rational agents the environment must be _____?
  - Often yield infinite loops
    - Randomization can help but only so much

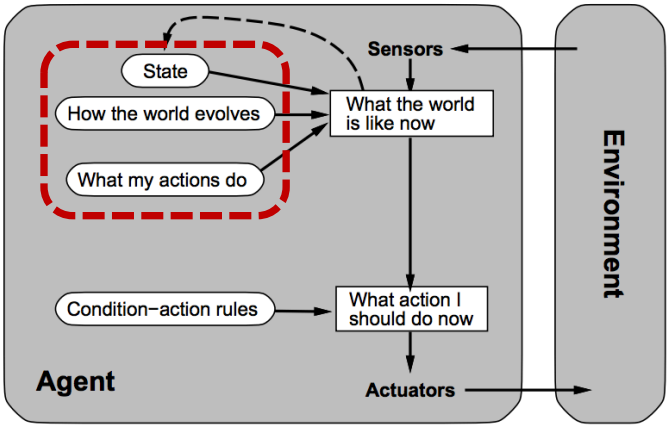
2. Model-based Reflex Agents

- That didn't turn out so well
- Let's try **keeping track of** what we can't currently observe
- **Internal state** to track
  - Is that car braking?
  - Is there a car in my blind spot??
  - Where did I park???

2. Model-based Reflex Agents

"Agents that keep track of the world"

- 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"**
- The agent should **maintain** some sort of **internal state** that depends on the percept history (*best guess*) and reflects at least some of the unobserved aspects of the current state



2. Model-based Reflex Agents

- Updating the internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program
- In other words, the environment/world is modeled based on
  1) Information about ***how the world evolves independently of the agent***
  2) Information about ***how the agent's own actions affects the world***
- This knowledge about "*how the world works*" is called a **model of the world**

2. Model-based Reflex Agents

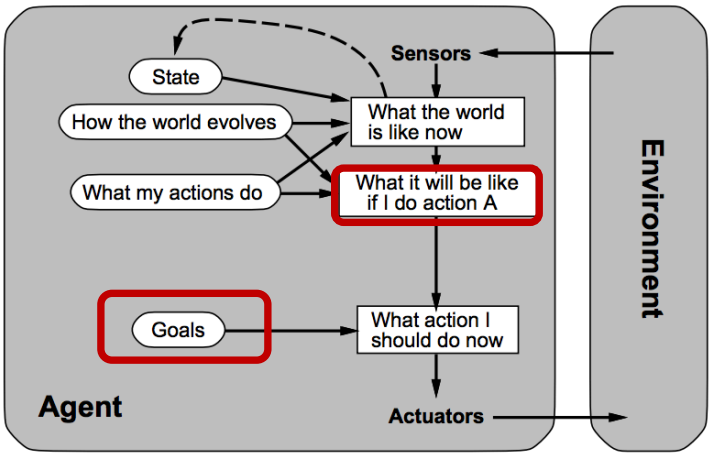
- Getting better!
- But sometimes model is insufficient:
  - Impossible to model hidden world accurately and reliably
  - What to do when environment doesn't imply action?
    - i.e., fork in the road

3. Goal-based Agents

- An improvement over model-based agents, and used in cases where knowing the current state of the environment is not enough
  - Additional to Model-based agents, these agents need some **goal information**
- Agents combine the provided goal information with the environment model to choose the actions which achieve that goal
  - Goals decide the actions of the agent
- Agents will account for the future
  - Consider the future with "What will happen if I do A?"

3. Goal-based Agents

- Knowing about the current state of the environment is not always enough to decide what to do (e.g., decision at a road junction)
- The agent needs some sort of **goal** information that **describes situations that are desirable**
- The agent program can combine this with information about the results of possible actions in order to **choose actions that achieve the goal**
- Usually requires search and planning



3. Goal-based Agents

- The goal-based decision making is different from the condition-action rules in that **the future is considered**
  - What will happen if I do that?
  - Will that make me satisfied?
- Goal-based agents vs reflex-based agents
  - The reflex agent brakes when it sees brake lights
  - A goal-based agent could reason that if the car in front has its brake lights on, it will slow down. Given the way the world usually evolves, the only action that will achieve the goal of not hitting other cars is to brake.

Goal-based Agents vs Reflex-based Agents

- Although the goal-based agent appears less efficient, it is more flexible because **"the knowledge that supports its decisions is represented explicitly and can be modified"**
  - On the other hand, for the reflex-agent, we would have to rewrite many condition-action rules
- The goal-based agent's behavior can easily be changed to go to a different location, but the reflex agent's rules work only for a single destination
  - The reflex agent's rules must be changed for a new situation
- Sometimes goal-based selection is not trivial when the agent must find a long sequence of actions to achieve the goal
  - Search and planning are devoted to this

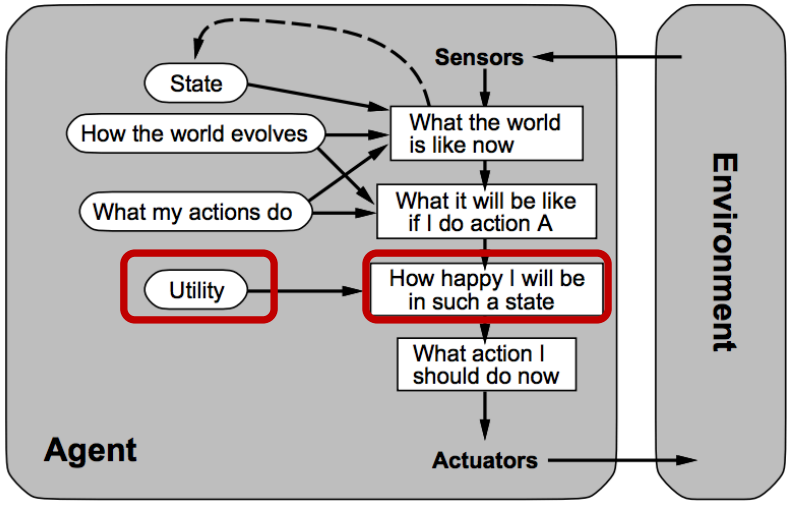
4. Utility-based Agents

- Sometimes achieving the desired goal is not enough to generate high-quality behavior in most environments
  - Goals just roughly provide a binary distinction between happy and unhappy states
  - Did you meet it or not?
- We may look for quicker, safer, cheaper trip to reach a destination
- A more general performance measure should allow a comparison of different world states according to exactly **how happy** they would make the agent if they could be achieved
  - Agent **happiness** should be taken into consideration → we call it **utility**
  - A **utility function** is the agent's performance measure – maps a state onto a real number which describes the associated degree of happiness
- Because of the uncertainty in the world, a utility agent chooses the action that maximizes the expected (average) utility

4. Utility-based Agents

Needed when:

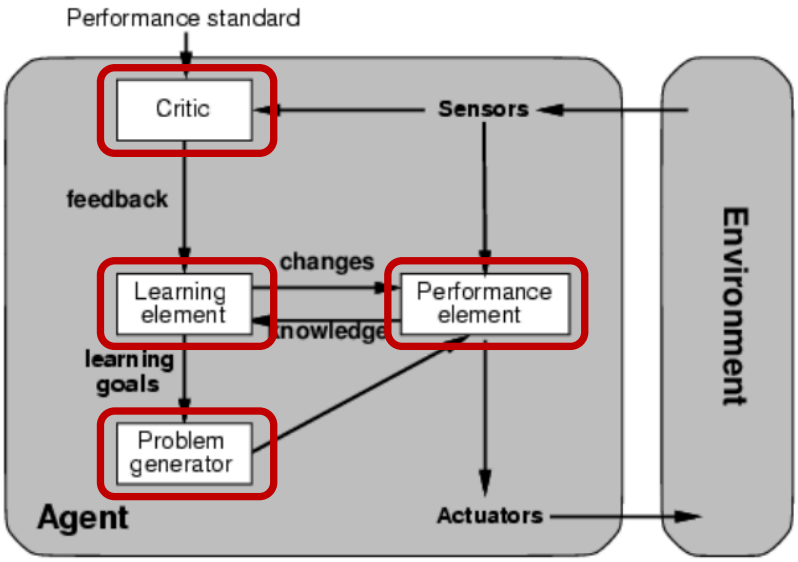
- Goals conflict
  - tradeoff between conflicting objectives (speed vs. safety)
- Uncertain goals
  - likelihood of success



Learning Agents

- That's nice – but how do we program these thing?
- By hand?
  - Manual programming is too slow and tiring
  - "Some more expeditious method seem desirable", Alan Turing, 1950
- Let the agent figure things out by itself!
- A learning agent
  - This generalized all the previous agents we have seen

Learning Agents

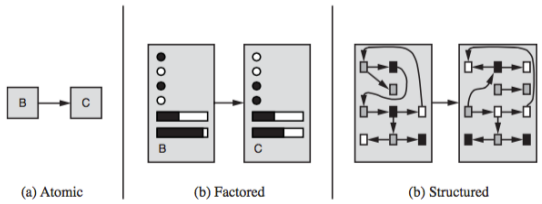


Learning Agents

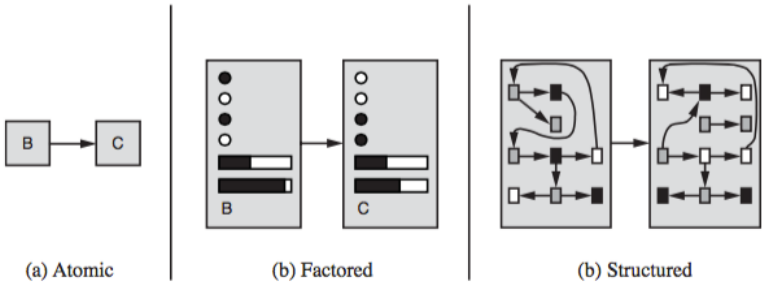
- A learning agent has four conceptual components
  - **Learning Element**: responsible for making improvements to performance element
  - **Performance Element**: responsible for selecting external actions
    - It is what we considered as agent so far
  - **Critic**: How well is the agent doing w.r.t. a fixed performance standard
    - Learning element uses feedback from the critic on how the agent is doing and determines how the performance element should be modified to do better in the future
  - **Problem Generator**: suggests innovative actions
    - Responsible for suggesting actions that will lead to new and informative experiences
- Learning is a process of modification of each component to bring them into closer agreement with the available feedback information

How the components of agent programs work?

- We have described agent programs (in very high-level terms) as consisting of various components, whose function is to answer questions such as:
  - "What is the world like now?"
  - "What action should I do now?"
  - "What my actions do?"
- The next question for a student of AI is
  - "How on earth do these components work?"
- Roughly speaking, we can place the representations along an axis of increasing complexity and expressive power — **atomic**, **factored**, and **structured**



How the components of agent programs work?



Three ways to represent states and the transitions between them

a) **Atomic Representation:** a state (such as B or C) is a black box with no internal structure

b) **Factored Representation:** a state consists of a vector of attribute values; values can be Boolean, real-valued, or one of a fixed set of symbols

c) **Structured Representation:** a state includes objects, each of which may have attributes of its own as well as relationships to other objects

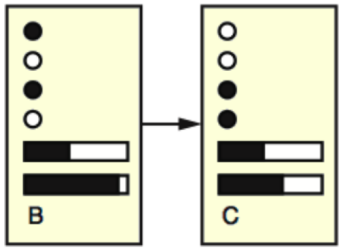
Atomic Representation

- Each state of the world is indivisible — is stored as a ***black-box***, i.e. *without any internal structure*
- ***Example:*** finding a driving route, each state is a city
  - Consider the problem of finding a driving route from one end of a country to the other via some sequence of cities
  - For the purposes of solving this problem, reduce the state of world to just the name of the city we are in—a single atom of knowledge; a "black box" whose only discernible property is that of being identical to or different from another black box
- Atomic representation works for **model-based** and **goal-based** agents
- Used in various **AI Algorithms**
  - Search
  - Game-playing
  - Hidden Markov models
  - Markov decision processes

all work with atomic representations
— or, at least, they treat representations
as if they were atomic

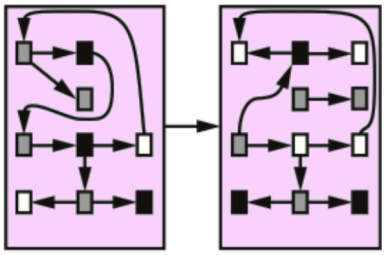
Factored Representation

- In factored representation, the state is no longer a black box. Now, ***each state has some attribute-value pairs, also known as variables that can contain a value***
  - A **factored representation** splits up each state into a fixed set of **variables** or **attributes**, each of which can have a **value**
  - While two different atomic states have nothing in common—they are just different black boxes—two different factored states can share some attributes (such as being at some particular GPS location) and not others (such as having lots of gas or having no gas); this makes it much easier to work out how to turn one state into another
- *Example:* While finding a route, you have a GPS location and amount of gas in the tank – adds a constraint to the problem
- With factored representations, we can also represent ***uncertainty***
  - Ignorance about the amount of gas in the tank can be represented by leaving that attribute blank
- Factored representation works for **goal-based** agents
- Many important areas of AI are based on factored representation
  - Constraint Satisfaction algorithms
  - Propositional Logic
  - Planning
  - Bayesian networks
  - Machine Learning algorithms



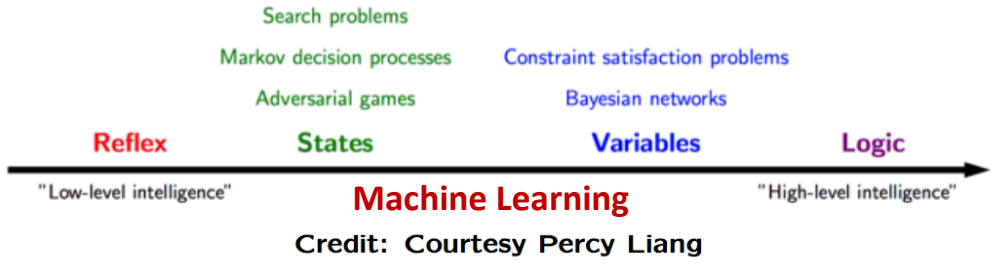
Structured Representation

- For many purposes, we need to understand the world as having ***things*** in it that are ***related*** to each other, not just variables with values
  - For example, we might notice that a large truck ahead of us is reversing into the driveway of a dairy farm but a cow has got loose and is blocking the truck's path
  - A factored representation is unlikely to be pre-equipped with the attribute ***TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow*** with value ***true*** or ***false***
  - Instead, we would need a **structured representation**, in which **objects** such as cows and trucks and their various **relationships** can be described explicitly
- In structured representation, ***we have relationships between the variables/factored states - This induces logic in the AI algorithms***
  - For example, in natural language processing, the states are whether the statement contains a reference to a person and whether the adjective in that statement represents that person. The relation in these states will decide, whether the statement was a sarcastic one
- High level AI, used in algorithms like
  - First-order logic
  - First-order probability models
  - Knowledge-based learning
  - Natural language understanding



Representation of States: Atomic, Factored, Structured

- The axis along which atomic, factored, and structured representations lie is the axis of increasing **expressiveness** power
- A more expressive representation can capture, at least as concisely, everything a less expressive one can capture, plus some more
- Often, the more expressive language is *much* more concise



Credit: Courtesy Percy Liang

Summary

- The concept of intelligent agent is central in AI
- AI aims to design intelligent agents that are useful, reactive, autonomous, and even social and pro-active
- An **agent** **perceives** its environment through **percepts**, and **acts** through **actuators**
- A **performance measure** evaluates the behavior of the agent
- An agent that acts to maximize its expected **performance measure** given the **percept** sequence and candidate **actions,** is called a **rational agent**
- We can define different criteria for rational agent environment, which is called **PEAS** -- a task environment specification that includes **P**erformance measure, **E**nvironment, **A**ctuators, and **S**ensors
- Task environments vary along several dimensions
- An agent is a set of **architecture**, or **hardware**, plus a **program** or **system**, where both must be compatible to work together

Agent = Architecture + Program

Summary

- There exists four types of agents
  - Simple reflex agents – that are the simplest ones
  - Model-based agents – that try to model the world by creating the model internally
  - Goal-based agents – that wants to model the world, but still have a goal to achieve
  - Utility-based agents – that try to increase their happiness
- All agents can improve their performance through ***learning***
  - ***which is a central piece today in AI that aims to make the agent aware of different situations, and learn from its experience***
- This was a high-level representation of the different agents' programs
- Lastly, we see that state representation can be either **atomic**, **factored**, or **structured**, which goes by increasing order of expressiveness

Next Week: Search Agents

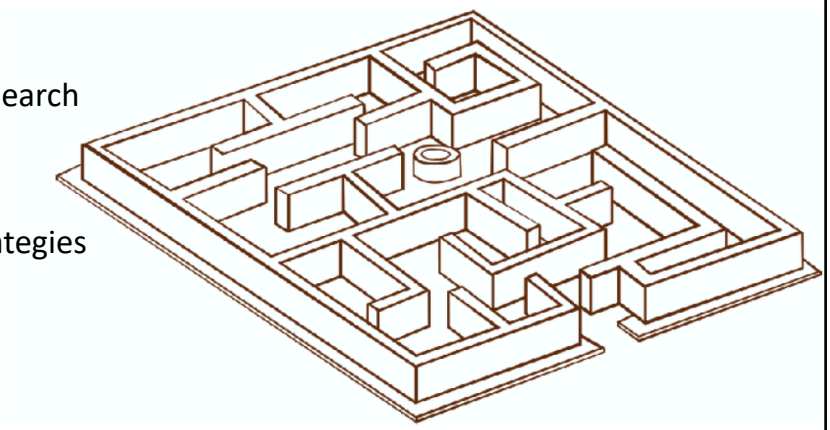
- Agents that work towards a **goal**
- Agents consider the impact of **actions** on future **states**
- Agent's job is to identify the action or series of actions that lead to the goal
- Paths come with different costs and depths
- Two kinds of search
  - **Uninformed Search** (use no domain knowledge): BFS, DFS, UCS, etc.
  - **Informed Search** (use heuristic to reach the goal faster): Greedy search, A*, etc.

BIM309 Artificial Intelligence

Week 4 – Search Agents & Uninformed Search

Outline

- Search Agents
- Problem Solving using Search
- Problem Formulation
  - Toy Problems
  - Real World Problems
- Uninformed Search Strategies
  - BFS
  - DFS
  - DLS
  - IDS
  - UCS
- Example



Introduction

- **Reflex agents:** use a mapping from states to actions
  - Have a loop-up table or condition-action rules, and know what to do according to the states
  - Therefore, they cannot operate well in environments where the mapping is too large to store or takes too much to learn
- **Goal-based agents:** can succeed by considering future actions and desirability of their outcomes
  - *Operate differently!!* Instead of having this predefined list of states to actions, the goal-based agents would have to find a solution for the problem

- Agents that don't have any intelligence
- Just have a look up table/condition-action rules
- Look what to do according to kind of dictionary/look up table/condition rules
- So, considered as one of the lowest level of intelligence

Goal-based Agents

- What are **Goal-based agents?**
  - Agents that work towards a **goal**
  - Agents consider the impact of **actions** on future **states**
    - Have a concept of **the future**
    - Capable of comparing desirability of states relative to a **goal**
  - Agent's job is to identify the action or series of actions that lead to the goal
  - In AI, this is formalized as a **search** through possible set of **solutions** (actions or series of actions)

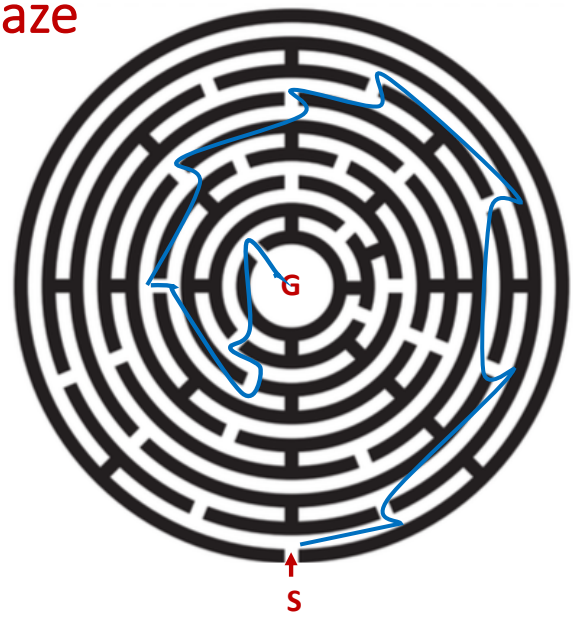
    - In general, the search space of the possible solutions is large enough to prevent from putting this information into a look-up table → reflex agents won't be effective
    - We need a goal-based agent that knows how to search through the possibilities, and get the best solutions to the goal

Problem Solving Agents (Search Agents)

- "**Problem-solving agent**" is one kind of **goal-based agent**
  - Use **atomic representation** – states are atomic
    - Indivisible black boxes
    - As opposed to factored and structured representation
  - Problem solving agent is a goal-based agent that decides what to do by finding **sequences of actions** that lead to the **goal** (desirable state)

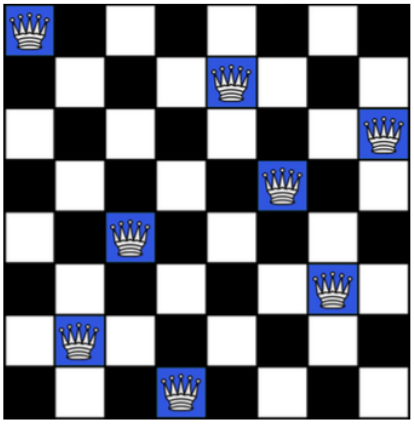
*Note: Goal-based agents that use more advanced **factored** or **structured** representations are called "**planning agents**" (Chapter 7, 10)*

Example: Maze



EXPLORE!

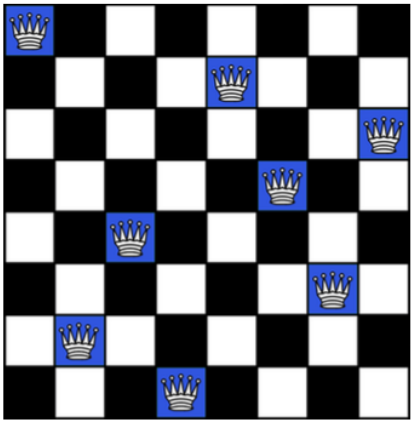
Example: The 8-queen problem



Goal State

The 8-queen problem: on a chess board, place 8 queens so that no queen is attacking any other horizontally, vertically or diagonally

Example: The 8-queen problem



Idea of a goal-based agent is to find one successful possibility to put the queens on the chess board

Number of possible sequences to investigate (size of the search space):

$$64 * 63 * 62 * 61 * 60 * 59 * 58 * 57 = \mathbf{1.8 \times 10^{14}}$$

Problem Solving as Search

1. **Define the problem through**
   (a) **Goal formulation:** what is the goal to achieve
   (b) **Problem formulation:** the process of deciding what sorts of actions and states to consider, given a goal
2. **Solving the problem as a 2-stage process**
   (a) **Search:** "mental" or "offline" exploration of several possibilities
   (b) **Execute** the solution found

"formulate, search, execute"

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: 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)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

"formulate, search, execute"

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Environment Assumptions

In general, ***an agent with several immediate options of unknown value can decide what to do by first examining*** future ***actions that eventually lead to states of known value***

- What does it mean "**examining future actions**"?
- Let's look at the properties of the environment

We assume that …

- The environment is **observable**, so the agent always knows the current state
- The environment is **discrete**, so at any given state there are only finitely many actions to choose from
- The environment is **known**, so the agent knows which states are reached by each action
- The environment is **deterministic**, so each action has exactly one outcome

→ ***Under these assumptions, the solution to any problem is a fixed sequence of actions***

Problem Formulation

A **problem** can be defined formally by

- **Initial state:** The state in which the agent starts from
- **States:** All states reachable from the initial state by any sequence of actions **(State space)**
- **Actions:** Possible actions available to the agent
  - At a state ***s***, ***Actions(s)*** returns the set of actions that can be executed in state ***s*** **(Action space)**
- **Transition model:** A description of what each action does
  - Specified by a function ***Results(s, a)*** that returns the state results from doing action ***a*** in state ***s***
- **Goal test:** Determines if a given state is a goal state
- **Path cost:** function that assigns a numeric cost to a path w.r.t. performance measure

Problem Formulation Example: Vacuum World

- **States:** The state is determined by both **agent location** and **dirt locations**
  - The agent is in one of two locations, each of which might or might not contain dirt
    - $2 \times 2^2 = 8$ possible states ($n \times 2^n$)
- **Initial state**: Any state can be designated as the initial state
- **Actions**: Each state has just three actions - ***Left***, ***Right***, and ***Suck***
- **Transition model**: The actions have their expected effects, except that moving *Left* in the leftmost square, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect
- **Goal test**: Checks whether all the squares are clean
- **Path cost**: Each step costs 1, so the path cost is the number of steps in the path

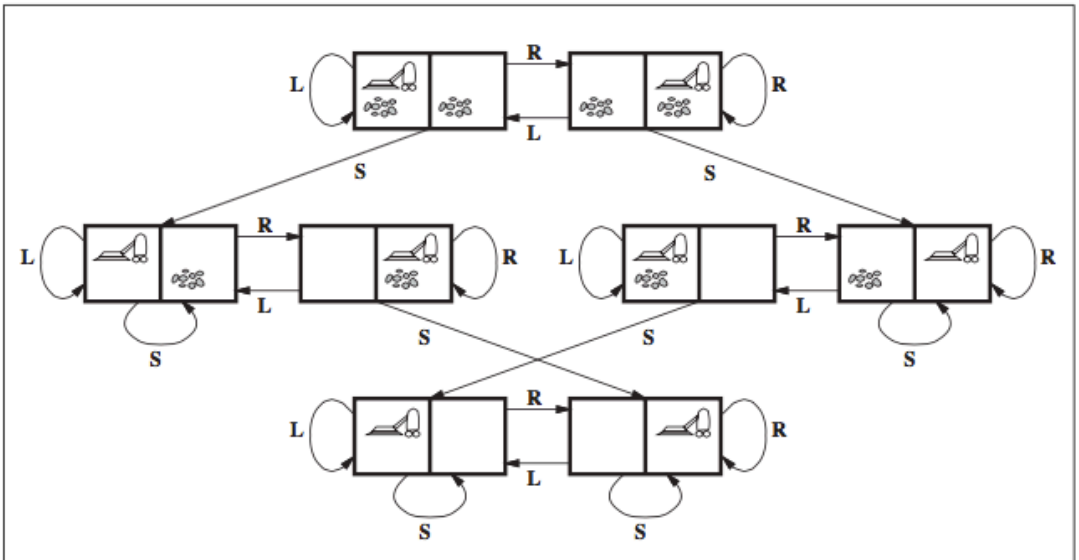
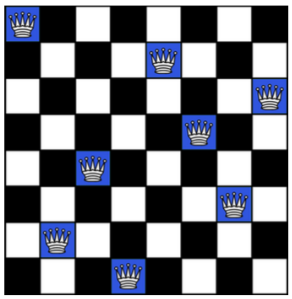


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

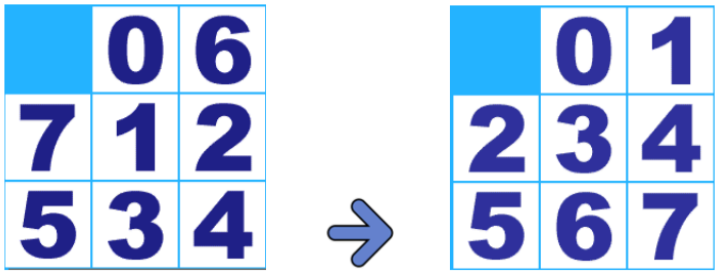
Problem Formulation Example: The 8-queen problem



In this game, we start with an empty board. The players take turns into putting the queen on the board. The goal is to have a board with eight queens on the board where no one is attacking another one diagonally, vertically, or horizontally.

- **States:** all arrangements of 0 to 8 queens on the board
- **Initial state:** No queen on the board (empty board)
- **Actions**: Add a queen to any empty square
- **Transition model:** updated board
  - Returns the board with a queen added to the specified square
- **Goal test:** 8 queens are on the board, none attacked

Problem Formulation Example: 8-puzzle

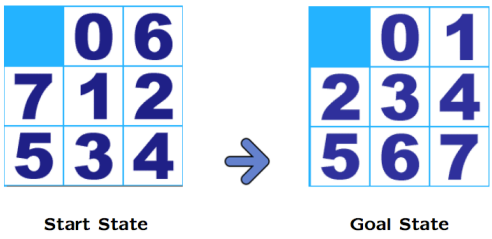


Start State Goal State

The **8-puzzle**

- Consists of a 3×3 board with 8 numbered tiles (moveable) and a blank square (“hole”)
- The blank square can be changed with a tile adjacent to it
- The object is to reach a specified goal state (such as the one shown on the right)

Problem Formulation Example: 8-puzzle



- **States:** Location of each of the 8 tiles and the blank in one of the 3x3 grid
- **Initial state:** Any state
- **Actions:** Movements of the blank space ***Left***, ***Right***, ***Up*** or ***Down***
- **Transition model:** Given a state and an action, returns the resulting state
  - For example, if we apply ***Down*** to the start state, the resulting state has the 7 and the blank switched
- **Goal test:** Checks whether the state matches the goal state? *(Note that other goal configurations are also possible)*
- **Path cost:** total moves, each move costs 1

Problem Formulation Example: Route Finding

- Our agent is on a holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest
- Find a sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest to drive

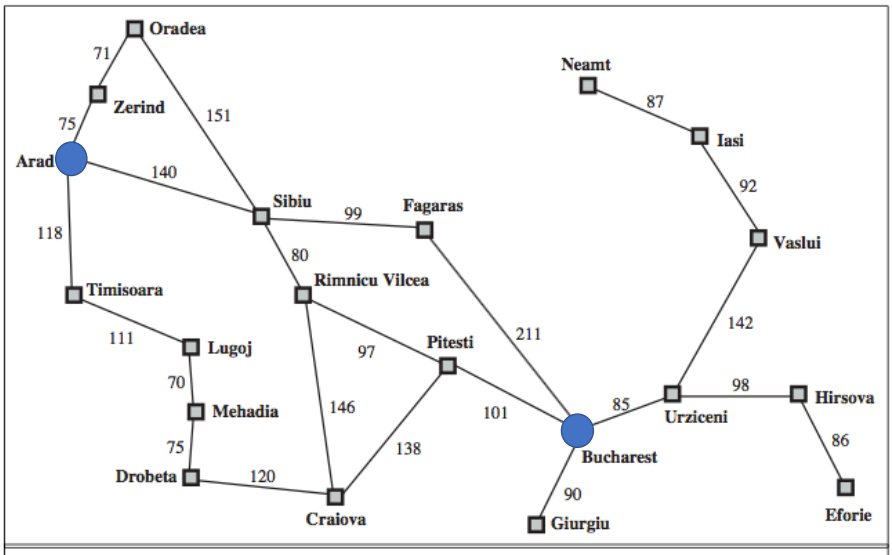


Figure 3.2 A simplified road map of part of Romania.

Problem Formulation Example: Romania

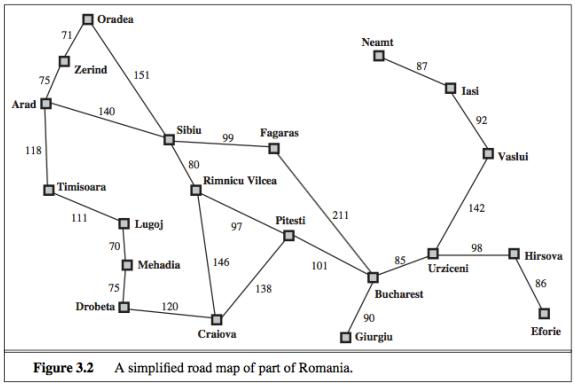
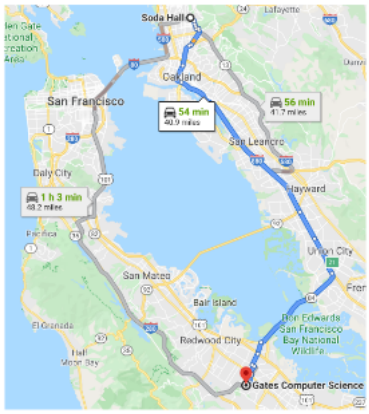


Figure 3.2 A simplified road map of part of Romania.

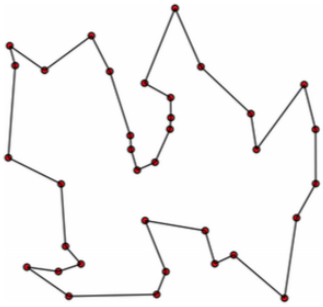
- **States:** Various cities
  In(city) where city ∈ {Sibiu, Fagaras, Iasi,…}
- **Initial state:** *In(Arad)*
- **Actions:** From the state *In(Arad)*, the applicable actions are {Go(Sibiu), Go(Timisoara), Go(Zerind)}
- **Transition model:** *Results(In(Arad), Go(Zerind)) = In(Zerind)*
- **Goal test:** Check if *In(Bucharest)*
- **Path cost:** Time to go Bucharest (so path length in kilometers)

Real-World Problems

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



Real-World Problems



- **Travelling salesperson problem (TSP) :**

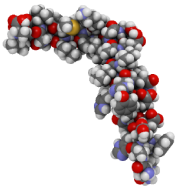
  Find the shortest tour to visit each city exactly once

- **VLSI layout design:** positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray and maximize manufacturing yield

  → ***Aim:*** put circuit components on a chip so as they don't overlap and leave space to wiring, which is a complex problem

  → ***Goal:*** optimize the area and optimize the position of the components with respect to each other

Real-World Problems

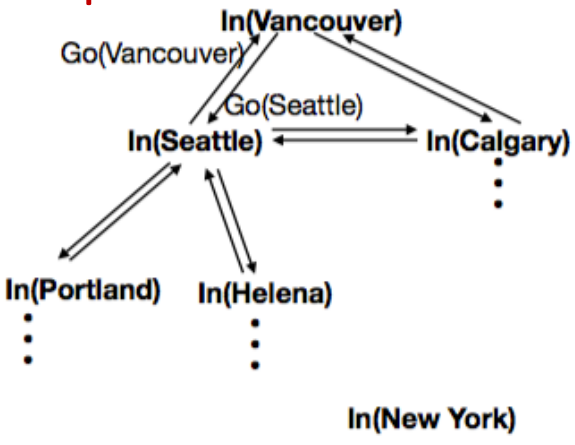


- **Robot Navigation:** Special case of route finding for robots with no specific routes or connections
  - The robot navigates in 2D or 3D space, where the state space and action space are potentially infinite
- **Automatic assembly sequencing**: Find an order in which to assemble parts of an object which is in general a difficult and expensive geometric search
- **Protein design**: Find a sequence of amino acids that will fold into a 3D protein with the right properties to cure some disease

Searching for Solutions

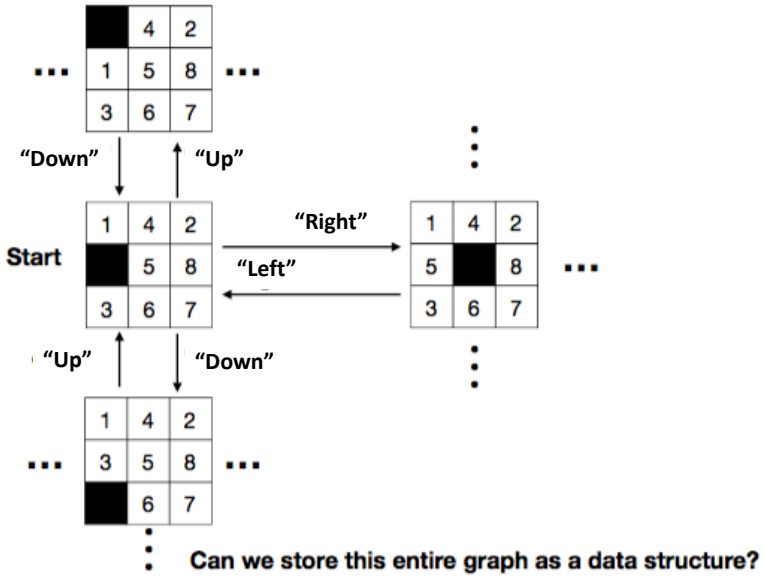
- The initial state, actions, and transition model implicitly define the **state space** of the problem
- Conceptually, the state space forms a directed graph in which
  - the vertices are states, and
  - the edges are actions
- The solution to the search problem is a path through the state space that starts in an initial state and ends in a state that passes the goal test

State Space



For route planning on a map, the state space is identical to the map itself

8-Puzzle State Space



State Space vs. Search Space

- **State space:** a ***physical*** configuration
  - Because state spaces can become extremely large (or infinite!), storing the entire state space as a graph is rarely a good idea
  - In many cases, we may not even have to explore some of the possible states in order to find a solution
- **Search space:** an ***abstract*** configuration represented by a search tree or graph of possible solutions
  - Search algorithms construct a **search tree** one state at a time

Search Tree

In(Vancouver)

In each step of the search algorithm, one **leaf node** in the search tree is expanded

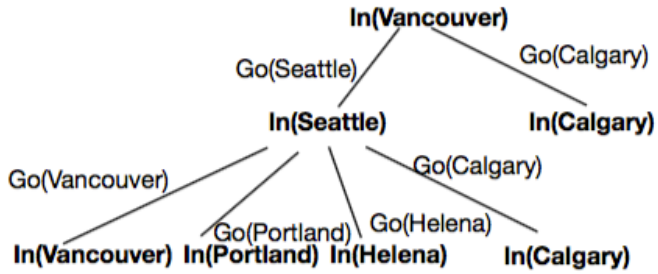
Note: Different search algorithms construct the search tree in different ways

Search Tree

In each step of the search algorithm, one **leaf node** in the search tree is expanded

Note: Different search algorithms construct the search tree in different ways

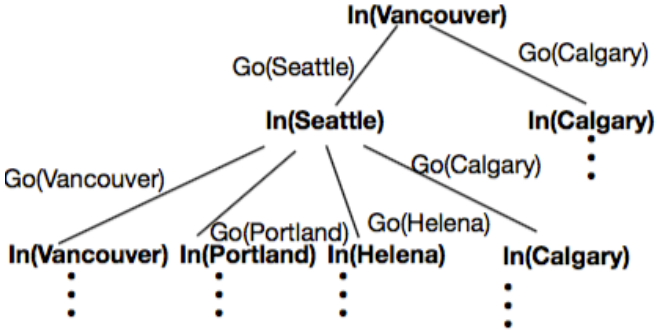
Search Tree



In each step of the search algorithm, one **leaf node** in the search tree is expanded

Note: Different search algorithms construct the search tree in different ways

Search Tree



In each step of the search algorithm, one **leaf node** in the search tree is expanded

Note: Different search algorithms construct the search tree in different ways

Search Tree

- **State space:** a ***physical*** configuration
- **Search space:** an ***abstract*** configuration represented by a search tree or graph of possible solutions
- **Search tree: models the sequence of actions**
  - **Root:** initial state
  - **Branches:** actions
  - **Nodes:** results from actions
    - *A node has:* parent, children, depth, path cost, associated state in the state space
- **Expand:** A function that given a node, creates all children nodes

Implementation: States vs. Nodes

- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree includes **state**, **parent node**, **action**, **path cost** *g(x)*, **depth**

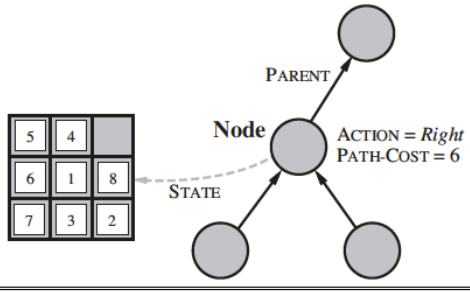


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Search Space Regions

- The search space is divided into three regions:
  1. **Explored** (a.k.a. Closed List, Visited Set)
  2. **Frontier** (a.k.a. Open List, the Fringe)
  3. **Unexplored**

- ***The essence of search*** is moving nodes from regions (3) to (2) to (1), and ***the essence of search strategy*** is deciding the order of such moves

Tree Search

```
function TREE-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    initialize frontier with initialState

    while not frontier.isEmpty():
        state = frontier.remove()

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            frontier.add(neighbor)

    return FAILURE
```

Tree Search

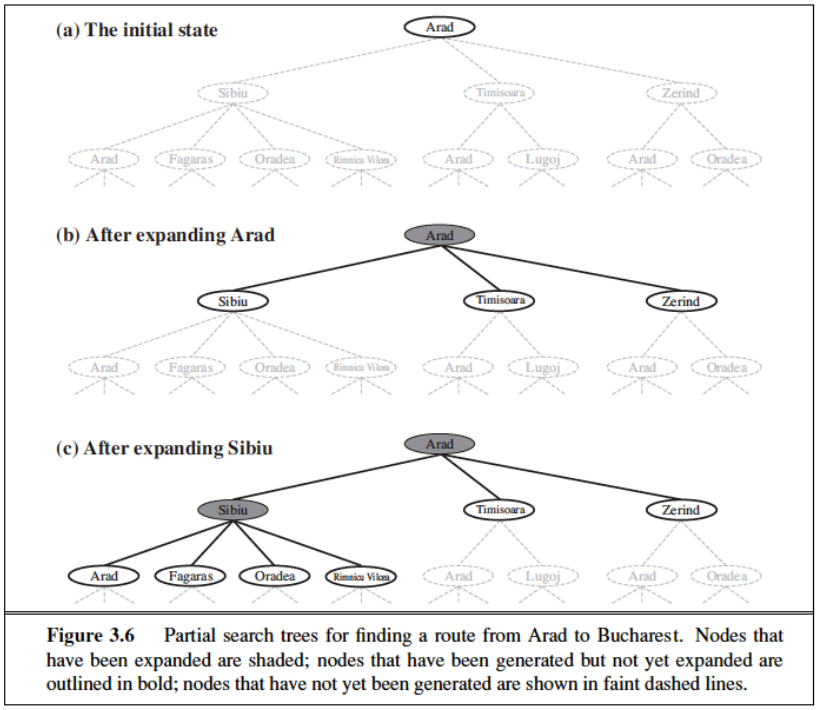
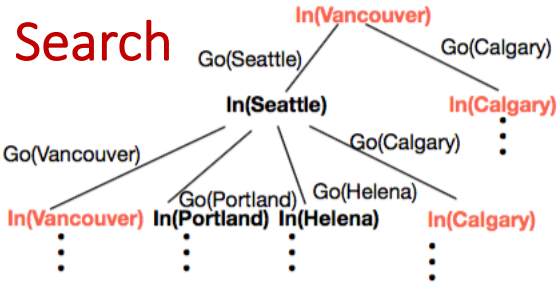


Figure 3.6 Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded; nodes that have been generated but not yet expanded are outlined in bold; nodes that have not yet been generated are shown in faint dashed lines.

States in search tree may repeat themselves !!

Loops in Tree Search



- The search tree may contain duplicate states
  - Loopy paths
    – StateA → StateB → StateA
  - Redundant Paths
    – StateA → StateZ
    – StateA → StateB → StateC → StateD → … → StateZ
- We would like to expand these states only once

How to handle repeated states?

- **Solution:** ***Graph Search***
  - Turn tree search into graph search by tracking redundant paths via an explored list
    - Starts out empty
    - Add node after goal test
    - Only expand node if not explored

Tree Search

```
function TREE-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    initialize frontier with initialState

    while not frontier.isEmpty():
        state = frontier.remove()

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            frontier.add(neighbor)

    return FAILURE
```

Graph Search

```
function GRAPH-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    initialize frontier with initialState
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.remove()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.add(neighbor)

    return FAILURE
```

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    initialize the explored set to be empty
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        add the node to the explored set
        expand the chosen node, adding the resulting nodes to the frontier
            only if not in the frontier or explored set
```

Figure 3.7 An informal description of the general tree-search and graph-search algorithms. The parts of GRAPH-SEARCH marked in bold italic are the additions needed to handle repeated states.

Search Strategies

- A **strategy** is defining the way to pick the **order of node expansion**
  - How the frontier is maintained
  - The way we pick those nodes is called an order of node expansion
  - Search algorithms vary primarily according to how they choose which state to expand next
- Strategies are evaluated along the following dimensions
  - ✓ **Completeness**
    - Does the algorithm guaranteed to find a solution if there is one?
  - ✓ **Time complexity**
    - How long does it take to find a solution? – Number of nodes generated/expanded
  - ✓ **Space complexity**
    - How much memory is needed to perform the search? – Max. number of nodes in memory
  - ✓ **Optimality**
    - Does it always find a least-cost (optimal) solution?

Search Strategies

- Time and space complexity are measured in terms of:
  - ✓ ***b:*** maximum **branching factor** of the search tree (possible actions per state)
  - ✓ ***d:*** **depth** of the solution
  - ✓ ***m:*** maximum depth of the state space (may be infinite)

- Two kinds of search strategies
  - Uninformed Search (or Blind Search)
  - Informed Search (or Heuristic Search)

Uninformed Search

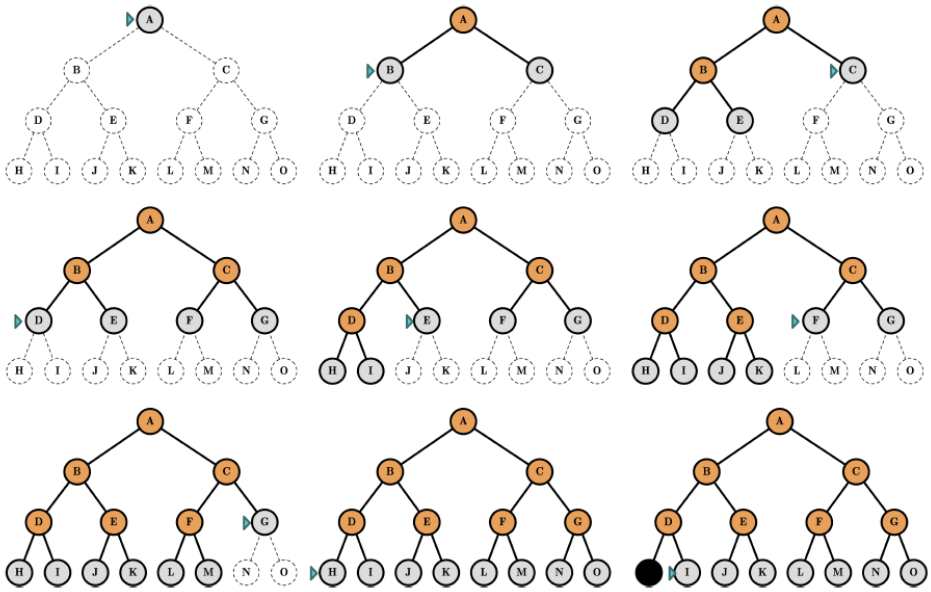
Use no domain knowledge!

Strategies:

1. **Breadth-first search (BFS):** Expand shallowest node
2. **Depth-first search (DFS):** Expand deepest node
3. **Depth-limited search (DLS):** Depth first with depth limit
4. **Iterative-deepening search (IDS):** DLS with increasing limit
5. **Uniform-cost search (UCS):** Expand least cost node

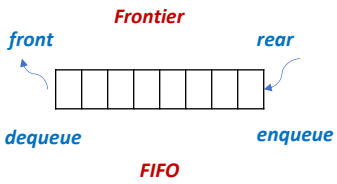
1. Breadth-first Search (BFS)

BFS: Expand the **shallowest** first



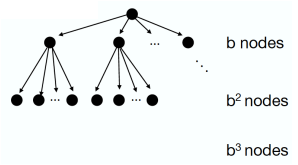
1. Breadth-first Search (BFS)

```
function BREADTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Queue.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.dequeue()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.enqueue(neighbor)

    return FAILURE
```



1. Breadth-first Search (BFS)

Branching factor **b** and solution depth **d**
Nodes are explored level-by-level



BFS criteria?

- **Complete** Yes (if b is finite)
- **Time** $1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$
- **Space** $O(b^d)$
  - *Note:* Need to store all visited nodes in memory
- **Optimal** Yes (if cost = 1 per step)
- **implementation:** fringe: FIFO (Queue)

Question: *If time and space complexities are exponential, why use BFS?*

1. Breadth-first Search (BFS)

How bad is BFS?

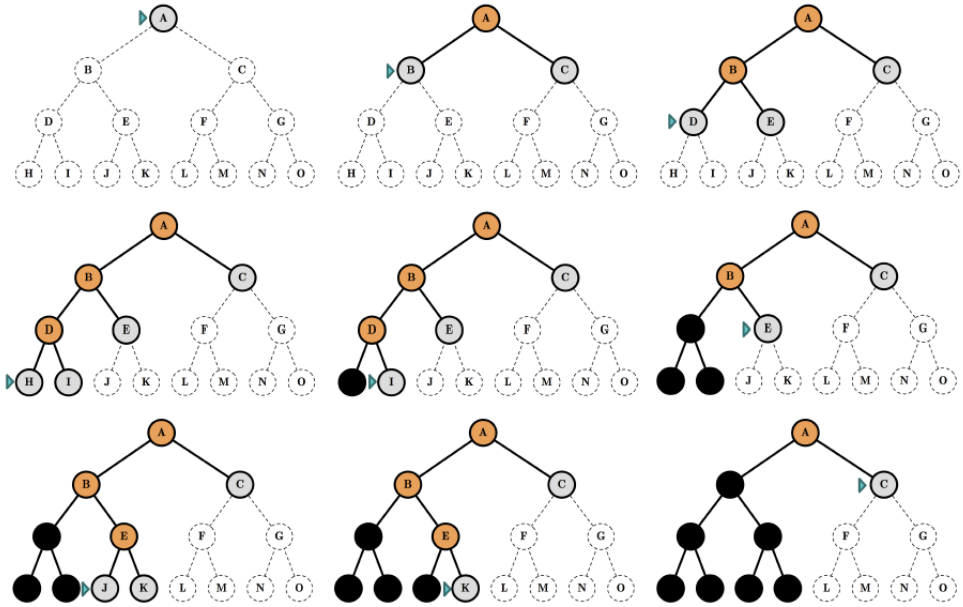
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Time and **Memory** requirements for breadth-first search for a branching factor b=10; 1 million nodes per second; 1,000 bytes per node

Memory requirement + exponential time complexity are the biggest handicaps of BFS!

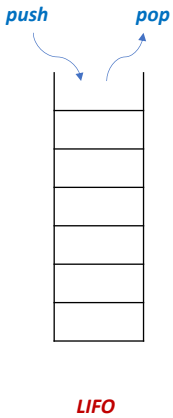
2. Depth-first Search (DFS)

DFS: Expand **deepest** first



2. Depth-first Search (DFS)

```
function DEPTH-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :

    frontier = Stack.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.pop()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.push(neighbor)

    return FAILURE
```



2. Depth-first Search (DFS)

DFS criteria?

- **Complete** No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → Complete in finite spaces
- **Time** $1 + b + b^2 + b^3 + \cdots + b^m = O(b^m)$
  - bad if ***m*** is much larger than ***d***
  - but if solutions are dense, may be much faster than BFS
- **Space** $O(bm)$ linear space complexity!
  - Needs to store only a single path from the root to a leaf node, **along with the remaining unexpanded sibling nodes for each node on the path, hence the *m* factor**
- **Optimal** No
- **implementation:** fringe: LIFO (Stack)

2. Depth-first Search (DFS)

How bad is DFS?

Recall for BFS …

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Depth =16

We go down from 10 exabytes in BFS to **156** kilobytes in DFS!

3. Depth-limited Search (DLS)

DLS: Depth first with **depth limit**

- DFS with depth limit *l* (nodes at level *l* has no successors)
- Select some limit *L* in depth to explore with DFS
- Iterative deepening: increasing the limit *l*

3. Depth-limited Search (DLS)

- If we know some knowledge about the problem, may be we don't need to go to a full depth

Idea: any city can be reached from another city in at most L steps with L < 16

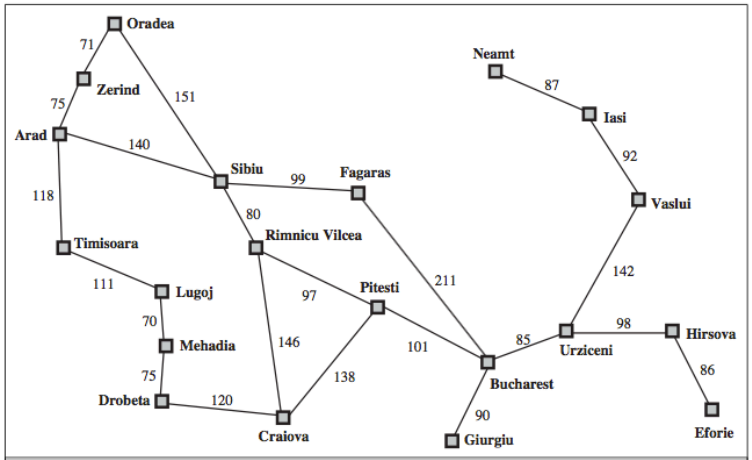


Figure 3.2 A simplified road map of part of Romania.

4. Iterative-deepening Search (IDS)

IDS: DLS with increasing **limit**

- Combines the benefits of BFS and DFS
- *Idea:* Iteratively increase the search limit until the depth of the shallowest solution *d* is reached
- Applies **DLS with increasing limits**
- The algorithm will stop
  - if a solution is found or
  - if DLS returns a failure (no solution)
- Because most of the nodes are on the bottom of the search tree, it is not a big waste to iteratively re-generate the top

4. Iterative-deepening Search (IDS)

IDS: DLS with increasing **limit**

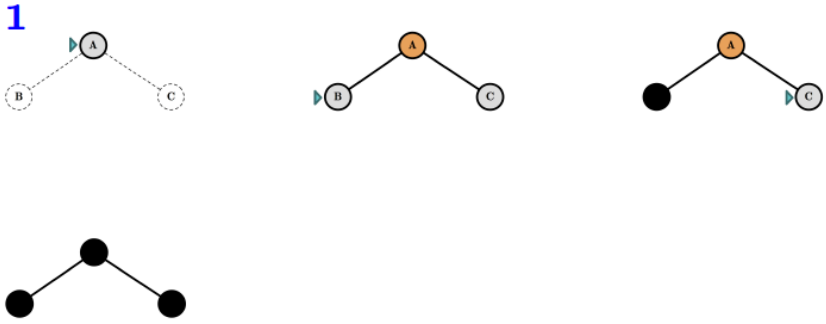
Let's take an example with a depth limit between 0 and 3

4. Iterative-deepening Search (IDS)

IDS: DLS with increasing **limit**

Let's take an example with a depth limit between 0 and 3

Limit = 1

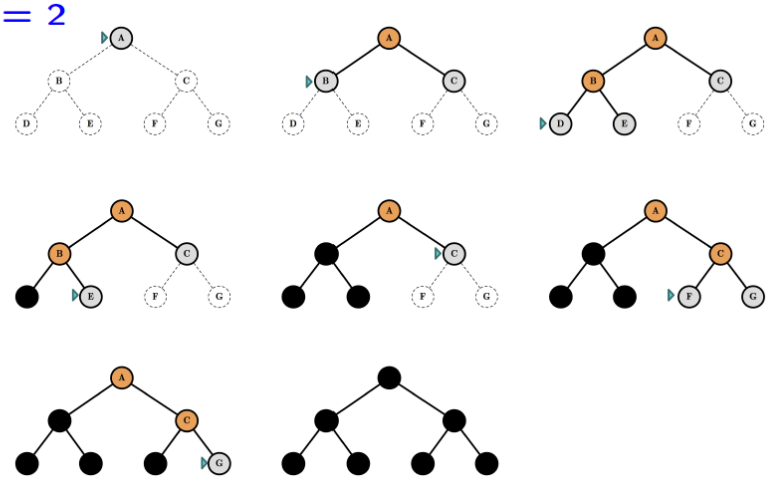


4. Iterative-deepening Search (IDS)

IDS: DLS with increasing **limit**

Let's take an example with a depth limit between 0 and 3

Limit = 2

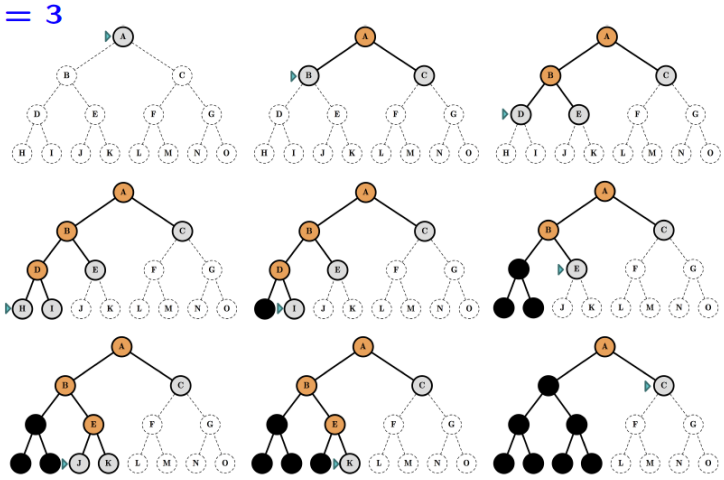


4. Iterative-deepening Search (IDS)

IDS: DLS with increasing **limit**

Let's take an example with a depth limit between 0 and 3

Limit = 3



4. Iterative-deepening Search (IDS)

IDS criteria?

- **Complete** Yes
- **Time** $O(b^d)$
- **Space** $O(bd)$
- **Optimal** Yes, if step cost = 1

5. Uniform-cost Search (UCS)

UCS: Expand **least cost** node

- The arcs in the search graph may have weights (costs attached)
  - How to leverage this information?
- BFS will find the shortest path which may be costly
- We want the **cheapest** not shallowest solution
- **Modify BFS:** Prioritize by cost not depth
  - **Expand node *n* with the lowest path cost *g(n)***
- Explores increasing costs, rather than increasing depth

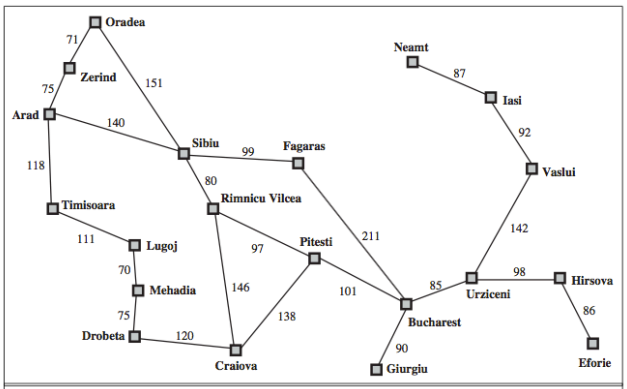


Figure 3.2 A simplified road map of part of Romania.

5. Uniform-cost Search (UCS)

```
function UNIFORM-COST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Cost from root to n

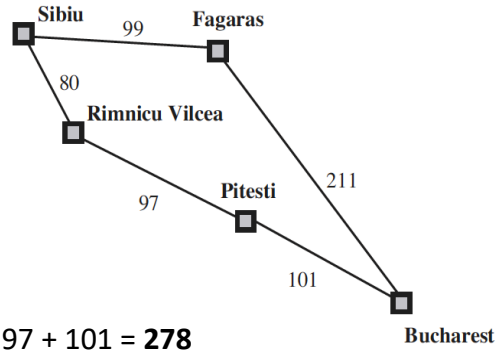
UCS vs. BFS

- **BFS** uses a **queue**, which is simply a list in which we put the elements, and we pick them **FIFO**
- In the case of **UCS**, we use a **heap**, which is a **priority queue**, which means to pick the elements or the nodes from the fringe by prioritizing according to the function g of n

Search by cost, not depth!!

5. Uniform-cost Search (UCS)

Problem is to go from Sibiu to Bucharest



Part of the Romania state space, selected to illustrate UCS

- Using BFS, we would find
  Sibiu – Fagaras – Bucharest → 99 + 211 = **310**

- However, using UCS, we would find
  Sibiu – RimnicuVilcea – Pitesti – Bucharest → 80 + 97 + 101 = **278**
  which is actually the **shortest path**!

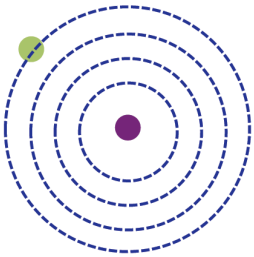
5. Uniform-cost Search (UCS)

UCS criteria?

- **Complete** Yes, if the solution has a finite cost
- **Time**
  - Suppose C^*: cost of the optimal solution
  - Every action costs at least ϵ (bound on the cost)
  - The effective depth is roughly C^*/ϵ (how deep the cheapest solution could be)
  - $O\left(b^{C^*/\epsilon}\right)$ → which can be much greater than $O(b^d)$
- **Space** # of nodes with $g \leq$ cost of optimal solution, $O\left(b^{C^*/\epsilon}\right)$
- **Optimal** Yes, nodes expanded in increasing order of *g(n)*
- **implementation:** fringe = queue ordered by path cost *g(n)*, lowest first = Heap!

5. Uniform-cost Search (UCS)

- While complete and optimal, UCS explores the space in every direction because no information is provided about the goal!
- BFS stops as soon as it generates a goal, whereas UCS examines all the nodes at the goal's depth to see if one has a lower cost
  - Thus uniform-cost search does strictly more work by expanding nodes at depth *d* unnecessarily



Bidirectional Search

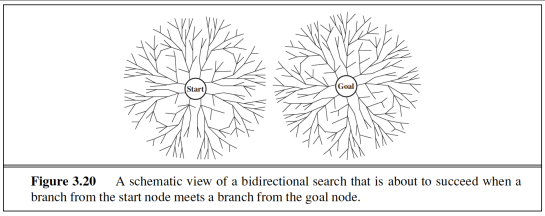
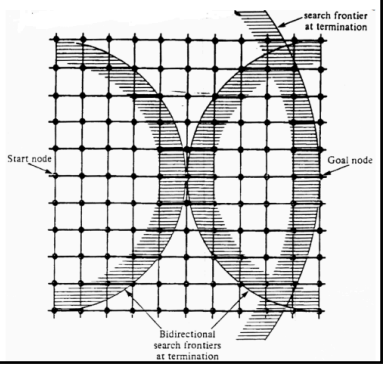


Figure 3.20 A schematic view of a bidirectional search that is about to succeed when a branch from the start node meets a branch from the goal node.

- Idea
  - Run two simultaneous searches – one forward from the initial state, and the other backward from the goal
  - Stopping when two searches "meet in the middle"
  - Need to keep track of the intersection of 2 open sets of nodes
- $b^{d/2} + b^{d/2} < b^d$
  - Time and space complexity are $O(b^{d/2})$
- What does searching backward from goal mean?
  - Need a way to specify the predecessors of goal
    - Can be difficult - e.g., predecessors of checkmate in chess?
- When there is a single goal, it is like forward search
- For more than a single goal – dummy goal state

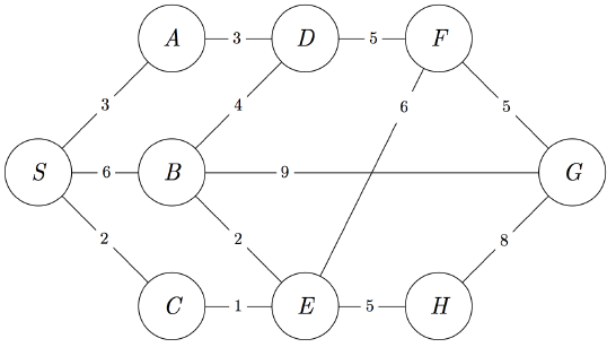


Comparing Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

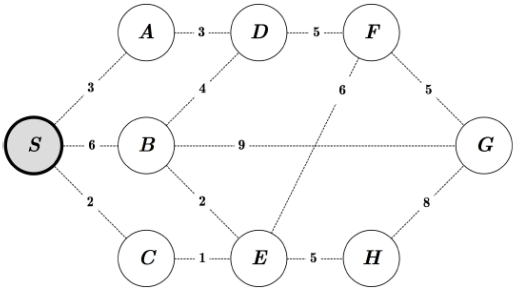
Example



Question: What is the **order of visits of the nodes** and the **path** returned by BFS, DFS and UCS?

- ***Order of Visit***: List the order that the nodes are expanded (a node is considered expanded when it is dequeued from the fringe)
- ***Path***: What is the path returned?
- Please assume that nodes are always visited in **lexicographical** order

Exercise: BFS

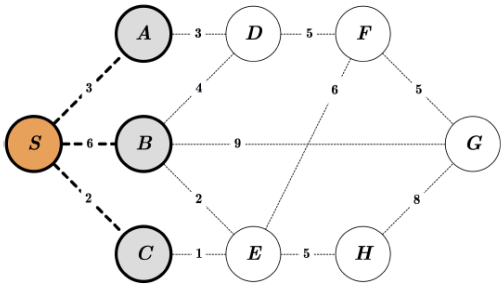


Queue:

S

Order of Visit:

Exercise: BFS



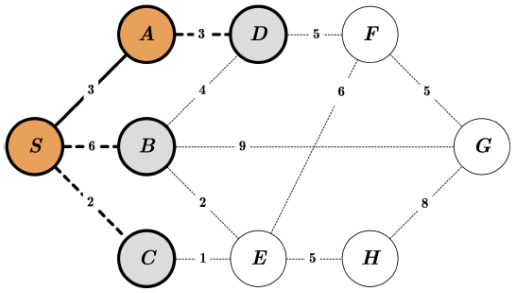
Queue:

S	A	B	C

Order of Visit:

S

Exercise: BFS



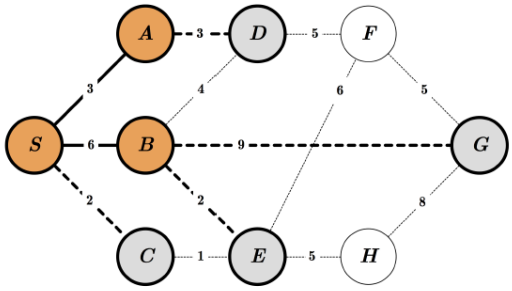
Queue:

S	A	B	C	D

Order of Visit:

S *A*

Exercise: BFS



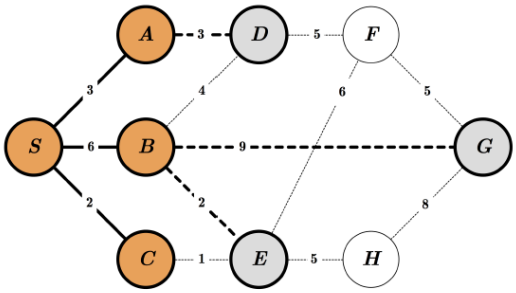
Queue:

S	A	B	C	D	E	G

Order of Visit:

S *A* *B*

Exercise: BFS



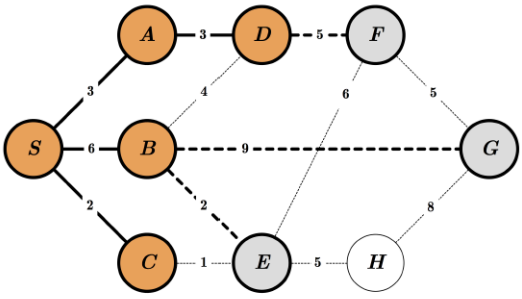
Queue:

S	A	B	C	D	E	G

Order of Visit:

S *A* *B* *C*

Exercise: BFS



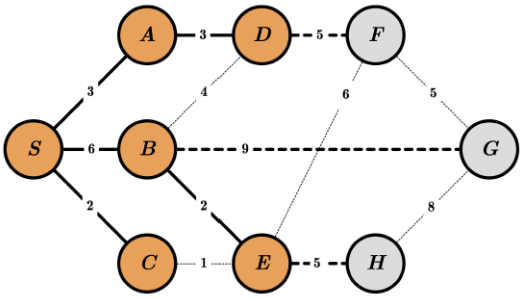
Queue:

S	A	B	C	D	E	G	F

Order of Visit:

S *A* *B* *C* *D*

Exercise: BFS



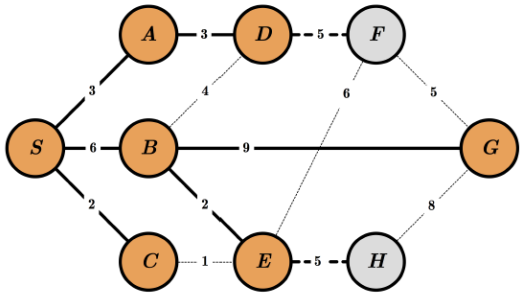
Queue:

S	A	B	C	D	E	G	F	H

Order of Visit:

S *A* *B* *C* *D* *E*

Exercise: BFS



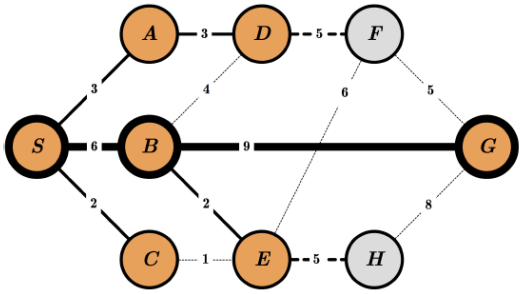
Queue:

S	A	B	C	D	E	G	F	H

Order of Visit:

S *A* *B* *C* *D* *E* *G*

Exercise: BFS



Path:
S B G

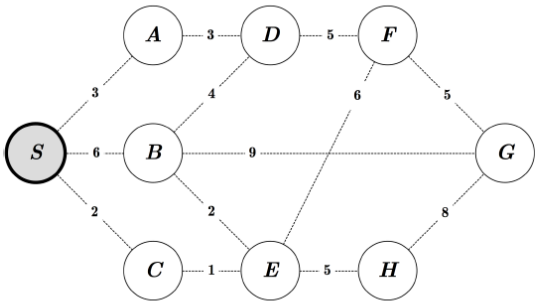
Queue:

S	A	B	C	D	E	G	F	H

Order of Visit:

S A B C D E G

Exercise: DFS

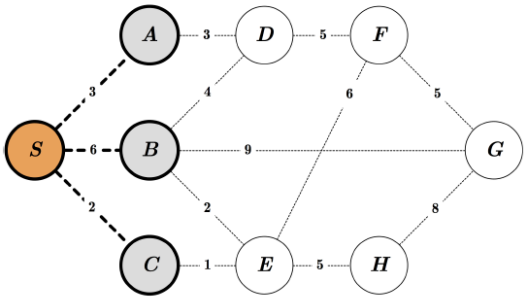


Stack:

S

Order of Visit:

Exercise: DFS



Stack:

S	*C*	*B*	*A*

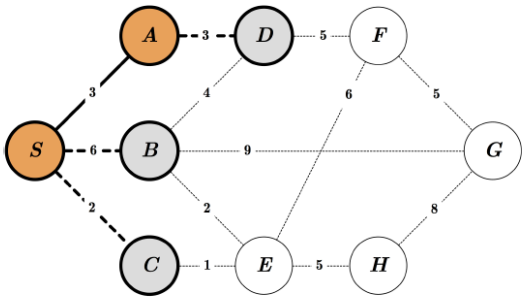
Order of Visit:

S

We're going to push the elements ABC in the stack, but we're going to do that in the reverse lexicographic order, and the main reason is that we want to pop them out in lexicographic order.

So we are going to push them as CBA, and we're going to pop them out as ABC

Exercise: DFS



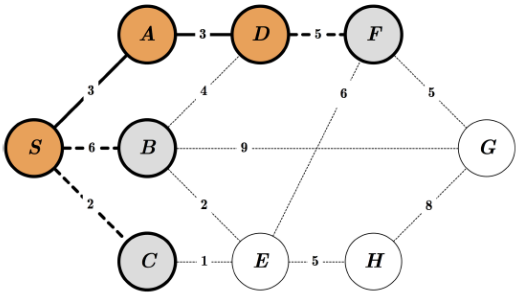
Stack:

S	*C*	*B*	*A*	*D*

Order of Visit:

S *A*

Exercise: DFS



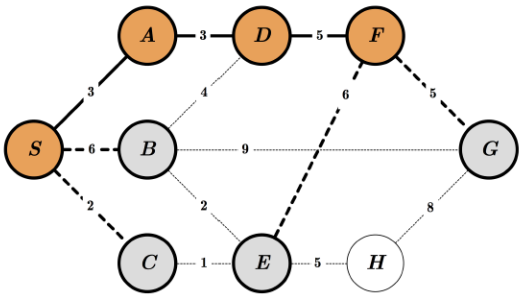
Stack:

S	C	B	A	D	F

Order of Visit:

S *A* *D*

Exercise: DFS



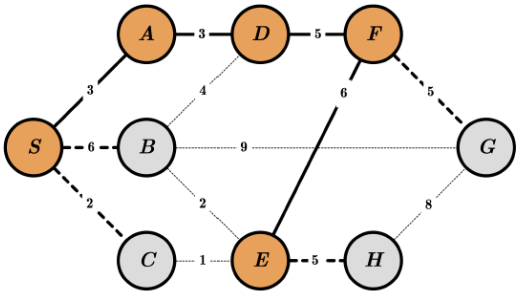
Stack:

S	C	B	A	D	F	G	E

Order of Visit:

S *A* *D* *F*

Exercise: DFS



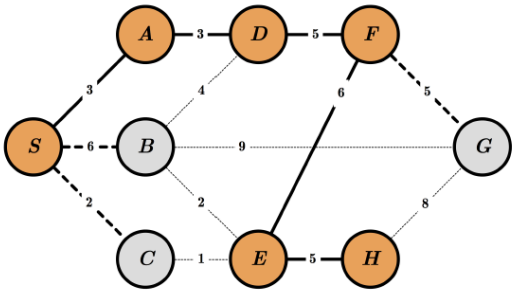
Stack:

S	*C*	*B*	*A*	*D*	*F*	*G*	*E*	*H*

Order of Visit:

S *A* *D* *F* *E*

Exercise: DFS



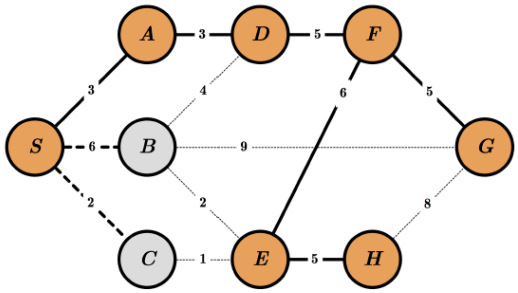
Stack:

S	*C*	*B*	*A*	*D*	*F*	*G*	*E*	*H*

Order of Visit:

S *A* *D* *F* *E* *H*

Exercise: DFS



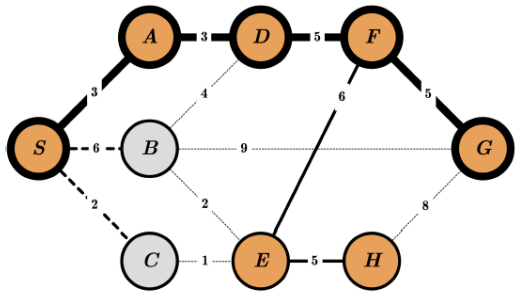
Stack:

S	C	B	A	D	F	G	E	H

Order of Visit:

S A D F E H G

Exercise: DFS



Path:
S A D F G

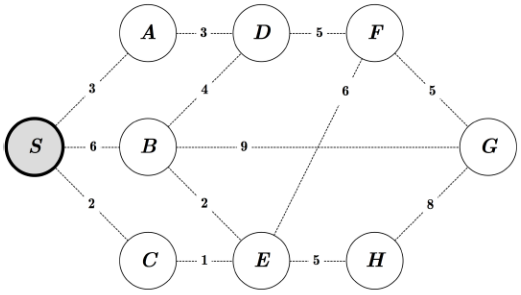
Stack:

S	C	B	A	D	F	G	E	H

Order of Visit:

S A D F E H G

Exercise: UCS

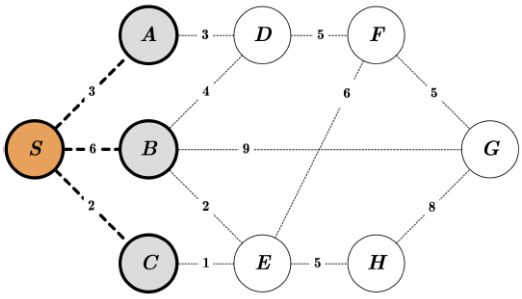


Priority Queue:

S_0

Order of Visit:

Exercise: UCS



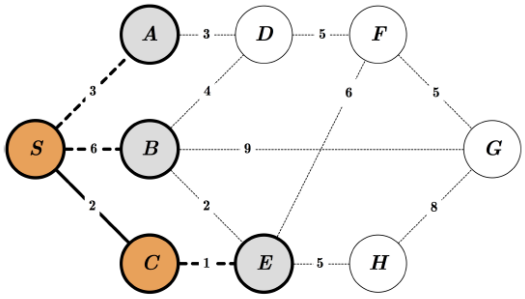
Priority Queue:

S_0	C_2	A_3	B_6

Order of Visit:

S

Exercise: UCS



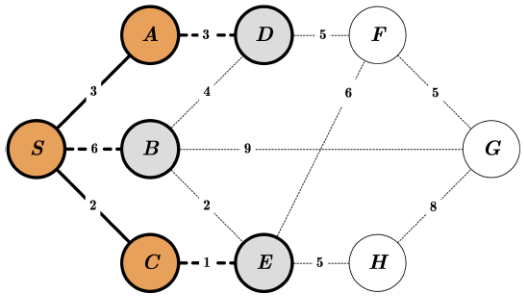
Priority Queue:

S_0	C_2	A_3	E_3	B_6

Order of Visit:

S *C*

Exercise: UCS



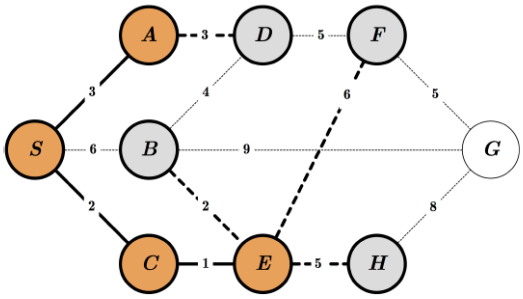
Priority Queue:

S_0	C_2	A_3	E_3	B_6	D_6

Order of Visit:

S *C* *A*

Exercise: UCS



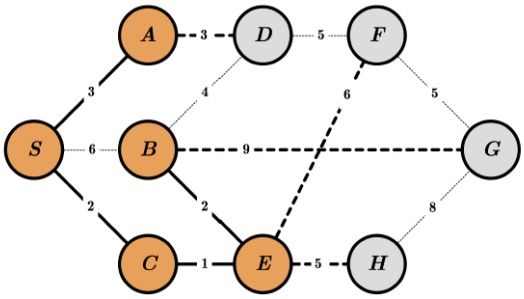
Priority Queue:

S_0	C_2	A_3	E_3	B_5	D_6	H_8	F_9

Order of Visit:

S *C* *A* *E*

Exercise: UCS



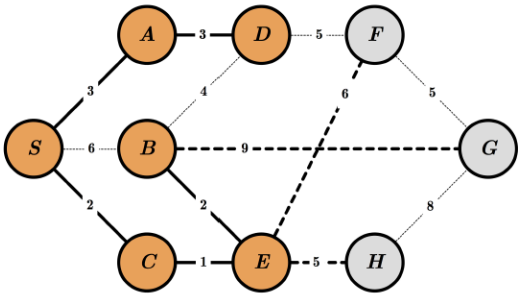
Priority Queue:

S_0	C_2	A_3	E_3	B_5	D_6	H_8	F_9	G_{14}

Order of Visit:

S *C* *A* *E* *B*

Exercise: UCS



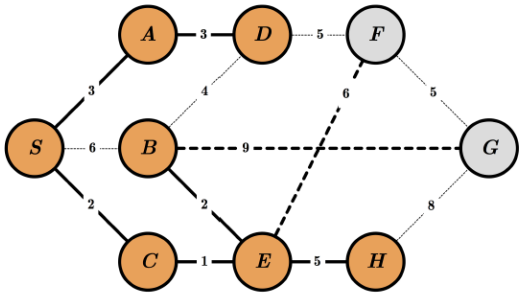
Priority Queue:

S_0	C_2	A_3	E_3	B_5	D_6	H_8	F_9	G_{14}

Order of Visit:

S C A E B D

Exercise: UCS



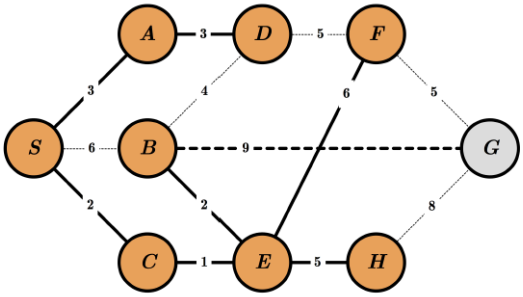
Priority Queue:

S_0	C_2	A_3	E_3	B_5	D_6	H_8	F_9	G_{14}

Order of Visit:

S C A E B D H

Exercise: UCS



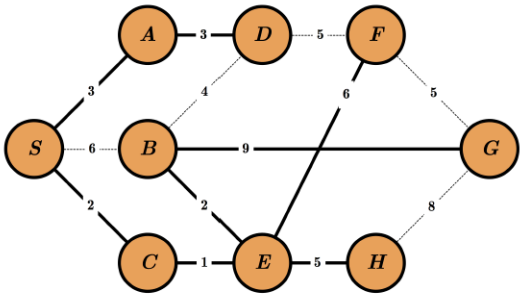
Priority Queue:

S_0	C_2	A_3	E_3	B_5	D_6	H_8	F_9	G_{14}

Order of Visit:

S C A E B D H F

Exercise: UCS



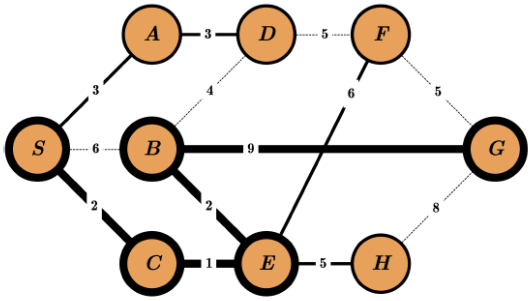
Priority Queue:

S_0	C_2	A_3	E_3	B_5	D_6	H_8	F_9	G_{14}

Order of Visit:

S C A E B D H F G

Exercise: UCS



Path:
S C E B G

Priority Queue:

S_0	C_2	A_3	E_3	B_5	D_6	H_8	F_9	G_{14}

Order of Visit:

S C A E B D H F G

Step	Expand (Pop)	Cities that should be in the priority queue	f = g	Priority Queue
0		S	0	S
1	S	A B C	0+3=3 0+6=6 0+2=2	C (2) A (3) B (6)
2	C	E A B	2+1=3 3 6	A (3) E (3) B (6)
3	A	D E B	3+3=6 3 6	E (3) B (6) D (6)
4	E	B H F D	3+2=5 3+5=8 3+6=9 6	B (5) D (6) H (8) F (9)
5	B	G D H F	5+9=14 6 8 9	D (6) H (8) F (9) G (14)
6	D	H F G	8 9 14	H (8) F (9) G (14)
7	H	F G	9 14	F (9) G (14)
8	F	G	14	G (14)
9	G	*GOAL reached!!*		

UCS from S to G

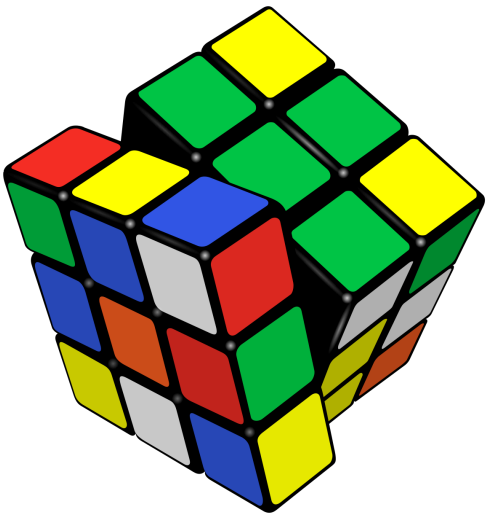
Order of Visit: S C A E B D H F G
Path: S C E B G
Path Cost: 14

BIM309 Artificial Intelligence

Week 5 – Search Agents & Informed (Heuristic) Search

Outline

- Heuristic Search
- Greedy Search Algorithms
- A* Search and Optimality
- Recap of Search Algorithms



Informed Search

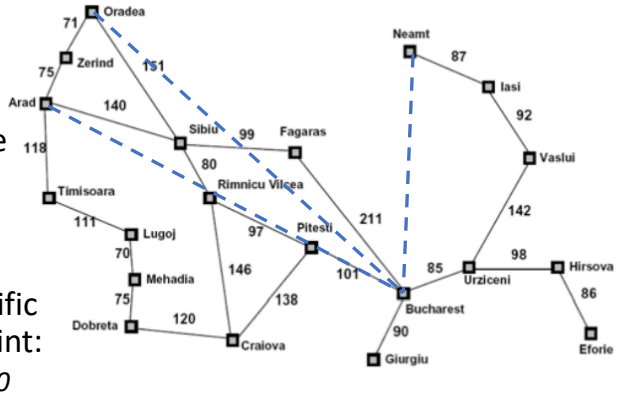
Basic Idea: **Beyond the definition of the problem itself use domain knowledge!** we may have about the problem to guide the search

- Are we getting close to the goal?
  - What if we know something about the search?
  - How should we include that knowledge?
  - In what form should it be expressed to be useful?
- Use a ***heuristic function*** that ***estimates*** how close a state is to the goal
- A heuristic is designed for a particular search problem
  - Problem specific (hence informed)
  - *Example:* Manhattan distance, Euclidean distance for pathing
- ***A heuristic does NOT have to be perfect!***
- Example of informed search strategies:
  1. Greedy best-first search
  2. A* search
  3. IDA*

Example Heuristic: Straight Line Distance (SLD)

Heuristic functions

- Most common form for additional knowledge of the problem is imparted to the search algorithm
- Considered as arbitrary, nonnegative, problem-specific functions, with one constraint:
  if *n* is a goal node, then *h(n) = 0*
- Cannot be computed from the problem description itself



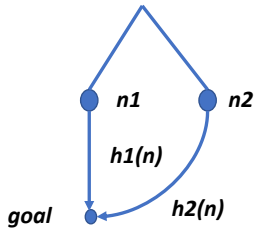
Heuristic!

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

h(x)

Straight- line distance heuristic can be used for route-finding problem in Romania. The goal is to get Bucharest, so all the distances are from each city to Bucharest - h(In(Arad))=366

Greedy Best-First Search



- Evaluation function ***h(n)*** (heuristic)
- ***h(n)*** = estimates the cost from *n* to the closest goal
  - Example: $h_{SLD}(n)$ = straight-line distance from *n* to Bucharest
- Greedy search expands the node that **appears** to be closest to goal
  - Keep nodes on a priority queue sorted by their heuristic value only
    - Priority queue sort function = *h(n)*
  - Compute heuristic function *h(n)* when a node is placed in the frontier
  - Expand the node that appears to be closest to the goal

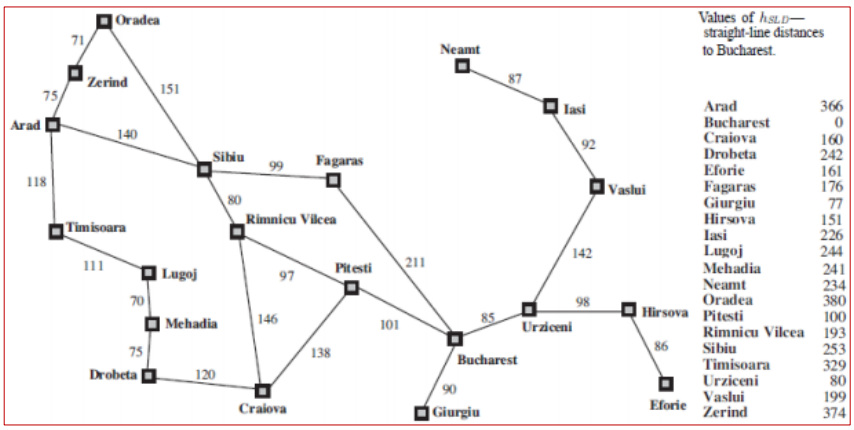
Greedy Best-First Search

```
function GREEDY-BEST-FIRST-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Example:
Greedy Best-First Search

(a) The initial state

Nodes are labeled with their *h*-values

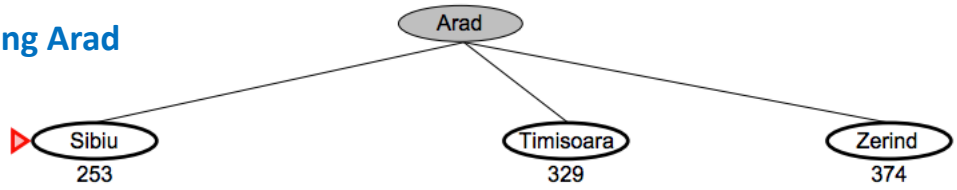


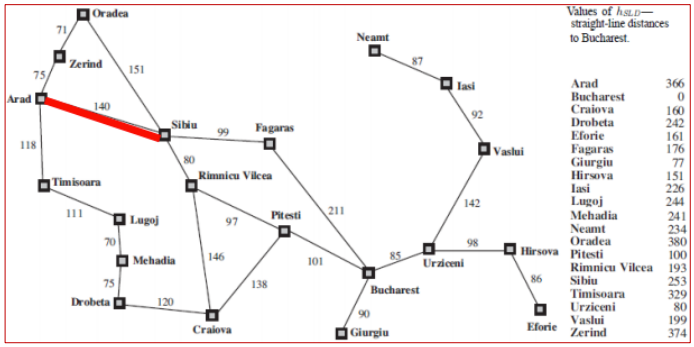
Values of h_{SLD}—straight-line distances to Bucharest.

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

Example:
Greedy Best-First Search

(b) After expanding Arad



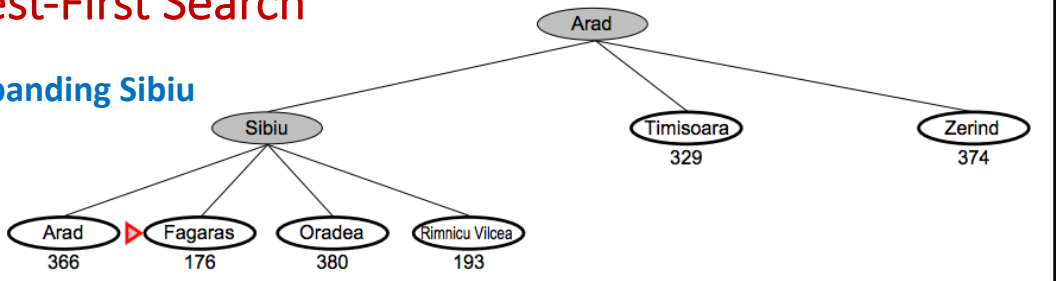


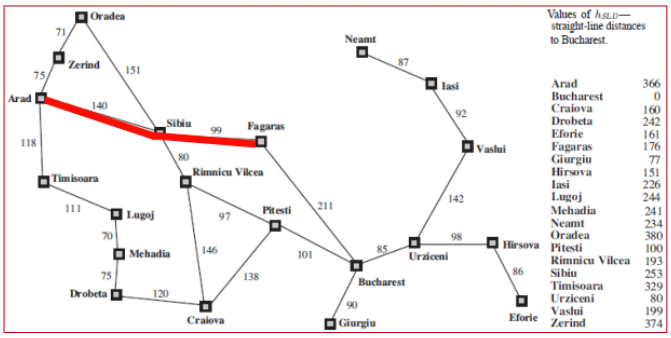
Values of h_{SLD}—straight-line distances to Bucharest.

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

Example:
Greedy Best-First Search

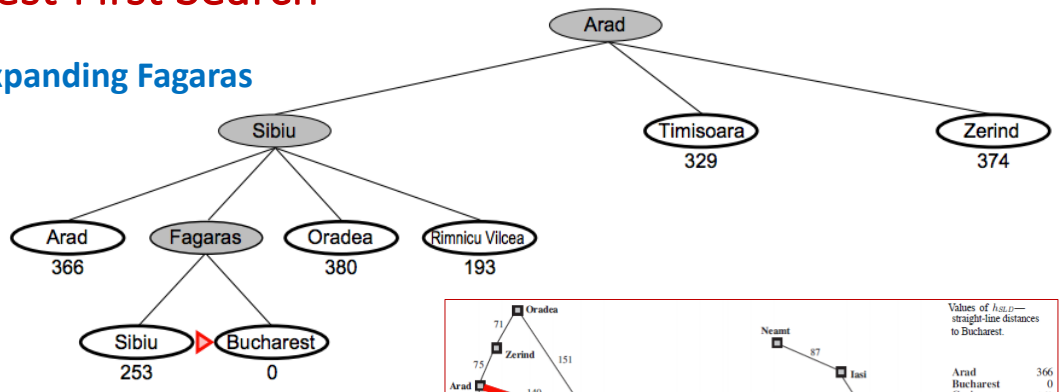
(c) After expanding Sibiu





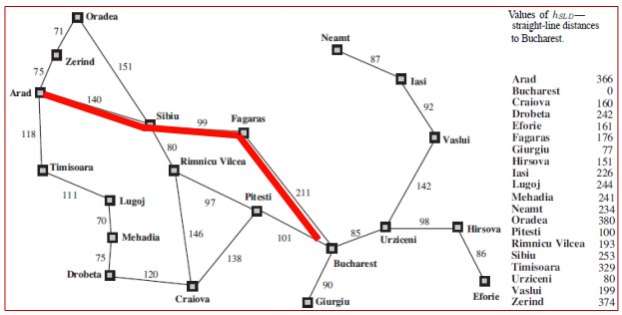
Example:
Greedy Best-First Search

(d) After expanding Fagaras



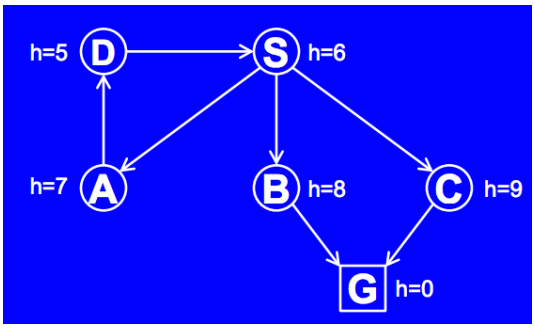
Path: Arad → Sibiu → Fagaras → Bucharest

140 + 99 + 211 = 450

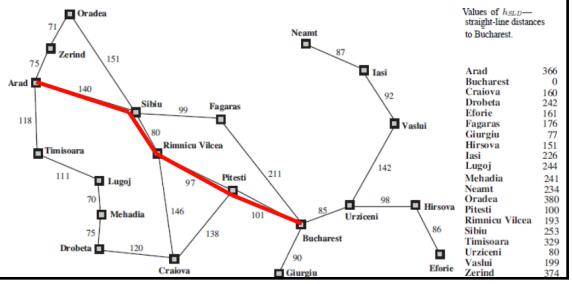


Greedy Search Criteria

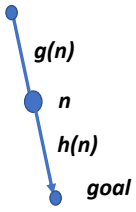
Order of node expansion: S A D S A D S A D. . .
Path found: none, *Cost of path found:* none



- **Complete** No
  - Tree version can get stuck in loops
  - Graph version is complete in finite spaces 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
  - E.g., "Arad → Sibiu → Rimnicu Vilcea → Pitesti → Bucharest" is shorter!



A* Search



- **Minimize the total estimated solution cost**
- Use a heuristic, but take into account the path cost for each node
  - Optimal if heuristic is: ***admissible*** (tree search)/***consistent*** (graph search)
- Combines:
  - ✓ ***g(n):*** cost to reach node ***n***
  - ✓ ***h(n):*** cost to get from ***n*** to the goal
  - ✓ ***f(n) = g(n) + h(n)*** ← ***Evaluation Function***

g(n) gives the path cost from the start node to node n, and
h(n) is the estimated cost of the cheapest path from n to the goal, so
f(n)* is the estimated cost of the cheapest solution through *n

A* Search

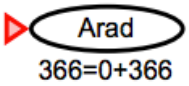
```
function A-STAR-SEARCH(initialState, goalTest)
    returns SUCCESS or FAILURE :  /* Cost f(n) = g(n) + h(n) */

    frontier = Heap.new(initialState)
    explored = Set.new()

    while not frontier.isEmpty():
        state = frontier.deleteMin()
        explored.add(state)

        if goalTest(state):
            return SUCCESS(state)

        for neighbor in state.neighbors():
            if neighbor not in frontier ∪ explored:
                frontier.insert(neighbor)
            else if neighbor in frontier:
                frontier.decreaseKey(neighbor)

    return FAILURE
```

Example: A* Search

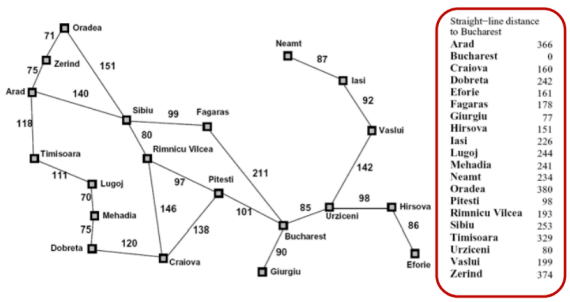
Frontier Queue

Arad 366

(a) The initial state

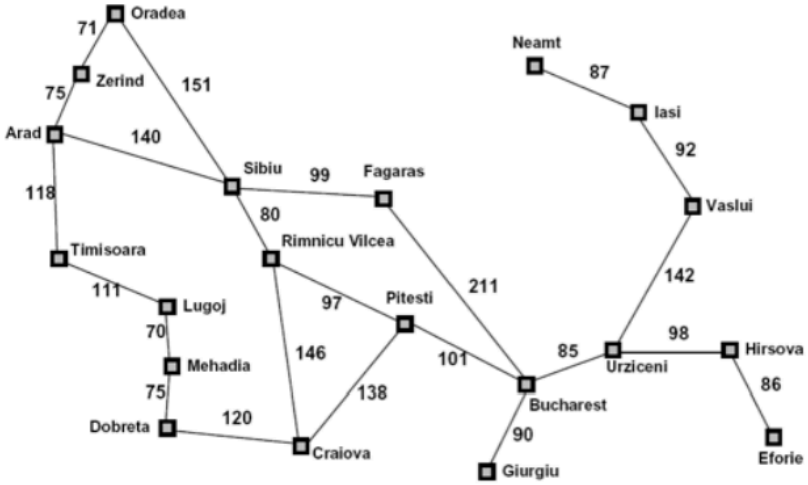


Nodes are labeled with ***f=g+h***



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

h(x)



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

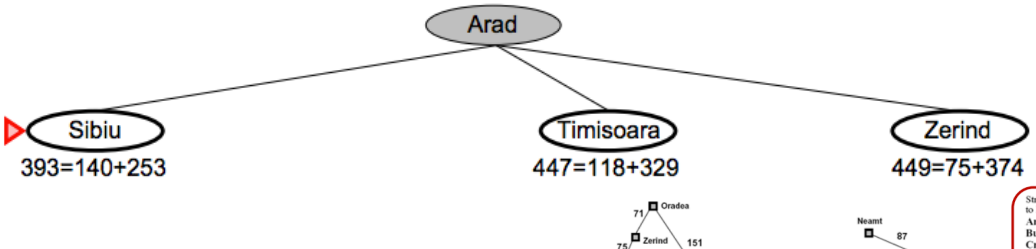
h(x)

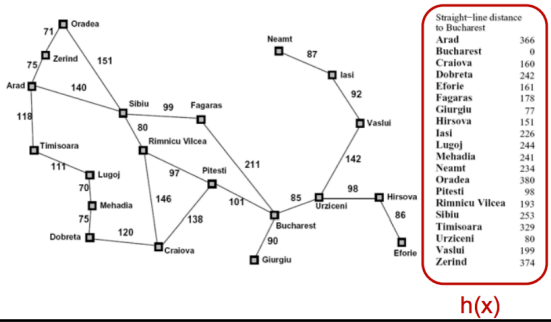
Example:
A* Search

Frontier Queue

Sibiu	393
Timisoara	447
Zerind	449

(b) After expanding Arad

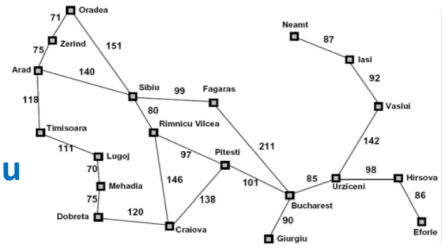




h(x)

Example: A* Search

(c) After expanding Sibiu

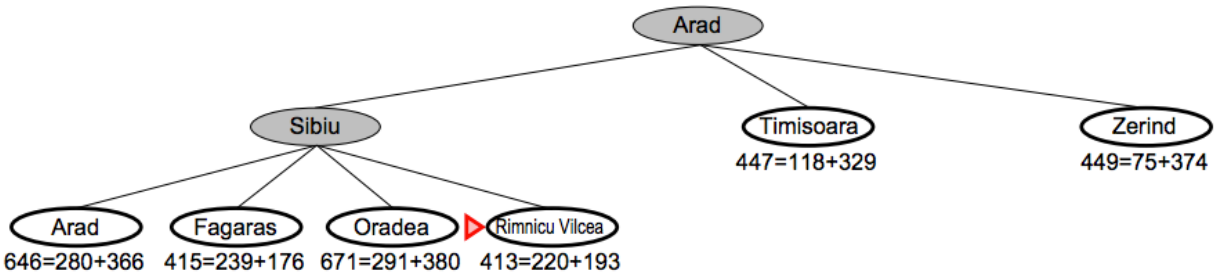


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

h(x)

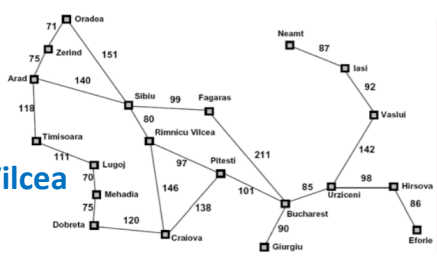
Frontier Queue

Rimricu Vicea	413
Fagaras	415
Timisoara	447
Zerind	449
Oradea	671



Example: A* Search

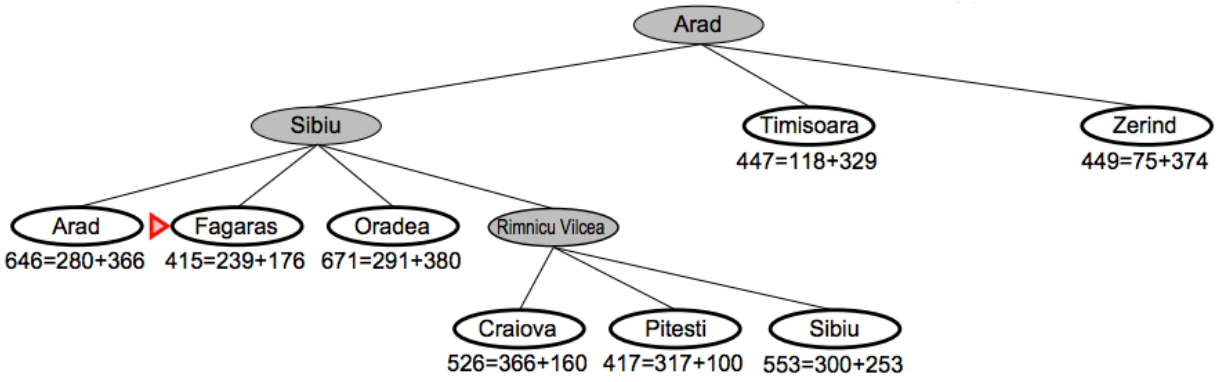
(d) After expanding Rimnicu Vilcea



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

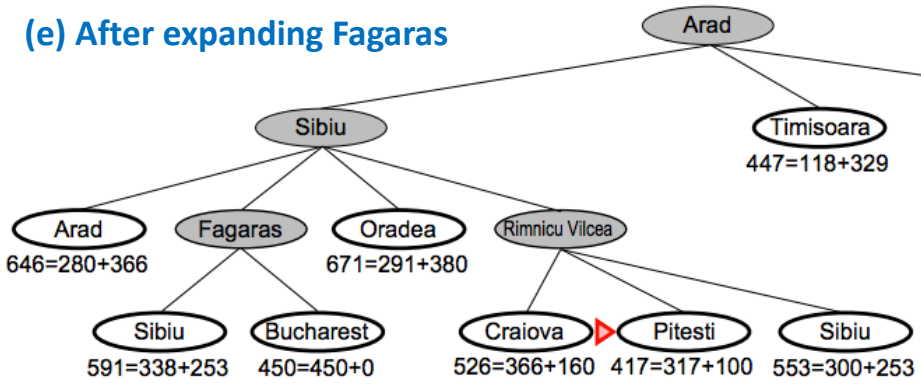
Frontier Queue

Fagaras	415
Pitesti	417
Timisoara	447
Zerind	449
Craiova	526
Oradea	671



Example:
A* Search

(e) After expanding Fagaras

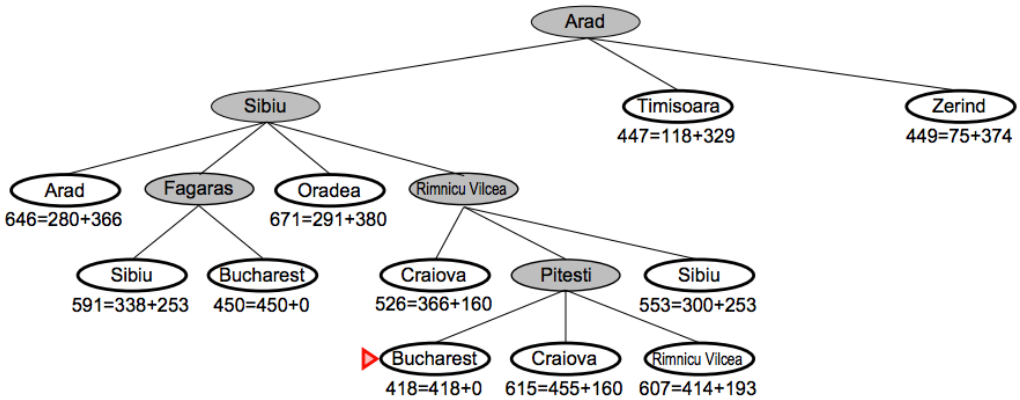


Frontier Queue

Pitesti	417
Timisoara	447
Zerind	449
Bucharest	450
Craiova	526
Oradea	671

Example:
A* Search

(f) After expanding Pitesti



Frontier Queue

Bucharest	418
Timisoara	447
Zerind	449
Bucharest	450
Craiova	526
Oradea	671

Admissible Heuristics

A good heuristic can be powerful.

Only if it is of a "good quality"

A good heuristic must be admissible

Admissible Heuristics

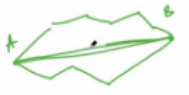
- An **admissible** heuristic **never** overestimates the cost to reach the goal, that is, it is **optimistic**
- A heuristic ***h*** is admissible if

$$\forall\, node\ n, h(n) \leq\ h^*(n)$$

  where $\boldsymbol{h}^*$ is **true** cost to reach the goal from ***n***

- Example

  h_{SLD} (used as a heuristic in the map example) is admissible, because it is by definition the shortest distance between two points
  - Never overestimates the actual road distance

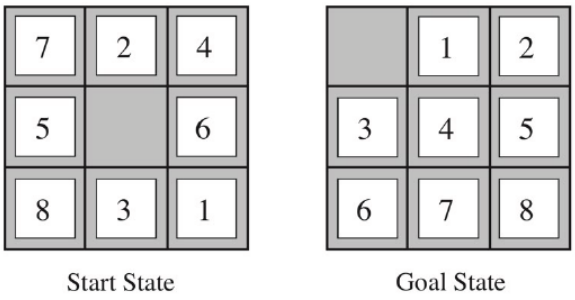


A* Search Criteria

- **Complete** Yes
- **Time** exponential (depends on heuristic)
- **Space** Keeps every node in memory (***the biggest problem***)
- **Optimal** Yes

Example: Admissible Heuristics for the 8-puzzle

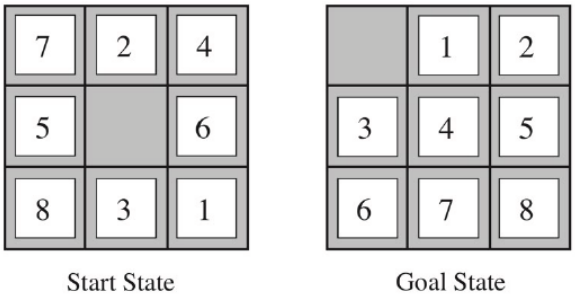
- The solution is 26 steps long
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (i.e., no. of squares from desired location of each tile - sum of the horizontal and vertical distances)



- **$h_1(n)$ = ??**
- **$h_2(n)$ = ??**

Example: Admissible Heuristics for the 8-puzzle

- The solution is **26** steps long
- $h_1(n)$ = number of misplaced tiles
- $h_2(n)$ = total Manhattan distance (i.e., no. of squares from desired location of each tile - sum of the horizontal and vertical distances)



Start State Goal State

- $h_1(S)$ = **8**
- $h_2(S)$ = Tiles 1 to 8 in the start state gives:
  $h_2(S)$ = 3+1+2+2+2+3+3+2 = **18**

which does not overestimate the true solution

Search Algorithms: Recap

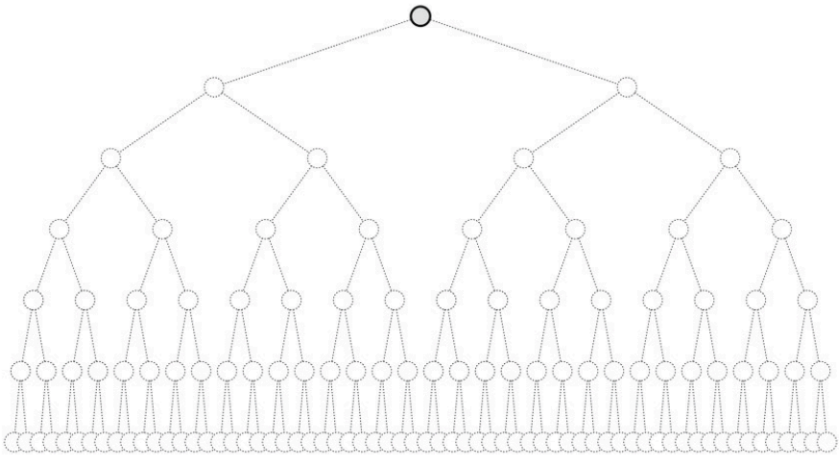
- **Uninformed Search:** Use no domain knowledge

  BFS, DFS, DLS, IDS, UCS

- **Informed Search:** Use a heuristic function that estimates how close a state is to the goal

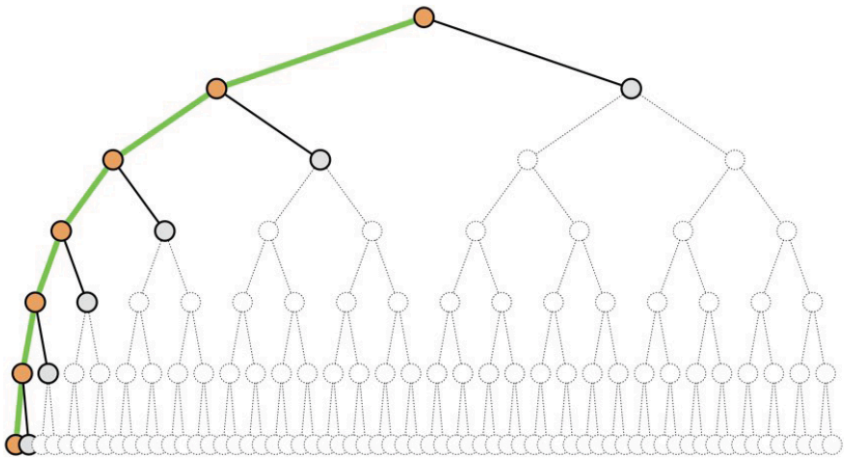
  Greedy search, A*, IDA*

DFS



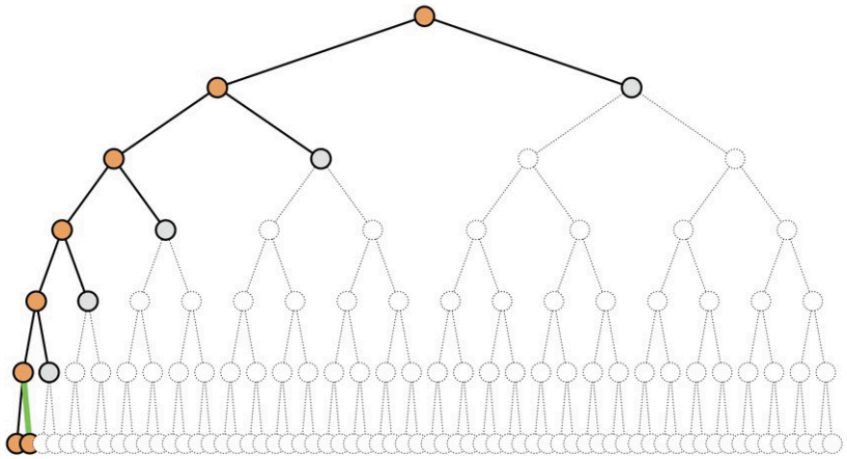
Searches branch by branch ...
... with each branch pursued to maximum depth.

DFS



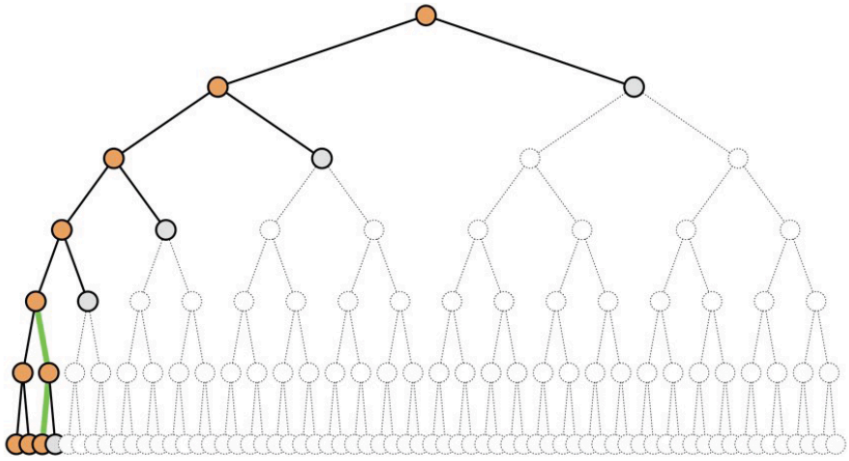
Searches branch by branch ...
... with each branch pursued to maximum depth.

DFS



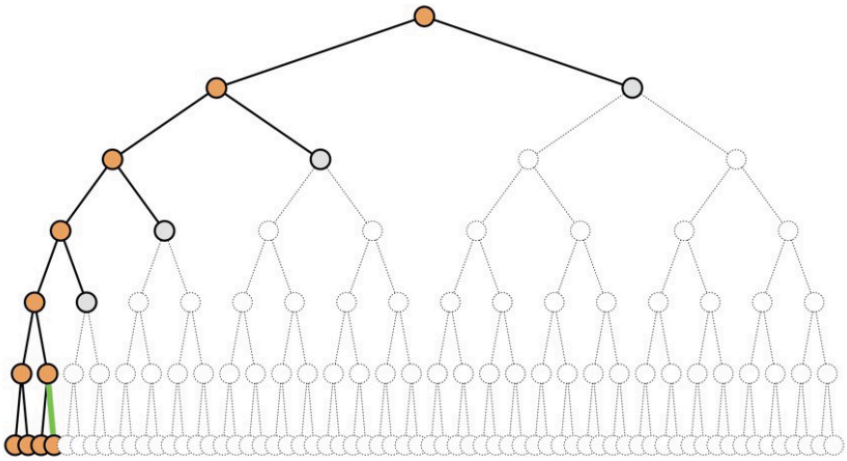
Searches branch by branch ...
... with each branch pursued to maximum depth.

DFS



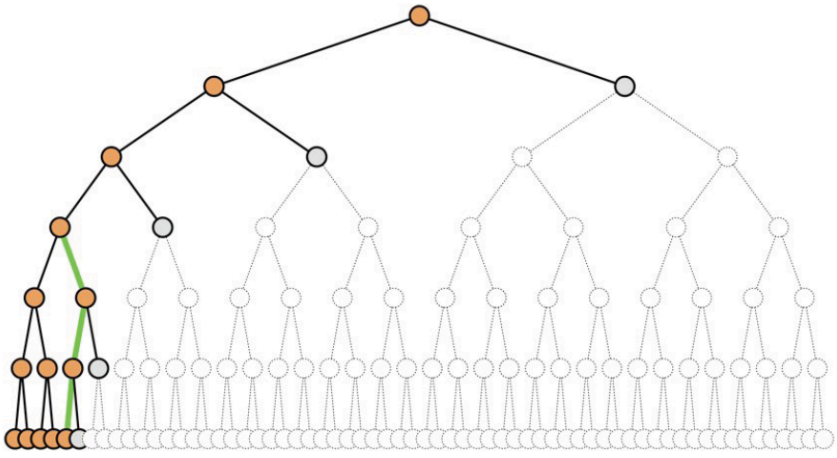
Searches branch by branch ...
... with each branch pursued to maximum depth.

DFS



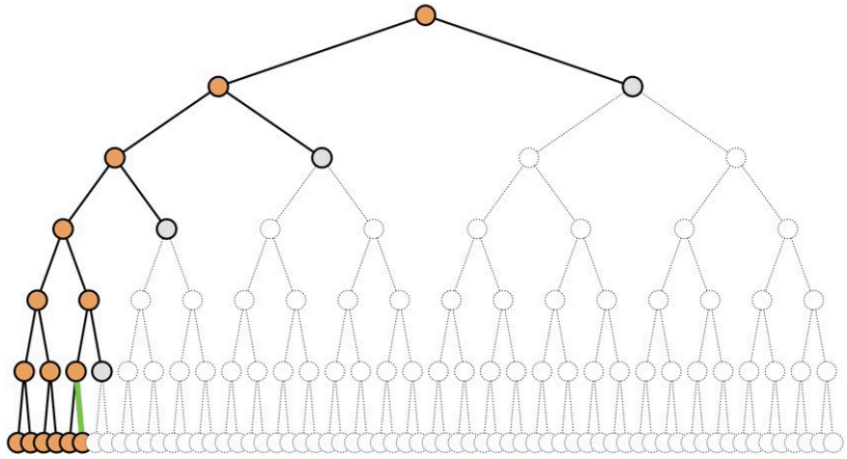
Searches branch by branch ...
... with each branch pursued to maximum depth.

DFS



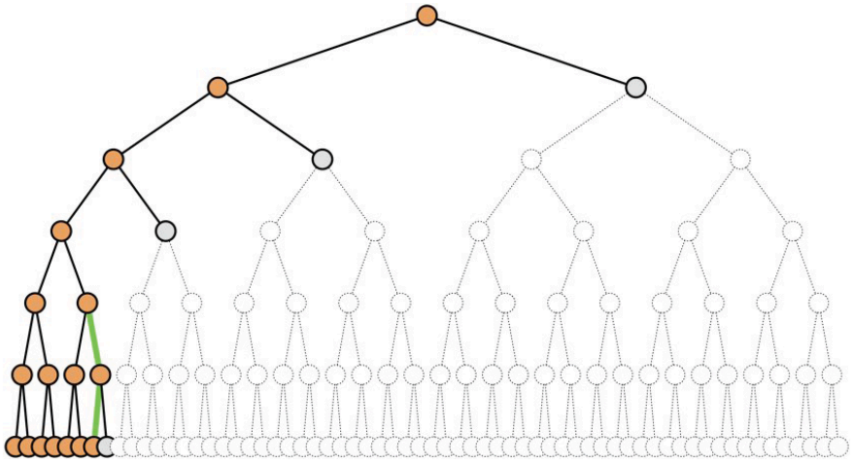
Searches branch by branch ...
... with each branch pursued to maximum depth.

DFS



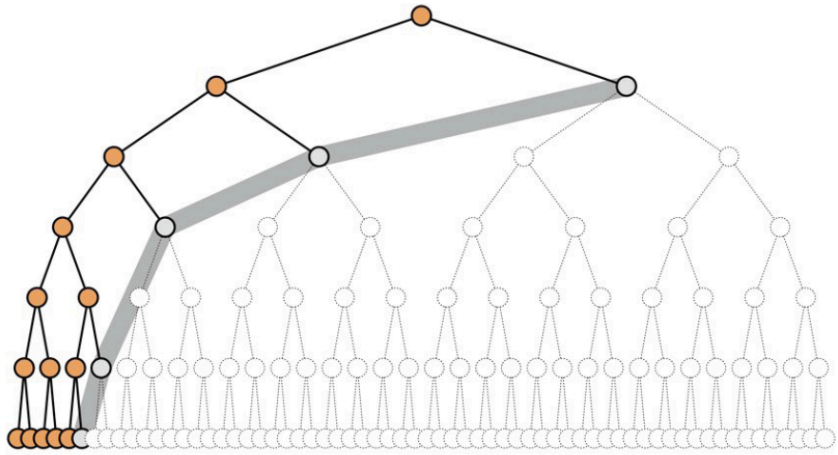
Searches branch by branch ...
... with each branch pursued to maximum depth.

DFS



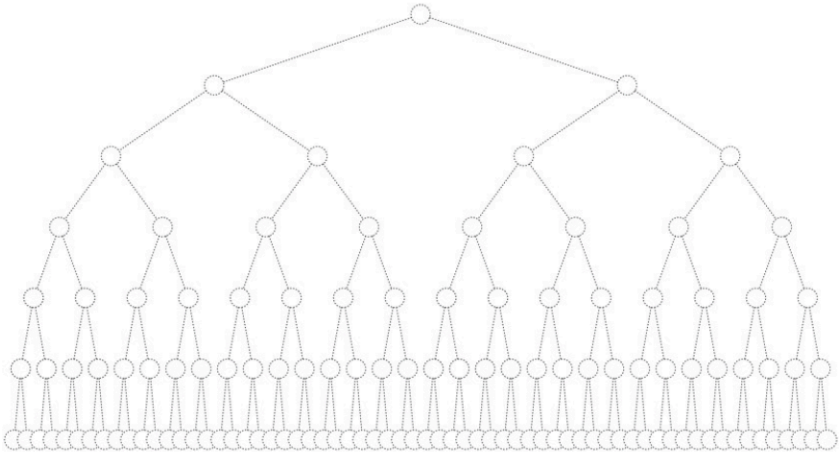
Searches branch by branch ...
... with each branch pursued to maximum depth.

DFS



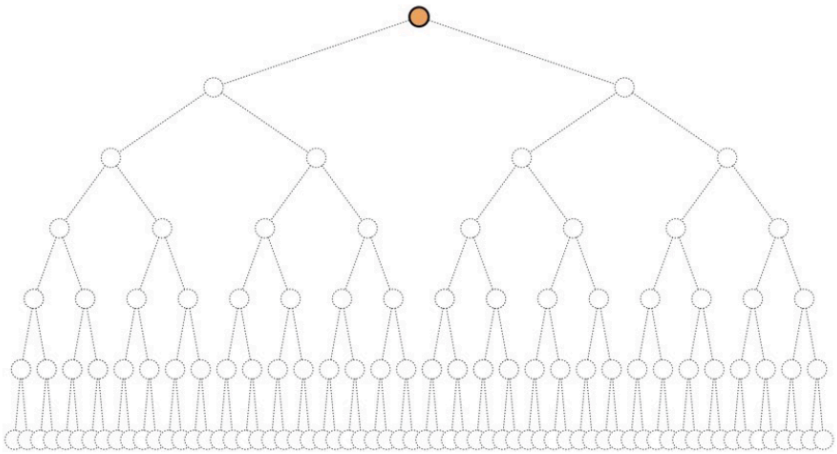
The frontier consists of unexplored siblings of all ancestors.
Search proceeds by exhausting one branch at a time.

IDS



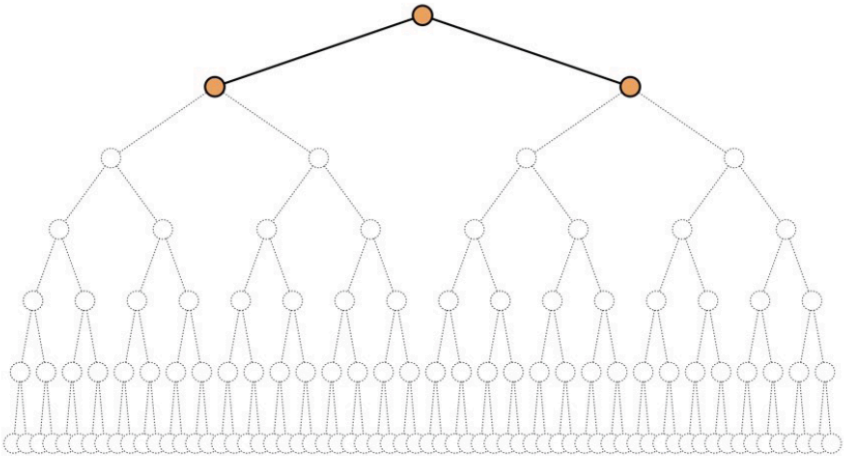
Searches subtree by subtree ...
... with each subtree increasing by depth limit.

IDS



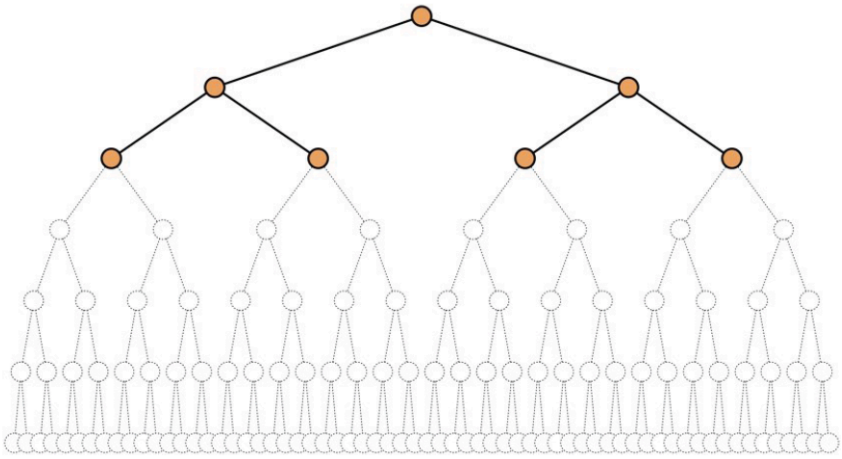
Searches subtree by subtree ...
... with each subtree increasing by depth limit.

IDS



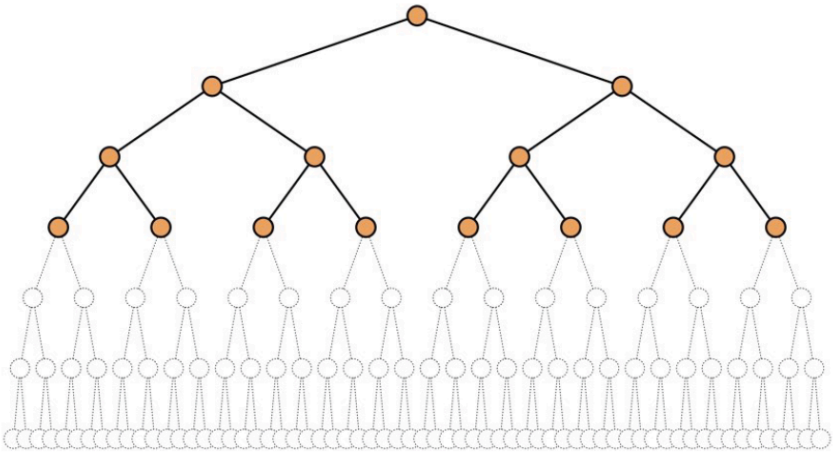
Searches subtree by subtree ...
... with each subtree increasing by depth limit.

IDS



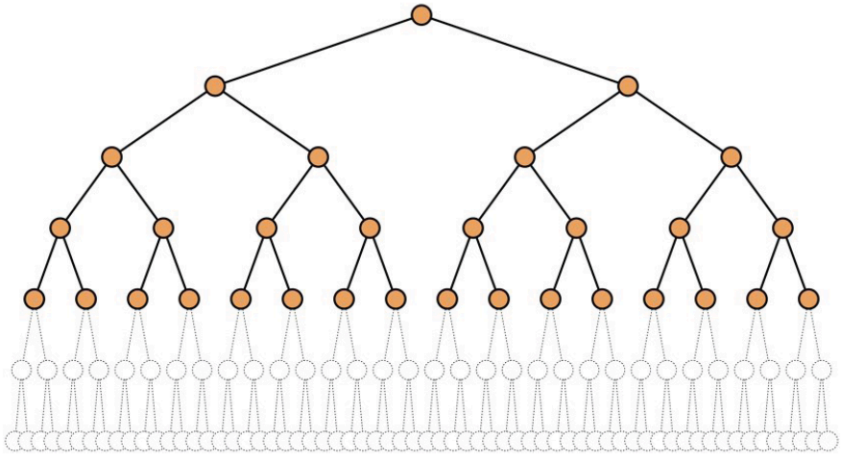
Searches subtree by subtree ...
... with each subtree increasing by depth limit.

IDS



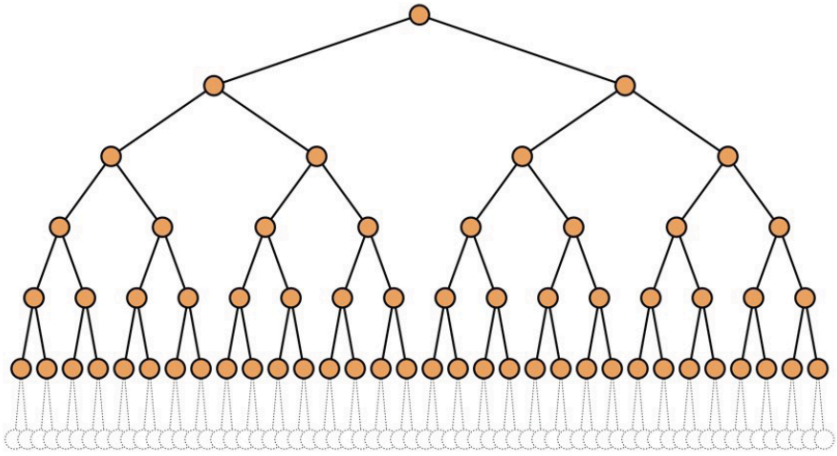
Searches subtree by subtree ...
... with each subtree increasing by depth limit.

IDS



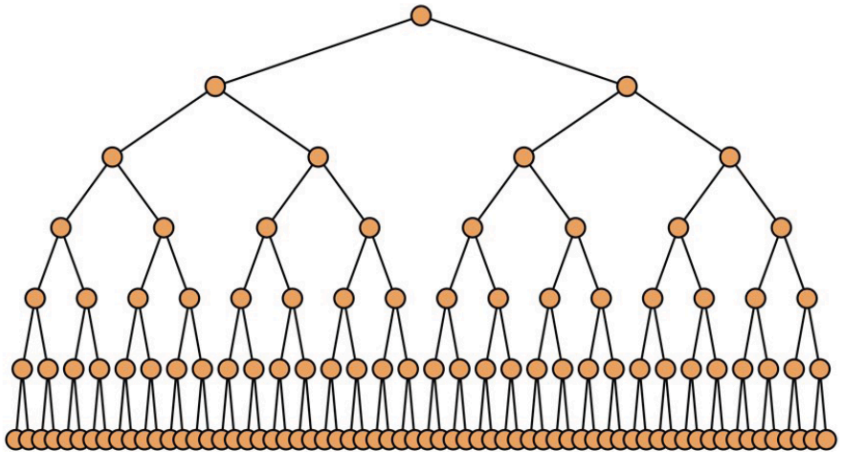
Searches subtree by subtree ...
... with each subtree increasing by depth limit.

IDS



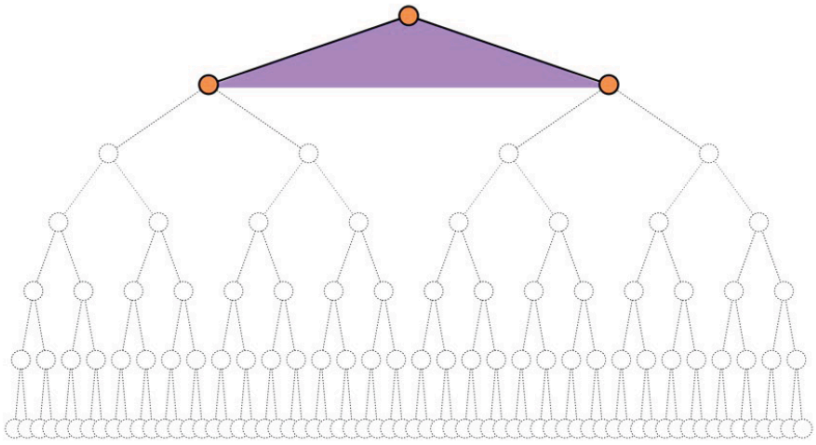
Searches subtree by subtree ...
... with each subtree increasing by depth limit.

IDS



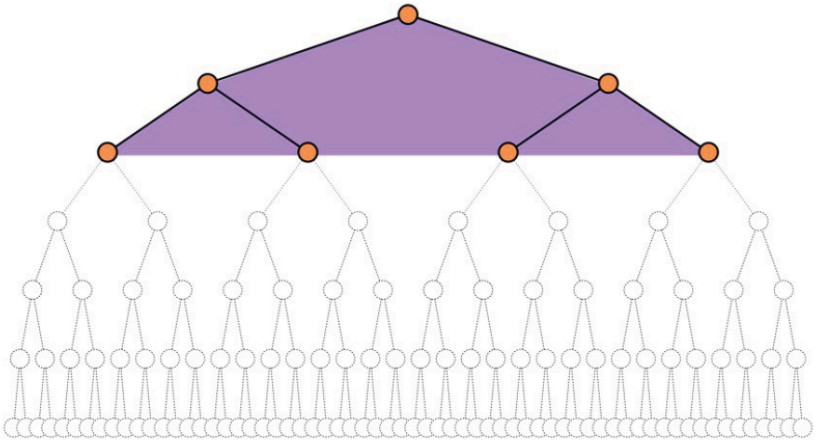
Searches subtree by subtree ...
... with each subtree increasing by depth limit.

IDS



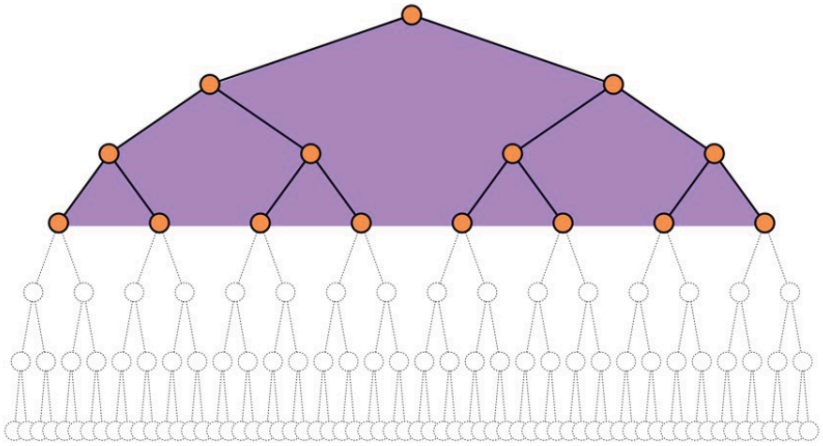
Each iteration is simply an instance of depth-limited search.
Search proceeds by exhausting larger and larger subtrees.

IDS



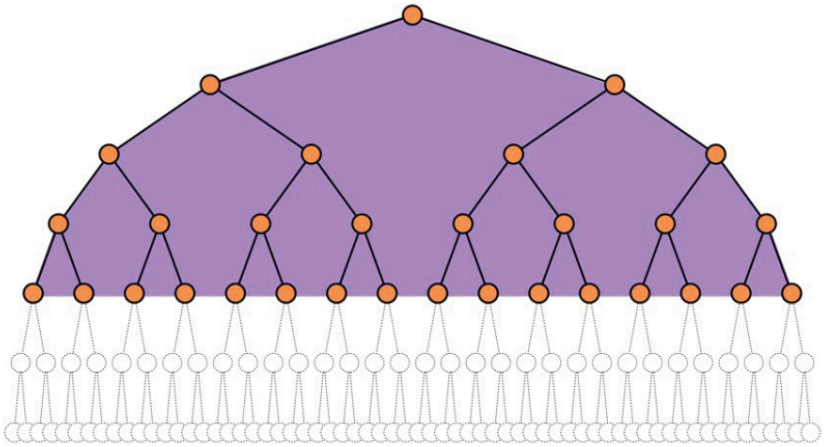
Each iteration is simply an instance of depth-limited search.
Search proceeds by exhausting larger and larger subtrees.

IDS



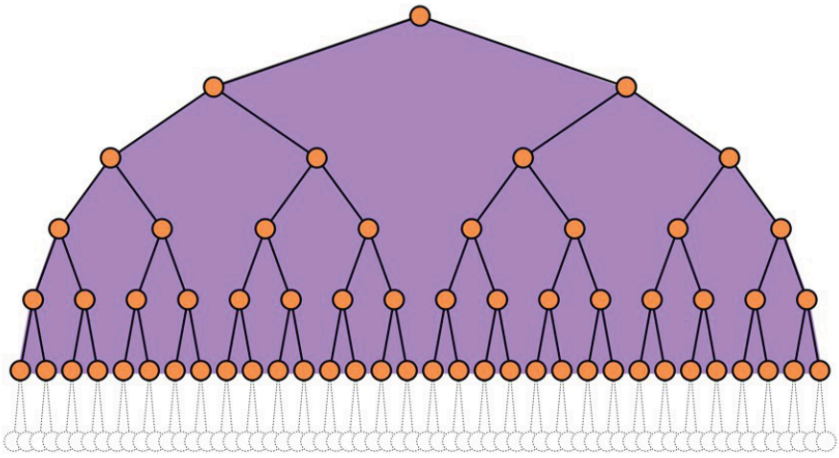
Each iteration is simply an instance of depth-limited search.
Search proceeds by exhausting larger and larger subtrees.

IDS



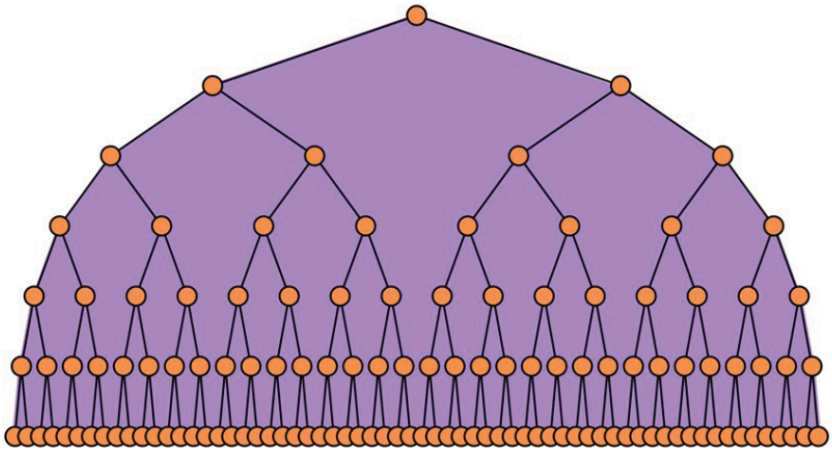
Each iteration is simply an instance of depth-limited search.
Search proceeds by exhausting larger and larger subtrees.

IDS



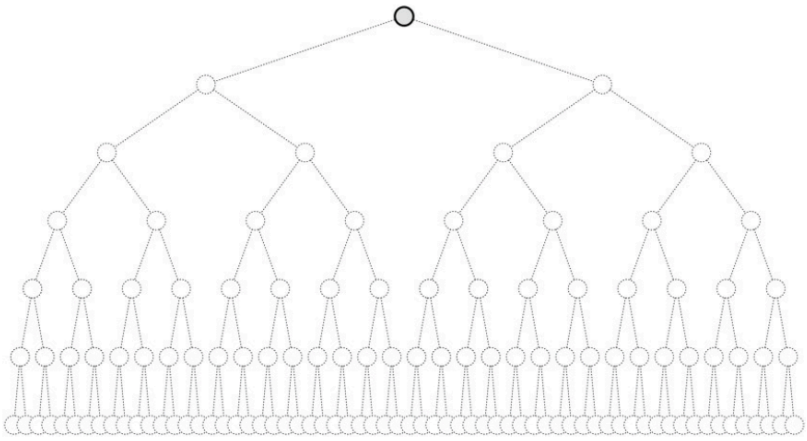
Each iteration is simply an instance of depth-limited search.
Search proceeds by exhausting larger and larger subtrees.

IDS



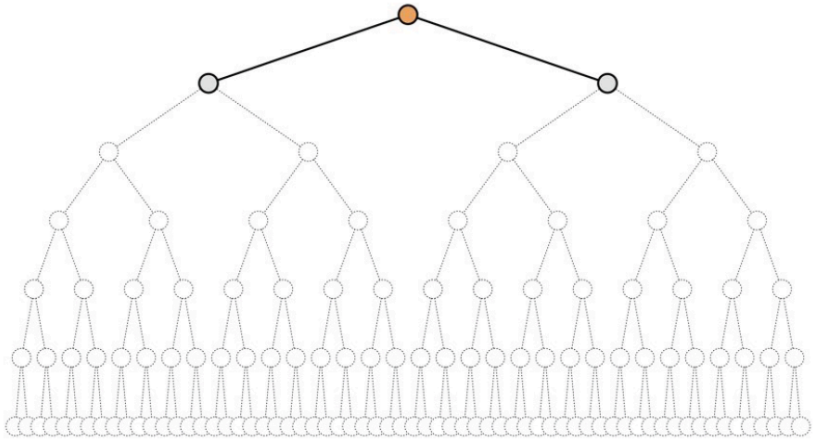
Each iteration is simply an instance of depth-limited search.
Search proceeds by exhausting larger and larger subtrees.

BFS



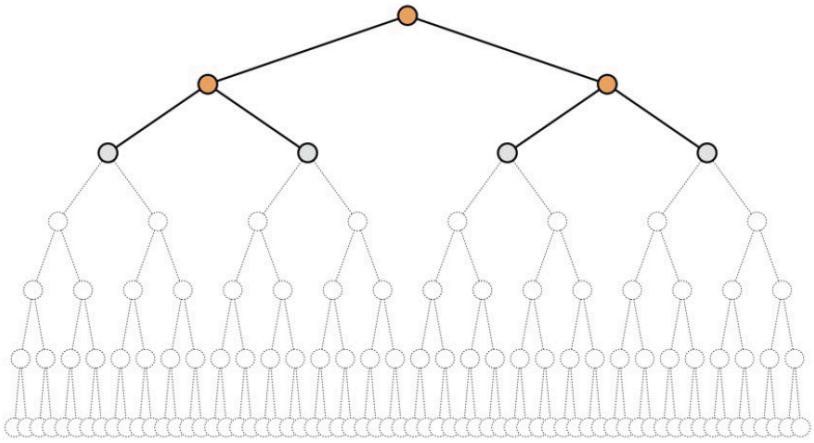
Searches layer by layer ...
... with each layer organized by node depth.

BFS



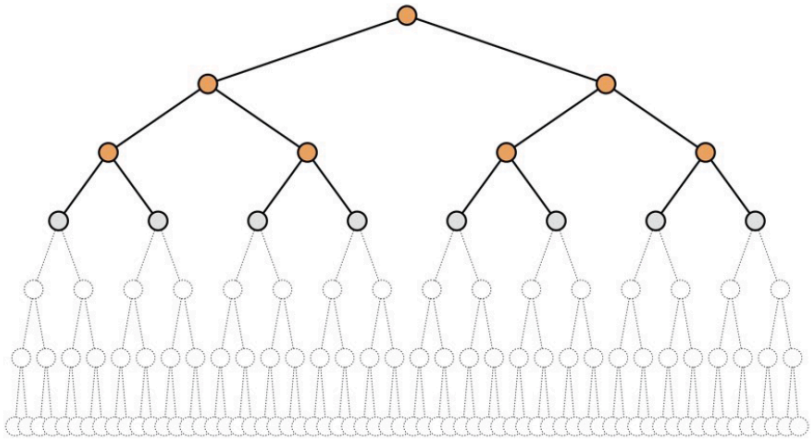
Searches layer by layer ...
... with each layer organized by node depth.

BFS



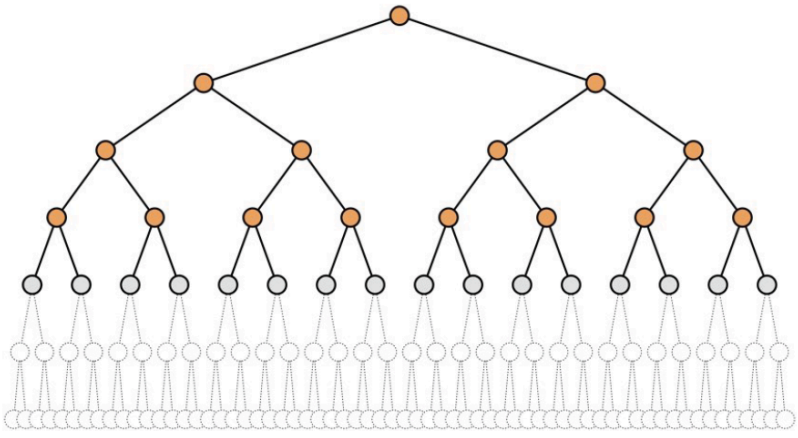
Searches layer by layer ...
... with each layer organized by node depth.

BFS



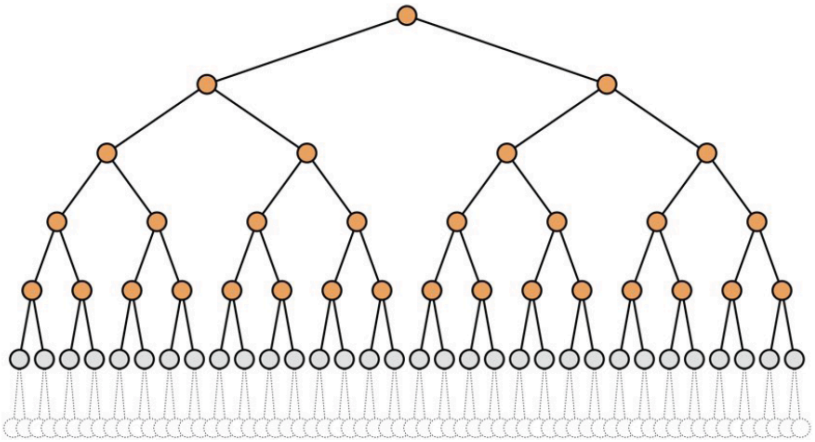
Searches layer by layer ...
... with each layer organized by node depth.

BFS



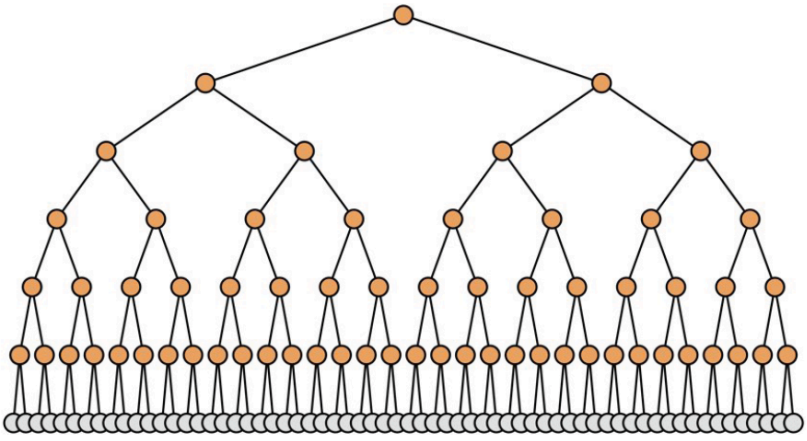
Searches layer by layer ...
... with each layer organized by node depth.

BFS



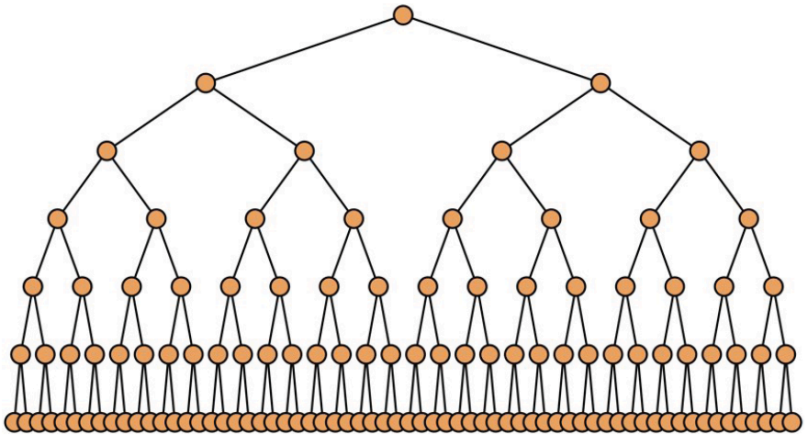
Searches layer by layer ...
... with each layer organized by node depth.

BFS



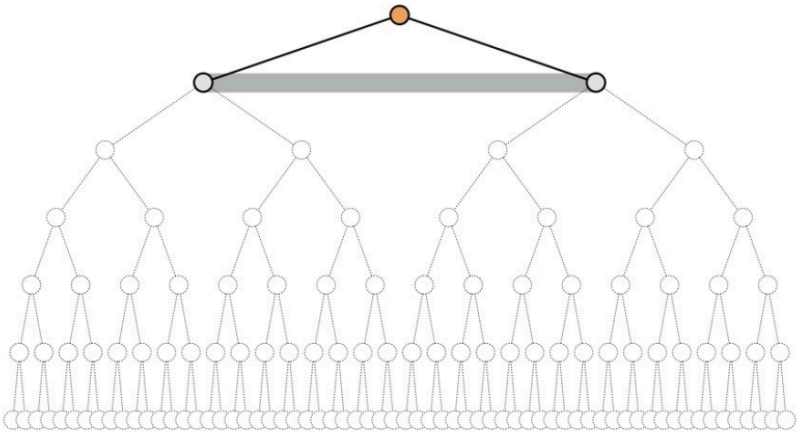
Searches layer by layer ...
... with each layer organized by node depth.

BFS



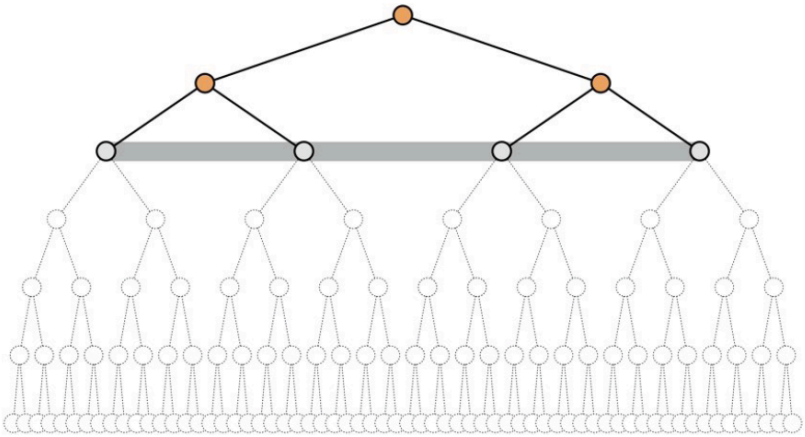
Searches layer by layer ...
... with each layer organized by node depth.

BFS



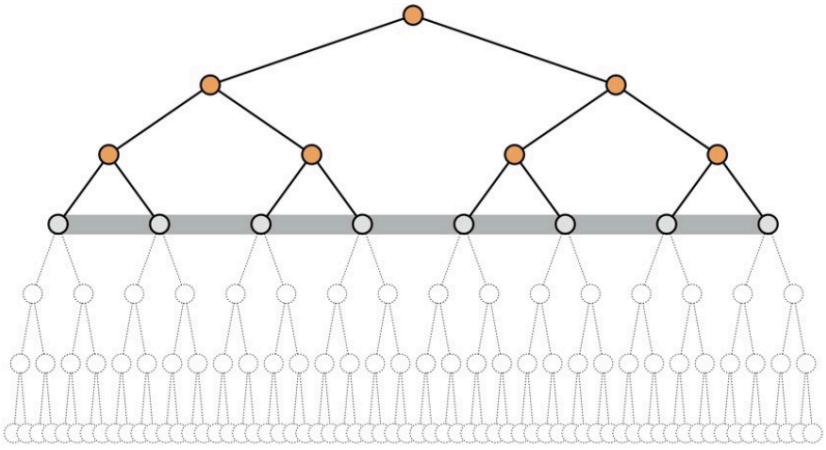
The frontier consists of nodes of similar depth (horizontal).
Search proceeds by exhausting one layer at a time.

BFS



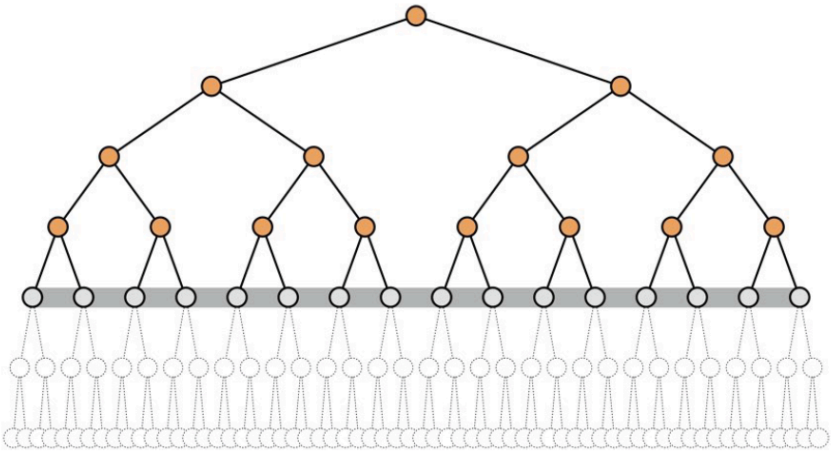
The frontier consists of nodes of similar depth (horizontal).
Search proceeds by exhausting one layer at a time.

BFS



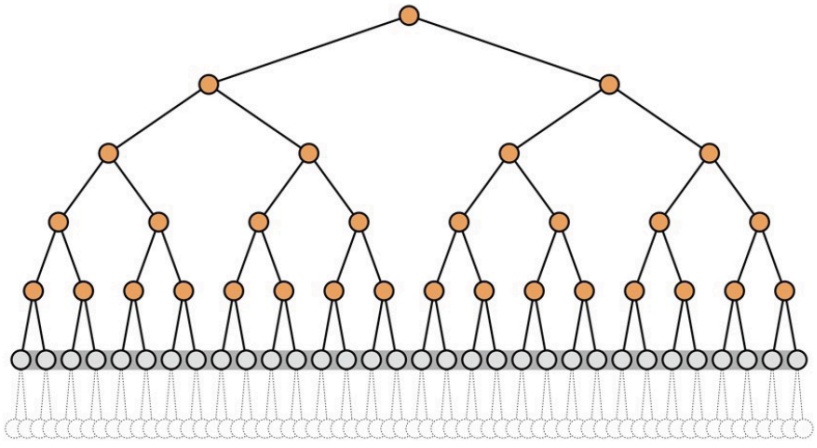
The frontier consists of nodes of similar depth (horizontal).
Search proceeds by exhausting one layer at a time.

BFS



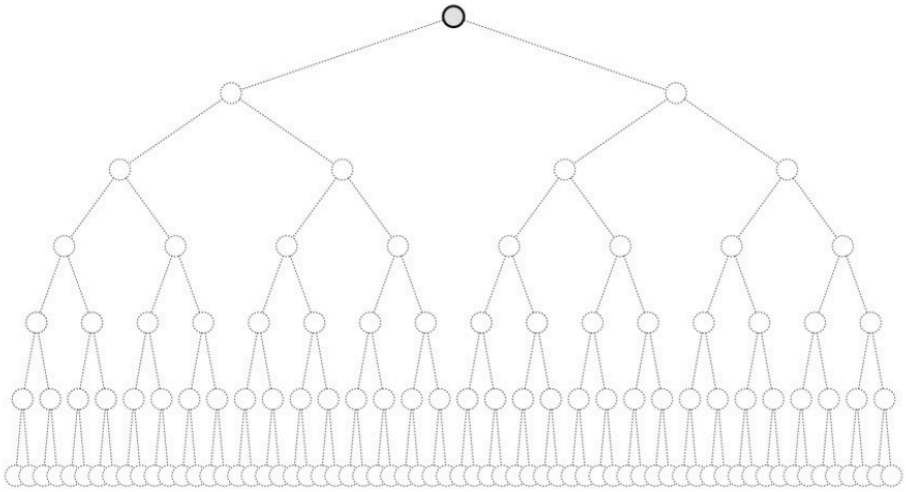
The frontier consists of nodes of similar depth (horizontal).
Search proceeds by exhausting one layer at a time.

BFS



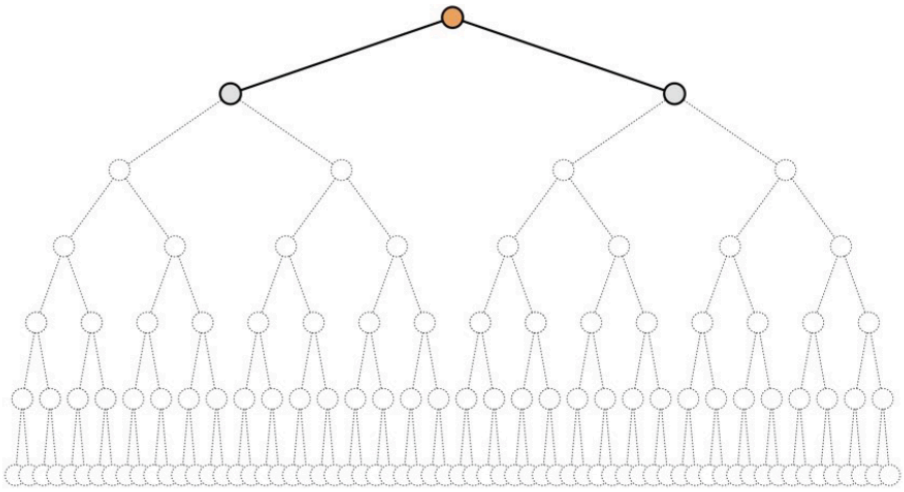
The frontier consists of nodes of similar depth (horizontal).
Search proceeds by exhausting one layer at a time.

UCS



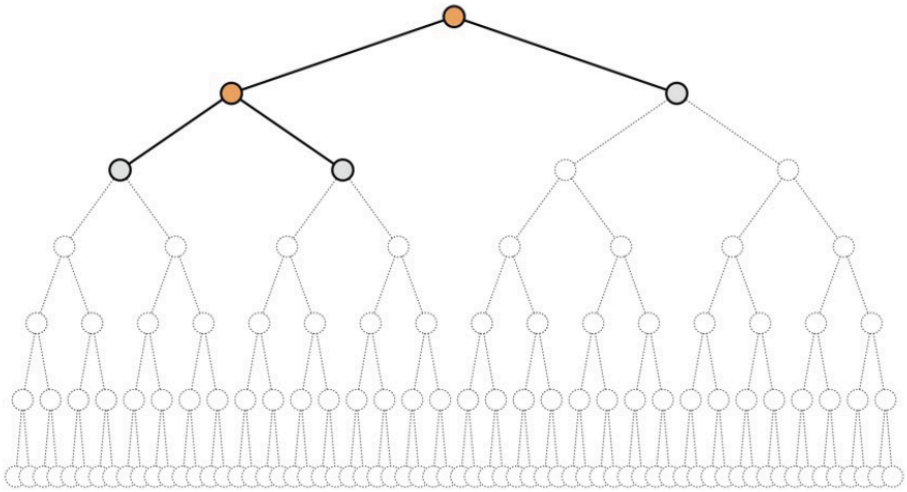
Searches layer by layer …
… with each layer organized by path cost.

UCS



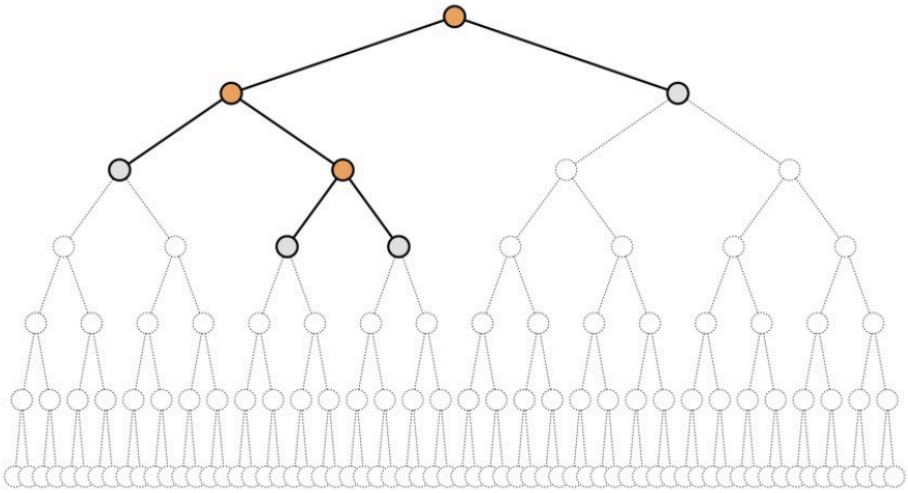
Searches layer by layer …
… with each layer organized by path cost.

UCS



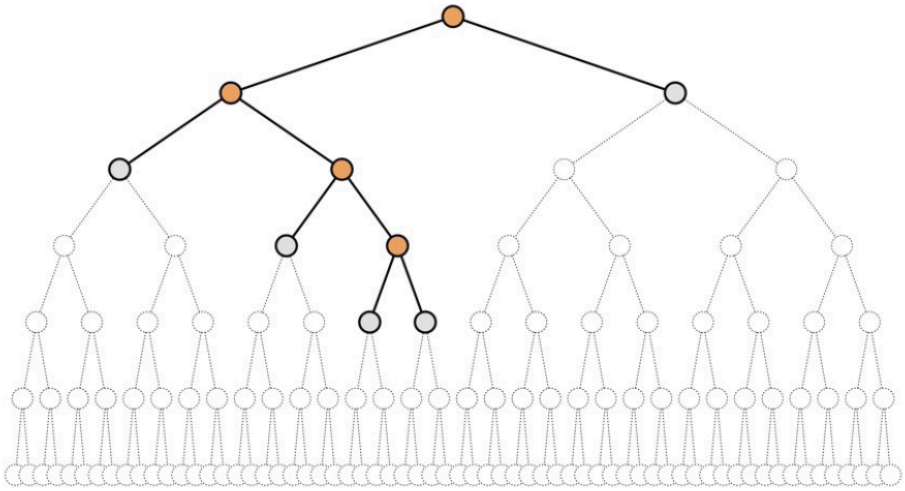
Searches layer by layer ...
... with each layer organized by path cost.

UCS



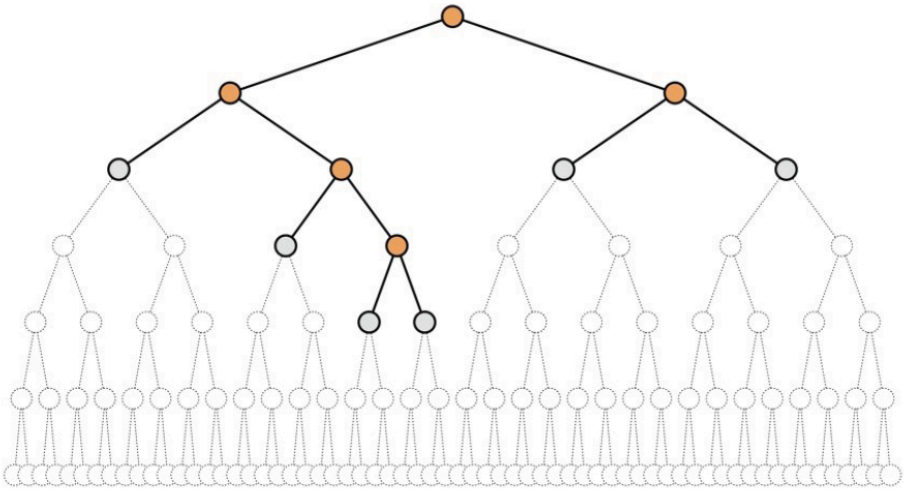
Searches layer by layer ...
... with each layer organized by path cost.

UCS



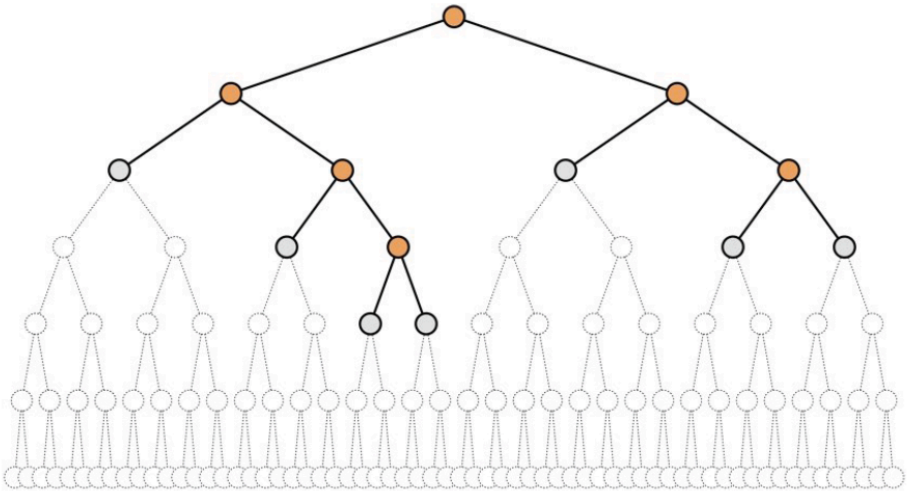
Searches layer by layer ...
... with each layer organized by path cost.

UCS



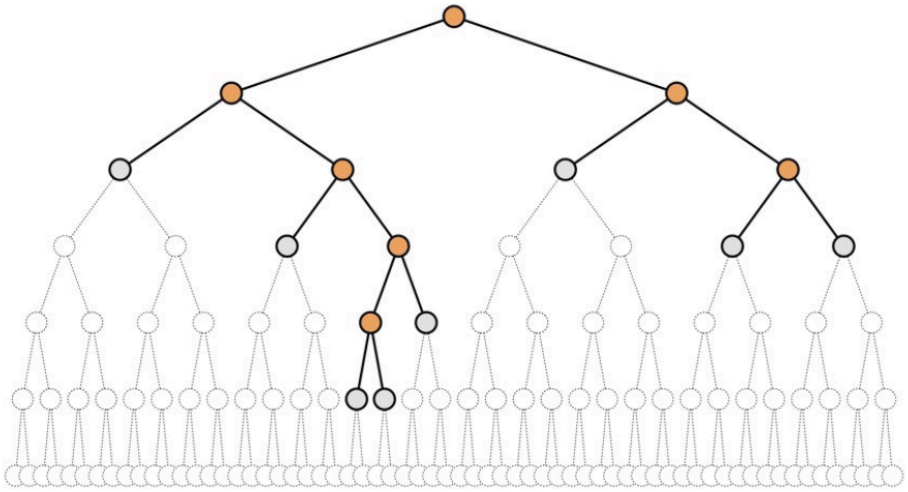
Searches layer by layer ...
... with each layer organized by path cost.

UCS



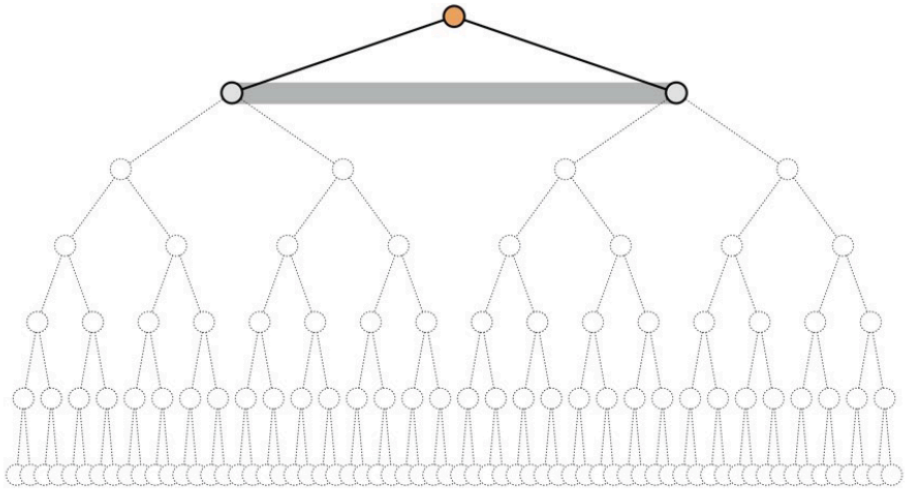
Searches layer by layer ...
... with each layer organized by path cost.

UCS



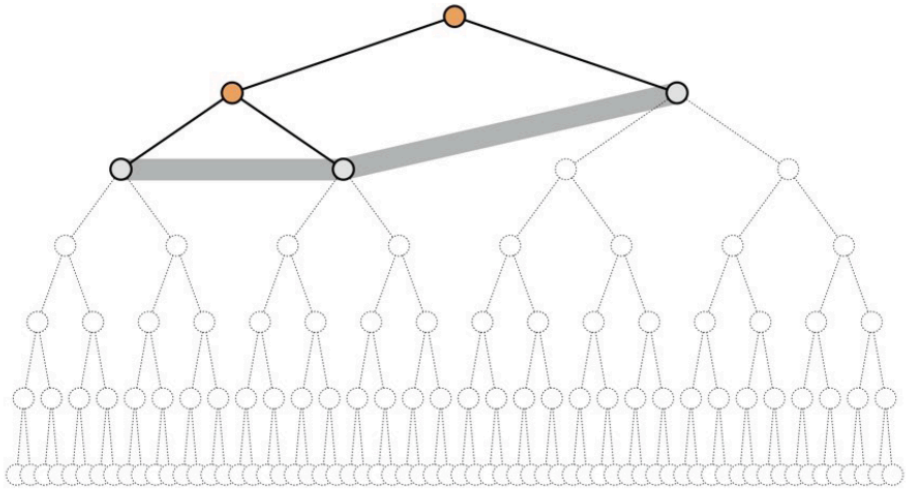
Searches layer by layer ...
... with each layer organized by path cost.

UCS



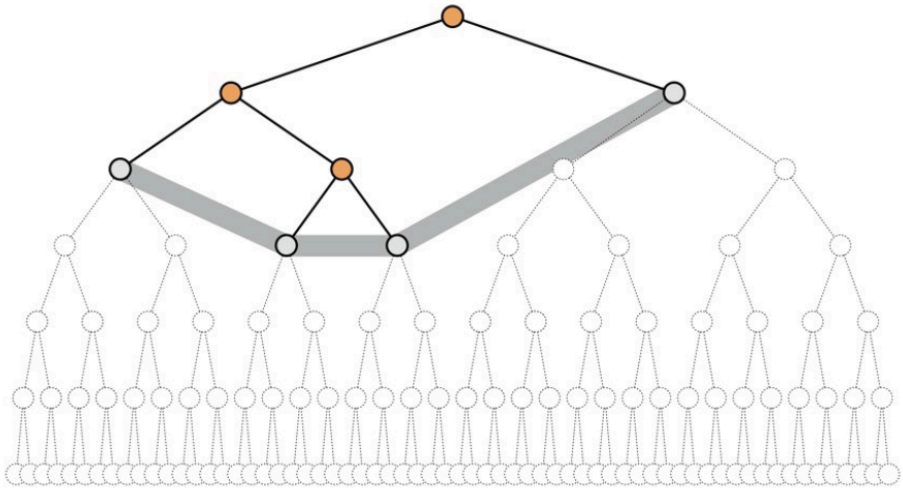
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



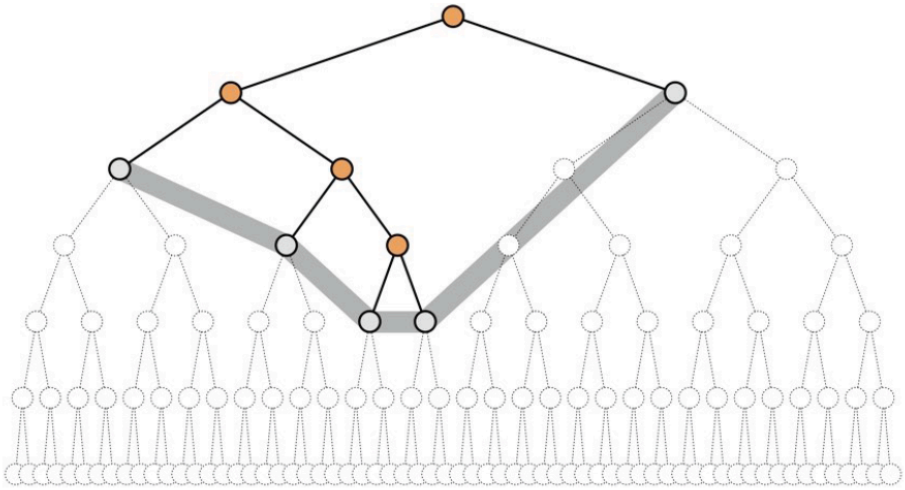
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



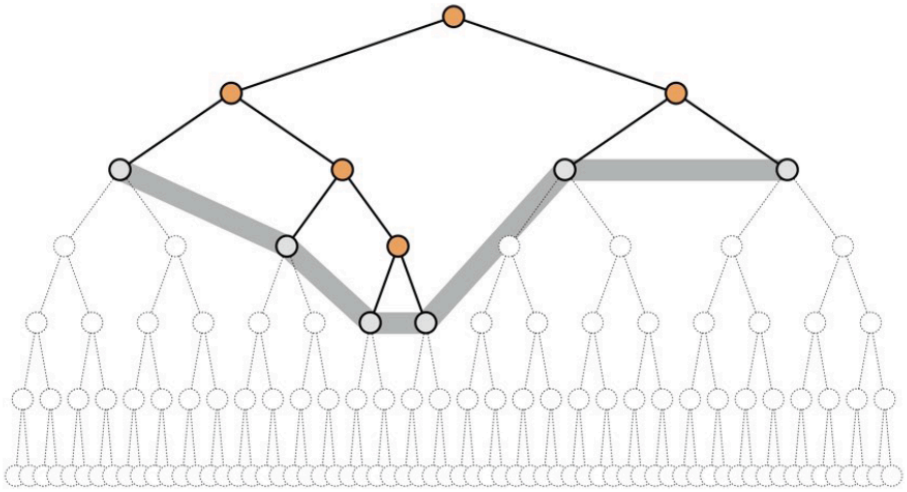
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



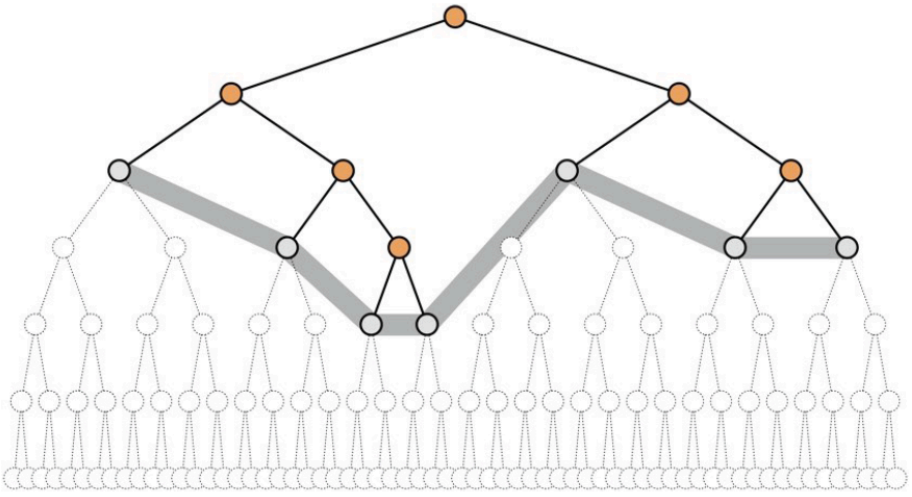
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



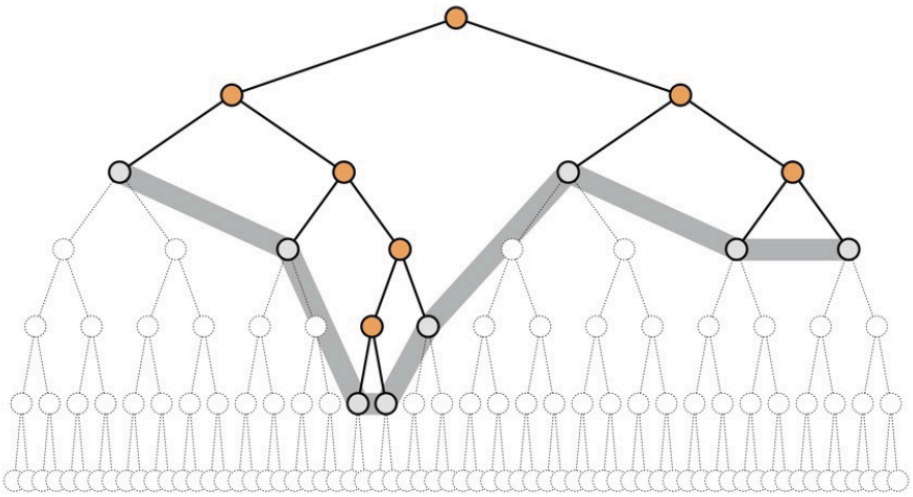
The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

UCS



The frontier consists of nodes of various depths (jagged).
Search proceeds by expanding the lowest-cost nodes.

Recap

We can organize the algorithms into pairs where the first proceeds by **layers**, and the other proceeds by **subtrees**

(1) Iterate on Node Depth:

- BFS searches **layers** of increasing node depth
- IDS searches **subtrees** of increasing node depth

(2) Iterate on Path Cost + Heuristic Function:

- A* searches **layers** of increasing path cost + heuristic function
- IDA* searches **subtrees** of increasing path cost + heuristic function

Recap

Which cost function?

- UCS searches layers of increasing **path cost** - ***g***
- Greedy best first search searches layers of increasing **heuristic function** - ***h***
- A* search searches layers of increasing **path cost** + **heuristic function**

$$f = g + h$$

Summary

- Before an agent can start searching for solutions, a **goal** must be identified and a well-defined **problem** must be formulated
- A problem consists of five parts:
  - the **initial state**
  - a set of **actions**
  - a **transition model** describing the results of those actions
  - a **goal test** function
  - a **path cost** function
- The environment of the problem is represented by a **state space**
  - A **path** through the state space from the initial state to a goal state is a **solution**
- Search algorithms treat states and actions as **atomic**: they do not consider any internal structure they might possess
- A general ***TREE-SEARCH*** algorithm considers all possible paths to find a solution, whereas a ***GRAPH-SEARCH*** algorithm avoids consideration of redundant paths

Summary

- Search algorithms are judged on the basis of **completeness**, **optimality**, **time complexity**, and **space complexity**
  - Complexity depends on ***b*** - the branching factor in the state space, and ***d*** - the depth of the shallowest solution
- **Uninformed search** methods have access only to the problem definition
  - **Breadth-first search (BFS)** expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity
  - **Uniform-cost search (UCS)** expands the node with lowest path cost, *g(n),* and is optimal for general step costs
  - **Depth-first search (DFS)** expands the deepest unexpanded node first, neither complete nor optimal, but has linear space complexity
  - **Depth-limited search (DLS)** adds a depth bound
  - **Iterative deepening search (IDS)** calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal for unit step costs, has time complexity comparable to breadth-first search, and has linear space complexity
  - **Bidirectional search** can enormously reduce time complexity, but it is not always applicable and may require too much space

Summary

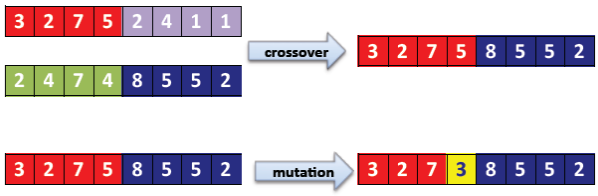
- **Informed search** methods may have access to a **heuristic** function ***h(n)*** that estimates the cost of a solution from ***n***
  - The generic **best-first search** algorithm selects a node for expansion according to an **evaluation function**
  - **Greedy best-first search** expands nodes with minimal h(n)
    - It is not optimal but is often efficient
  - **A∗ search** expands nodes with minimal ***f(n) = g(n) + h(n)***
    - A∗ is complete and optimal, provided that ***h(n)*** is admissible (for *TREE-SEARCH*) or consistent (for *GRAPH-SEARCH*)
    - The space complexity of A∗ is still prohibitive
- The performance of heuristic search algorithms depends on the ***quality of the heuristic function***

BIM309 Artificial Intelligence

Week 5 – Search Agents – Local Search

Outline

- Local Search
- Local Search Algorithms
  - Hill Climbing
  - Simulated Annealing
  - Local Beam Search
  - Genetic Algorithms



Local Search

- Search algorithms seen so far are designed to **explore search spaces systematically**
- **Problems:** observable, deterministic, known environments where the solution is a sequence of actions
- Real-World problems are more complex
- When a goal is found, the path to that goal constitutes a solution to the problem. But, depending on the applications, the path may or may not matter
- If the path does not matter/systematic search is not possible, then consider another class of algorithms → **Local Search**

Local Search

- In such cases, we can use iterative improvement algorithms, **Local Search**
- Also useful in pure **optimization problems** where the goal is to find the best state according to an **optimization function**
- ***Examples***
  - Integrated circuit design, telecommunications network optimization, airline flight scheduling, etc.
  - *N-puzzle or 8-queen*: what matters is the final configuration of the puzzle, not the intermediary steps to reach it

Local Search

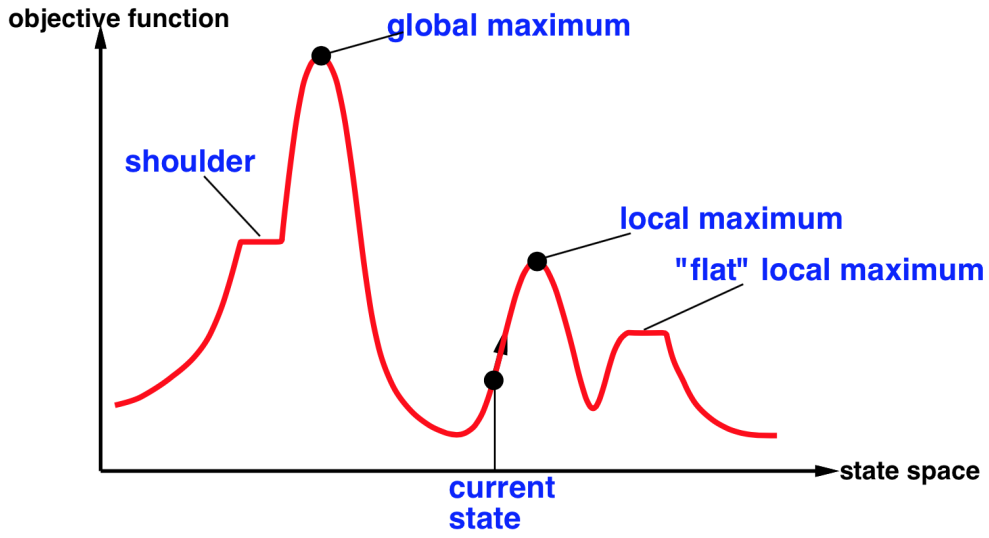
- ***Idea:*** **keep a single "current" state** *(rather than multiple paths)***, and try to improve it**
- Move only to the neighbors of that node
- ***Advantages***
  1. No need to maintain a search tree
  2. Use very little memory
  3. Can often find good enough solutions in continuous or large state spaces

Local Search

Local Search Algorithms

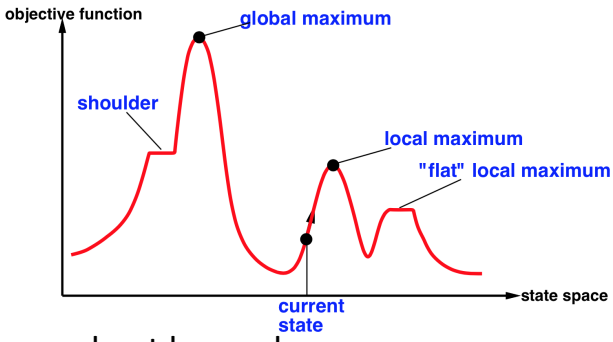
1. **Hill Climbing** (steepest ascent / descent)
2. **Simulated Annealing**: inspired by statistical physics
3. **Local Beam Search**
4. **Genetic Algorithms**: inspired by evolutionary biology

State Space Landscape for Local Search



State Space Landscape

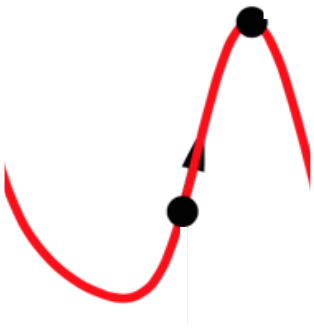
1. Hill Climbing



- Also called **Greedy Local Search**
- Looks only to immediate good neighbors and not beyond
- Search moves uphill:
  - moves in the direction of increasing elevation/value to find the top of the mountain
- Terminates when it reaches a **peak**
- Can terminate with a local maximum, global maximum or can get stuck and no progress is possible
- A **node** is **a state** and **a value**

"Like climbing Everest in thick fog with amnesia"

1. Hill Climbing



function HILL-CLIMBING(initialState)
returns State that is a local maximum

initialize current **with** initialState

loop do
neighbor = a highest-valued successor of current

if neighbor.value ≤ current.value:
return **current.state**

current = neighbor

1. Hill Climbing

A flat area of the state-space landscape (such as a flat local maximum from which no uphill exits, or a shoulder)

- Other variants of hill climbing include
  - **Sideways moves** escape from **plateaux** where best successor has same value as the current state
  - **Random-restart hill climbing** overcomes local maxima: keep trying! (either find a goal or get several possible solution and pick the max)
  - **Stochastic hill climbing** chooses at random among the uphill moves
- **Hill climbing** effective in general but depends on shape of the landscape
- Successful in many real-problems after a reasonable number of restarts

2. Simulated Annealing

- **Idea: Escape local maxima by allowing "downhill" moves**
  - But make them rarer as time goes on
- Combine hill climbing with a random walk in some way that yields both efficiency and completeness
- In metallurgy, **annealing** is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state
- Imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface
  - If we just let the ball roll, it will come to rest at a local minimum
  - If we shake the surface, we can bounce the ball out of the local minimum.
  - The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum
  - The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature)

3. Local Beam Search

- **Local beam search** keeps track of ***k*** states rather than one single state
  - Initially: Begin with ***k*** randomly generated states
  - Next: all the successors of all ***k*** states are generated
  - If any of successors is goal → finished
  - Else **select *k* best successors** from the complete list and repeat
- **Major difference with random-restart search**
  - In a random-restart search, each search process runs independently of the others
  - In a local beam search, useful information is passed among the parallel search threads (states)
- **Can suffer from lack of diversity among the *k* states** — they can quickly become concentrated in a small region (around the same part) of the state space
  - Help alleviate the problem of the states agglomerating, we can use **Stochastic Beam Search**, where we **choose *k* successors at random**

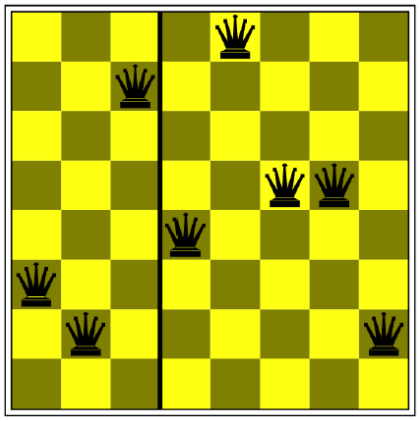
4. Genetic Algorithms

- **Genetic algorithms (GA)** is **a variant of stochastic beam search**
- Successor states are generated by combining ***two*** parents rather by modifying a single state
- The process is inspired by **natural selection**
  - Same as in stochastic beam search, except that now deal with sexual rather than asexual reproduction
- Starts with ***k* randomly generated states**, called **population**
- Each state is an **individual**
- An individual is usually represented by a **string** of **0**'s and **1**'s, or digits, in general a finite set
- The objective function is called **fitness function**: **better states have high values of fitness function**

4. Genetic Algorithms

- In the 8-queen problem, an individual can be represented by a string **digits** 1 to 8, that represents the position of the 8 queens in the 8 columns

[3 2 7 4 8 5 5 2]

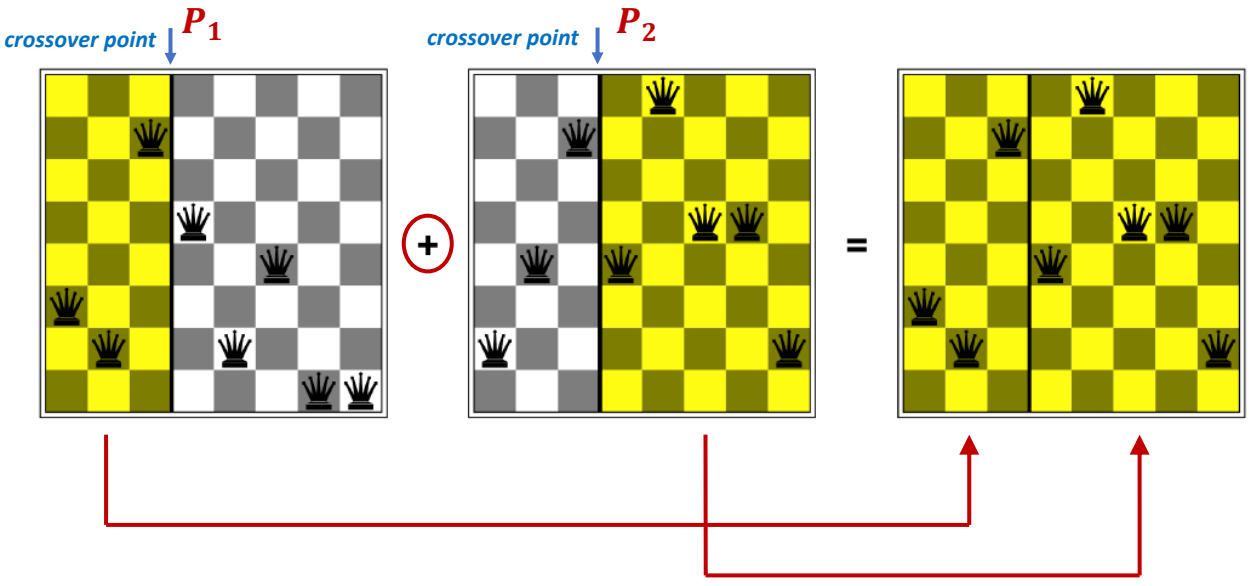


4. Genetic Algorithms

The objective function is called fitness function: better states have high values of fitness function

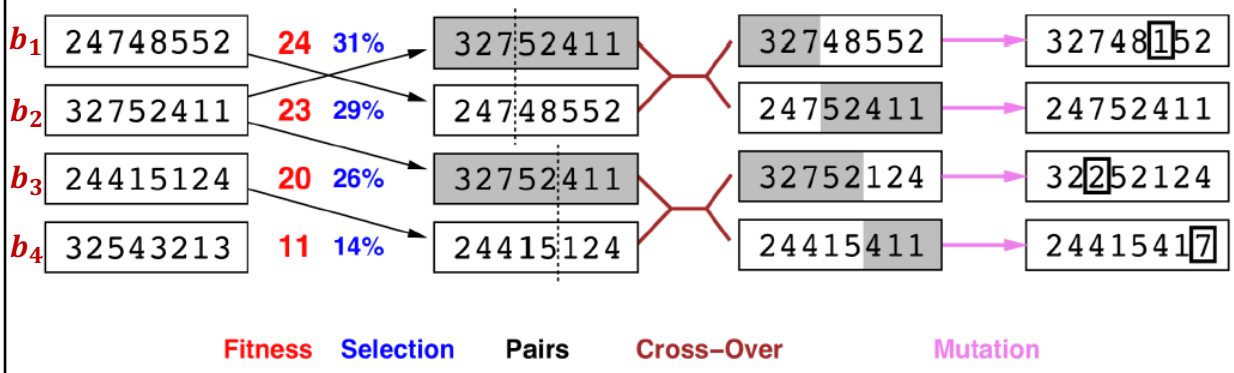
- Possible fitness function is **the number of non-attacking pairs of queens**
- Fitness function of the solution: 7+6+5+4+3+2+1 = **28**
- Pairs of individuals are selected at random for **reproduction** with respect to some probabilities
- A **crossover** point is chosen randomly in the string
- **Offspring** are created by crossing the parents at the crossover point
- Each element in the string is also subject to some **mutation** with a small probability

4. Genetic Algorithms: Crossover



4. Genetic Algorithms

Generate successors from pairs of states



4. Genetic Algorithms

The Basic Genetic Algorithm

1. Generate random population
2. Until the end condition is met, create a new population by repeating following steps
   - Evaluate the **fitness function**
   - **Select two parents** from a population, weighed by their fitness function values
   - With probability p_c **cross over the parents** to form a new offspring
   - With probability p_m **mutate new offspring**
   - Place new offspring in the new population
3. Return ***the best solution in current population***

4. Genetic Algorithms

```
function GENETIC-ALGORITHM(population, fitness-function)
    returns an individual

    repeat

        initialize new-population with ∅

        for i=1 to size(population) do

            x = random-select(population,fitness-function)
            y = random-select(population,fitness-function)
            child = cross-over(x,y)
            mutate (child) with a small random probability
            add child to new-population

        population = new-population

    until some individual is fit enough or enough time has elapsed

    return the best individual in population w.r.t. fitness-function
```

Summary

- So far, we keep the paths to the goal!
- For some problems (like 8-queens) we don't care about the path, we only care about the solution - Many real problems are of this form
- **Local search** algorithms operate using a single **current state** and generally move to neighbors of that state
- There is an **objective function** that tells the value of each state, and the goal has the highest value (global maximum)
- Algorithms like **Hill Climbing** try to move to a neighbour with the highest value
- Danger of being stuck in a **local maximum**. So some randomness is added to "shake" out of local maxima
- **Simulated Annealing**: Instead of the best move, take a random move and if it improves the situation then always accept, otherwise accept with a probability < 1
- A **genetic algorithm** is a stochastic hill-climbing search in which a large population of states is maintained
  - New states are generated by **mutation** and by **crossover**, which combines pairs of states from the population

Grading Policy

- Midterm: 20% → **16-20 November 2020**
- Homework: 10%
- Project: 30%
- Final: 40% → **11-22 January 2021**

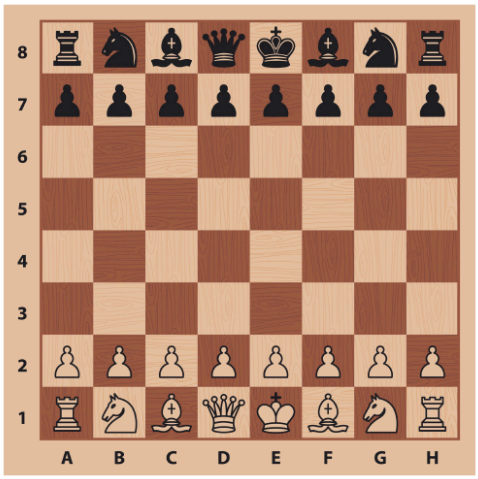
Note: **04 November 2020 - Deadline for Course Withdrawal!**

BIM309 Artificial Intelligence

Week 6 – Adversarial Search & Games

Outline

- Adversarial Search
- Minimax Algorithm
- Alpha-Beta Pruning
- Stochastic Games



Multiagent Environments

- Multiagent environments come in two versions:
  - ***cooperative*** and ***competitive***
- While problem solving in a multiagent environment, an agent's actions may depend on other agent's actions
- Typically, we are uncertain about the actions of other agents
  - But, we can often infer what actions they might take

Adversarial Search

- **Adversarial Search Problems ≡ Games**
- They occur in multi-agent ***competitive*** environments
  - There is an **opponent** we can't control planning again us!
  - Agent's goals are in conflict!! Each agent tries to optimize a ***utility***
  - ***Zero-sum-game***: One agent's gain in utility, is balanced by the opponent's loss in utility
- We will assume that there are only two agents, the environment is deterministic, and fully observable

Adversarial Search

- ***Game vs. Search:***
  - In games, we look for the optimal solution, which is not a sequence of actions anymore, but a **strategy** ("**policy**") that help us win the game
  - If opponent does ***a***, agent does ***b***, else if opponent does ***c***, agent does ***d***, etc.
- This can be expressed as a set of rules (i.e., implemented with rules)
  - However, tedious and fragile if hard-coded
- ***Good news:***

  Games are modeled as (1) **search problems, algorithms** and (2) use **heuristic evaluation functions**

two main ingredients to model and solve games in AI

Games: *Hard Topic in AI*

- Games are a big deal in AI
- Games are interesting to AI because they are too hard to solve
- Well defined problems, clear basic solution strategies, but typically very ***huge search space***
  - Chess has a branching factor of 35
  - If each player plays 50 moves, the game tree has $35^{100}\ nodes \approx 10^{154}\ nodes$
- Main work is in figuring out how to make "good" decisions in the game, even when the optimal decision cannot be computed (infeasible)
  - Key to solve games in AI → "***Approximation***"

Adversarial Search

Checkers

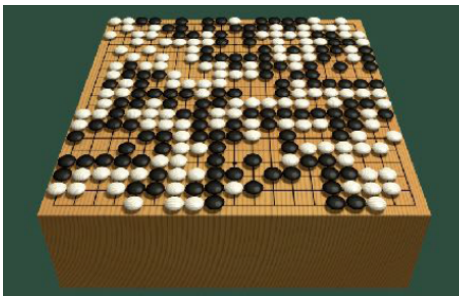
- In 1994: "**Chinook**" ended 40-year-reign of human world champion Marion Tinsley
  - World's first computer program to win a world championship against a human in checkers
  - Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 (443 billion) positions
- In 2007: Jonathan Schaeffer et al. "**Checkers Is Solved**"
  - Shows that perfect play by both players leads to a draw
  - https://spectrum.ieee.org/computing/software/checkers-solved

Adversarial Search

Chess

- In 1949: Caude E. Shannon in his paper "*Programming a Computer for Playing Chess*", suggested ***Chess* as an AI problem** for the community
  - Invented the **minimax algorithm** and suggested **heuristic evaluation functions** that actually help cutoff the search space
- In 1997: **Deep Blue** (the IBM intelligent machine) defeated human world champion Gary Kasparov in a six-game match
- In 2006: The undisputed world champion, Vladmir Kramnik, was defeated 4-2 by **Deep Fritz**

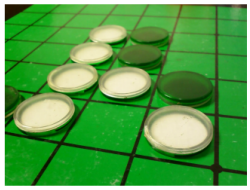
Adversarial Search



Go

- Players can put a stone anywhere on the board

  19 x 19 = 361 possibilities for the 1st move
- b > 300!
- In 2016: Google Deep mind Project **AlphaGo** beat both Fan Hui, the European Go champion and Lee Sedol the worlds best player
  - Deep neural networks



Othello / Reversi

- Several computer othello exists and human champions refuse to play against computers, that are too good (no chance to win)

Types of Games

	Deterministic	**Chance** *(Nondeterministic or Stochastic)*
Perfect Information *(Fully Observable)*	Chess, Checkers, Go, Othello	Backgammon, Monopoly
Imperfect Information *(Partially Observable)*	Battleships, Blind Tic-Tac-Toe	Bridge, Poker, Scrabble, Nuclear War

We are mostly interested in deterministic games, fully observable environments, **zero-sum**, where two agents act alternately

Zero-Sum Games

- Adversarial Games that involve pure competition
- Agents have different values on the outcomes
- One agent maximizes one single value, while the other minimizes it
- Each move in the game by one of the players is called a "***ply***"

One function: *one agents maximizes it and one minimizes it!*

Embedded Thinking…

Embedded thinking or backward reasoning!

- **One agent is trying to figure out what to do**
- How to decide?
  - He thinks about the consequences of the possible actions
  - He needs to think about his opponent as well...
  - The opponent is also thinking about what to do etc.
  - Each will imagine what would be the response from the opponent to their actions
- This leads to ***embedded thinking***!
- Assume that the players are rational: They will play the move that optimizes their performance measure

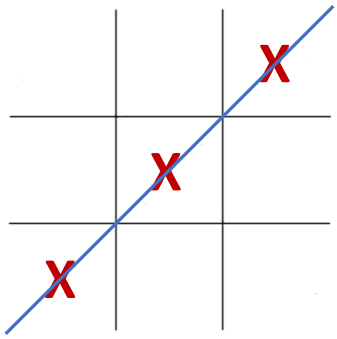
Formalization Adversarial Search Problems

A game can be formally defined as a kind of search problem

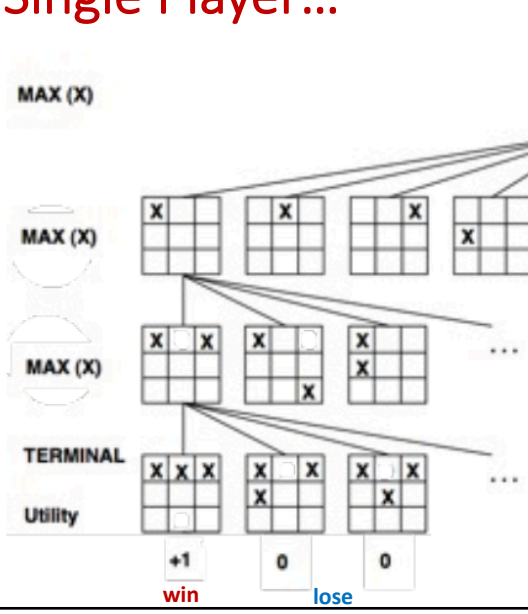
- The **initial state** - S_o - model of the state of the environment that the agent starts from
- **Player(s):** Whose turn is it?
  - Defines which player has the move in state ***s***. Usually taking turns
- **Actions(s):** returns the set of legal moves in state ***s***
- **Transition function:** $S \times A \rightarrow S$ defines the result of a move - *Result(s,a)*
- **Terminal test:** Is terminal? Game over?
  - ***True*** when the game is over, ***False*** otherwise
  - States where game ends are called **terminal states**
- ***Utility(s, p):*** **utility function** or **objective function** for a game that ends in terminal state ***s*** for player ***p***
  - In Chess, the outcome is a **win**, **loss**, or **draw** with values **+1**, **0**, **½**
  - For tic-tac-toe we can use a utility of **+1**, **-1**, **0**, if it's a **win**, **loss**, or **draw**, respectively

Single Player…

- Assume we have a tic-tac-toe with one player
- Let's call him ***Max*** and have him play ***three moves*** only for the sake of the example



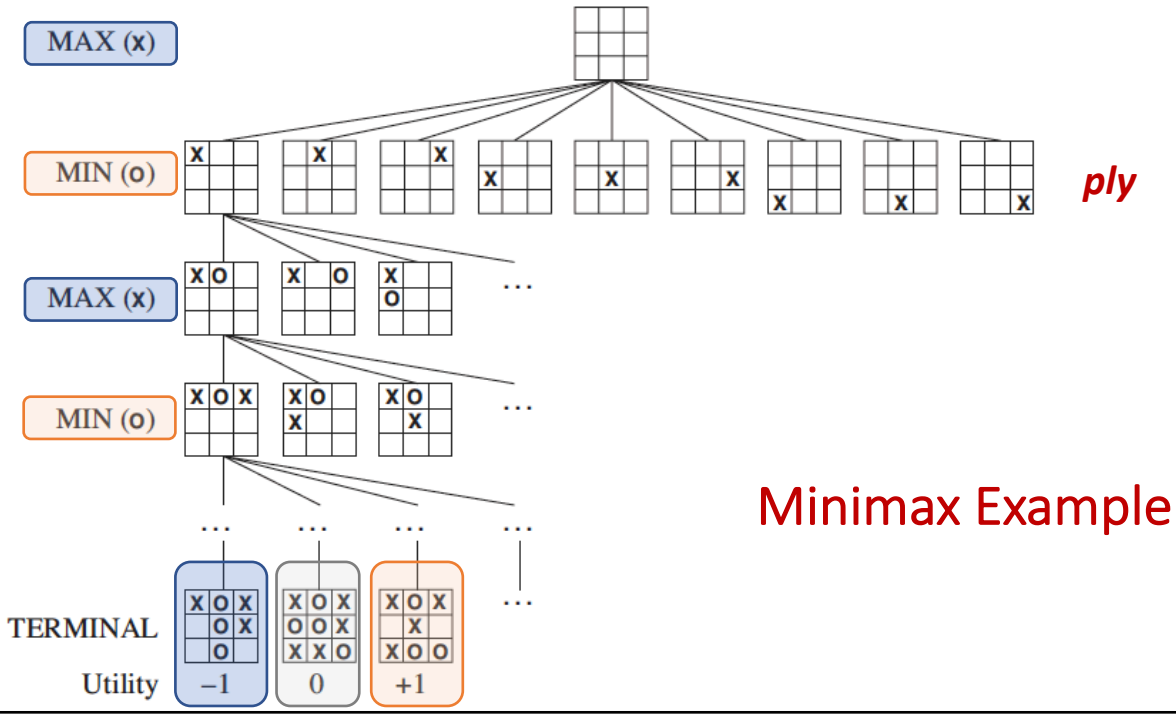
Single Player…



In the case of one player, nothing will prevent Max from winning (choose the path that leads to the desired utility, here 1), unless there is another player who will do everything to make Max lose, let's call him *Min*

Adversarial Search: Minimax

- Two players: ***Max*** and ***Min***
- Players alternate turns
- Max moves first
- Max maximizes results
- Min minimizes the result
- Compute each node's minimax value's the best achievable utility against an optimal adversary
- ***Minimax value*** ≡ ***best achievable payoff against best play***



Minimax Example

Adversarial Search: Minimax

- Aims to find the optimal strategy for Max to win:
  - Depth-first search of the game tree
  - An optimal leaf node could appear at any depth of the tree
  - ***Minimax principle:*** compute the utility of being in a state, assuming both players play optimally from there until the end of the game
  - *Idea of the minimax algorithm:* Propagate **minimax** values up the tree once terminal nodes are discovered
    - Go all the way down to the tree and down to the leaves in the tree

The Minimax Algorithm

- Minimax algorithm is designed to determine the optimal strategy for MAX: *Perfect play for deterministic, perfect-information games*
- Idea: choose move to position with ***highest minimax value*** ≡ **best achievable payoff against best play**
- An optimal procedure
  1. Depth-first search of the game tree
  2. Apply utility (payoff) function to each leaf
  3. Back-up values from leaves through branch nodes:
     - A MAX node computes the max of its child values
     - A MIN node computes the min of its child values
  4. At root: Choose move leading to the child of highest value

Adversarial Search: Minimax

- If state is **terminal node**: Value is ***utility(state)***
- If state is **MAX node**: Value is ***highest value of all successor node values*** (children)
- If state is **MIN node**: Value is ***lowest value of all successor node values*** (children)
- ***Recursive Definition:***

For a state s minimax(s) =

$$\begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Max} \\ min_{a \in Actions(s)} \text{ minimax(Result(s,a))} & \text{if Player(s) = Min} \end{cases}$$

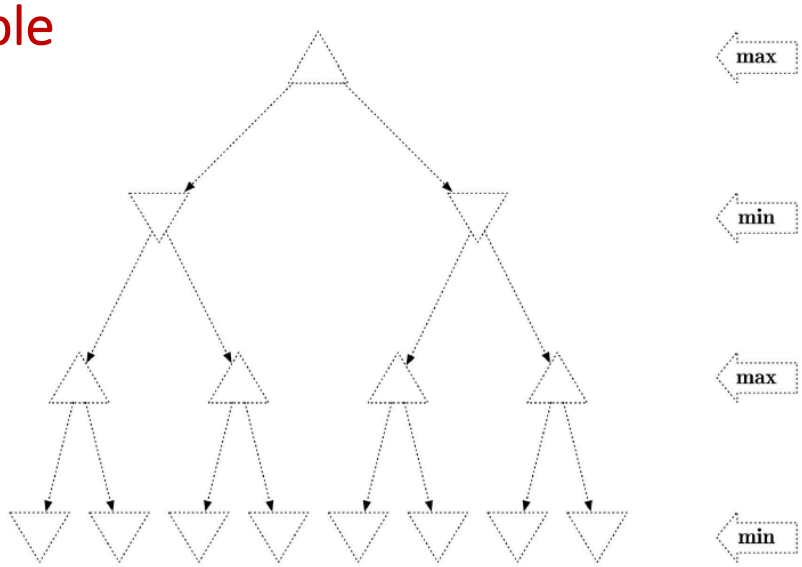
The Minimax Algorithm

```
/* Find the child state with the lowest utility value */

function MINIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

    return ⟨minChild, minUtility⟩
```

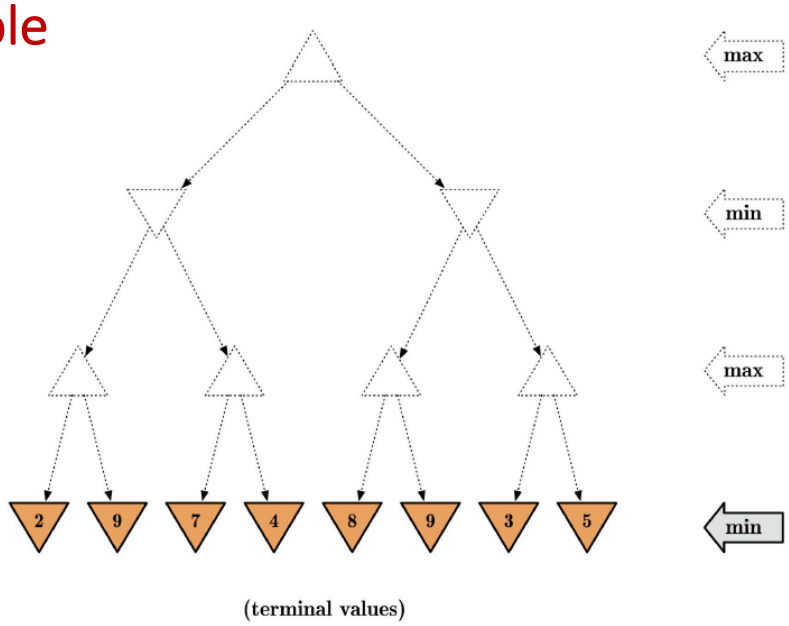
```
/* Find the child state with the highest utility value */

function MAXIMIZE(state)
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */

function DECISION(state)
    returns STATE :

    ⟨child, _⟩ = MAXIMIZE(state)

    return child
```

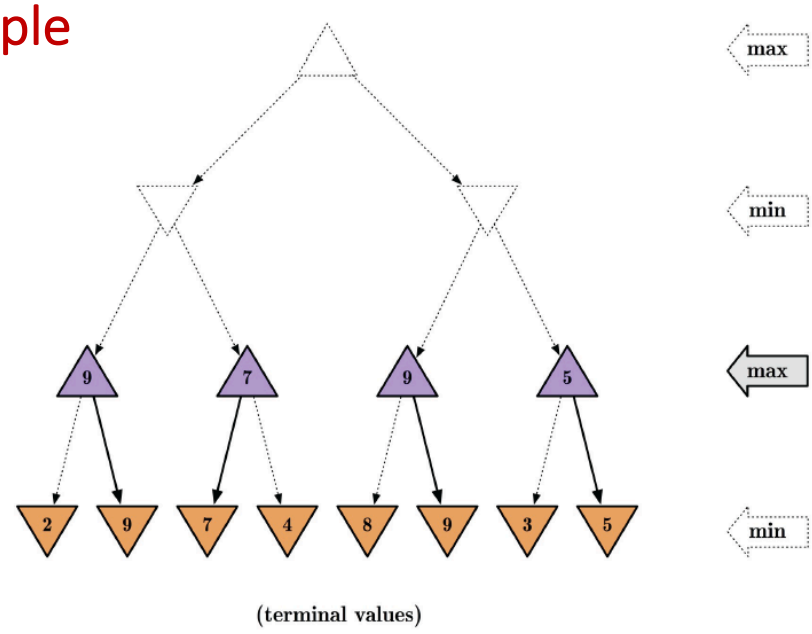
Minimax Example



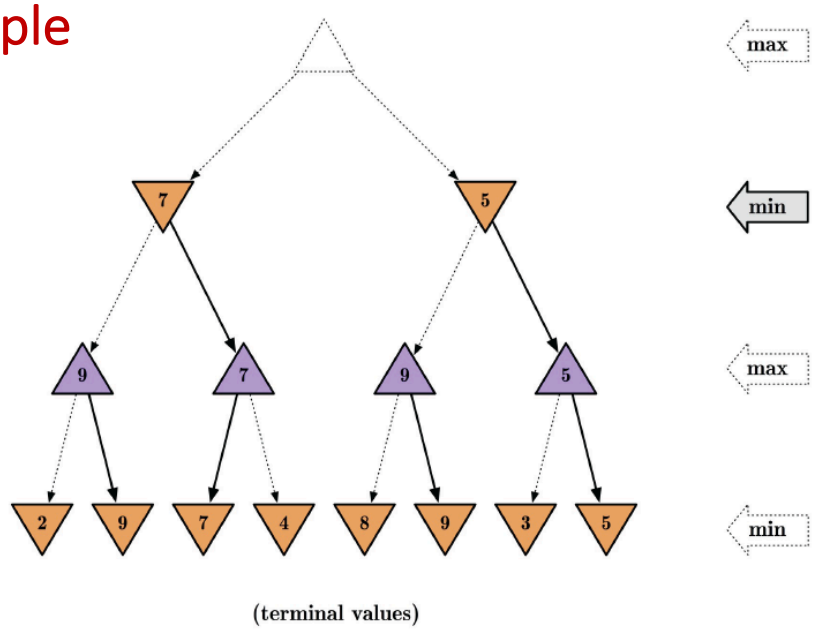
Minimax Example



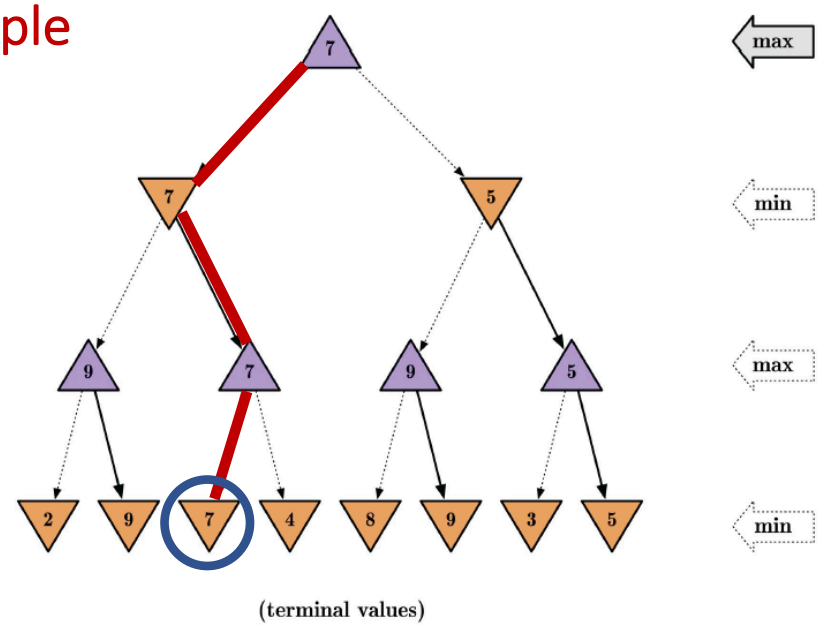
Minimax Example



Minimax Example



Minimax Example



Properties of Minimax

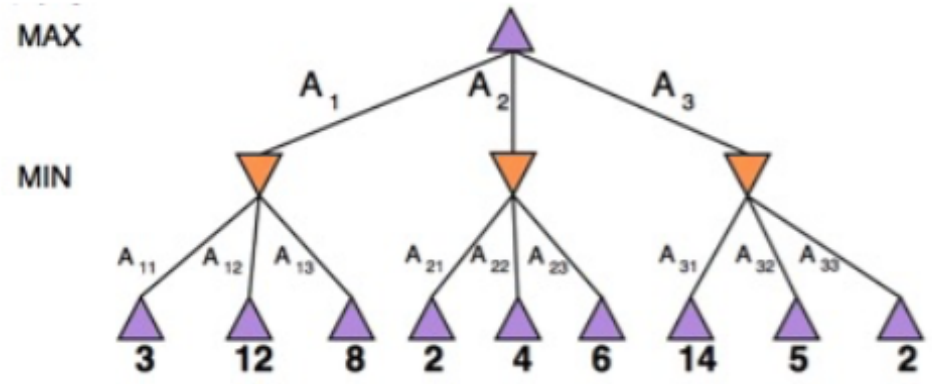
- ***Optimal*** (opponent plays optimally)
- ***Complete*** (finite search tree – possibly huge)
- ***DFS space***: $O(bm)$
- ***DFS time***: $O(b^m)$
  - **Tic-Tac-Toe**
    - $b \approx 5$ legal moves on average, total of 9 moves (9 plies), so $m = 9$
    - $5^9 = 1{,}953{,}125$
    - $9! = 362{,}880$ terminal nodes
  - **Chess**
    - $b \approx 35$ (average branching factor)
    - $d \approx 100$ (depth of game tree for a typical game)
    - $b^d \approx 35^{100} \approx 10^{154}$ nodes
  - **Go** branching factor starts at 361 (19×19 board)

Real Games are Hard!!

Case of Limited Resources

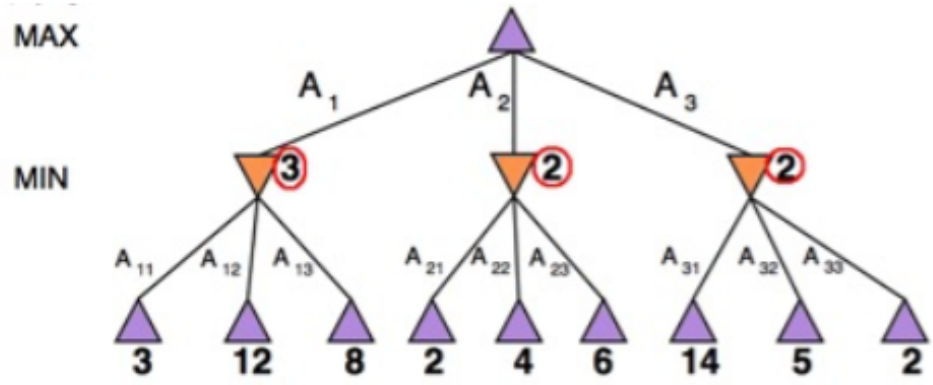
- ***Problem:*** In real games, **we are limited in time, so we can't search the leaves**
  - Complete Minimax search is infeasible
- To be practical and run in a reasonable amount of time, minimax can only search to some depth
- But, more plies make a big difference
  - You can ***plan better*** when you know more about the different consequences of those choices of actions
- ***Solution:*** *(for reducing the search space)*
  1. Replace terminal utilities with an evaluation function for non-terminal positions
  2. Use Iterative Deepening Search (IDS)
  3. Use pruning: Eliminate large parts of the tree

$\alpha - \beta$ pruning



A two-ply game tree

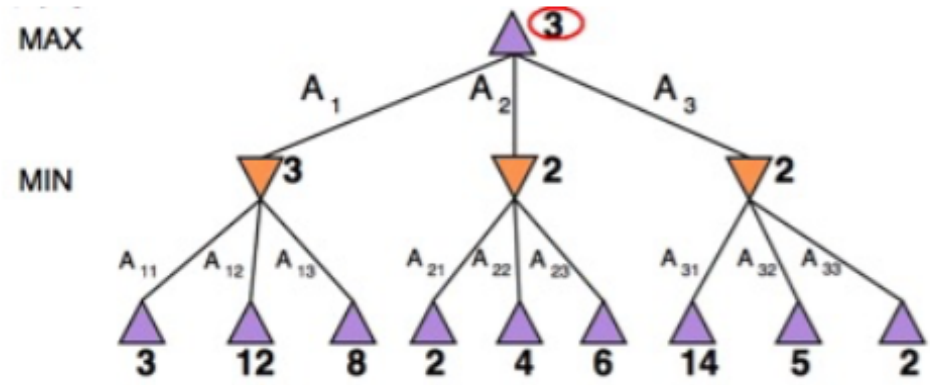
$\alpha - \beta$ pruning



A two-ply game tree

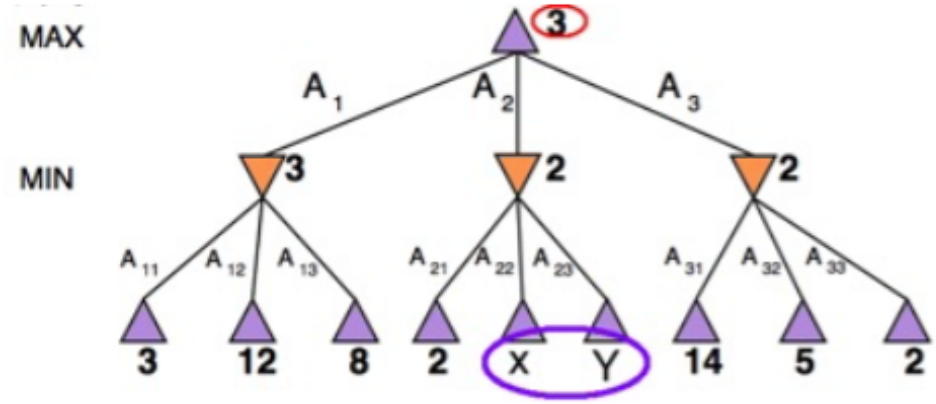
$\alpha - \beta$ pruning

Do we really need to explore the whole tree to find a minimax strategy ?



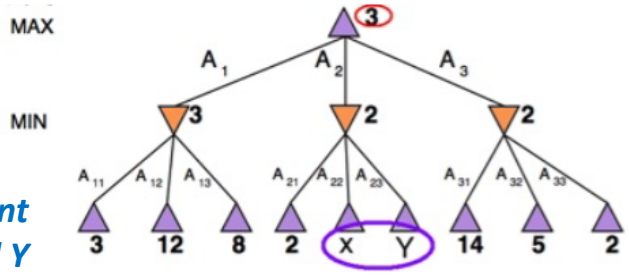
A two-ply game tree

$\alpha - \beta$ pruning



Which values are necessary?

$\alpha - \beta$ pruning



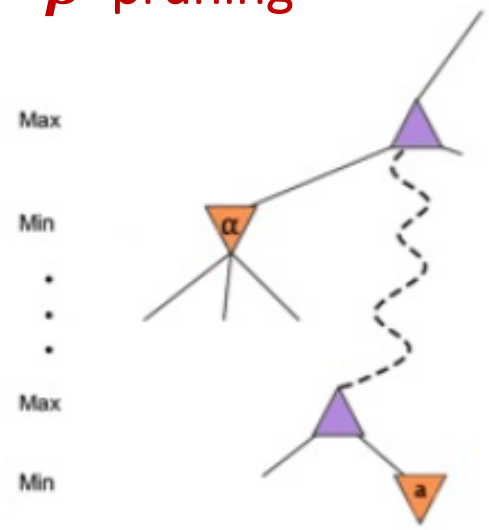
Minimax decisions are independent of the values of X and Y

$$Minimax(root) = max(min(3,12,8), min(2,X,Y), min(14,5,2))$$
$$= max(3, min(2,X,Y), 2)$$
$$= max(3,Z,2) \qquad \text{where } Z = min(2,X,Y) \leq 2$$
$$= 3$$

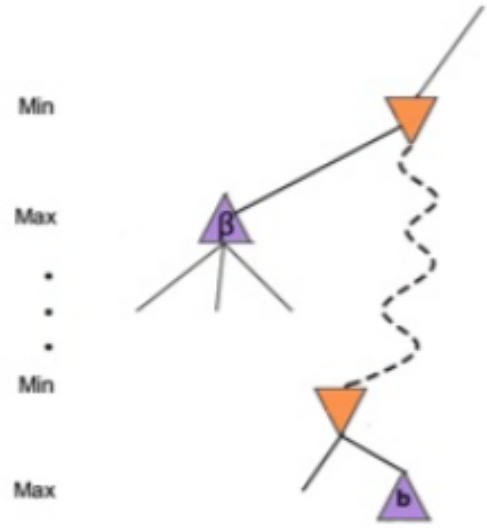
$\alpha - \beta$ pruning

- **Strategy:** Just like minimax, it performs a DFS
- **Parameters:** Keep track of two bounds
  - α : **largest** value for **MAX** across seen children (current lower bound on MAX's outcome)
  - β : **lowest** value for **MIN** across seen children (current upper bound on MIN's outcome)
- **Initialization:** $\alpha = -\infty, \quad \beta = \infty$
- **Propagation:** Send α, β values ***down*** during the search to be used for pruning
  - Update α, β values by ***propagating upwards*** values of terminal nodes
  - Update α only at **MAX nodes** and update β only at **MIN nodes**
- **Pruning:** Prune any remaining branches whenever $\alpha \geq \beta$

$\alpha - \beta$ pruning



If α is better than **a** for **Max**, then **Max** will avoid it, that is prune that branch



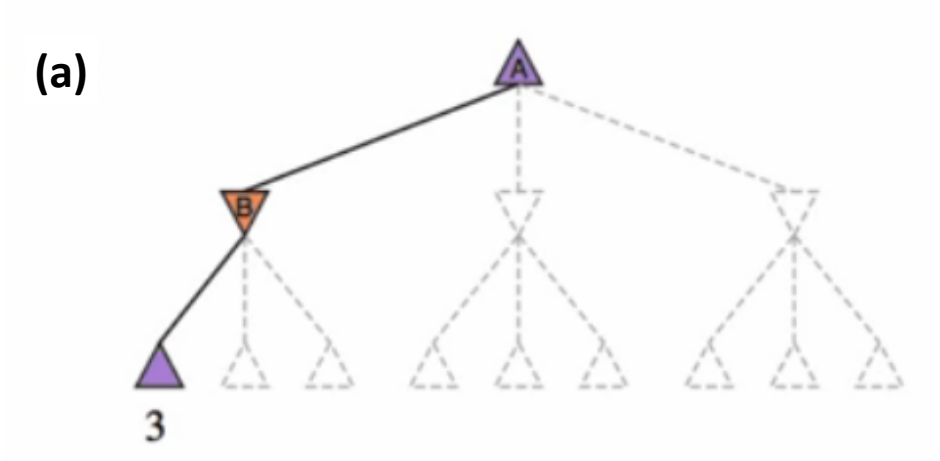
If β is better than **b** for **Min**, then **Min** will avoid it, that is prune that branch

Example: $\alpha - \beta$ pruning

MAX

MIN

(a)

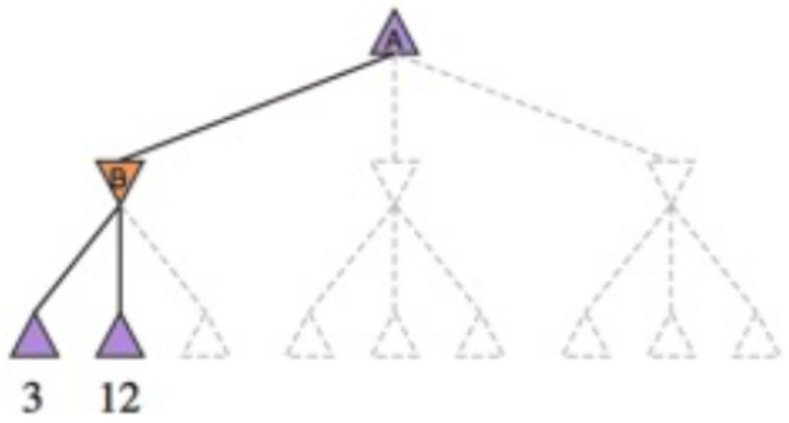


Example: $\alpha - \beta$ pruning

MAX

MIN

(b)

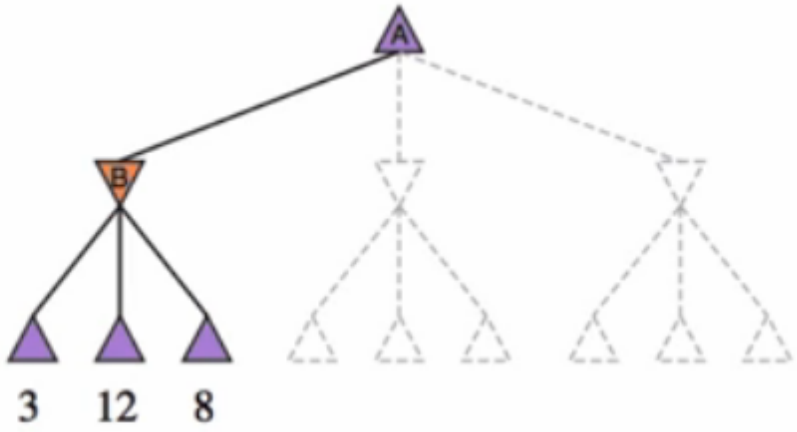


Example: $\alpha - \beta$ pruning

MAX

MIN

(c)

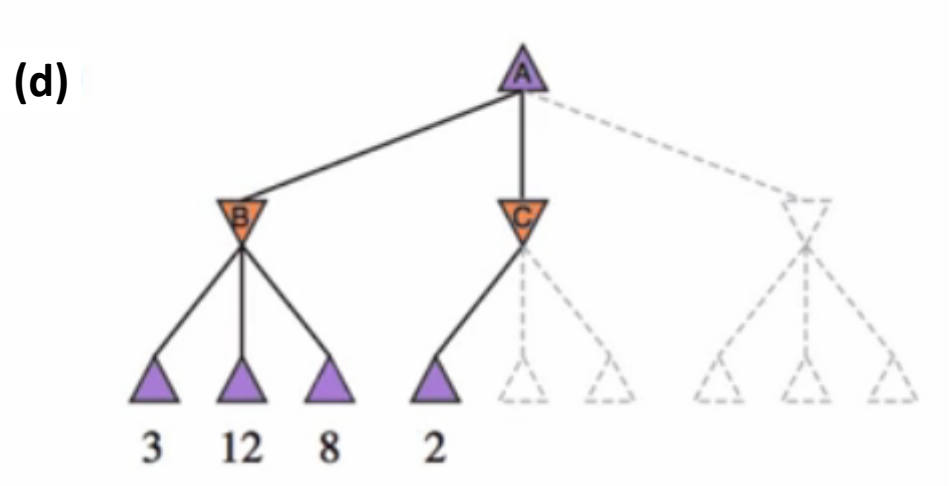


Example: $\alpha - \beta$ pruning

MAX

MIN

(d)

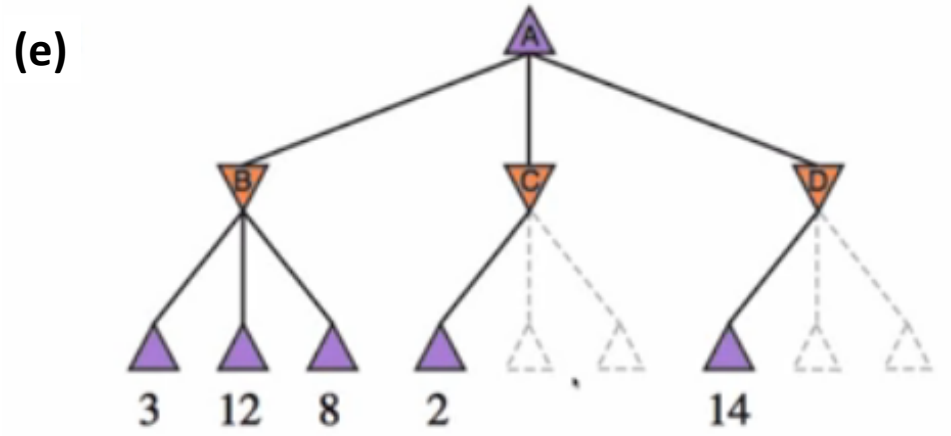


Example: $\alpha - \beta$ pruning

MAX

MIN

(e)

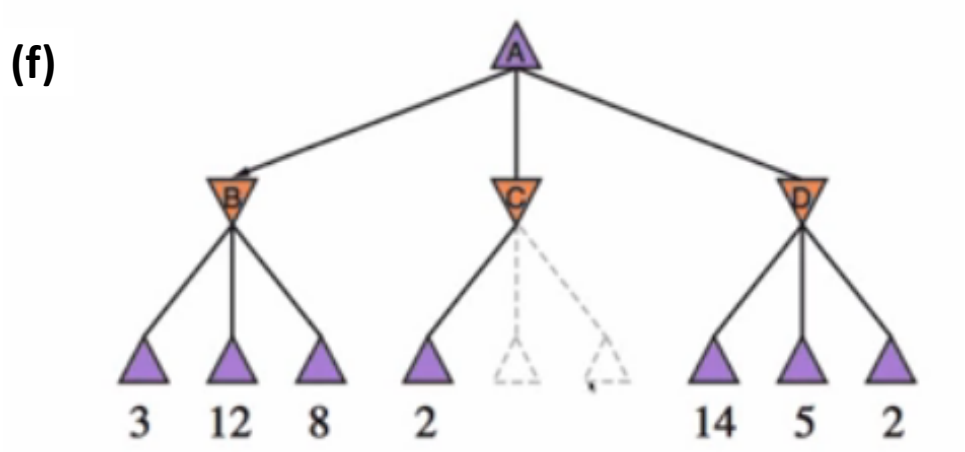


Example: $\alpha - \beta$ pruning

MAX

MIN

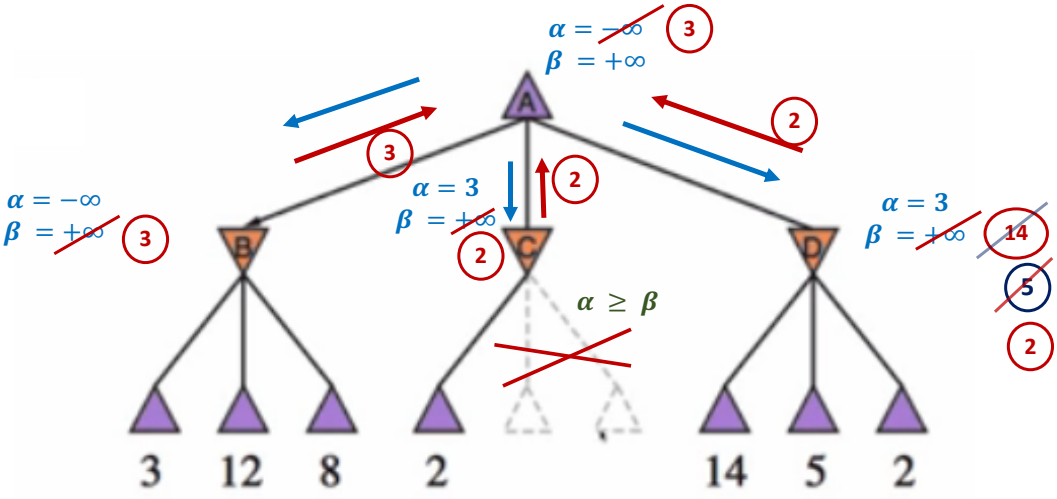
(f)



Example: $\alpha - \beta$ pruning

MAX

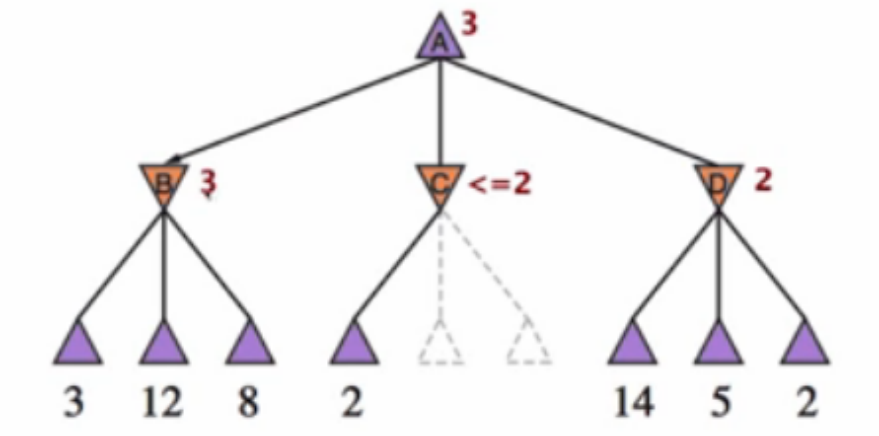
MIN



Example: $\alpha - \beta$ pruning

MAX

MIN



$\alpha - \beta$ pruning Algorithm

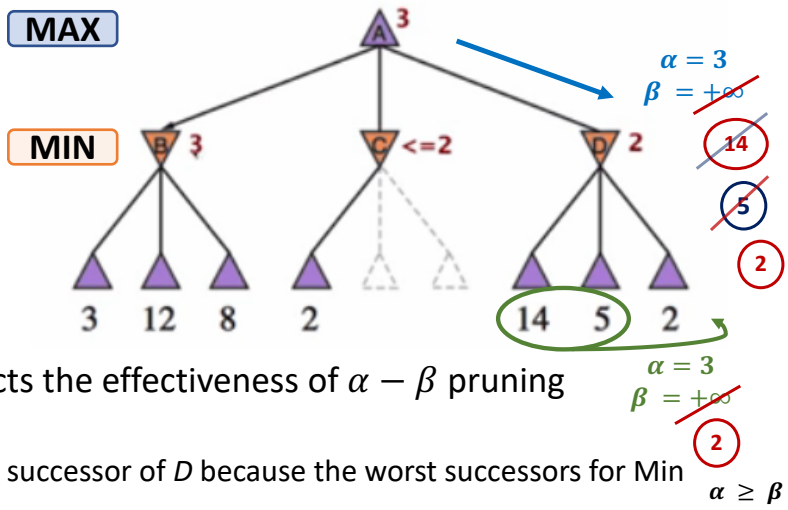
```
/* Find the child state with the lowest utility value */
function MINIMIZE(state, α, β)                                   MIN
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨minChild, minUtility⟩ = ⟨NULL, ∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MAXIMIZE(child, α, β)

        if utility < minUtility:
            ⟨minChild, minUtility⟩ = ⟨child, utility⟩

        if minUtility ≤ α:                                       Cut
            break
        if minUtility < β:                                       Update β
            β = minUtility
    return ⟨minChild, minUtility⟩
```

```
/* Find the child state with the highest utility value */
function MAXIMIZE(state, α, β)                                   MAX
    returns TUPLE of ⟨STATE, UTILITY⟩ :

    if TERMINAL-TEST(state):
        return ⟨NULL, EVAL(state)⟩

    ⟨maxChild, maxUtility⟩ = ⟨NULL, −∞⟩

    for child in state.children():
        ⟨_, utility⟩ = MINIMIZE(child, α, β)

        if utility > maxUtility:
            ⟨maxChild, maxUtility⟩ = ⟨child, utility⟩

        if maxUtility ≥ β:                                       Cut
            break
        if maxUtility > α:                                       Update α
            α = maxUtility
    return ⟨maxChild, maxUtility⟩
```

```
/* Find the child state with the highest utility value */
function DECISION(state)
    returns STATE :                                              α  β

    ⟨child, _⟩ = MAXIMIZE(state, −∞, ∞)

    return child
```

$\alpha \geq \beta$ is the condition to prune (cut) for both maximize and minimize functions

Move Ordering



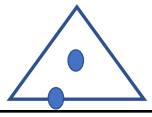
- It does matter as it affects the effectiveness of $\alpha - \beta$ pruning
- *Example*
  - We could not prune any successor of *D* because the worst successors for Min were generated first
  - If the third one (leaf 2) was generated first we would have pruned the two others (14 and 5)
- *Idea of ordering:* examine first successors that are likely best

Move Ordering

- **Worst ordering:** no pruning happens (best moves are on the right of the game tree)
  - **Complexity** $O(b^m)$ - which makes the run time of a search with pruning the same as plain Minimax
- **Ideal ordering:** lots of pruning happens (best moves are on the left of the game tree)
  - **Complexity** $O(b^{m/2})$ (in practice) - solves tree twice as deep as minimax in the same amount of time
  - The search can go deeper in the game tree
  - In Deep Blue, they found that alpha beta pruning meant the average branching factor at each node about 6 instead of 35
- **How to find good orderings?**
  - Remember the best moves from shallowest nodes
  - Order the nodes so as the best are checked first
  - Use domain knowledge: Need to select "Killer moves", which requires domain knowledge
    - E.g., for chess, try order: *captures* first, then *threats*, then *forward* moves, *backward* moves
  - Bookkeep the states, they may repeat! ("Transposition table")

Real-Time Decisions

- **Minimax:** generates the entire game search space
  - Real games are too large to enumerate tree
- $\alpha - \beta$ **algorithm:** prune large chunks of the trees
- BUT $\alpha - \beta$ still has to go all the way to the leaves of the tree (terminal)
- Impractical in real-time (moves has to be done in a reasonable amount of time)
- ***Solution:*** bound the depth of search (cut off search) and replace ***utiliy(s)*** with ***eval(s)***, an evaluation function to **estimate** value of current board configurations – turns non-terminal nodes into terminal nodes



Real-Time Decisions

- ***eval(s)*** is a heuristic at state ***s***
  - E.g., **Tic-Tac-Toe**:
    h(n) = [# of 3 lengths that are left open for player A] – [# of 3 lengths that are left open for player B]
  - E.g., Alan Turing's function for **Chess**: *h(n) = A(n) - B(n),* where
    - *A(n)* is the sum of the point value for player A's pieces
    - *B(n)* is the sum for player B's piece values
    - Each piece has a material value →

Pawn	1.0
Knight	3.0
Bishop	3.25
Rook	5.0
Queen	9.0

- An ideal evaluation function should rank terminal states in the same way as the utility function, but must be fast
- Typical to define features, make the function a linear weighted sum of the features
- Use domain knowledge to craft the best and useful features

Real-Time Decisions

- Most evaluation functions are specified as a weighted sum of features
- How does it work?
  - Select useful features $f_1(s), f_2(s), \ldots, f_n(s)$
    - Typical evaluation functions use features of the state
    - **Chess:** # of pieces on board (#white pawns, #black pawns, #white queens, #black queens, …)
  - Weighted linear function:

$$eval(s) = \sum_{i=1}^{n} w_i f_i(s)$$

  - Learn w_i from examples
  - Deep blue uses about 6,000 features in its evaluation function!

Pawn	1.0
Knight	3.0
Bishop	3.25
Rook	5.0
Queen	9.0

value of pieces

Stochastic Games

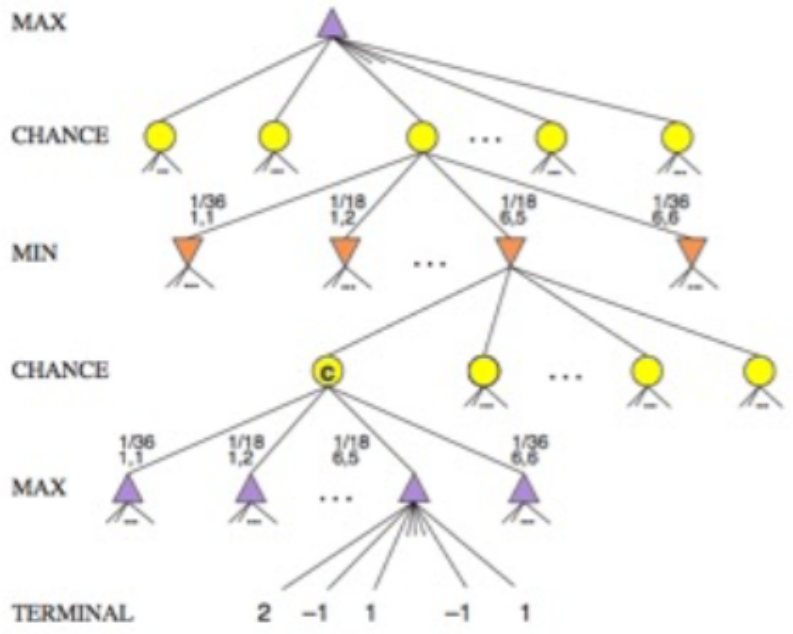
	Deterministic	**Chance** *(Nondeterministic or Stochastic)*
Perfect Information *(Fully Observable)*	Chess, Checkers, Go, Othello	Backgammon, Monopoly
Imperfect Information *(Partially Observable)*	Battleships, Blind Tic-Tac-Toe	Bridge, Poker, Scrabble, Nuclear War

Stochastic Games

- Include a random element (e.g., throwing a die, shuffling of cards)
- Include **chance nodes**

- **Backgammon:** old board game combining skills and chance
  - The goal is that each player tries to move all of his pieces off the board before his opponent does

Stochastic Games



Partial game tree for Backgammon

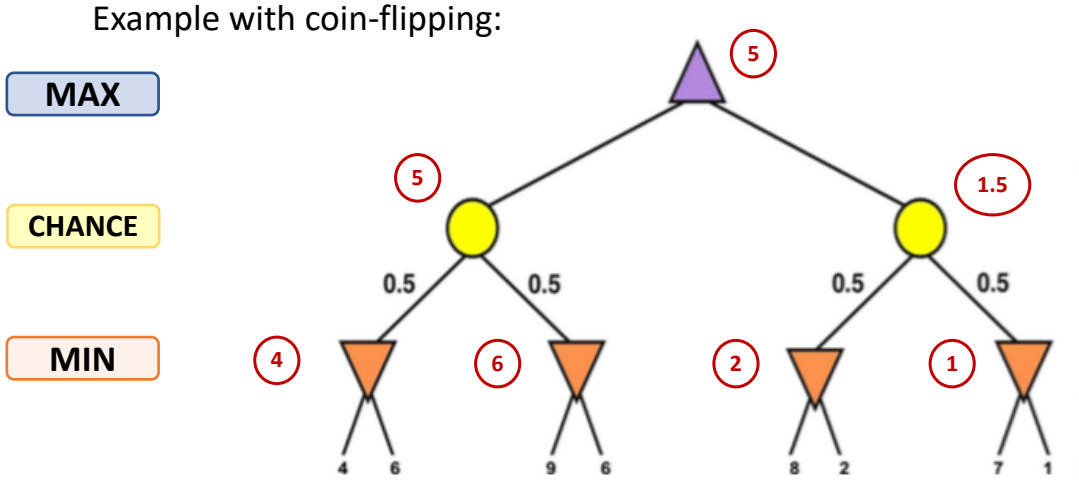
Stochastic Games

Algorithm **Expectiminimax** generalized Minimax to handle chance nodes as follows:

- If state is a **Max** node then
  return the **highest** Expectiminimax-Value of Successors(state)
- If state is a **Min** node then
  return the **lowest** Expectiminimax-Value of Successors(state)
- If state is a **chance** node then
  return **average** of Expectiminimax-Value of Successors(state)

Stochastic Games

Example with coin-flipping:



Expectiminimax

For a state s:

Expectiminimax(s) =

$$\begin{cases} Utility(s) & \text{if Terminal-test(s)} \\ max_{a \in Actions(s)} \text{ Expectiminimax(Result(s,a))} & \text{if Player(s) = Max} \\ min_{a \in Actions(s)} \text{ Expectiminimax(Result(s,a))} & \text{if Player(s) = Min} \\ \sum_r P(r) \text{ Expectiminimax(Result(s,r))} & \text{if Player(s) = Chance} \end{cases}$$

Where ***r*** represents all chance events (e.g., dice roll), and ***Result(s,r)*** is the same state as s with the result of the chance event is r

Games: Conclusions

- Games are modeled in AI as a search problem and use heuristic to evaluate the game
- Minimax algorithm choses the best move given an optimal play from the opponent
- Minimax goes all the way down the tree which is not practical give game time constraints
- Alpha-Beta pruning can reduce the game tree search which allow to go deeper in the tree within the time constraints
- Pruning, bookkeeping, evaluation heuristics, node re-ordering and IDS are effective in practice

Games: Conclusions

- Games is an exciting and fun topic for AI
- Devising adversarial search agents is challenging because of the huge state space
- We have just scratched the surface of this topic
- Further topics to explore include partially observable games (card games such as bridge, pocker, etc.)
- Except for robot football (a.k.a. soccer), there was no much interest from AI in physical games (see http://www.robocup.org/)