

Artificial Intelligence (IT38005)

Course Motivation and Introduction to AI

Anup Kumar Gupta

Instructional Objectives (Module 1)

- Understand the definition of artificial intelligence (AI)
- Understand the different faculties involved with intelligent behavior
- Examine the different ways of approaching AI
- Look at some example systems that use AI
- Have a fair idea of the types of problems that can be currently solved by computers and those that are as yet beyond its ability.
- Trace briefly the history of AI

Instructional Objectives (Module 1)

We will introduce the following entities:

- An agent
- An intelligent agent
- A rational agent

We will explain and discuss:

- We will explain the notions of rationality and bounded rationality.
- We will discuss different types of environment in which the agent might operate.
- We will also talk about different agent architectures.

Instructional Objectives (Module 1)

On completion of this module the students will be able to:

- Understand what an agent is and how an agent interacts with the environment.
- Given a problem situation, the student should be able to:
 - identify the percepts available to the agent and
 - the actions that the agent can execute.
- Understand the performance measures used to evaluate an agent.

Instructional Objectives (Module 1)

The students will become familiar with the following agent architectures:

- Stimulus response agents
- State based agents
- Deliberative / goal-directed agents
- Utility based agents

The student should be able to analyze a problem situation and be able to

- Identify the characteristics of the environment.
- Recommend the architecture of the desired agent.

1.1.1 Definition of AI

What is AI ?

Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by *McCarthy* in 1956.

There are two ideas in the definition.

1. Intelligence
 2. Artificial device
-
- What is intelligence?
 - Is it that which characterize humans?
 - Is there an absolute standard of judgement?

1.1.1 Definition of AI

Is it that which characterize humans? Or is there an absolute standard of judgement?

- Accordingly there are two possibilities:
 - A system with intelligence is expected to behave **as intelligently as a human.**
 - A system with intelligence is expected to behave **in the best possible manner.**
- Secondly what type of behavior are we talking about?
 - Are we looking at the thought process or reasoning ability of the system?
 - Or are we only interested in the final manifestations of the system in terms of its actions?

1.1.1 Interpretations of AI (Cognitive science approach)

One view is that artificial intelligence is about designing systems that are as intelligent as humans.

This view involves trying to understand human thought and an effort to build machines that emulate the human thought process.

This view is the cognitive science approach to AI.

Cognitive: Capability of relating to, being, or involving in conscious intellectual activity (such as thinking, reasoning, or remembering). [Britannica Dictionary]

1.1.1 Interpretations of AI (Acting Human)

- The second approach is best embodied by the concept of the Turing Test.
- Turing held that in future computers can be programmed to acquire abilities rivaling human intelligence.
- As part of his argument Turing put forward the idea of an 'imitation game', in which a human being and a computer would be interrogated under conditions where the interrogator would not know which was which, the communication being entirely by textual messages.
- Turing argued that if the interrogator could not distinguish them by questioning, then it would be unreasonable not to call the computer intelligent.
- Turing's 'imitation game' is now usually called 'the Turing test' for intelligence.

1.1.1 Turing Test and the Imitation Games

- **Alan Turing** (23 June 1912 – 7 June 1954) was an English mathematician, computer scientist, and logician.
- Turing was highly influential in the development of theoretical computer science, providing a formalisation of the concepts of algorithm and computation with the Turing machine, which can be considered a model of a general-purpose computer.
- He is widely considered to be the father of theoretical computer science and artificial intelligence.



Fig 1. Alan Mathison Turing
Ref: [Wikipedia](#)



Fig 2. “*The Imitation Game*”, a movie based on the life of Alan Turing
Ref: [IMDB](#)

1.1.1 Turing Test

Consider the following setting.

- There are two rooms, A and B.
- One of the rooms contains a computer. The other contains a human.
- The interrogator is outside and does not know which one is a computer.
- He / She can ask questions through a teletype and receives answers from both A and B.
- The interrogator needs to identify whether A or B are humans.
- To pass the Turing test, the machine has to fool the interrogator into believing that it is human.

The Turing Test

Turing test

During the Turing test, the human questioner asks a series of questions to both respondents. After the specified time, the questioner tries to decide which terminal is operated by the human respondent and which terminal is operated by the computer.

■ QUESTION TO RESPONDENTS ■ ANSWERS TO QUESTIONER



ILLUSTRATION: GSTUDIO GROUP/ADDOBE STOCK

©2017 TECHTARGET. ALL RIGHTS RESERVED



Fig 3. Turing Test. The test involves two humans and one machine.

Ref: [TechTarget](#)

1.1.1 Interpretations of AI (Reasoning and Inference)

- Logic and laws of thought deals with studies of ideal or rational thought process and inference.
- The emphasis in this case is on the inferencing mechanism, and its properties.
- That is how the system arrives at a conclusion, or the reasoning behind its selection of actions is very important in this point of view.
- The soundness and completeness of the inference mechanisms are important here.

1.1.1 Interpretations of AI (Rationality of Agents)

- The fourth view of AI is that it is the study of rational agents.
- This view deals with building machines that act rationally.
- The focus is on how the system acts and performs, and not so much on the reasoning process.
- A rational agent is one that acts rationally, that is, is in the best possible manner.

1.1.2 Typical AI problems

- We expect an “intelligent entity” to perform, we need to consider both “common-place” tasks as well as expert tasks.

Examples of “common-place” tasks include:

- Recognizing people, objects.
- Communicating (through natural language).
- Navigating around obstacles on the streets.

These tasks are done matter of routinely by people and some other animals.

Expert tasks include:

- Medical diagnosis.
- Mathematical problem solving
- Playing games like chess

These tasks cannot be done by all people, and can only be performed by skilled specialists.

1.1.2 Typical AI problems

- Now, which of these tasks are easy and which ones are hard?
- Clearly tasks of the first type are easy for humans to perform, and almost all are able to master them.
- The second range of tasks requires skill development and/or intelligence and only some specialists can perform them well.
- However, when we look at what computer systems have been able to achieve to date, we see that their achievements include performing sophisticated tasks like medical diagnosis, performing symbolic integration, proving theorems and playing chess.
- On the other hand it has proved to be very hard to make computer systems perform many routine tasks that all humans and a lot of animals can do.

1.1.2 Typical AI problems

- On the other hand it has proved to be very hard to make computer systems perform many routine tasks that all humans and a lot of animals can do.
- Examples of such tasks include navigating our way without running into things, catching prey and avoiding predators.
- Humans and animals are also capable of interpreting complex sensory information.
- We are able to recognize objects and people from the visual image that we receive.
- We are also able to perform complex social functions.

1.1.2 Some more division of Tasks

- Mundane Tasks:
 - Perception
 - Vision
 - Speech
 - Natural Languages
 - Understanding
 - Generation
 - Translation
 - Common sense reasoning
- Expert Tasks:
 - Engineering (Design, Fault finding, Manufacturing planning)
 - Scientific Analysis
 - Medical Diagnosis
 - Financial Analysis
- Formal Tasks
 - Games : chess, checkers.
 - Mathematics: Geometry, logic, proving properties of programs

1.1.3 Intelligent Behavior

- This discussion brings us back to the question of what constitutes intelligent behavior.
- Some of these tasks and applications are:
 - Perception involving image recognition and computer vision.
 - Reasoning
 - Learning
 - Understanding language involving natural language processing, speech processing
 - Solving problems
 - Robotics

[Also discussed towards the end of the lecture]

1.1.4 Approaches to AI

- **Strong AI** aims to build machines that can truly reason and solve problems. These machines should be self aware and their overall intellectual ability needs to be indistinguishable from that of a human being. Strong AI maintains that suitably programmed machines are capable of cognitive mental states.
- **Weak AI** deals with the creation of some form of computer-based artificial intelligence that cannot truly reason and solve problems, but can act as if it were intelligent. Weak AI holds that suitably programmed machines can simulate human cognition.
- **Applied AI** aims to produce commercially viable "smart" systems such as, for example, a security system that is able to recognize the faces of people who are permitted to enter a particular building. Applied AI has already enjoyed considerable success.
- **Cognitive AI** computers are used to test theories about how the human mind works, for example, theories about how we recognize faces and other objects, or about how we solve abstract problems.

What is Artificial Intelligence?

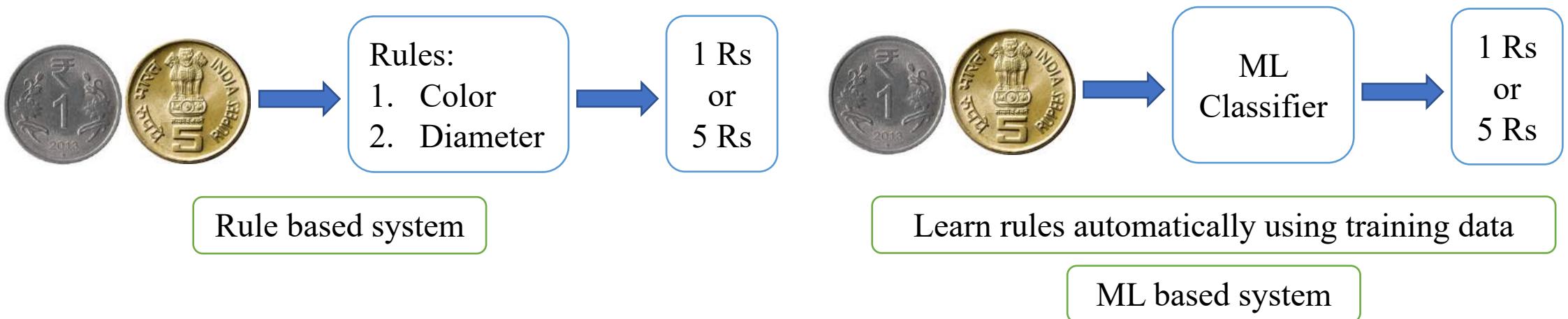
- Artificial intelligence (AI) is the capability of a computer system to mimic human cognitive functions.
- Through AI, a computer system uses math and logic to simulate the reasoning that people use to learn from new information and make decisions.

What is Machine Learning?

- Machine learning (ML) is an application of AI.
- It's the process of using mathematical models of data to help a computer learn without direct instruction.
- This enables a computer system to continue learning and improving on its own, based on experience.

1.1.5 Machine Learning

- Some computational problems like sorting and searching are rule-driven.
- For some problems, it is difficult to define rules. ML is used to solve them.



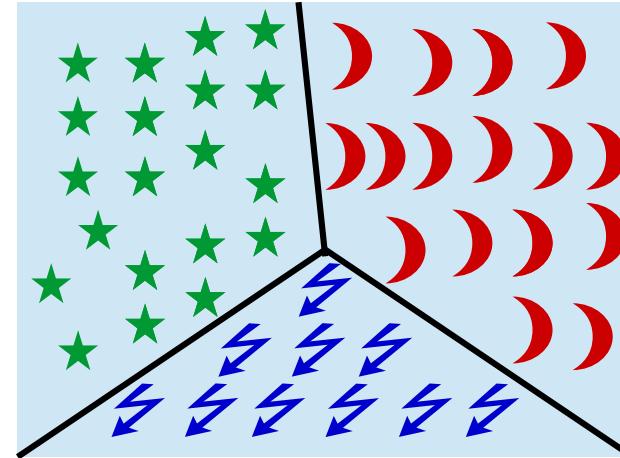
- Rules should correctly perform on unseen or future data: **Generalization**.
- Consider color-based rule, it is not easily generalized.
- Example of classifier which can be easily generalized: OCR
- Performance is highly dependent on training data and training methodology.



1.1.6 ML: Some Predictive Tasks

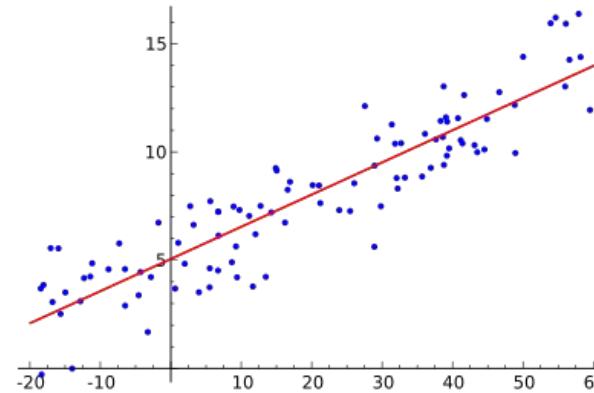
Classification:

- It maps data into predefined labels.
- E.g.: Spam vs non-spam by analysing frequency of words.



Regression:

- It map data into real value variable.
- E.g.: Estimate cost of a building by analysing location, plot area, construction year, etc.



Time Series Analysis:

- Values of attribute are examined over time.
- Eg: Comparing stocks or weather prediction by analysing past and current estimates.
- Eg: Weather forecasting by (20 Jul, No-rain), (21 Jul, Rain), (24 Jul, Rain), (26 Jul, ...?...)

- Fingerprint Verification.
- Captioning.
- Spelling Correction on mobile.

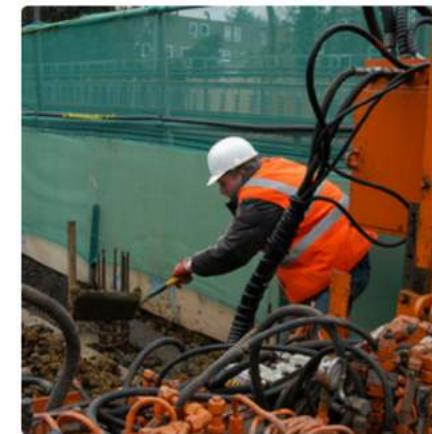
1.1.7 Computer Vision



Robotics



"man in black shirt is playing guitar."



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



"two young girls are playing with lego toy."

Image Captioning



What color are her eyes?
What is the mustache made of?



How many slices of pizza are there?
Is this a vegetarian pizza?

Visual Question Answering

1.1.8 ML in other Areas

Natural Language Processing

- Text summarization
- Language translation
- Topic modelling
- Searching documents

Finance

- Stock prediction
- Fraud detection

Healthcare

- Predicting diseases
- Performing surgeries

Entertainment

- Movie recommendation
- Ads placements and impact
- Animation

Audio

- Speech understanding
- Siri, Alexa or google assistant
- Improving CV
- Speech translation

Internet

- Spam vs Non-spam
- Webpage clustering

Code completion

1.1.4 Limits of AI

- Machines cannot learn how to do something without clear, replicable examples.
- The real advancements in AI haven't been in "creative thinking," but in accuracy and efficiency.
- Because AI learns from the past, the biggest concern with the technology is bias.
- AI cannot multitask.
- AI cannot always explain its decisions.
- AI is not capable of making moral judgements.
- AI cannot feel empathy, sympathy, or anything else for that matter.
- AI is susceptible to adversarial attacks.

1.1.4 History of AI

- AI is not a new word and not a new technology for researchers.
- Even there are the myths of Mechanical men in Ancient Greek and Egyptian Myths.
- Intellectual roots of AI date back to the early studies of the nature of knowledge and reasoning.
- The dream of making a computer imitate humans also has a very early history.
- The concept of intelligent machines is found in Greek mythology.
- There is a story in the **8th century A.D** about Pygmalion Olio, the legendary king of Cyprus. He fell in love with an ivory statue he made to represent his ideal woman. The king prayed to the goddess Aphrodite, and the goddess miraculously brought the statue to life. Other myths involve human-like artifacts. As a present from Zeus to Europa, Hephaestus created Talos, a huge robot. Talos was made of bronze and his duty was to patrol the beaches of Crete.

1.1.4 History of AI

- Aristotle (**384-322 BC**) developed an informal system of syllogistic logic, which is the basis of the first formal deductive reasoning system.
- Early in the **17th century**, Descartes proposed that bodies of animals are nothing more than complex machines.
- Pascal in **1642** made the first mechanical digital calculating machine.

1.1.4 History of AI - Maturation of AI (1943-1952)

- **Year 1943:** The first work which is now recognized as AI was done by Warren McCulloch and Walter Pitts in 1943. They proposed a model of **artificial neurons**.
- **Year 1949:** Donald Hebb demonstrated an updating rule for modifying the connection strength between neurons. His rule is now called **Hebbian learning**.
- **Year 1950:** The Alan Turing who was an English mathematician and pioneered Machine learning in 1950. Alan Turing publishes "**Computing Machinery and Intelligence**" in which he proposed a test. The test can check the machine's ability to exhibit intelligent behaviour equivalent to human intelligence, called a **Turing test**.

1.1.4 History of AI – Coining of term AI (1943-1952)

- **Year 1955:** An Allen Newell and Herbert A. Simon created the "first artificial intelligence program" which was named as "**Logic Theorist**". This program had proved 38 of 52 Mathematics theorems, and find new and more elegant proofs for some theorems.
- **Year 1956:** The word "Artificial Intelligence" first adopted by American Computer scientist John McCarthy at the Dartmouth Conference. For the first time, AI coined as an academic field.
- At that time high-level computer languages such as FORTRAN, LISP, or COBOL were invented. And the enthusiasm for AI was very high at that time.

1.1.4 History of AI – The early enthusiasm (1956-1974)

- **Year 1966:** The researchers emphasized developing algorithms which can solve mathematical problems. Joseph Weizenbaum created the first chatbot in 1966, which was named as ELIZA.
- **Year 1972:** The first intelligent humanoid robot was built in Japan which was named as WABOT-1.

1.1.4 History of AI – The first AI winter (1974-1980)

- The duration between years **1974 to 1980** was the first *AI winter* duration.
- *AI winter* refers to the time period where computer scientist dealt with a severe shortage of funding from government for AI researches.
- During *AI winters*, an interest of publicity on artificial intelligence was decreased.



Winter is Coming, Game of Thrones

Credits: GOT

1.1.4 History of AI – AI bounces back (1980-1987)

- **Year 1980:** After AI winter duration, AI came back with "Expert System". Expert systems were programmed that emulate the decision-making ability of a human expert.
- In the Year **1980**, the first national conference of the American Association of Artificial Intelligence was held at **Stanford University**.

1.1.4 History of AI – The second AI winter (1987-1993)

- The duration between the years 1987 to 1993 was the second *AI Winter* duration.
- Again Investors and government stopped in funding for AI research as due to high cost but not efficient result.

1.1.4 History of AI – Emergence of intelligent agents (1993-2011)

- **Year 1997:** In the year 1997, IBM Deep Blue beats world chess champion, Gary Kasparov, and became the first computer to beat a world chess champion.
- **Year 2002:** for the first time, AI entered the home in the form of Roomba, a vacuum cleaner.
- **Year 2006:** AI came in the Business world till the year 2006. Companies like Facebook, Twitter, and Netflix also started using AI.
- **Year 2011:** In the year 2011, IBM's Watson won jeopardy, a quiz show, where it had to solve the complex questions as well as riddles. Watson had proved that it could understand natural language and can solve tricky questions quickly.

1.1.4 History of AI – Deep Learning (2011-present)

- **Year 2012:** Google has launched an Android app feature "Google now", which was able to provide information to the user as a prediction.
- **Year 2014:** In the year 2014, Chatbot "Eugene Goostman" won a competition in the infamous "Turing test."
- **Year 2016:** IBM Creates First Movie Trailer by AI. Scientists at IBM Research have collaborated with 20th Century Fox to create the first-ever cognitive [movie trailer](#) for the movie Morgan.
- **Year 2018:** The "Project Debater" from IBM debated on complex topics with two master debaters and also performed extremely well.
- Google has demonstrated an AI program "Duplex" which was a virtual assistant and which had taken hairdresser appointment on call, and lady on other side didn't notice that she was talking with the machine. - [Video Link](#)

Contact Details

- Anup Kumar Gupta
- PhD scholar, IIT Indore
- Email: msrphd2105101002@iiti.ac.in
- Website: <https://anupkumargupta.github.io/>

Artificial Intelligence (IT38005)

Introduction to Agent

Anup Kumar Gupta

1.3.1 Introduction to Agents

- An agent acts in an **environment**.
- An agent perceives its environment through **sensors**.
- The complete set of inputs at a given time is called a **percept**.
- The current percept, or a sequence of percepts can influence the actions of an agent.
- The agent can change the environment through **actuators** or **effectors**.
- An operation involving an effector is called an **action**.
- More than one actions can be grouped into a single entity called an **action sequence**.
- The task(s) which the agent tries to achieve is/are called the **goal(s)** to the agent.

1.3.1.1 Performance of Agents

- The behavior and performance of intelligent agents have to be evaluated in terms of the agent function (or mapping).
- The **ideal mapping** specifies which actions an agent ought to take at any point in time.
- The performance measure is a *subjective* measure to characterize how successful an agent is.
- The success can be measured in various ways.
 - It can be measured in terms of speed or efficiency of the agent.
 - It can be measured by the accuracy or the quality of the solutions achieved by the agent.
 - It can also be measured by power usage, money, etc.

1.3.1.2 Examples of Agents

1. **Humans** can be looked upon as agents. They have eyes, ears, skin, taste buds, etc. for sensors; and hands, fingers, legs, mouth for actuators (or effectors).
2. **Robots** are agents. Robots may have camera, sonar, infrared, bumper, etc. for sensors. They can have grippers, wheels, lights, speakers, etc. for actuators.

1.3.1.2 Examples of Agents - Robots



Xavier, CMU
(1996).



T-HR3 Toyota (2017), is a humanoid robot that mimics the movements of its operator.



Sophia, Hanson
Robotics (2017).



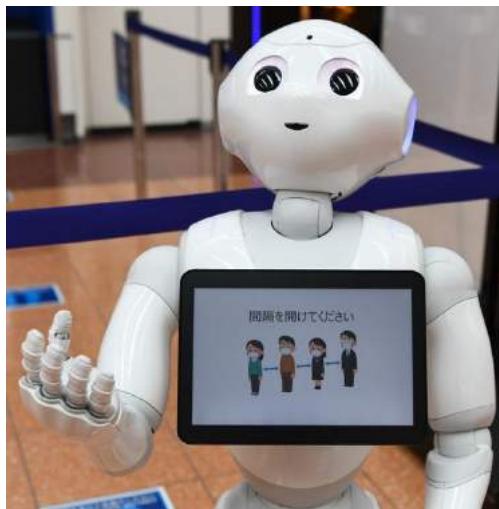
Digit, Agility Robotics (2018), is a delivery robot.



Surena, University of Tehran (2019)



Digital Humanoids, Samsung Technology and Advanced Research (2020)



Pepper, SoftBank Robotics, receptionist robot (2014).



Tesla cars and Optimus, Telsa (2009, 2023).



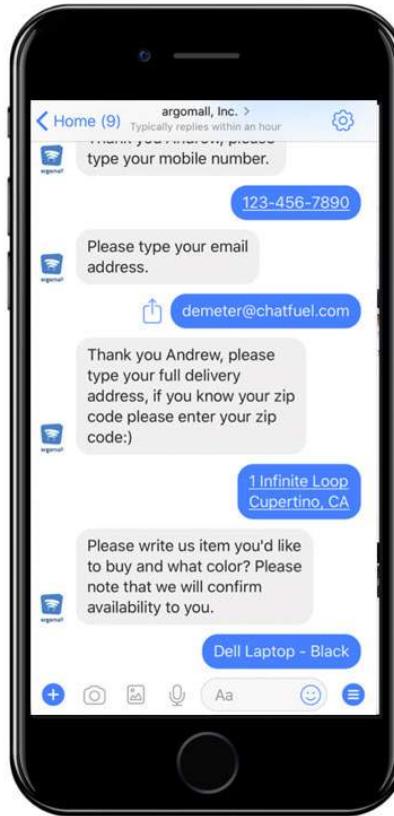
1.3.1.2 Examples of Agents

1. **Humans** can be looked upon as agents. They have eyes, ears, skin, taste buds, etc. for sensors; and hands, fingers, legs, mouth for actuators (or effectors).
2. **Robots** are agents. Robots may have camera, sonar, infrared, bumper, etc. for sensors. They can have grippers, wheels, lights, speakers, etc. for actuators.
3. **Softbots or chatbots**. We also have software agents or softbots or chatbots that have some *functions* as sensors and some *functions* as actuators.

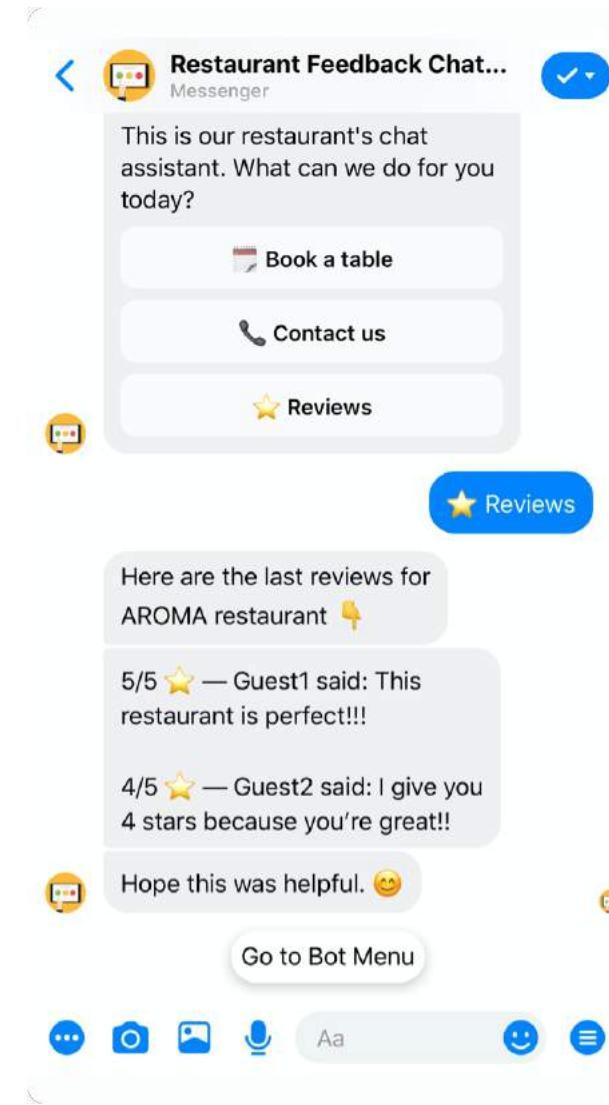
For example: the *function* or *method* which takes input from the chat window can be considered a sensor.

Similarly, the *method* or the *function* providing the output can be called an actuator.

1.3.1.2 Examples of Agents - Chatbots



SUMO



1.3.1.3 Agent Faculties

- The fundamental faculties of intelligence are
 - Sensing
 - Understanding, reasoning
 - Acting
 - Learning
- Blind action is not a characterization of intelligence.
- In order to act intelligently, one must sense.
- Understanding is essential to interpret the sensory percepts and decide on an action.
- Many robotic agents stress sensing and acting, and do not have understanding.

1.3.1.4 Intelligent Agents

- An Intelligent Agent must sense, must act, must be autonomous (to some extent),.
- It also must be rational. AI is about building rational agents.
- An agent is something that perceives and acts. A rational agent always does the right thing.

In order to build an intelligent agent, we must answer the following questions:

1. What are the functionalities (goals) ?
2. What are the components ?
3. How do we build them ?

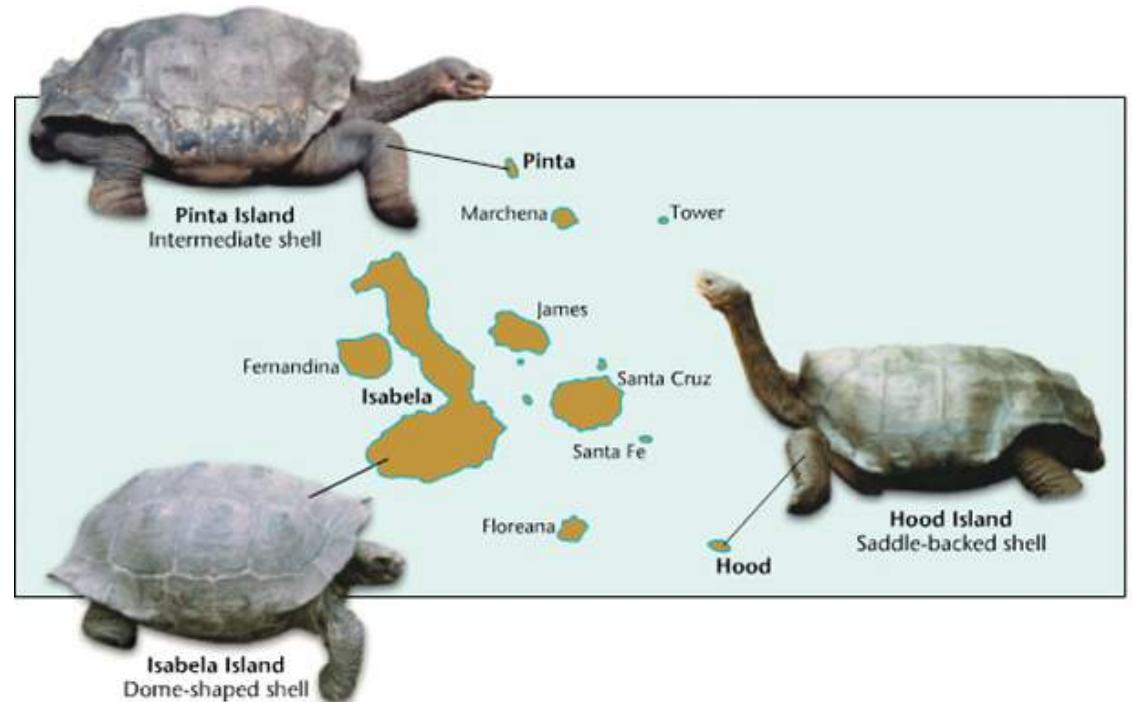
1.3.1.5 Rationality

- Perfect rationality assumes that the rational agent knows all and will take the action that maximizes its utility.
- Human beings do not satisfy **this** definition of rationality.
- Rational action is the action that maximizes the expected value of the performance measure given the percept sequence to date.
- However, a rational agent is **not omniscient**. It does not know the actual outcome of its actions, and it may not know certain aspects of its environment.
- Therefore rationality must take into account the limitations of the agent.
- The agent has to select the best action to the best of its knowledge depending on its **percept sequence**, its **background knowledge** and its **feasible actions**.
- An agent also has to deal with the expected outcome of the actions where the action effects are not deterministic.

1.3.1.6 Bounded Rationality

“Because of the limitations of the human mind, humans must use approximate methods to handle many tasks.” -- Herbert Simon, 1972.

- Evolution did not give rise to optimal agents, but to agents which are in some senses locally optimal at best.



1.3.1.6 Bounded Rationality

“Because of the limitations of the human mind, humans must use approximate methods to handle many tasks.” -- Herbert Simon, 1972.

- Evolution did not give rise to optimal agents, but to agents which are in some senses locally optimal at best.
- In 1957, Simon proposed the notion of Bounded Rationality:
“that property of an agent that behaves in a manner that is **nearly** optimal with respect to its goals as its resources will allow.”
- Under these promises an intelligent agent will be expected to act optimally to the best of its abilities and its resource constraints.

1.3.2 Agent Environment

- Environments in which agents operate can be defined in different ways.
- We will determine the different types of environment based on the following criterions:
 1. Observability
 2. Determinism
 3. Episodicity
 4. Dynamism
 5. Continuity

1.3.2.1 Observability

In terms of observability, an environment can be characterized as **fully observable** or **partially observable**.

- In a **fully observable** environment all of the environment relevant to the action being considered is observable. In such environments, the agent **does not need** to keep track of the changes in the environment. A chess playing system is an example of a system that operates in a fully observable environment.
- In a **partially observable** environment, the relevant features of the environment are only partially observable. In a partially observable system the observer may utilize a **memory system** in order to add information to the observer's understanding of the system. An example of a partially observable system would be a card game in which some of the cards are discarded into a pile face down.

1.3.2.2 Determinism

- In **deterministic environments**, the next state of the environment is completely described by the current state and the agent's action.
 - Image analysis systems are examples of this kind of situation. The processed image is determined completely by the current image and the processing operations.
 - If an element of interference or uncertainty occurs then the environment is stochastic. Note that a deterministic yet partially observable environment will *appear* to be stochastic to the agent.
 - Examples of this are the automatic vehicles that navigate a terrain, say, the Mars rovers robot. The new environment in which the vehicle is in is stochastic in nature.
-
- If the environment state is wholly determined by the preceding state and the actions of multiple agents, then the environment is said to be **strategic**.
 - Example: Chess. There are two agents, the players and the next state of the board is strategically determined by the players' actions.

1.3.2.3 Episodicity

- An **episodic environment** means that subsequent episodes do not depend on what actions occurred in previous episodes.
- In a **sequential environment**, the agent engages in a series of connected episodes.

1.3.2.4 Dynamism

- A **static environment** does not change from one state to the next while the agent is considering its course of action.
 - The only changes to the environment are those caused by the agent itself.
 - The agent doesn't need to observe the world during deliberation.
 - The passage of time as an agent deliberates is irrelevant.
- A **dynamic environment** changes over time independent of the actions of the agent.
 - If an agent does not respond in a timely manner, this counts as a choice to do nothing.

1.3.2.5 Continuity

- If the number of distinct percepts and actions is limited, the environment is **discrete**, otherwise it is **continuous**.

1.3.2.6 Presence of Other agents

Single agent versus Multi-agent:

- A multi-agent environment has other agents. If the environment contains other intelligent agents, the agent needs to be concerned about strategic, game-theoretic aspects of the environment (for *either* cooperative *or* competitive agents).
- Most engineering environments do not have multi-agent properties, whereas most social and economic systems get their complexity from the interactions of (more or less) rational agents.

1.3.3 Agent architectures

We will next discuss various agent architectures:

1.3.3.1 Table based agents

1.3.3.2 Percept based agent or reflex agents

1.3.3.3 Subsumption architecture base agents

1.3.3.4 State-based Agent or model-based agents

1.3.3.5 Goal-based agents

1.3.3.6 Utility-based agents

1.3.3.7 Learning agents

1.3.3.1 Table based agent

- In table based agent the action is looked up from a table based on information about the agent's percepts.
- A table is simple way to specify a mapping from percepts to actions. The mapping is implicitly defined by a program.
- The mapping may be implemented by a rule based system, by a neural network or by a procedure.
- There are several disadvantages to a table based system.
- The tables may become very large. Learning a table may take a very long time, especially if the table is large.
- Such systems usually have little autonomy, as all actions are pre-determined.

1.3.3.2 Percept based agent or reflex agents

In percept based agents:

1. information comes from **sensors or percepts**
 2. changes the agents current **state of the environment**
 3. triggers **actions** through the **effectors**.
- Such agents are called reactive agents or stimulus-response agents.
 - Reactive agents have no notion of history.
 - The current state is as the sensors see it right now.
 - The action is based on the current percepts only.

1.3.3.2 Percept based agent or reflex agents

The following are some of the characteristics of percept-based agents:

- Efficient
- No internal representation for reasoning, inference.
- No strategic planning, learning.
- Percept-based agents are not good for multiple, opposing, goals.

Assignment: Justify each of the characteristics mentioned above in context of percept-based agents.

(no need to submit the assignment, just to ponder upon)

1.3.3.3 Subsumption architecture base agents

- The subsumption architecture was proposed by *Rodney Brooks*, (1986).
- This architecture is based on reactive systems.
- Brooks notes that in lower animals there is no deliberation and the actions are based on sensory inputs.
- But even lower animals are capable of many complex tasks.
- His argument is to follow the evolutionary path and build simple agents for complex worlds.

1.3.3.3 Subsumption architecture base agents

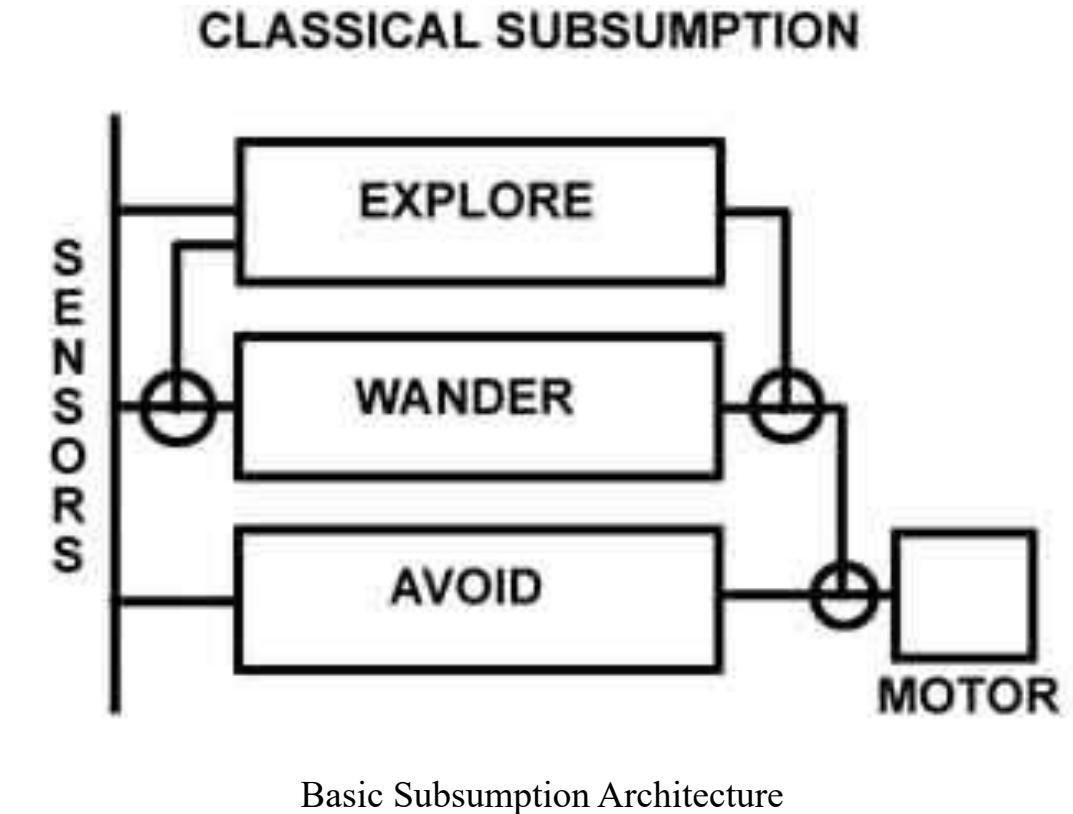
The main features of Brooks' architecture are:

- There is no explicit knowledge representation
- Behavior is distributed, not centralized
- Response to stimuli is reflexive
- The design is bottom up, and complex behaviors are fashioned from the combination of simpler underlying ones.
- Individual agents are simple

Note: *These architectural choices are analogous to the software engineering designs you must have studied earlier.*

1.3.3.3 Subsumption architecture base agents

- The Subsumption Architecture built in layers.
- There are different layers of behavior.
- The higher layers can override lower layers.
- Each activity is modeled by a finite state machine (FSM, or finite state automata).
- The subsumption architecture can be illustrated by Brooks' Mobile Robot example.



Basic Subsumption Architecture

1.3.3.3 Subsumption architecture base agents

The system is built in three layers:

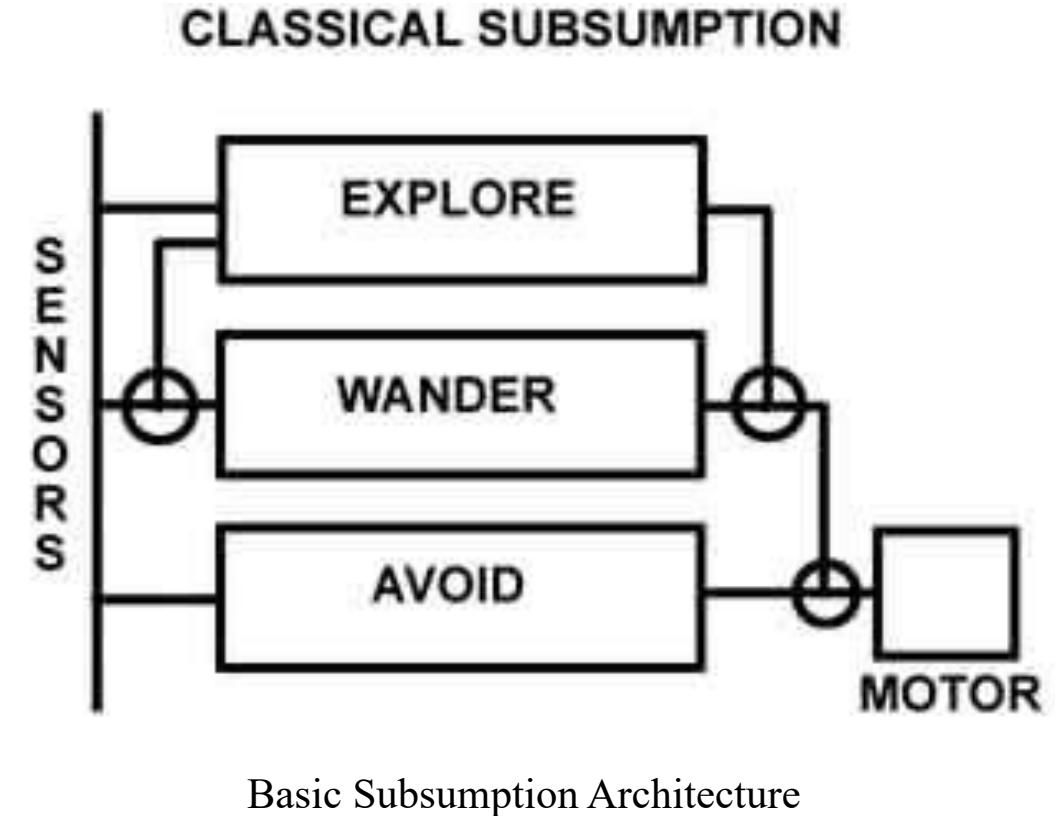
1. Layer 0: Avoid Obstacles.
2. Layer 1: Wander behavior.
3. Layer 2: Exploration behavior.

The information is received through sensors.

It is then processed by the architecture blocks

These blocks send command to the motor (actuator).

The motor then performs the operation.

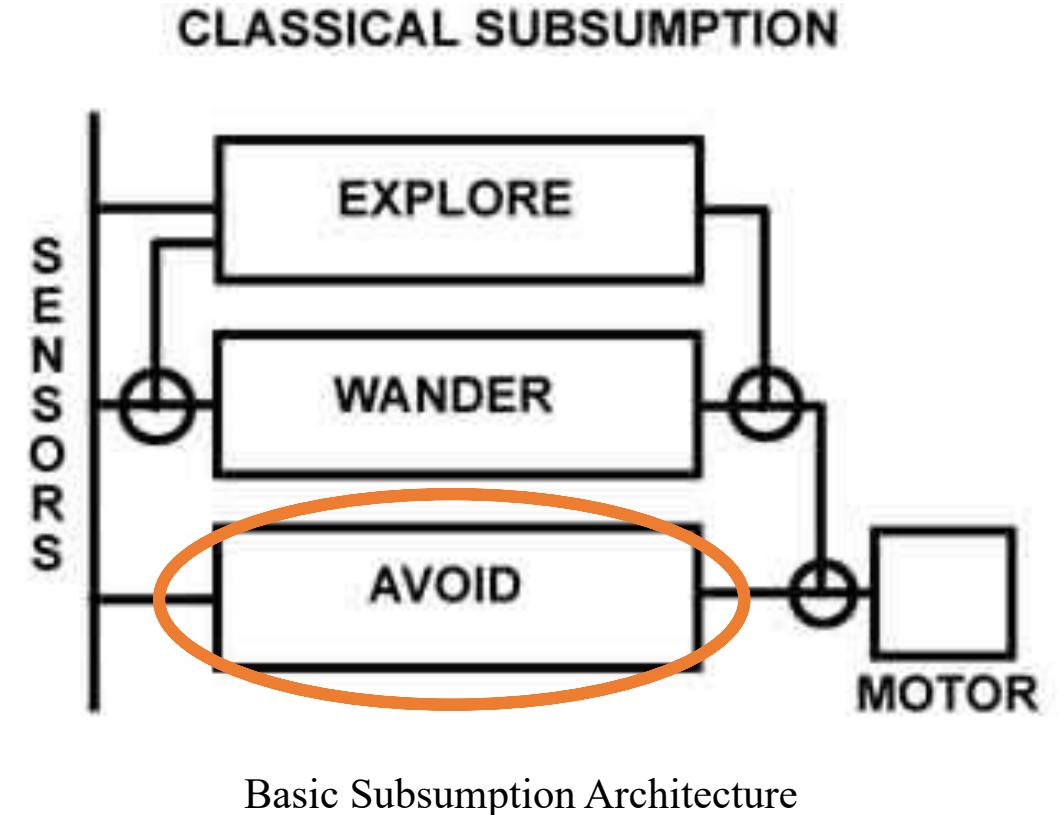


1.3.3.3 Subsumption architecture base agents

Layer 0 (Avoid Obstacles) has the following capabilities:

- **Sonar**: generate sonar scan
- **Collide**: send HALT message to *forward* if about to run into an object
- **Feel force**: Compute overall repulsive force.

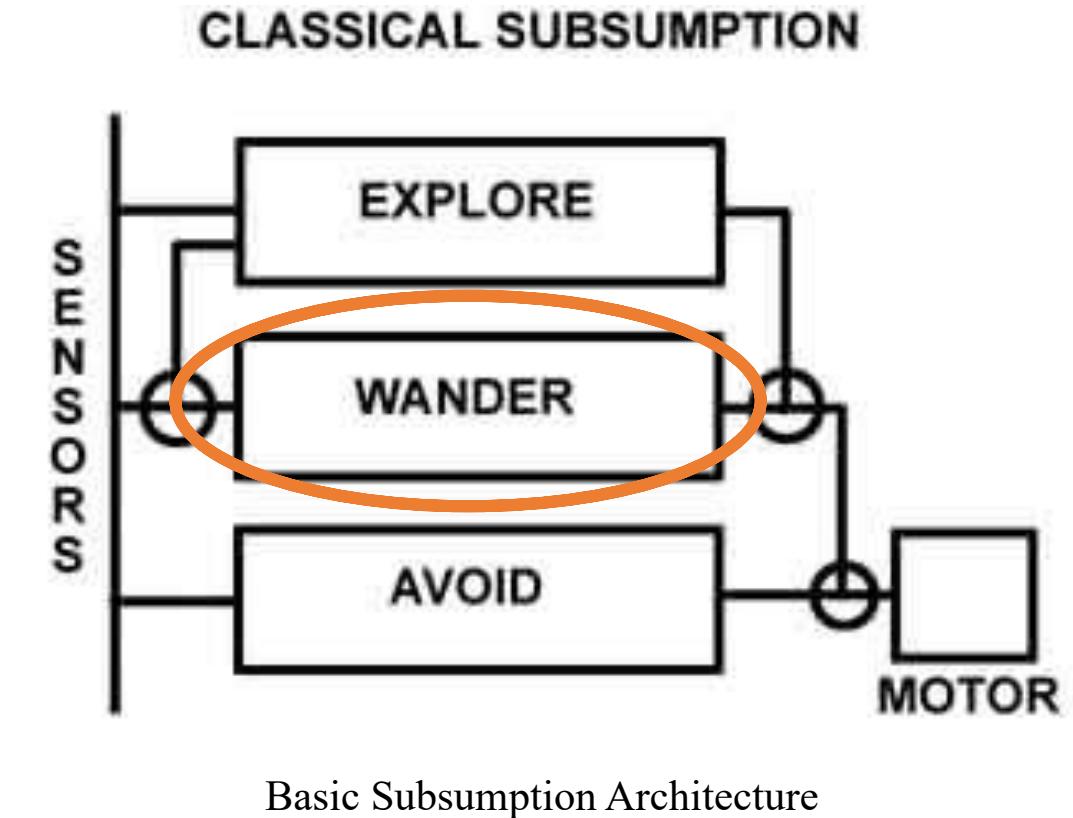
This vector is passed to *run-away*, which sends it to the *turn* cell



1.3.3.3 Subsumption architecture base agents

Layer 1 (Wander behavior)

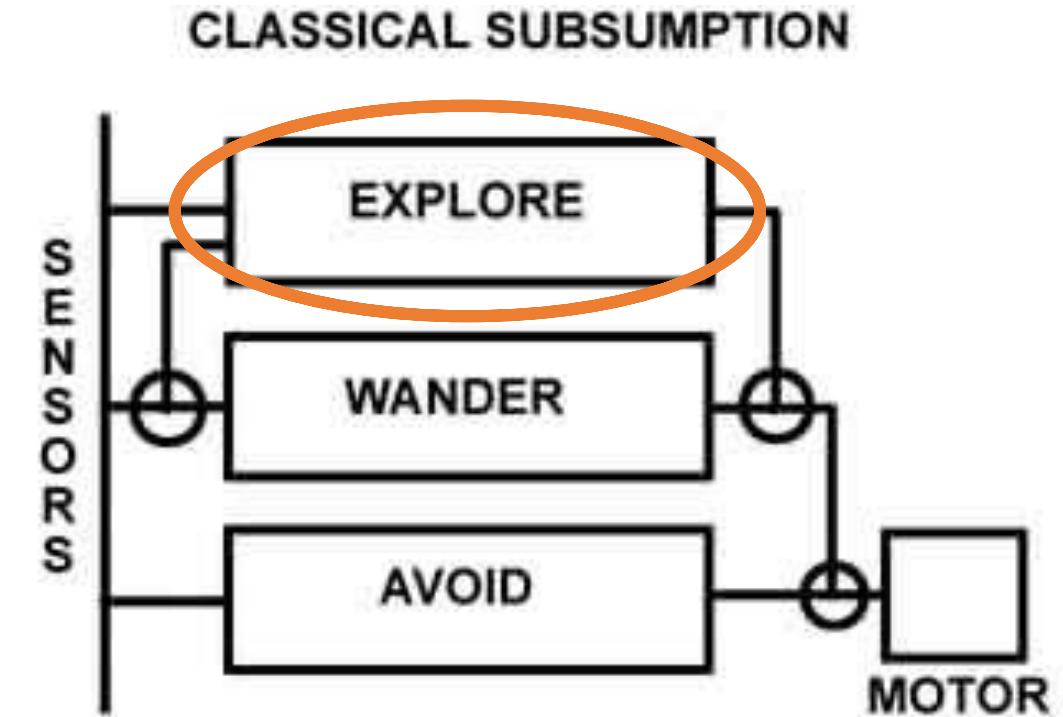
- *wander* generates a random heading.
- *avoid* reads repulsive force and new heading, it then generates new heading and feeds it to *turn* and *forward*.



1.3.3.3 Subsumption architecture base agents

Layer 2 (Exploration behavior)

- *whenlook* notices idle time and looks for an interesting place. Inhibits wandering
- *path-plan* sends new direction to *avoid*.
- *integrate* monitors path and sends them to the *path-plan*.



Basic Subsumption Architecture

1.3.3.4 State-based Agent or model-based agents

- State based agents differ from percept based agents in that such agents maintain some sort of *state* based on the percept sequence received so far.
- The *state* is updated regularly based on what the agent senses, and the agent's actions.
- Keeping track of the state requires that the agent has knowledge about how the world evolves, and how the agent's actions affect the world.

Thus a state based agent works as follows:

- information comes from **sensors (or percepts)**
- based on this, the agent changes the current **state of the environment**
- based on **state of the environment** and knowledge (**memory**), it triggers **actions** through the **effectors**.

1.3.3.5 Goal-based agents

Thus a state based agent works as follows:

- information comes from **sensors** (or **percepts**)
- based on this, the agent changes the current **state** of the **environment**
- based on **state** of the **environment**, knowledge (**memory**) and **goal/intentions**, it triggers **actions** through the **effectors**.
- The goal based agent has some goal which forms a basis of its actions.
- Goal formulation based on the current situation, is a way of solving many problems
- The sequence of steps required to solve a problem is not known a priori and must be determined by a systematic exploration of the alternatives present.

1.3.3.6 Utility-based agents

- Utility based agents provides a more general agent framework.
- In case that the agent has multiple goals, this framework can accommodate different preferences for the different goals.
- Such systems are characterized by a utility function that maps a state or a sequence of states to a real valued utility.
- The agent acts so as to maximize expected utility.

1.3.3.7 Learning agents

- Learning allows an agent to operate in initially unknown environments. The learning element modifies the performance element.
- Learning is required for true autonomy.

1.4 Conclusion

- AI deals with exciting but hard problems. One goal of AI is to build intelligent agents that act so as to optimize performance.
- An agent perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- An ideal agent always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An autonomous agent uses its own experience rather than built-in knowledge of the environment by the designer.
- An agent program maps from percept to action and updates its internal state.

1.4 Conclusion

- Representing knowledge is important for successful agent design.
- The most challenging environments are (based on categories of environment):
 - partially observable
 - stochastic
 - sequential
 - dynamic
 - continuous
 - contain multiple intelligent agents.

Contact Details

- Anup Kumar Gupta
- PhD scholar, IIT Indore
- Email: msrphd2105101002@iiti.ac.in
- Website: <https://anupkumargupta.github.io/>

Artificial Intelligence (IT38005)

Introduction to State Space Search

Anup Kumar Gupta

Instructional Objectives (Module 2)

- The students should understand the state space representation, and gain familiarity with some common problems formulated as state space search problems.
- The student will be made familiar with the following algorithms:
 - greedy search, DFS, BFS, uniform cost search, iterative deepening search, bidirectional search
- Given a problem description, the student should be able to formulate it in terms of a state space search problem.
- The student should be able to understand and analyze the properties of these algorithms in terms of
 - time complexity, space complexity, termination, optimality

2.2 State space search

- Formulate a problem as a state space search by showing the legal problem states, the legal operators, and the initial and goal states.
- A state is defined by the specification of the values of all attributes of interest in the world.
- An operator changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator.
- The initial state is where you start.
- The goal state is the partial description of the solution.

2.2.2 State Space Search Notations

- An *initial state* is the description of the starting configuration of the agent.
- An *action* or an *operator* takes the agent from one state to another state which is called a **successor state**.
- A state can have a number of successor states.
- A plan is a sequence of actions.
- The cost of a plan is referred to as the path cost.
- The path cost is a positive number, and a common path cost may be the sum of the costs of the steps in the path.

Now let us look at the concept of a search problem.

2.2.2 State Space Search Notations

The concept of a search problem:

- Problem formulation means choosing a relevant **set of states** to consider, and a feasible **set of operators** for moving from one state to another.
- Search is the process of considering various possible sequences of operators applied to the initial state, and finding out a sequence which culminates in a goal state.

One more definition:

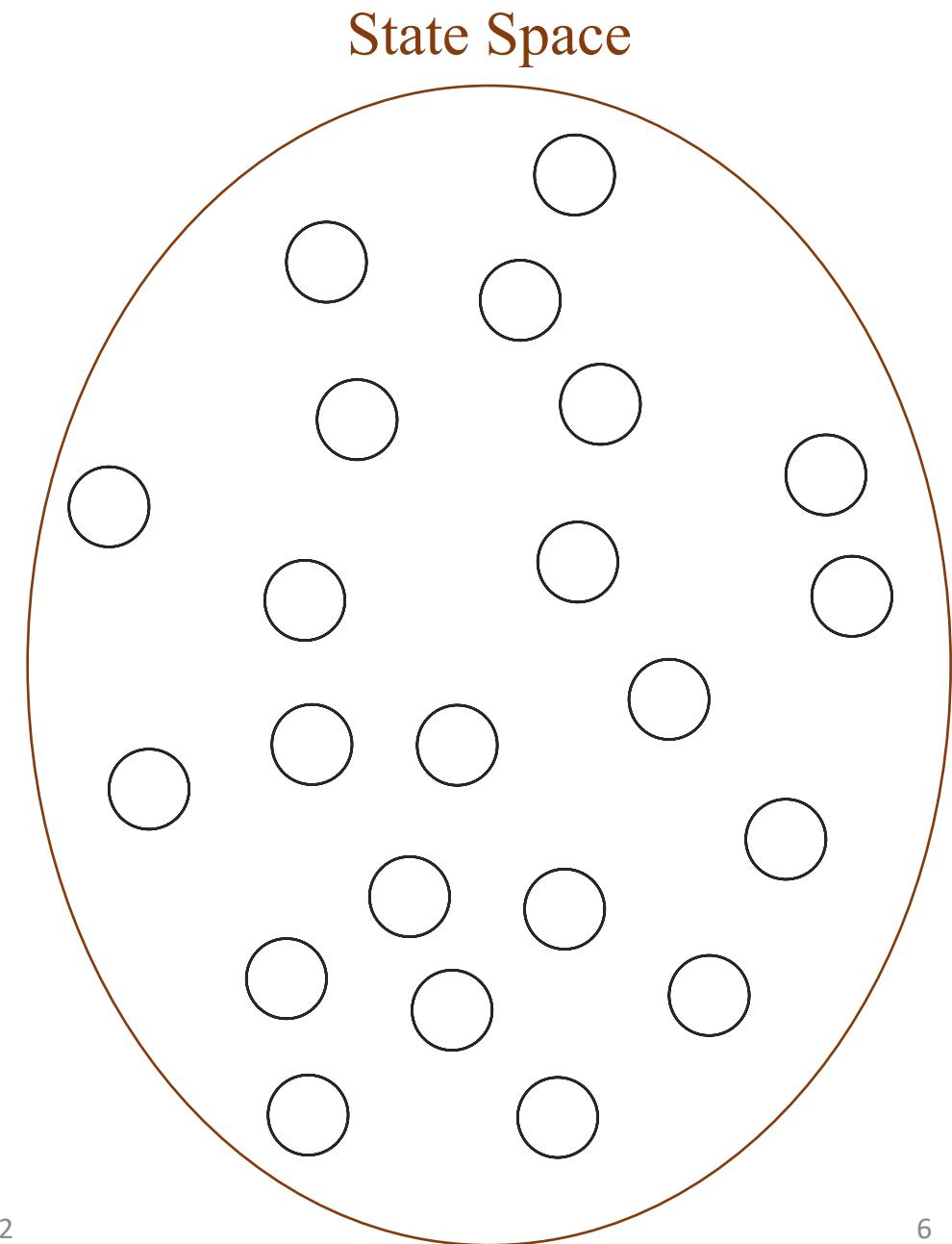
- A **goal** is a description of a set of desirable states of the world.
- **Goal states** are often specified by a goal test which any goal state must satisfy.
- A problem may have more than one more goal state. Example: goal states of a tic-tac-toe game.

2.2.3 Search Problem

We are now ready to formally define a search problem.

A search problem consists of the following:

- S : the full set of states
- s_0 : the initial state
- $A: S \rightarrow S$ is a set of operators or actions
- G is the set of final states.
- Note that $G \subseteq S$

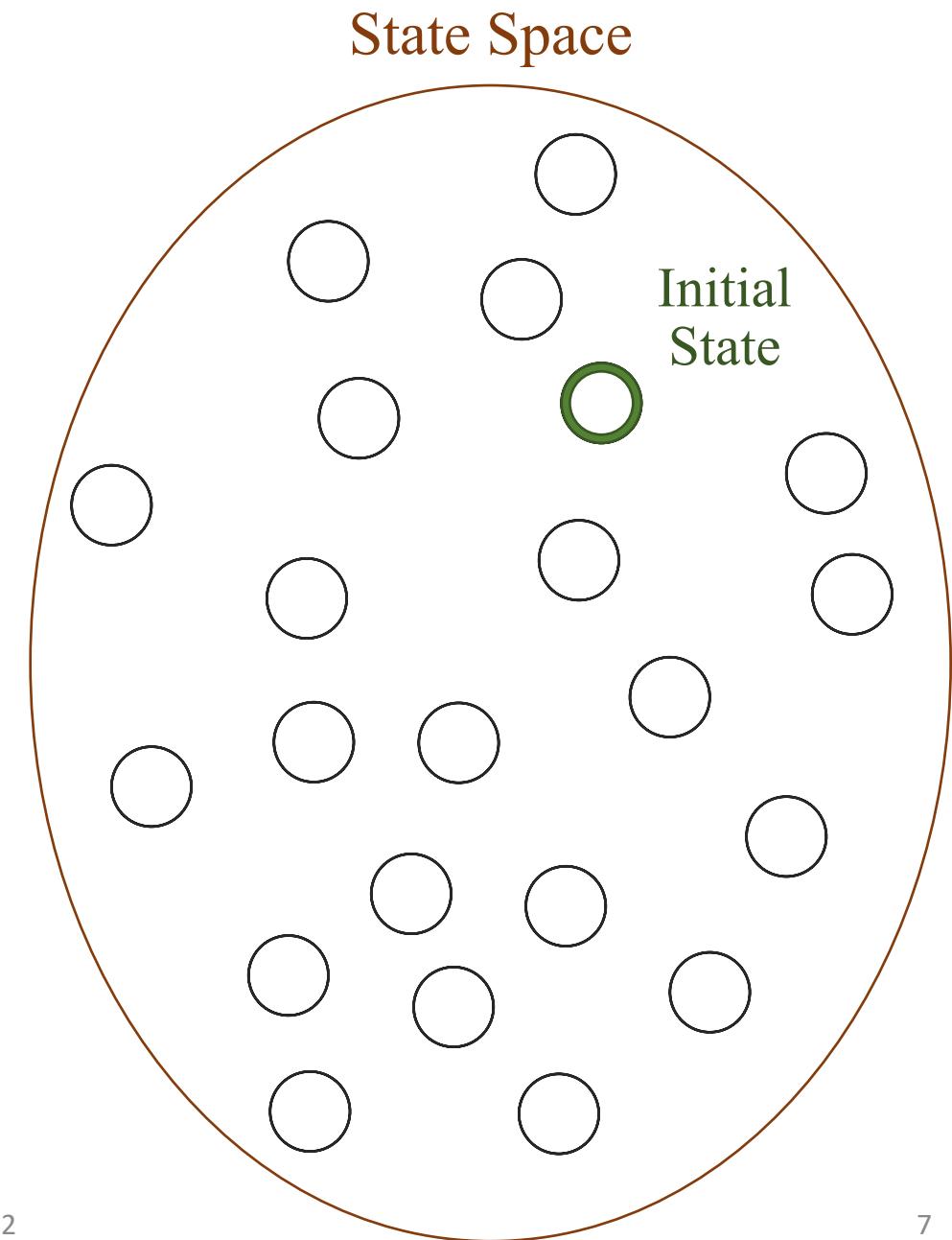


2.2.3 Search Problem

We are now ready to formally define a search problem.

A search problem consists of the following:

- S : the full set of states
- s_0 : the initial state
- $A: S \rightarrow S$ is a set of operators or actions
- G is the set of final states.
- Note that $G \subseteq S$

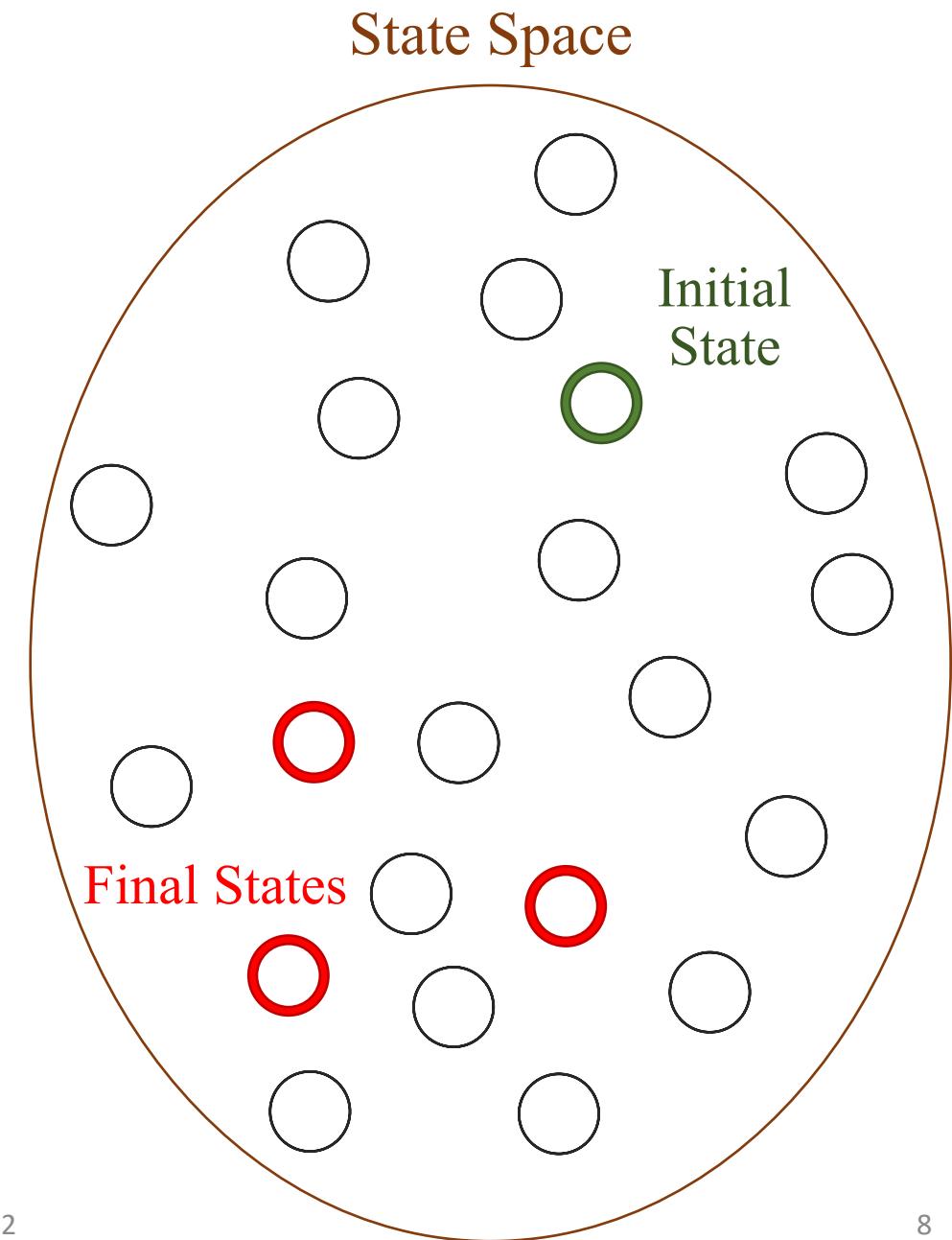


2.2.3 Search Problem

We are now ready to formally define a search problem.

A search problem consists of the following:

- S : the full set of states
- s_0 : the initial state
- $A: S \rightarrow S$ is a set of operators or actions
- G is the set of final states.
- Note that $G \subseteq S$

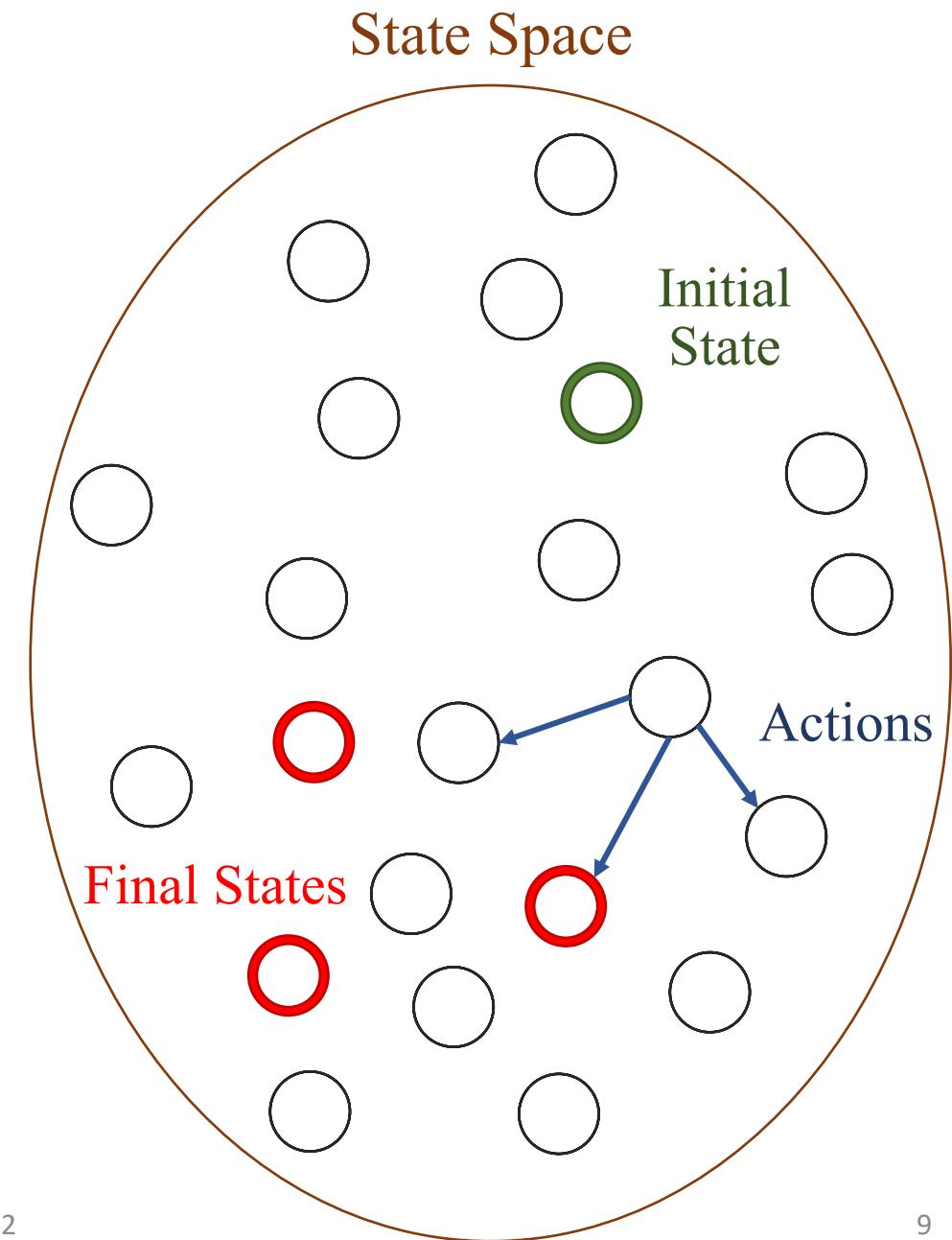


2.2.3 Search Problem

We are now ready to formally define a search problem.

A search problem consists of the following:

- S : the full set of states
- s_0 : the initial state
- $A: S \rightarrow S$ is a set of operators or actions
- G is the set of final states.
- Note that $G \subseteq S$

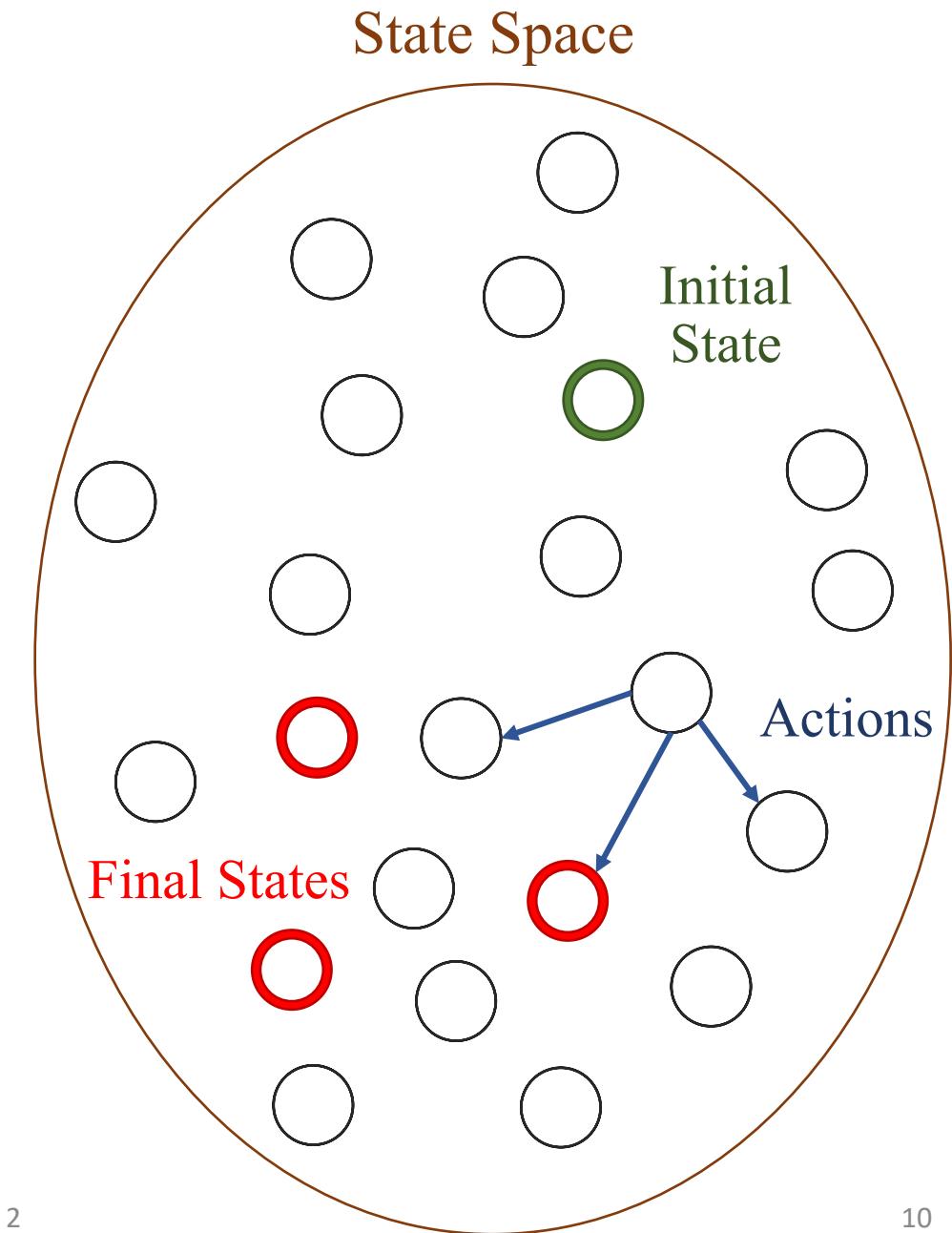


2.2.3 Search Problem

The search problem is to find a sequence of actions which transforms the agent from the initial state (s_0) to a goal state, $g \in G$.

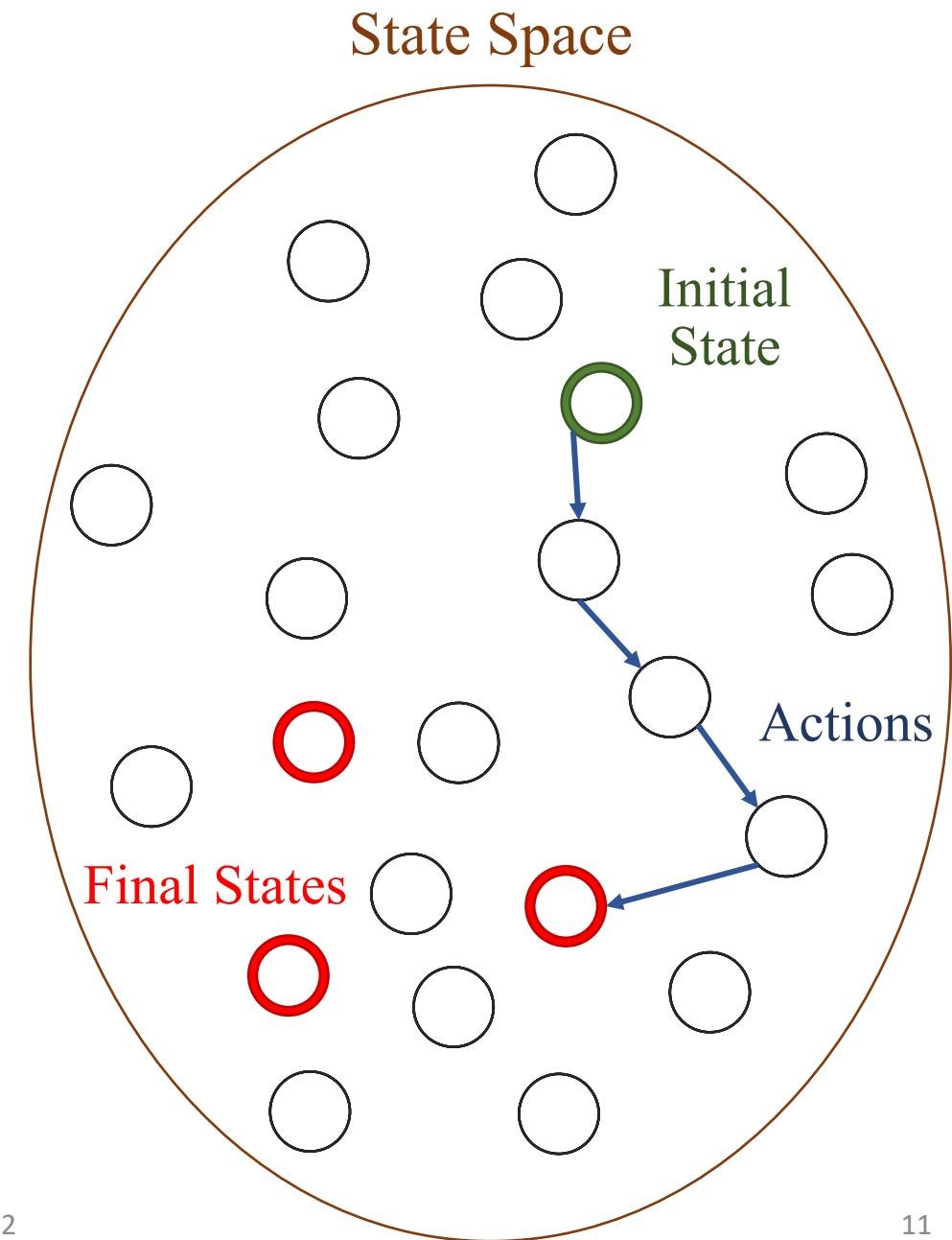
A search problem is represented by a 4-tuple $\{S, s_0, A, G\}$.

- S : the full set of states
- s_0 : the initial state
- A : $S \rightarrow S$ is a set of operators or actions
- G is the set of final states, where $G \subseteq S$



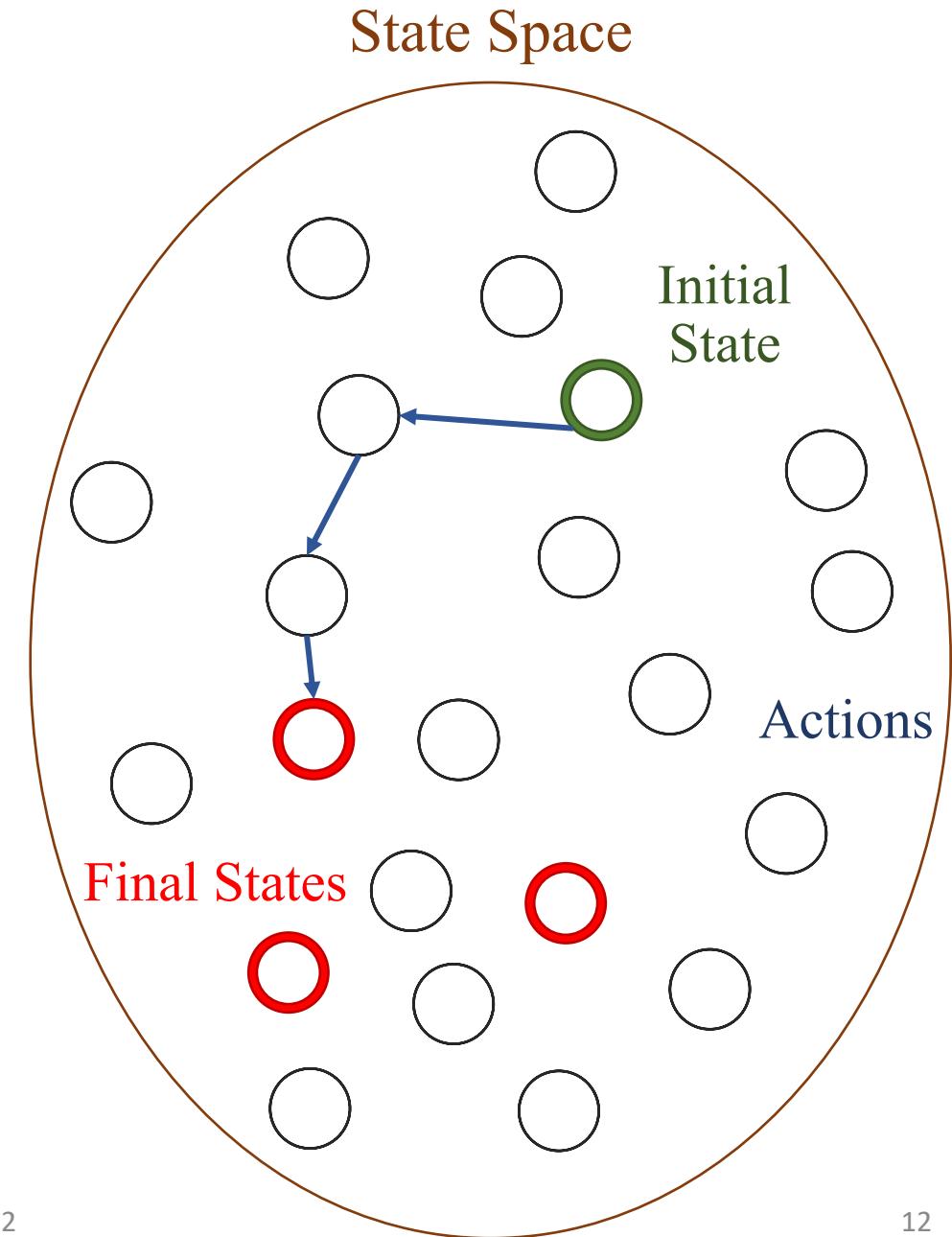
2.2.3 Search Problem

- That sequence of actions which leads the from initial state to a goal state is called a solution plan.



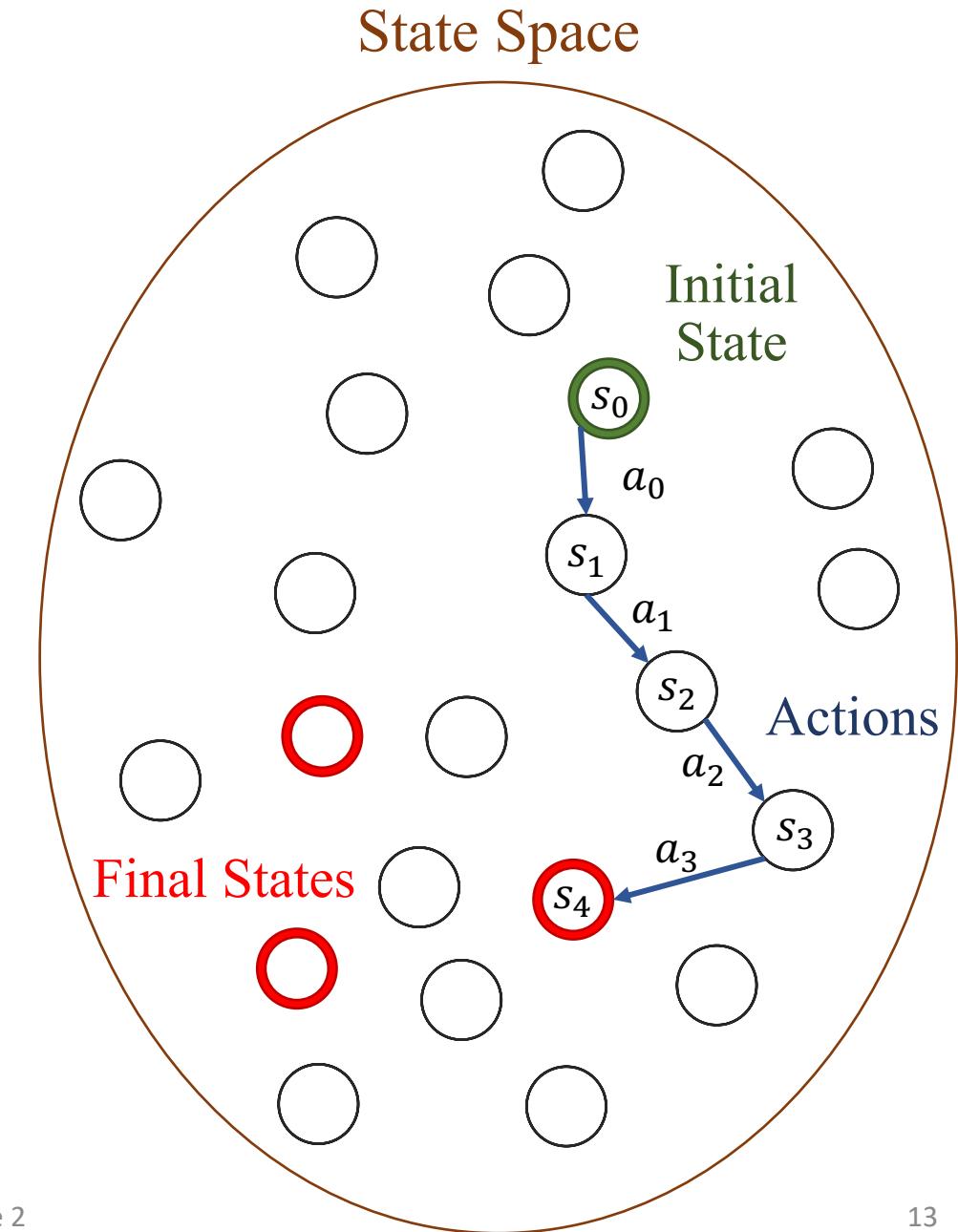
2.2.3 Search Problem

- That sequence of actions which leads the from initial state to a goal state is called a solution plan.



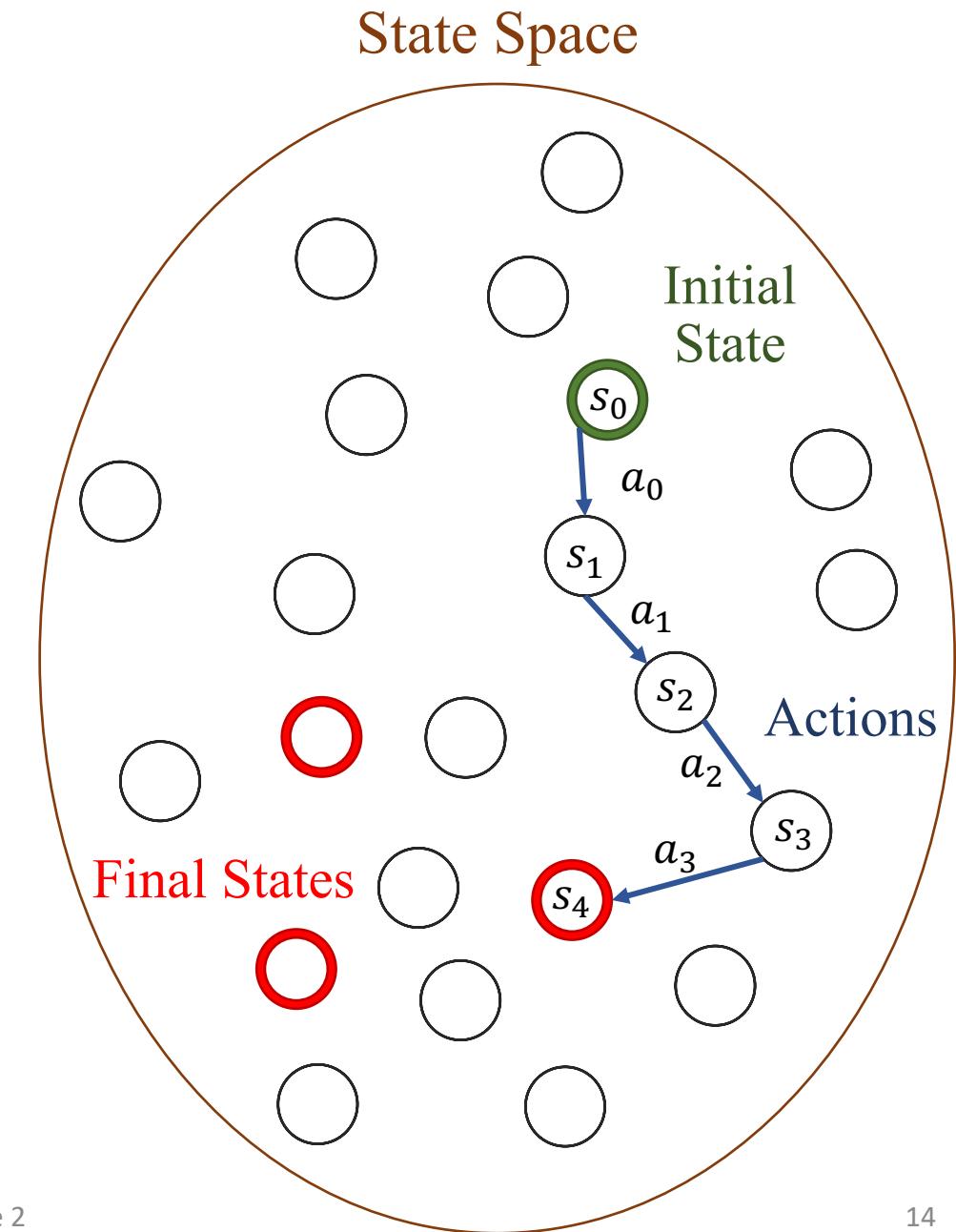
2.2.3 Search Problem

- That sequence of actions which leads the from initial state to a goal state is called a solution plan.
- A plan P is a sequence of actions.
- $P = \{a_0, a_1, \dots, a_N\}$ which leads to traversing a number of states $\{s_0, s_1, \dots, s_{N+1}\}$.



2.2.3 Search Problem

- That sequence of actions which leads the from initial state to a goal state is called a solution plan.
- A plan P is a sequence of actions.
- $P = \{a_0, a_1, \dots, a_N\}$ which leads to traversing a number of states $\{s_0, s_1, \dots, s_{N+1}\}$.
- A sequence of states is called a path.
- The cost of a path is a positive number, which is computed by taking the sum of the costs of each action.



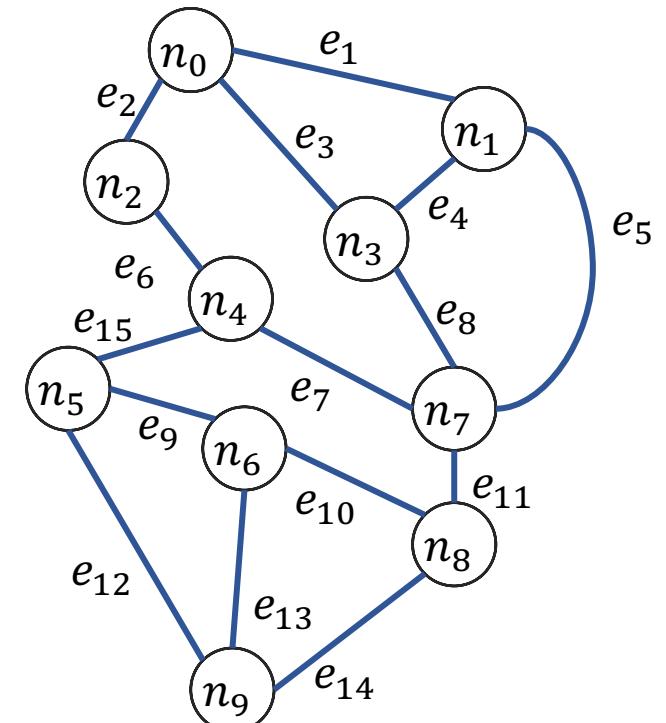
2.2.3.1 Representation of search problems

A search problem is represented using a **directed graph**. Specifically a **weighted directed graph**.

- The states are represented as nodes and actions are represented as arcs (or edges).

Graph Data Structure

- A graph is a non-linear data structure consisting of vertices and edges.
- The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a graph G is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by $G(E, V)$.



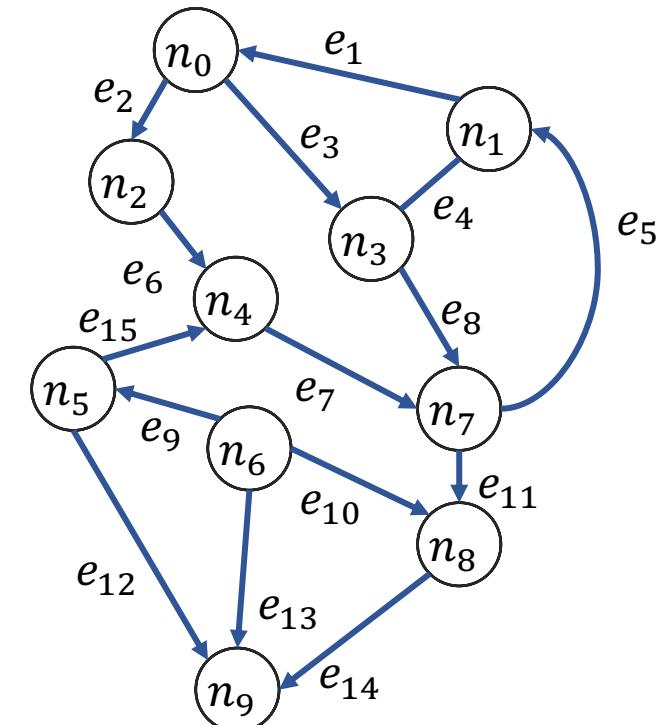
2.2.3.1 Representation of search problems

A search problem is represented using a **directed graph**. Specifically a **weighted directed graph**.

- The states are represented as nodes and actions are represented as arcs (or edges).

Graph Data Structure

- A graph is a non-linear data structure consisting of vertices and edges.
- The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a graph G is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by $G(E, V)$.



Edge	Weight
e_1	2
e_2	5
\vdots	\vdots
e_{14}	7
e_{15}	6

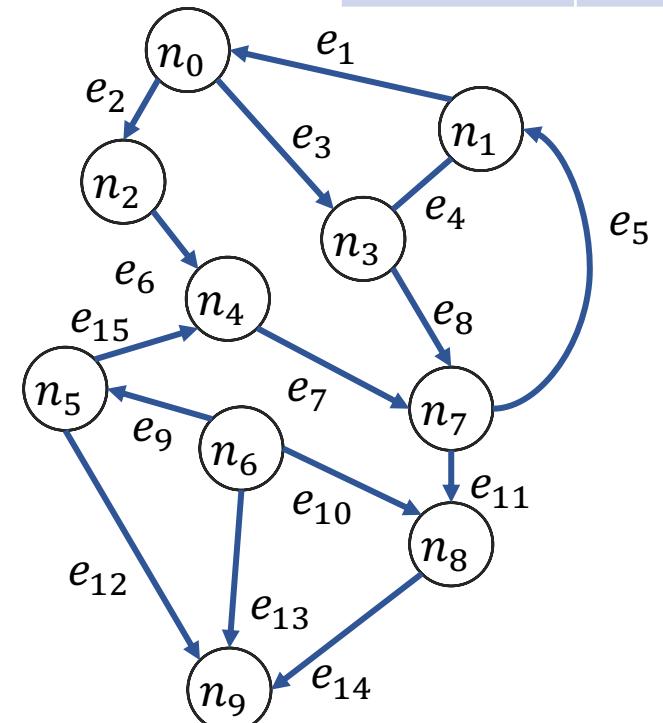
2.2.3.1 Representation of search problems

A search problem is represented using a **directed graph**. Specifically a **weighted directed graph**.

- The states are represented as nodes and actions are represented as arcs (or edges).

Graph Data Structure

- A graph is a non-linear data structure consisting of vertices and edges.
- The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a graph G is composed of a set of vertices (V) and a set of edges (E). The graph is denoted by $G(E, V)$.



2.2.2.2 Searching process

The generic searching process can be very simply described in terms of the following steps:

Do until a solution is found or the state space is exhausted.

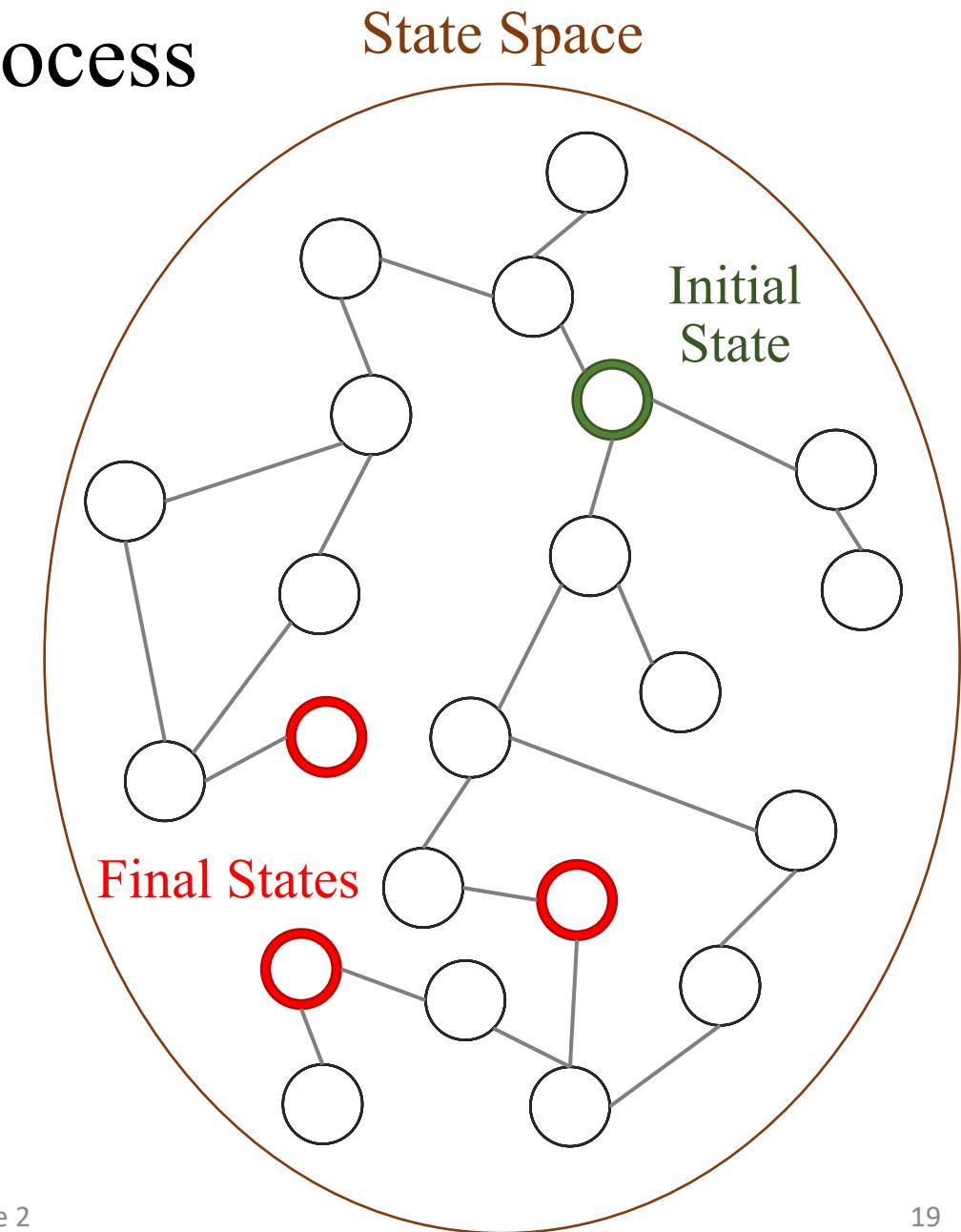
1. Check the current state.
2. Execute allowable actions to find the successor states.
3. Pick one of the new states.
4. Check if the new state is a solution state:
 - a) if **yes**, stop the process
 - b) if **no**, the new state becomes the current state and the process is repeated.

2.3.1 Illustration of a search process

We will now illustrate the searching process with the help of an example.

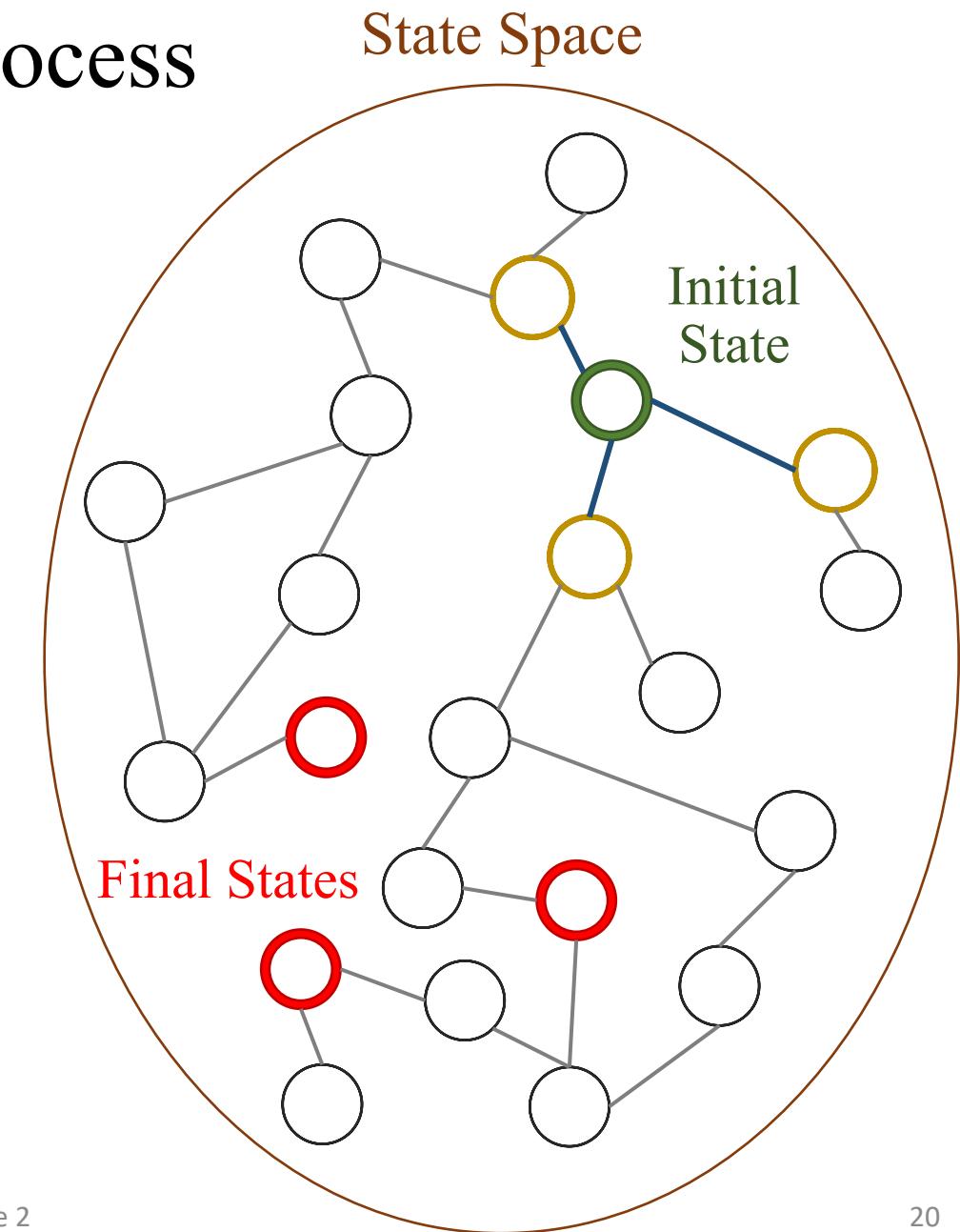
Consider the problem depicted in the figure.

- s_0 is the initial state.
- There are three **goal states**.
- The successor states are the adjacent states in the graph.



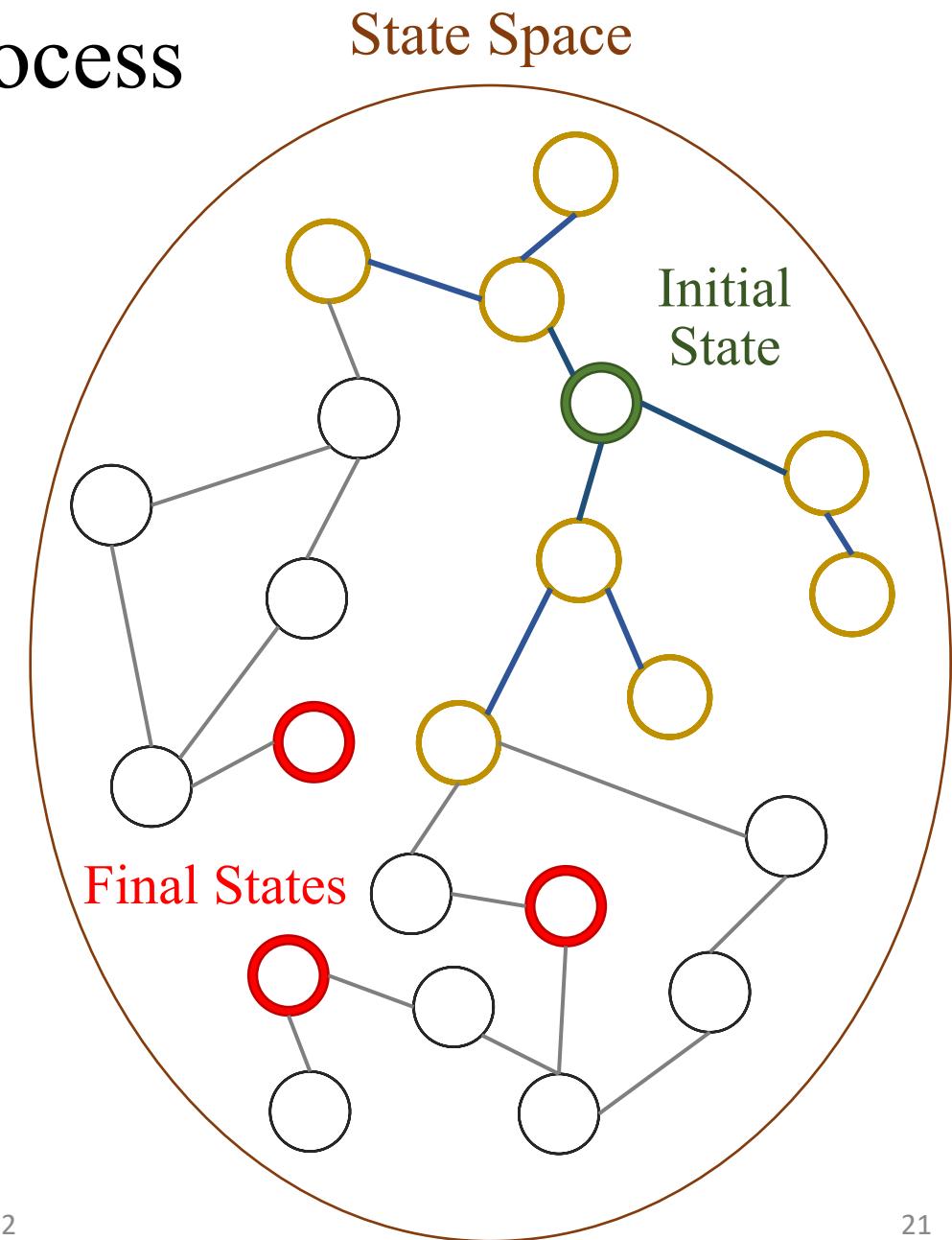
2.3.1 Illustration of a search process

- The three successor states of the initial state are generated.
- For each state we check, whether it is a goal state (or final state or solution state).
- Since none of the successor states are goal state, The successors of these states are picked and their successors are generated.



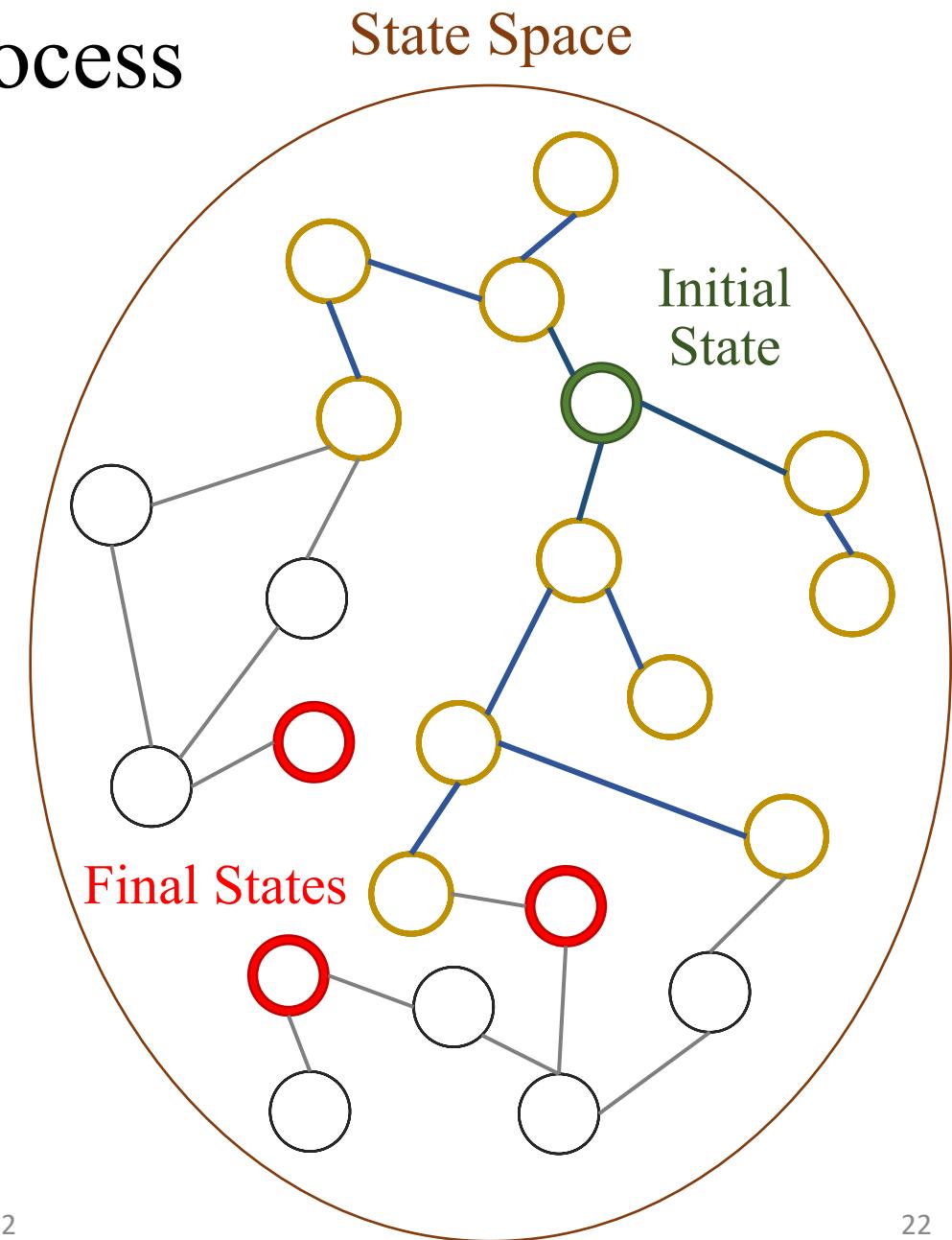
2.3.1 Illustration of a search process

- For each state we check, whether it is a goal state.
- Since none of the successor states are goal state, The successors of these states are picked and their successors are generated.



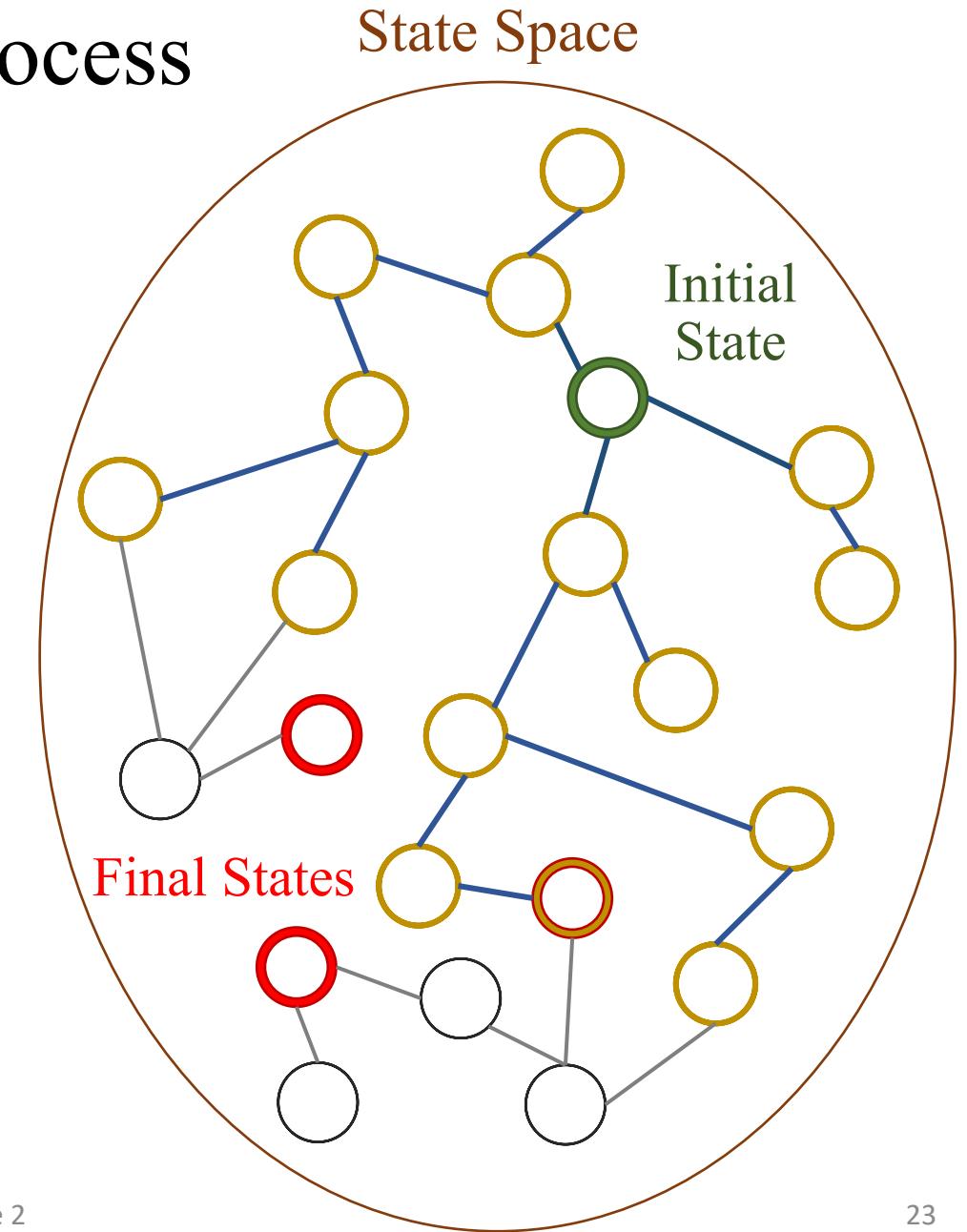
2.3.1 Illustration of a search process

- For each state we check, whether it is a goal state.
- Since none of the successor states are goal state, The successors of these states are picked and their successors are generated.



2.3.1 Illustration of a search process

- A goal state has been found. Stop the search.
- This example illustrates how we can start from a given state and follow the successors, and be able to find solution paths that lead to a goal state.
- The yellowed nodes define the search tree.
- Usually the search tree is extended one node at a time.
- The order in which the search tree is extended depends on the search strategy.



2.3.1 Illustration of a search process – Pegs and Disks problem

Problem statement

- Consider the following problem.
- We have 3 pegs and 3 disks.

Operators

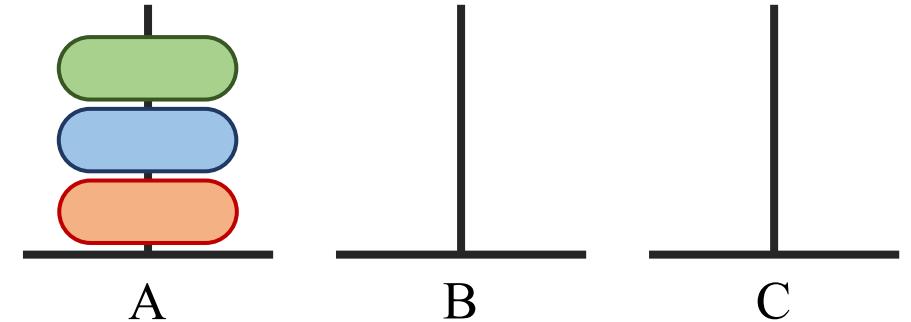
- One may move the topmost disk on any needle to the topmost position to any other needle

Goal

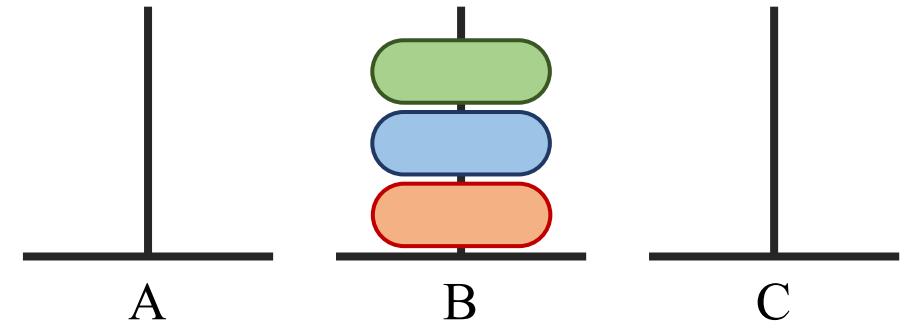
- In the goal state all the disks are in the peg B as shown in the figure.

Initial state

- In the initial state all the disks are in the peg A as shown in the figure.



Initial state



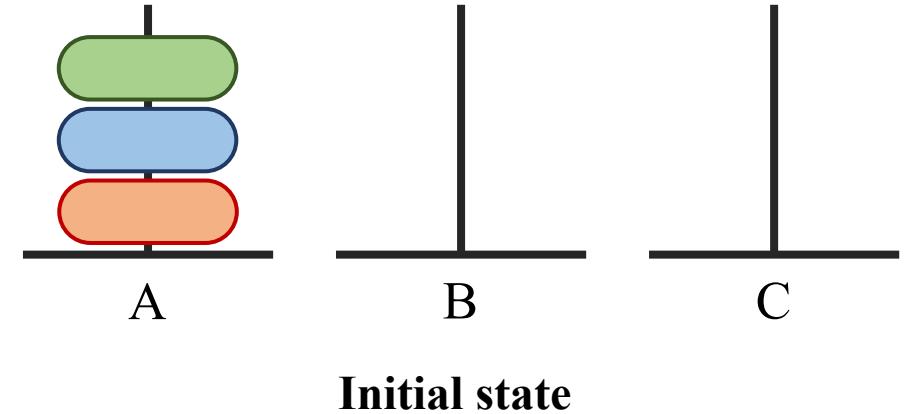
Goal state

2.3.1 Illustration of a search process – Pegs and Disks problem

- In the initial state all the disks are in the peg A as shown in the figure.

The agent may choose to perform either of the operations:

- Move green from A to C
- Move green from A to B



Recall that in our previous example we traversed multiple paths in single iteration.

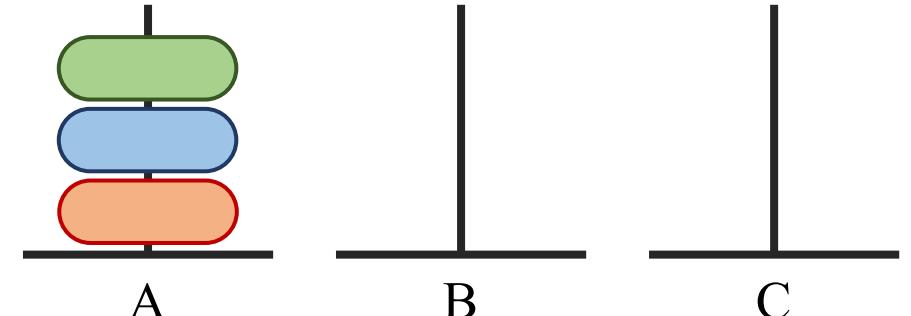
However, generally we traverse one node at a time.

The search tree is extended one node at a time.

2.3.1 Illustration of a search process – Pegs and Disks problem

The agent may choose to perform either of the operations:

- Move green from A to C
- Move green from A to B

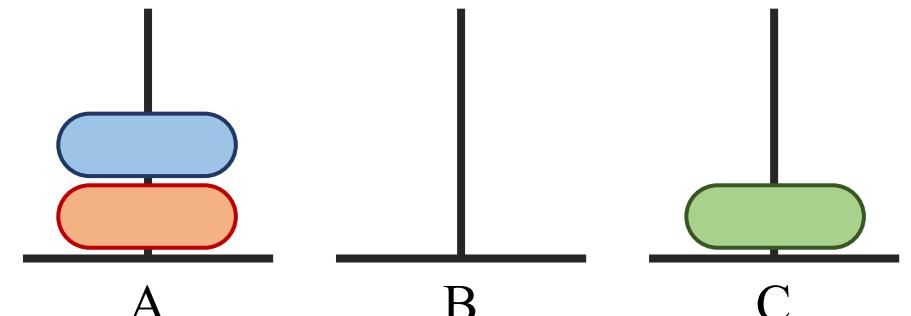


Initial state (Step 0)

We will choose one operation for now, and in future if it fails, we will backtrack and explore the remaining operations.

Which to choose?

- [For now: randomly]
- Move green from A to C

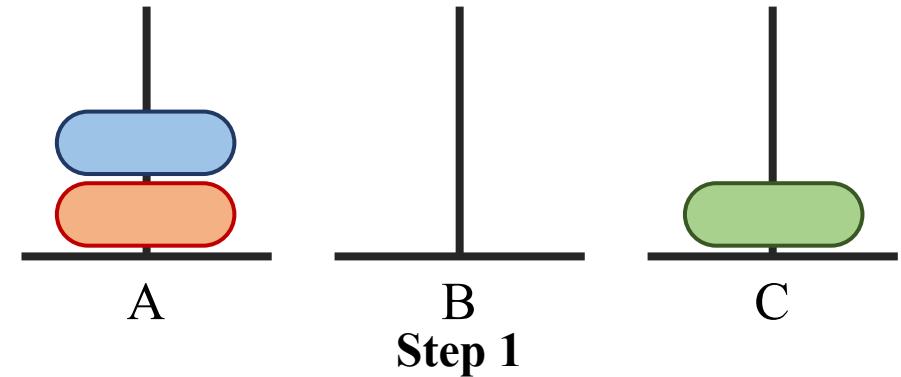


Step 1

2.3.1 Illustration of a search process – Pegs and Disks problem

Step 2: We again have multiple options:

- Move green from C to A
- Move green from C to B
- Move blue from A to B
- Move blue from A to C

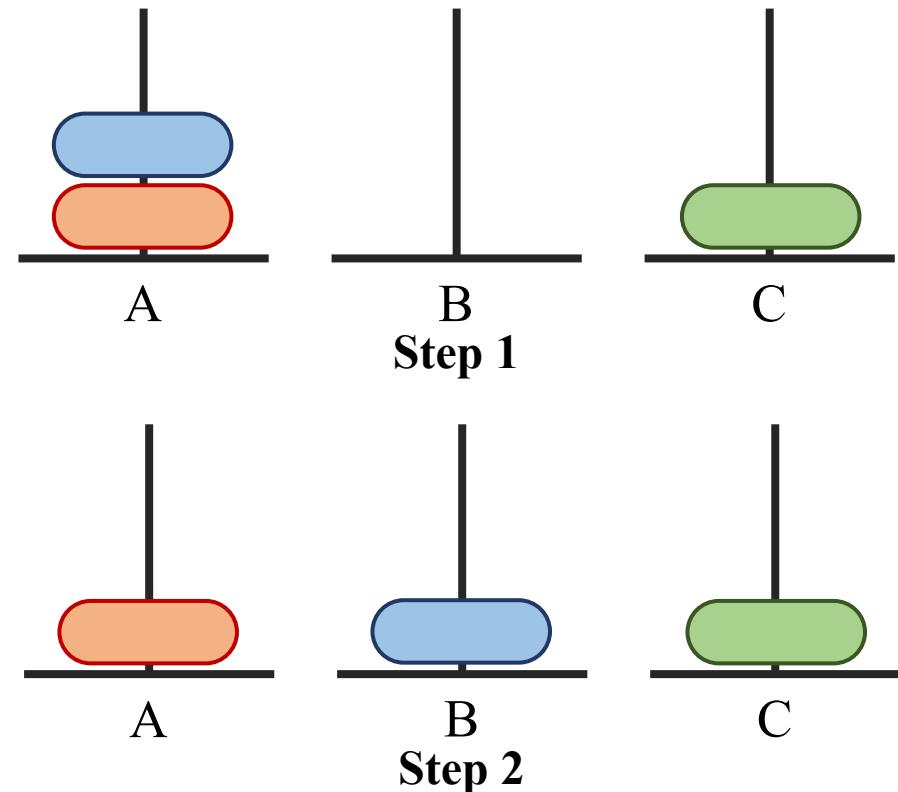


2.3.1 Illustration of a search process – Pegs and Disks problem

Step 2: We again have multiple options:

- ~~Move green from C to A~~
- Move green from C to B
- Move blue from A to B
- Move blue from A to C

We choose the operation [Move blue from A to B]

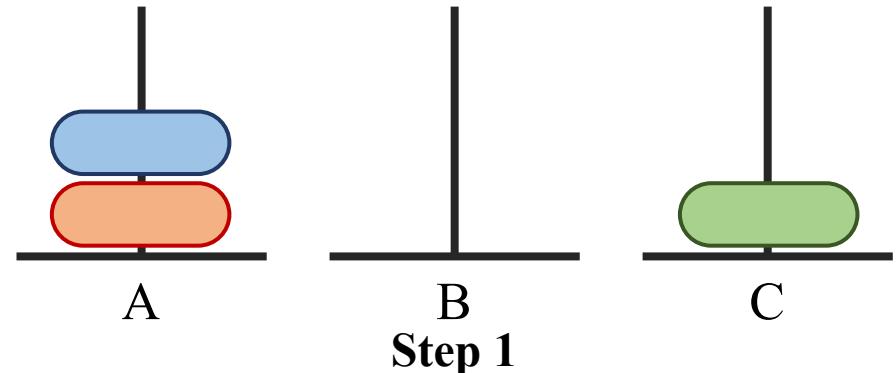


2.3.1 Illustration of a search process – Pegs and Disks problem

Step 2: We again have multiple options:

- ~~Move green from C to A~~
- Move green from C to B
- Move blue from A to B
- Move blue from A to C

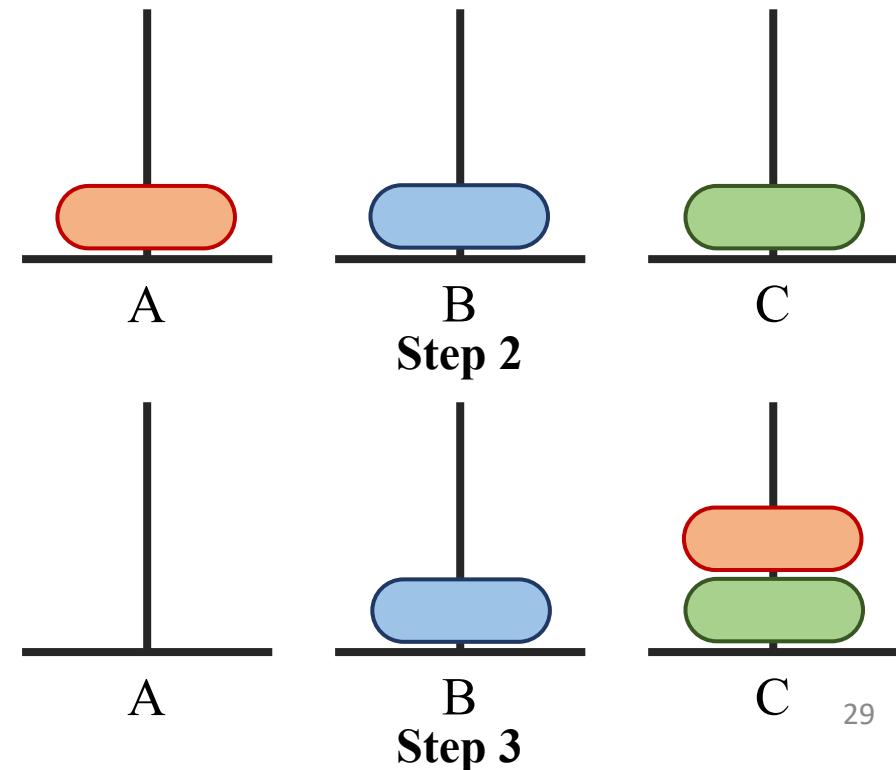
We choose the operation [Move blue from A to B]



Step 3: We again have multiple options:

- Move green from C, to A or B
- Move blue from B to C
- Move red from A, to B or C

We choose the operation [Move red from A to C]

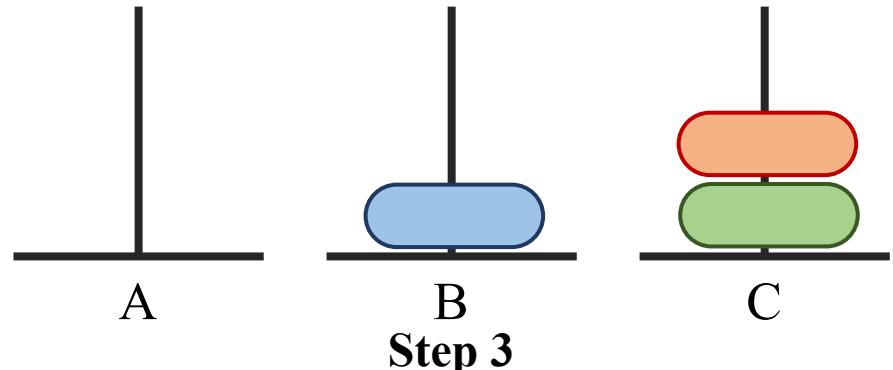


2.3.1 Illustration of a search process – Pegs and Disks problem

Step 4: We again have multiple options:

- Move blue from B, to A or C
- Move red from C to, A or B

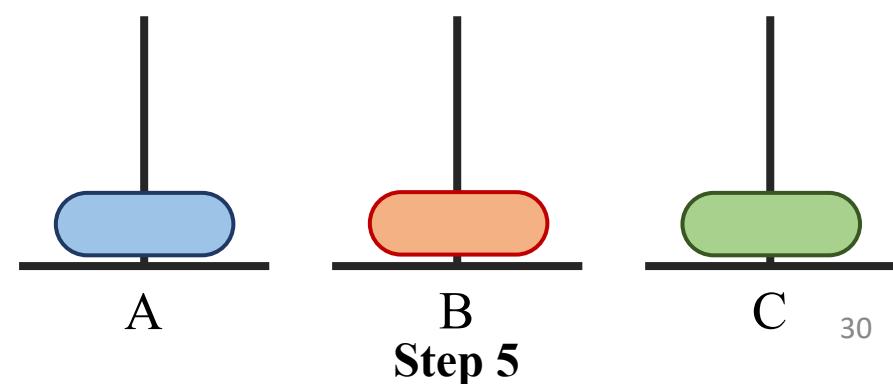
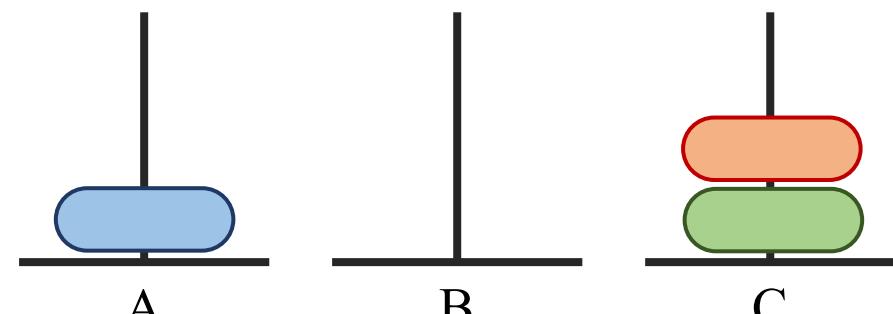
We choose the operation [Move blue from B to A]



Step 5: We again have multiple options:

- Move red from C to, A or B
- Move blue from A to C

We choose the operation [Move red from C to B]

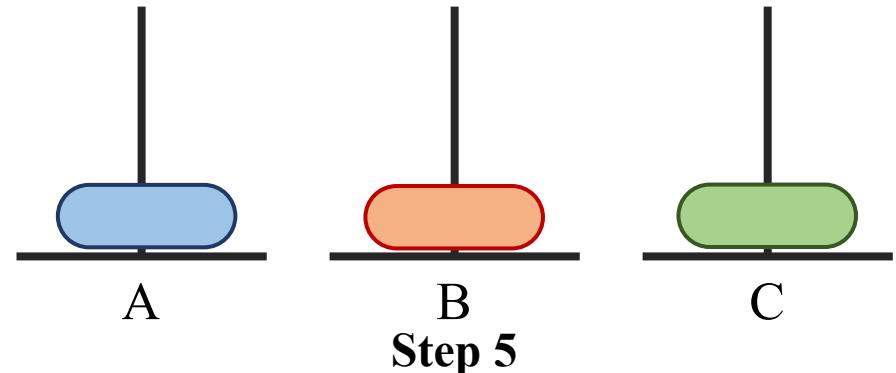


2.3.1 Illustration of a search process – Pegs and Disks problem

Step 6: We again have multiple options:

- Move red from B to A
- Move blue from A to, B or C
- Move green from C to, A or B

We choose the operation [Move blue from A to B]

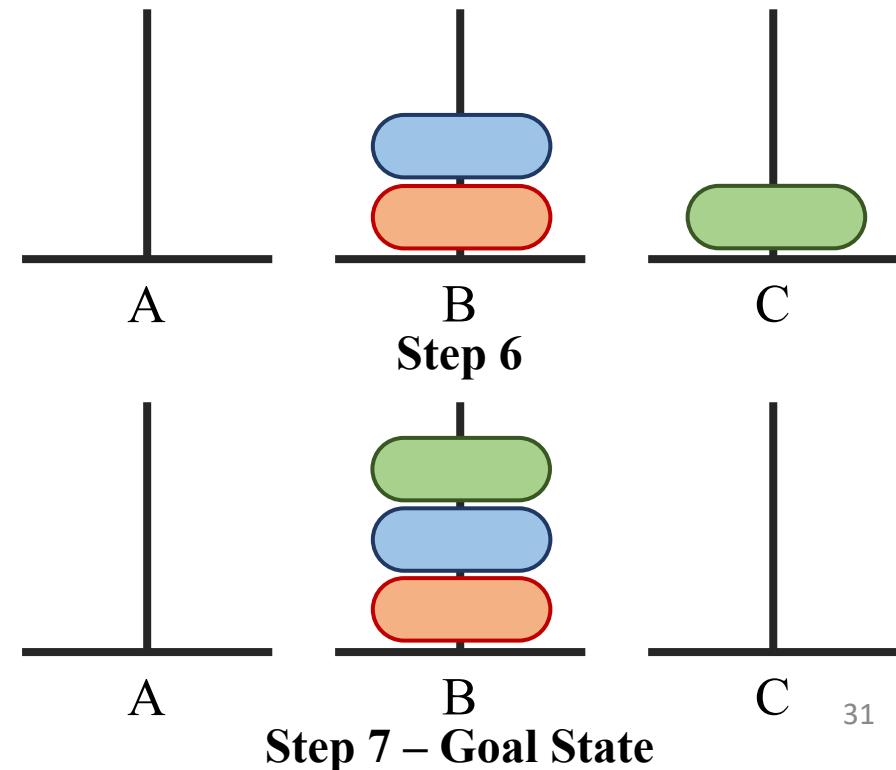


Step 7: We again have multiple options:

- Move green from C to, A or B
- Move blue from B to C

We choose the operation [Move red from C to B]

We reached our goal state. Hence Stop.



2.4 Search

Searching through a state space involves the following:

- A set of states
- Operators and their costs (In previous examples cost was same for every operator)
- Start state (or initial state)
- Goal state(s)

A test to check whether a state is a goal state or not.

2.4.1 Search algorithm: Key issues

Corresponding to a search algorithm, we get a search tree which contains the generated and the explored nodes.

- The search tree may be unbounded.
- This may happen if the state space is infinite.
- This can also happen if there are loops in the search space (*or cycle in the graph*).
- How can we handle loops?

2.4.1 Search algorithm: Key issues

Corresponding to a search algorithm, should we return a path or a node?

The answer to this depends on the problem.

- For problems like “Pegs and Disks”, we are interested in the solution path.
- For problems like “Magic Square” we are only interested in the goal state.

4	9	2
3	5	7
8	1	6

2	7	6
9	5	1
4	3	8

Magic Squares adding up to 15

2.4.1 Search algorithm: Key issues

- Depending on the search problem, we will have different cases.
- The search graph may be weighted or unweighted.
- In some cases we may have some knowledge about the quality of intermediate states and this can perhaps be exploited by the search algorithm.

The objective of a search problem is to find a path from the initial state to a goal state.

- If there are several paths which path should be chosen?
- Our objective could be to find any path, or
- Our objective could that we need to find the shortest path or the least cost path.

2.4.2 Evaluating Search strategies

1. **Completeness:** Is the strategy guaranteed to find a solution if one exists?
2. **Optimality:** Does the solution have low cost or the minimal cost?
3. What is the search cost associated with the time and memory required to find a solution?
 - a) **Time complexity:** Time taken (*or number of nodes expanded*), in worst or average cases, to find a solution.
 - b) **Space complexity:** Space used by the algorithm measured in terms of the maximum size of search tree.

2.5 Breadth First Search

BFS (u)

```
Q = {u} // Q is a normal queue  
while !Q.empty  
    u = Q.pop  
    for each neighbour v of u  
        if v is unvisited  
            Q.push(v) // tree edge  
        else if v is visited  
            we ignore this edge
```

BFS(G, s)

```
1 for each vertex  $u \in G.V - \{s\}$   
2      $u.color = \text{WHITE}$   
3      $u.d = \infty$   
4      $u.\pi = \text{NIL}$   
5      $s.color = \text{GRAY}$   
6      $s.d = 0$   
7      $s.\pi = \text{NIL}$   
8      $Q = \emptyset$   
9     ENQUEUE( $Q, s$ )  
10    while  $Q \neq \emptyset$   
11         $u = \text{DEQUEUE}(Q)$   
12        for each  $v \in G.Adj[u]$   
13            if  $v.color == \text{WHITE}$   
14                 $v.color = \text{GRAY}$   
15                 $v.d = u.d + 1$   
16                 $v.\pi = u$   
17                ENQUEUE( $Q, v$ )  
18             $u.color = \text{BLACK}$ 
```

2.6 Depth First Search

DFS (u)

for each neighbour v of u

if v is unvisited

 DFS (v) // tree edge

else if v is explored

 ignore

 // bidirectional/back edge

else if v is visited

 ignore

 // forward/cross edge

// Note: Recursion inherently uses stacks

DFS(G)

```
1 for each vertex  $u \in G.V$ 
2      $u.color = \text{WHITE}$ 
3      $u.\pi = \text{NIL}$ 
4      $time = 0$ 
5 for each vertex  $u \in G.V$ 
6     if  $u.color == \text{WHITE}$ 
7         DFS-VISIT( $G, u$ )
```

DFS-VISIT(G, u)

```
1  $time = time + 1$ 
2  $u.d = time$ 
3  $u.color = \text{GRAY}$ 
4 for each  $v \in G.Adj[u]$ 
5     if  $v.color == \text{WHITE}$ 
6          $v.\pi = u$ 
7         DFS-VISIT( $G, v$ )
8      $v.color = \text{BLACK}$ 
9      $time = time + 1$ 
10     $u.f = time$ 
```

Please note

Both the algorithms are borrowed verbatim from:

Cormen, Thomas H., et al. Introduction to algorithms. MIT press, 2022.

2.5.1 Properties of BFS

Breadth first search is:

- **Complete**
- The algorithm is **optimal** (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- The algorithm has **exponential time and space complexity**. Suppose the search tree can be modeled as a b -ary tree. Then the time and space complexity of the algorithm is $O(b^d)$ where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node.

2.6.1 Properties of DFS

- The algorithm takes exponential time.
- If N is the maximum depth of a node in the search space, in the worst case the algorithm will take time $O(b^d)$.
- However the space taken is linear in the depth of the search tree, $O(d)$.
- Note that the time taken by the algorithm is related to the maximum depth of the search tree.
- If the search tree has infinite depth, the algorithm may not terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycles.
- The latter case can be handled by checking for cycles in the algorithm. Thus Depth First Search is **not complete**.

2.6.2 Depth Limited Search

DLS (u)

for each neighbour v of u

if v is unvisited **and** $v.\text{depth} < \text{bound}$

 DLS (v)

else if v is explored

 ignore

else if v is visited

 ignore

else if $v.\text{depth} \geq \text{bound}$

 ignore

- A variation of Depth First Search circumvents the *infinite depth* problem by keeping a depth bound.

- Nodes are only expanded if they have depth less than the bound. This algorithm is known as depth-limited search.

Contact Details

- Anup Kumar Gupta
- PhD scholar, IIT Indore
- Email: msrphd2105101002@iiti.ac.in
- Website: <https://anupkumargupta.github.io/>

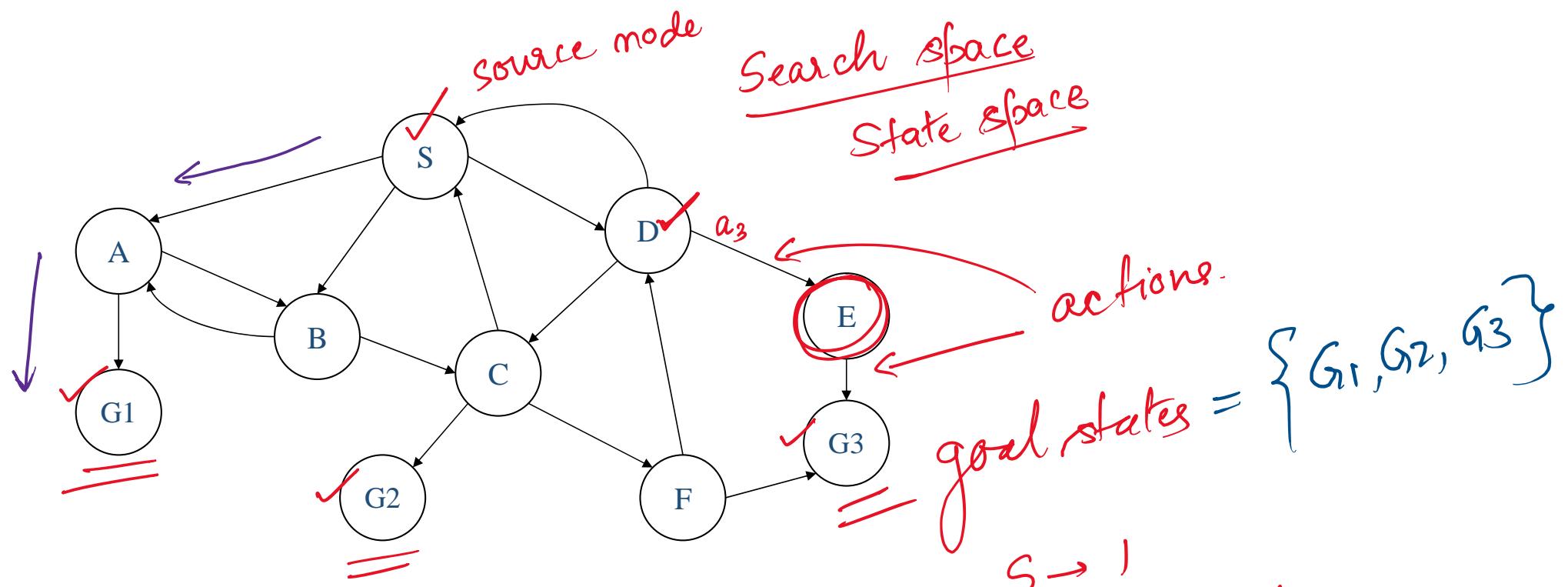
State based agents + Goal → Goal based agents → {G₁, G₂, G₃}

$S^{\checkmark} \xrightarrow{\text{task}} G^{\checkmark}$
task searching

Uninformed vs Informed Searches

Uninformed Searching ✓	Informed Searching ✓
1. Searching <u>without information</u>	1. Searching with <u>information</u> <u>heuristics</u>
2. No prior knowledge · <u> </u> <u>Blind search</u>	2. Uses <u>some prior knowledge</u> to move in direction of solution <u>decision</u>
3. Time Consuming ✓	3. Quick solution <u>(Quicker than us)</u>
4. More Complexity (Space and Time) ✓	4. Less Complexity (Space and Time) ✓
5. BFS, DFS, Iterative Deepening Search ✓	5. A*, Best First Search ✓

Anumaaan (hindi)



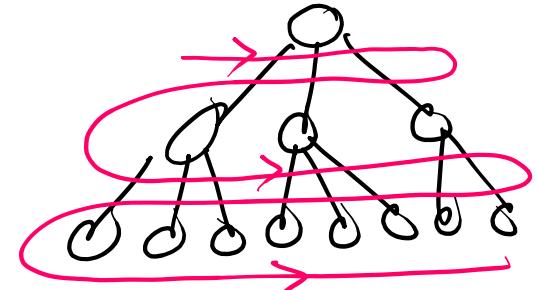
$S \xrightarrow{\text{path}} \{G_1, G_2, G_3\}$

searching

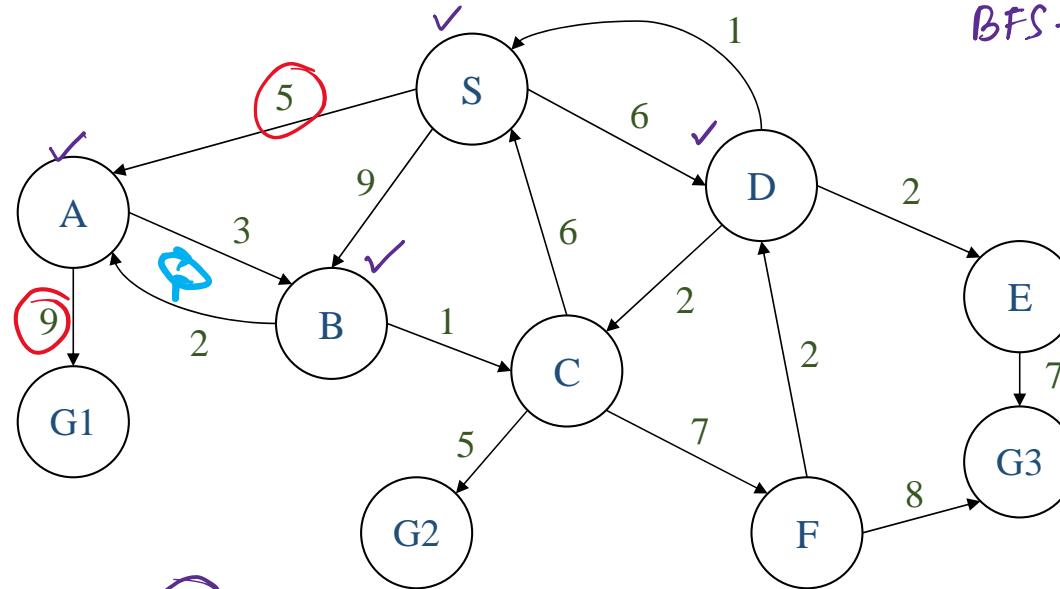
$S \rightarrow I$
 $G \rightarrow \text{multiple goal states}$

Properties of BFS (visit this slide after BFS)

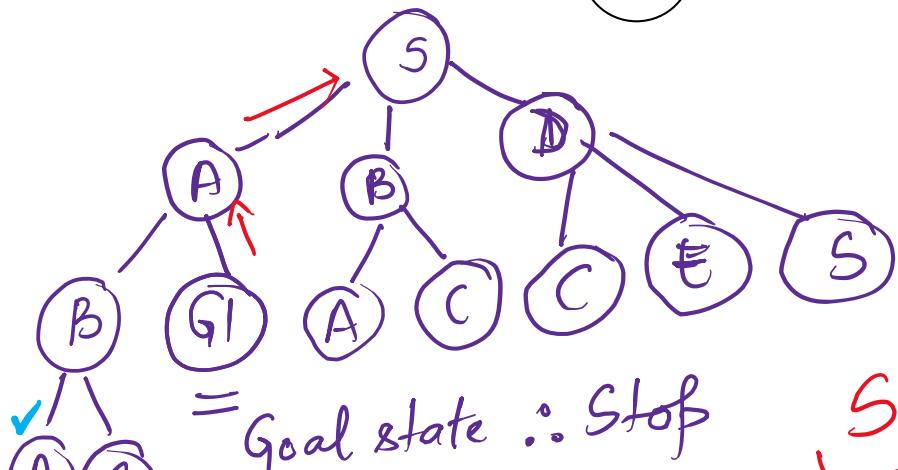
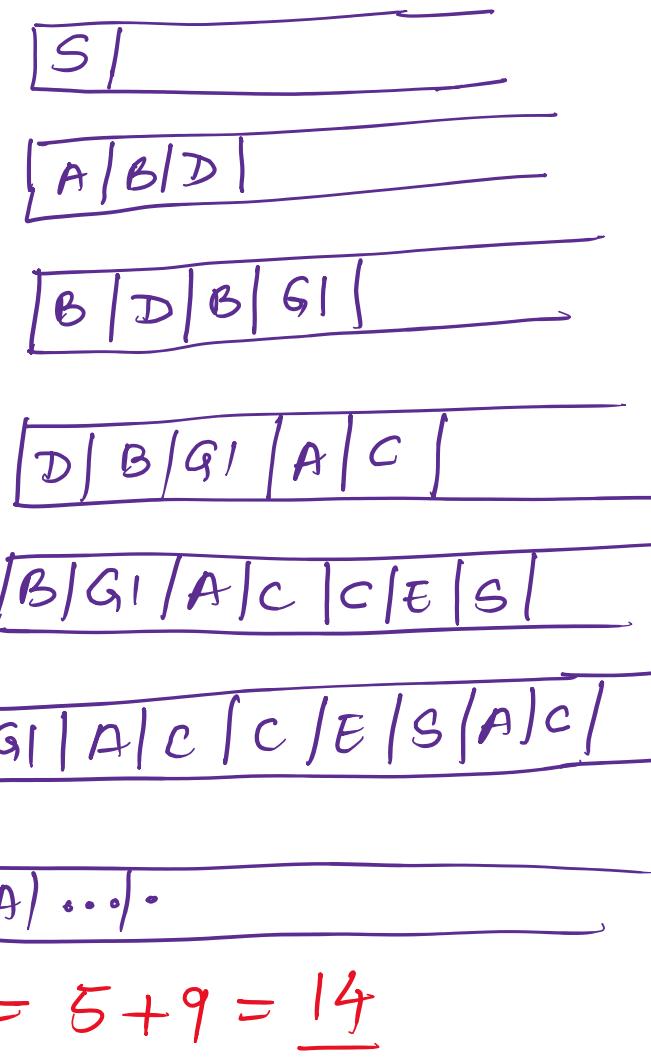
- ① uniformed. → (level wise search)
- ② FIFO (Queue)
- ③ Shallowest path to a given node.
- ④ Complete? — Yes. It will always provide an answer
- ⑤ Optimal — (Does it give the shortest path to the node?) → Yes ✓
- ⑥ Complexity — $O(V+E)$, $O(b^d)$ or $O(k^d)$ →
branching factor → k-ary
depth of tree / graph



BFS



BFS → Queue → FIFO

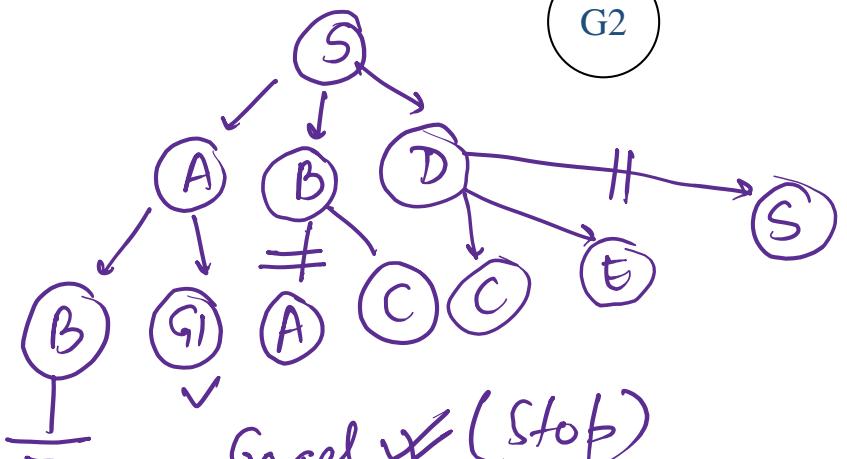
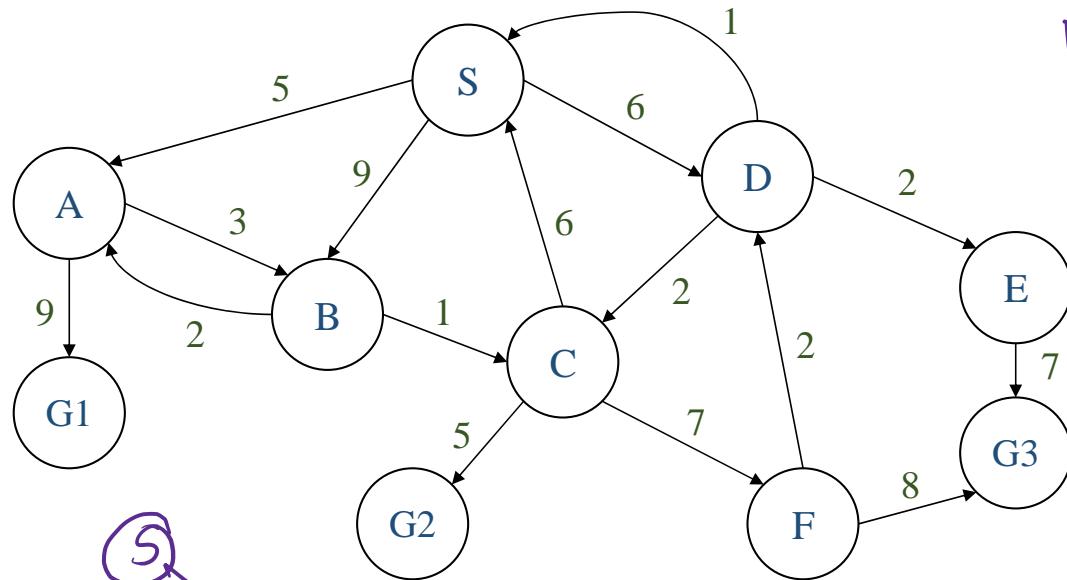


S-A-G1
path

$$\text{cost} = 5 + 9 = \underline{14}$$

visited list

BFS*



dead end

Goal ✓ (stop)

$S \rightarrow A \rightarrow G1$

S | | | | |

A | B | D | |

B | D | B | G1 |

D | B | G1 | C | |

B | G1 | C | C | E |

G1 | C | C | E |

G | C | E | ---

| | | | |

S | | | |

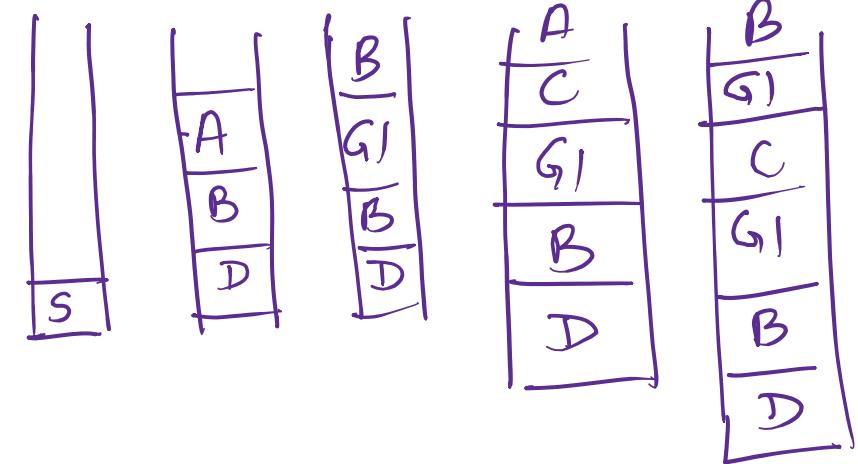
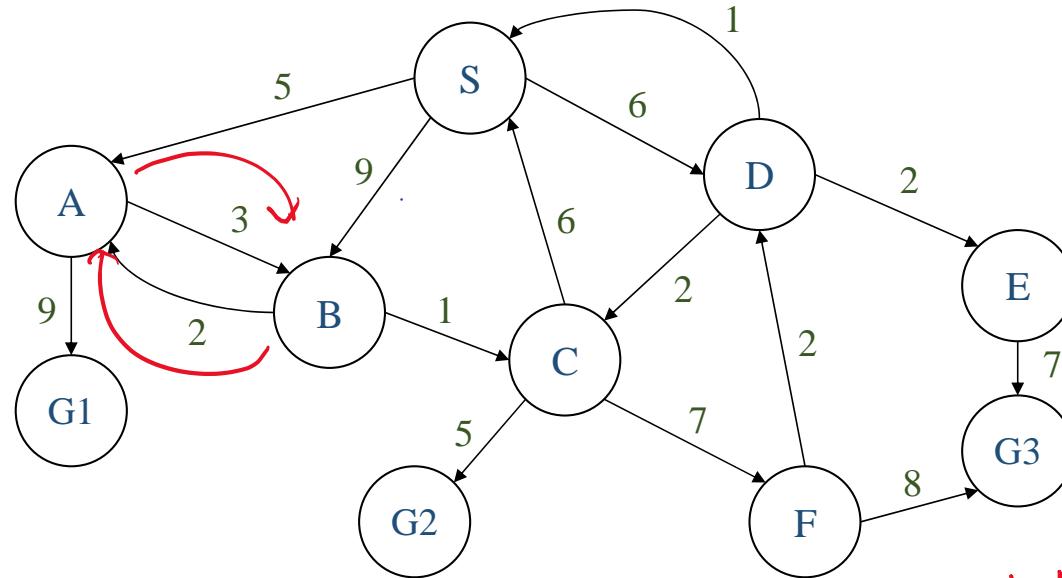
S | A | | |

S | A | B | |

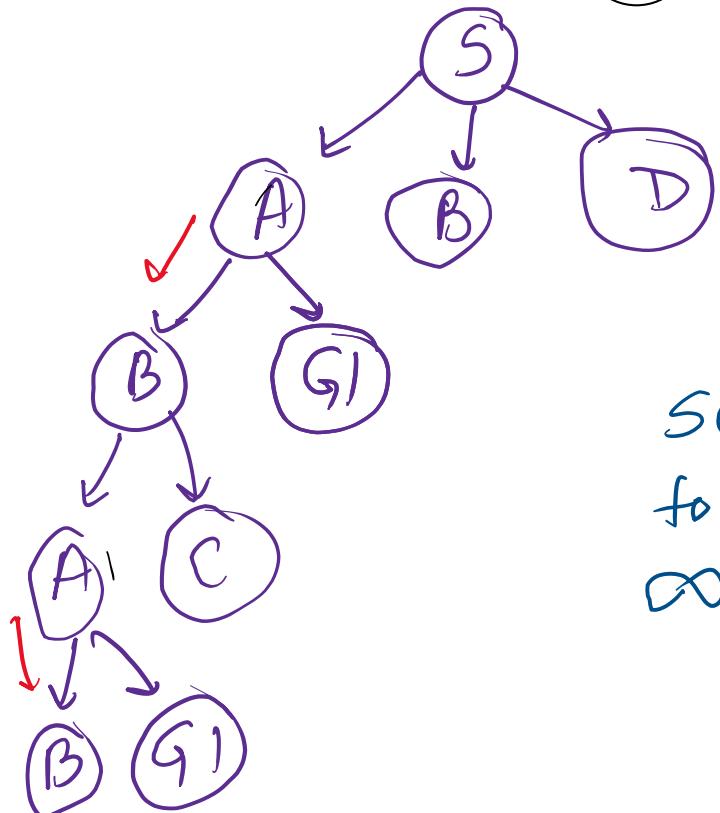
S | A | B | D |

S | P | B | D |

DFS (Stack) LIFO



depth can be seen as infinite depth
loops → not complete

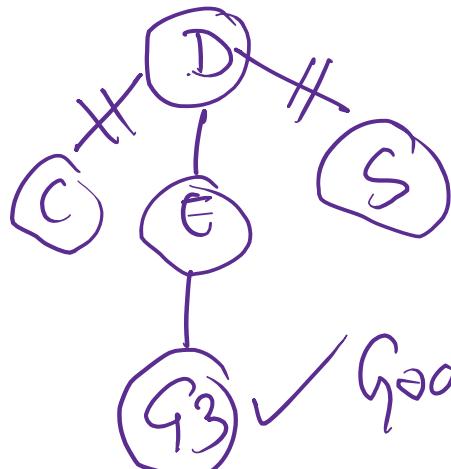
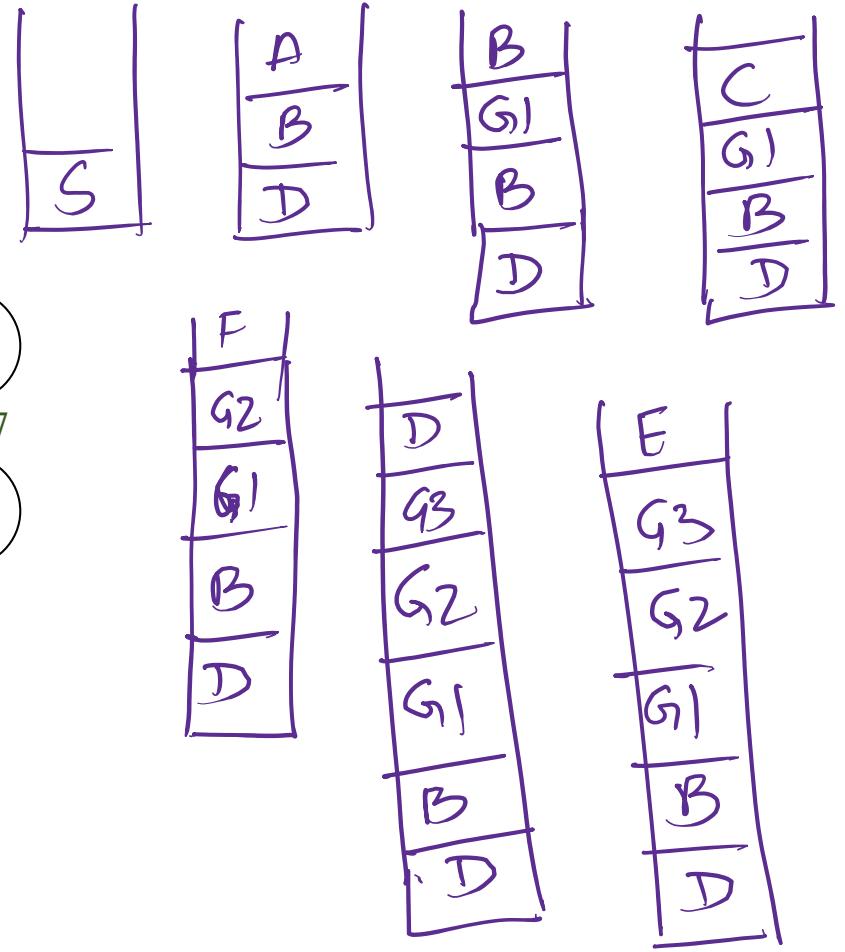
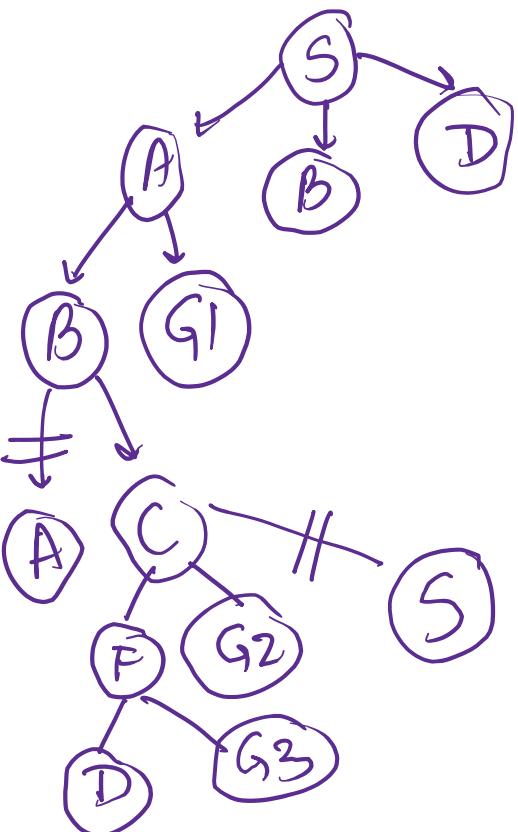
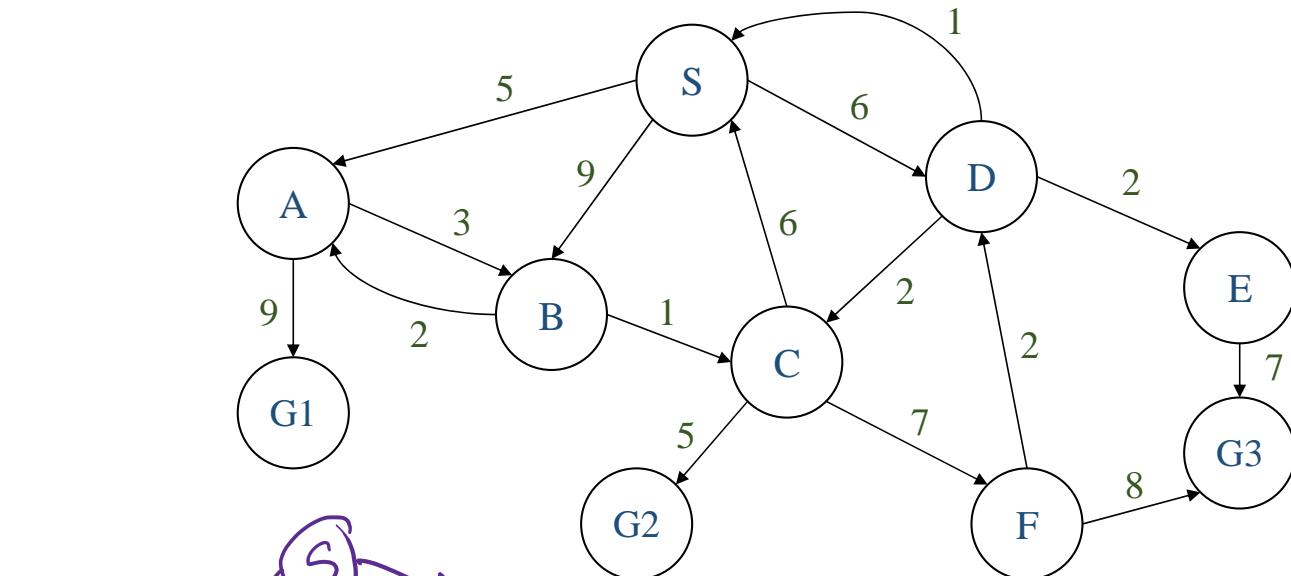


solution
to avoid
 ∞ loops

{ when you are expanding a
node and it gives us a node (C)
which we have already seen
earlier on that path.
then X Path is a
deadend

DFS⁺

e



✓ Goal!! Stop

$S \rightarrow A \rightarrow B \rightarrow C \rightarrow F \rightarrow D \rightarrow E \rightarrow G_3$

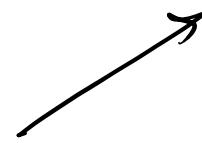
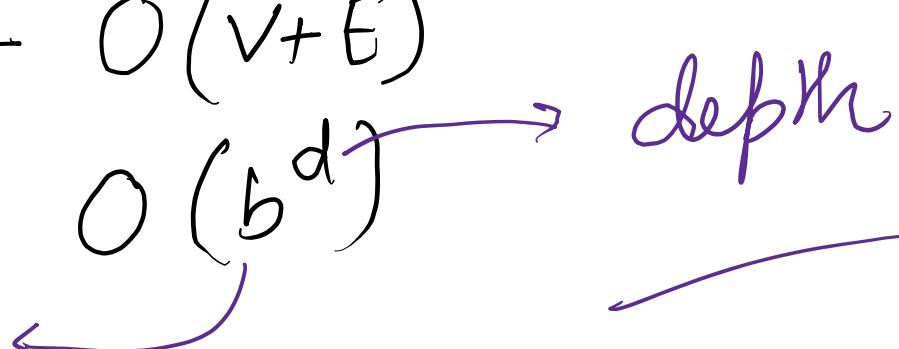
27

If DFS has all these issues, then why do we study DFS? Are there any benefits of DFS?

① If DFS does find a goal it is very fast in doing that.

② Lower Space complexity — what and how?
(Assignment).

Properties of DFS

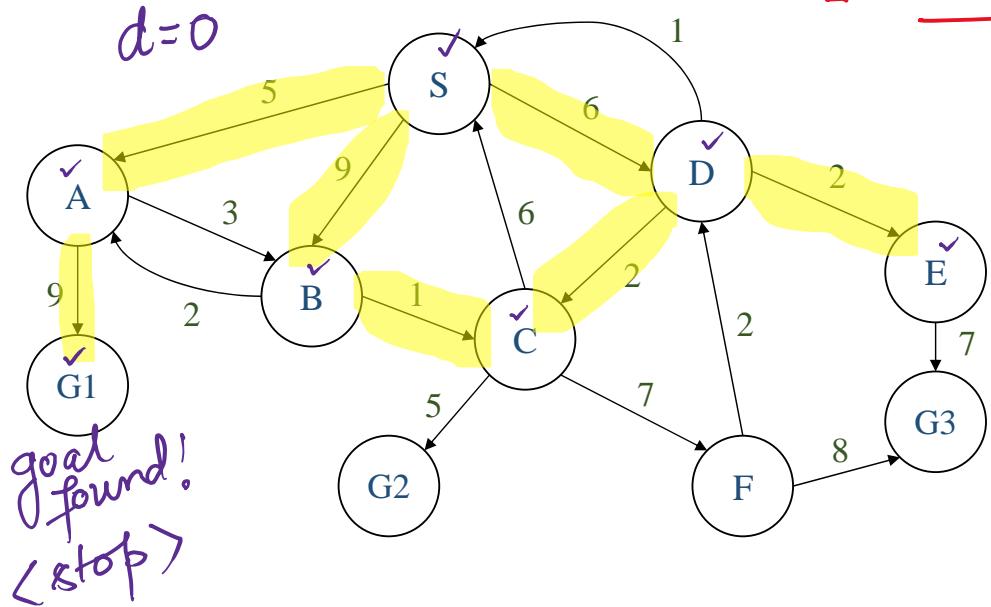
- ① uninformed
 - ② LIFO, (stack) 
 - ③ Gives us deepest path to the node.
 - ④ Not Complete (Possibility that it doesn't provide an answer, in case of loops)
 - ⑤ Not optimal (Doesn't provide shortest path to node)
 - ⑥ Time Complexity - $O(V+E)$
branching factor  $O(b^d)$ \rightarrow depth
- sometimes not explicitly used. But we perform recursion and in recursion we inherently use stack.

Iterative deepening DFS → what differs DFS

① fix the depth at every iteration say d

② and at that depth, we try to find whether we have solution or not.

③ if we find sol^{smiley face}
if not $d = d + 1$



Q1 Whether this algo is similar to BFS?

S - A - G1 Cost = 14

Heuristics ($31\overline{3}411-1$)

$$\textcircled{1} \quad x^2 = \underline{48}$$

$$\{x = 6.8\}$$

$$\{x = 4.5\} \quad \{x = 8.2\}$$

6.9282

Euclidean
dist

$$\textcircled{2} \quad x^2 = \underline{143}$$

$$\{x = 11.9\}$$

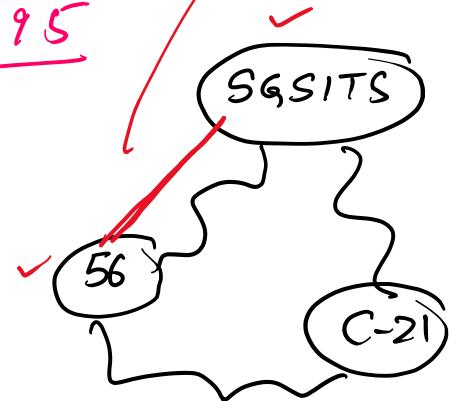
$$\{x = 8.5\} \quad \{x = 12.3\}$$

11.95

SGSITS

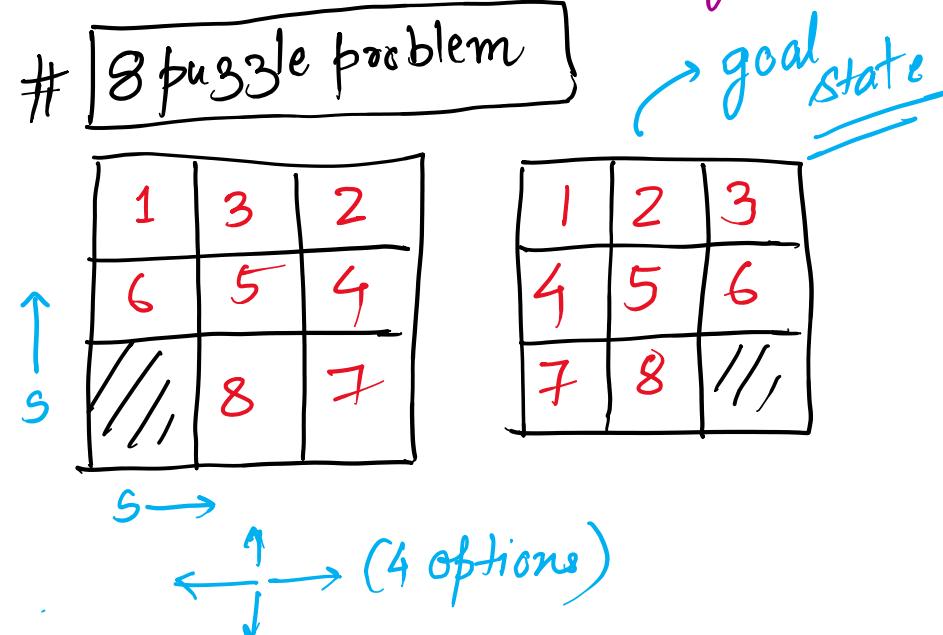
finding actual values could costly.

heuristics are quick and they provide some sense of direction!



(NP) problems → which cannot be solved in polynomial time.
using heuristics, we can aim / we may get answers in P time.

↳ Uniformed Searches are almost guaranteed to give correct answer, but cost is huge.



a) misplaced tiles count

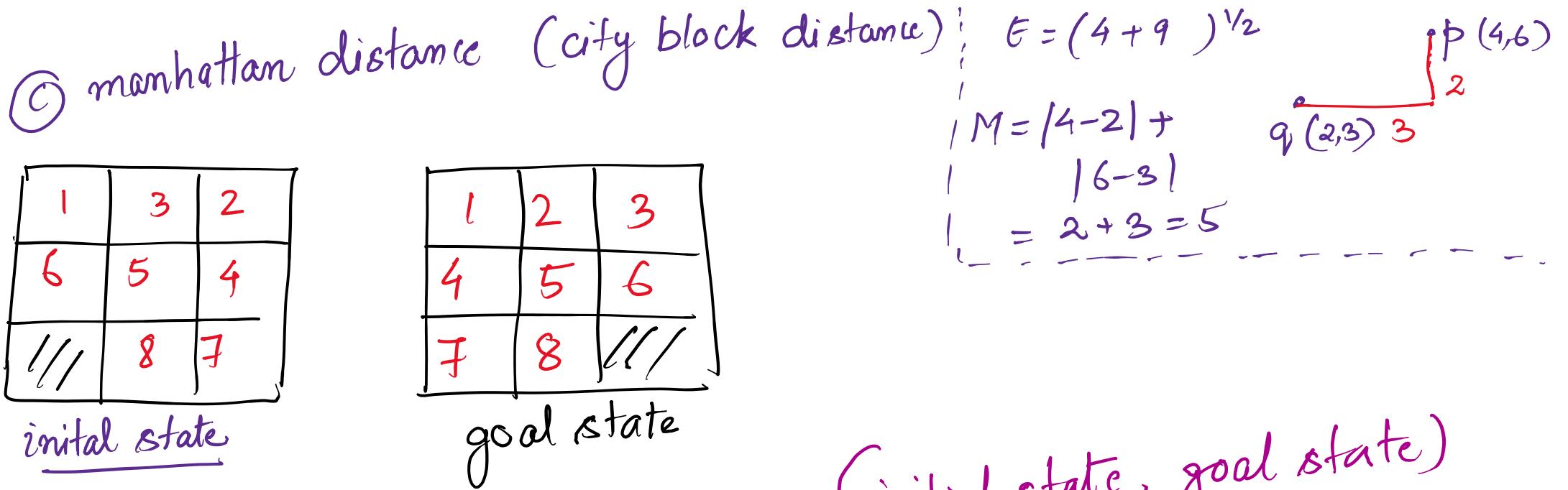
$$\{0 + 1 + 1 + 1 + 0 + 1 + 1 + 0 + 1\} = \underline{\underline{6}}$$

- higher the count, the far we are from the goal state.

b) correctly placed tiles count

$$\{1 + 0 + 0 + 0 + 1 + 0 + 0 + 1 + 0\} = \underline{\underline{3}}$$

- higher the count is, closer we are to the goal state.



1:0	5:0
2:1	6:2
3:1	7:2
4:2	8:0

Manhattan distance (initial state, goal state)

$$\begin{aligned}
 &= 0+1+1+2+0+2+2+0 \\
 &= \underline{\underline{8 \text{ avg}}}
 \end{aligned}$$

Priority Queue (ADT)

- Similar to queue, but every element here has certain priority.
- This priority of elements determines the order in which the element is removed (`poll()`) from the queue.
- $(\text{polled} \rightarrow \text{PQ}) \equiv (\text{popped} \rightarrow Q)$
- PQ supports only those types of element, that are comparable.

→ lower no.
↑↑
higher priority
(vice versa)

Example :- Assume you have certain numbers with you. $\{4, 8, 14, 22, 3\}$ → assume we want some order among the elements. lower the value higher is priority.

Operations

- ① `poll()` - 3
 - ② `add(1)`
 - ③ `poll()` - 1
 - ④ `add(47)`
 - ⑤ `poll()` - 4
 - ⑥ `poll_rest()` - 8, 14, 22, 47.
- $\{4, 8, 14, 22, 1\}$
 $\{4, 8, 14, 22\}$
 $\{4, 8, 14, 22, 47\}$
 $\{8, 14, 22, 47\}$

popping / polling

order :

3, 1, 4, 8, 14, 22, 47

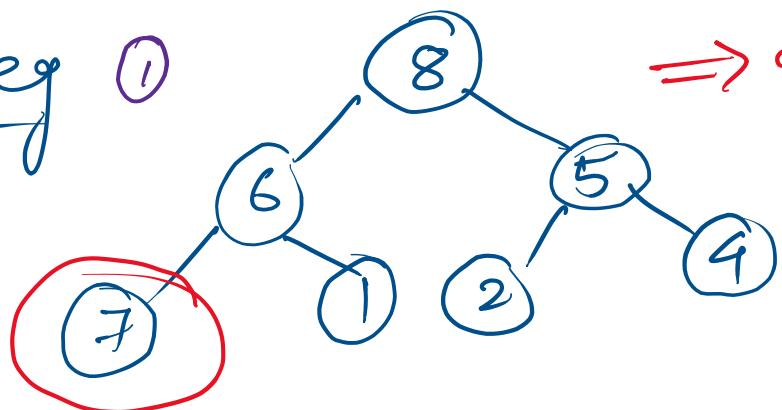
We focus only on the lowest value among the group at every step.

Heap → to implement PQ.

Why? → we only want to retain the knowledge of highest/lowest priority number among all nos.

↳ DS → Heap → it maintains max element at root → max heap
" min " → min heap.

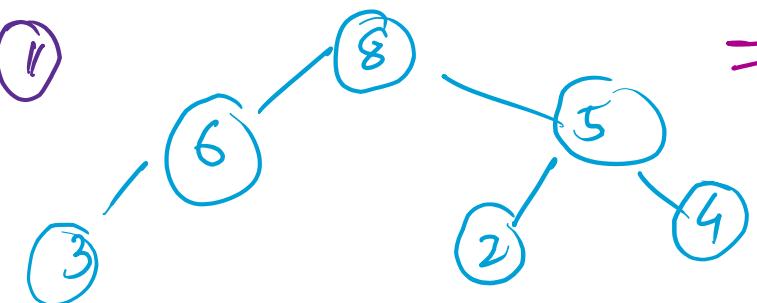
e.g. ①



⇒ max heap } if violates the following
} No. property:

parents > children in max
heap

②



⇒ max heap → No.

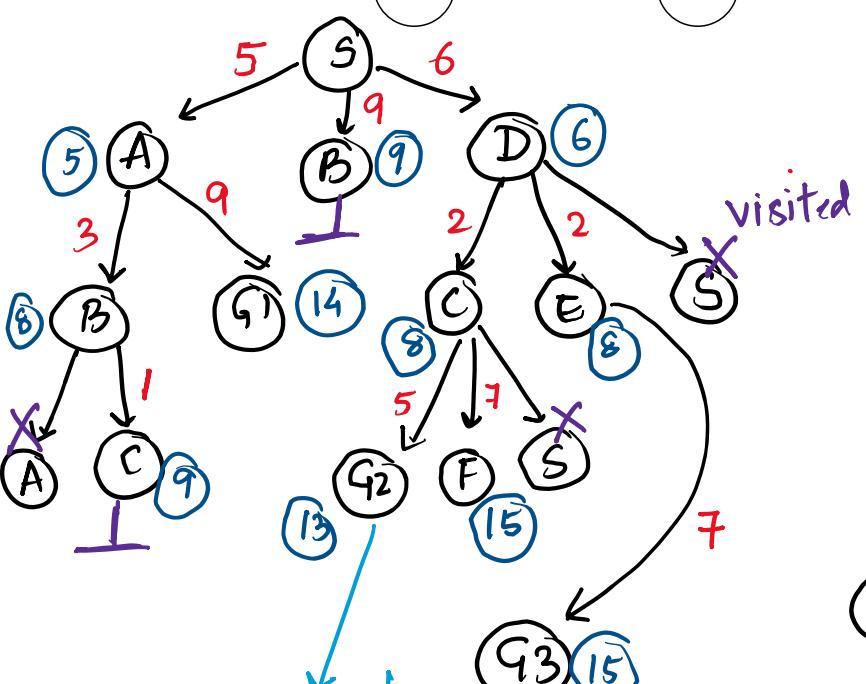
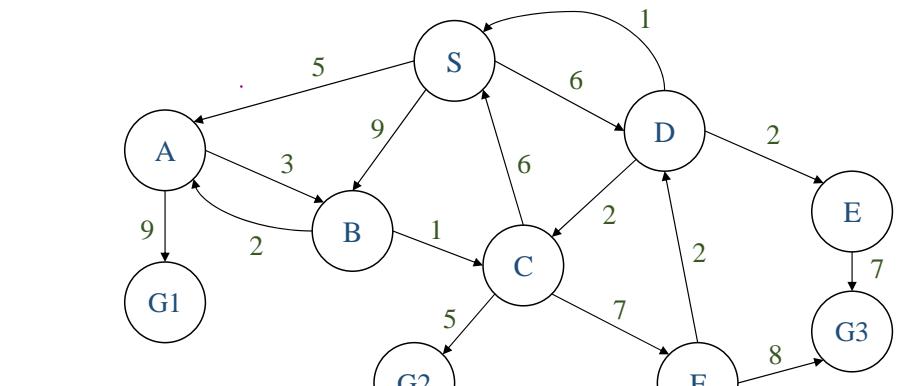
because heaps need to be
complete binary trees.

⇒ not even a valid heap.

Uniform Cost Search (Cheapest First Search)

uniformed search method.

we use priority queue and
the priority is the path value.
(cost)

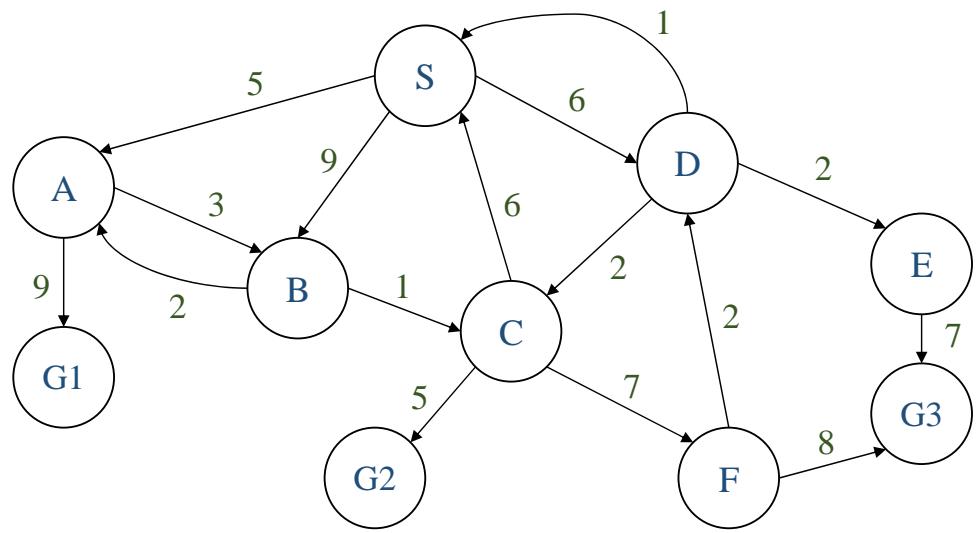


goal node
(stop)

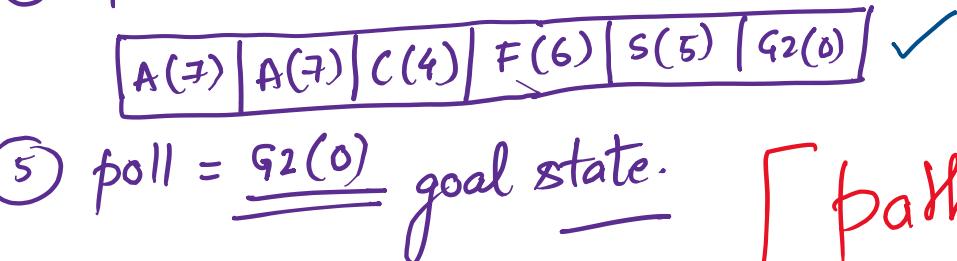
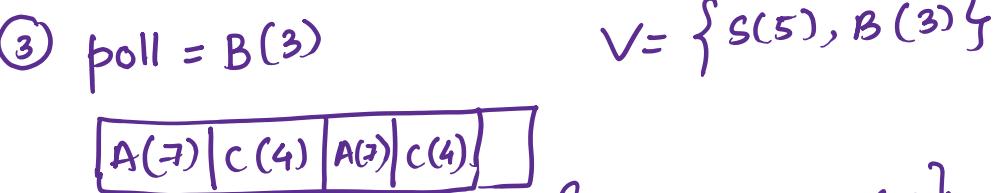
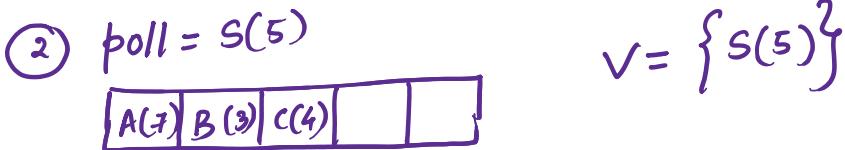
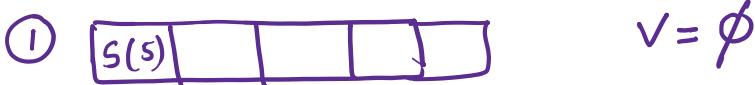
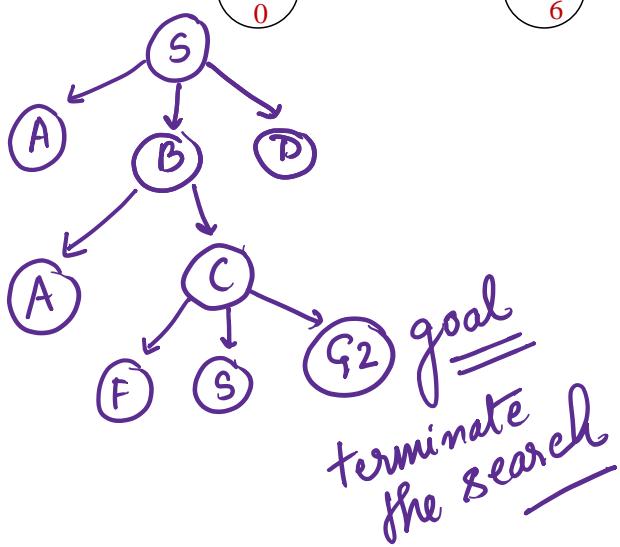
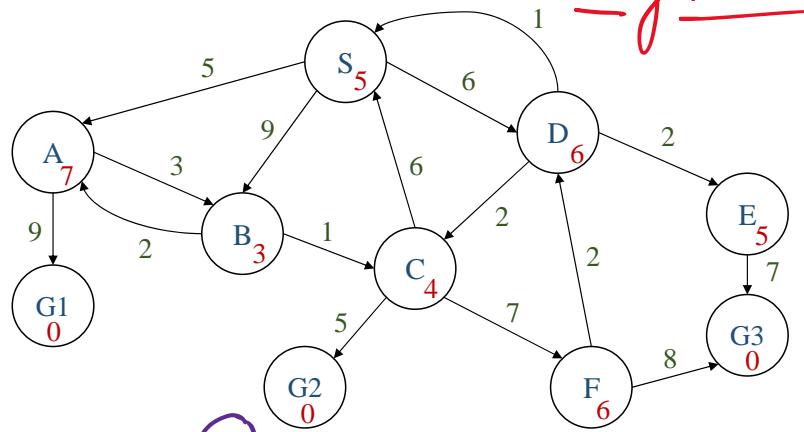
$[S - D - C - G_2]$

cost = 13

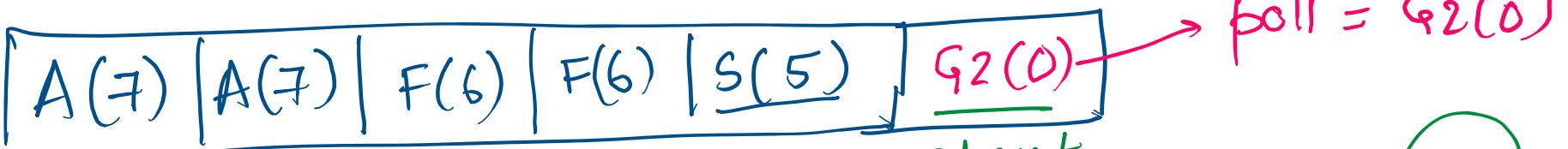
- | | | | |
|------|---|--|--|
| I | S(0) | $V = \{ S(0) \}$ | visited nodes set |
| II | A(5) B(9) D(6) | $V = \{ S(0) \}$ | |
| III | B(9) D(6) B(8) G ¹ (14) | $V = \{ S(0), A(5) \}$ | |
| IV | B(9) B(8) G ¹ (14) C(8) E(8) | $V = \{ S(0), A(5), D(6) \}$ | |
| V | B(9) G ¹ (14) C(8) E(8) G(9) | $V = \{ S(0), A(5), D(6), B(8) \}$ | $\text{poll} = B(8)$ (alphabetically) |
| VI | B(9) G ¹ (14) E(8) C(9) G ² (13) F(5) | $V = \{ S(0), A(5), D(6), B(8) \}$ | $V = \{ S(0), A(5), D(6), B(8), C(8) \}$ |
| VII | B(9) G ¹ (14) C(9) G ² (13) F(15) G ³ (15) | $V = \{ S(0), A(5), D(6), B(8), C(8), E(8) \}$ | |
| VIII | G ¹ (14) G ² (13) F(15) G ³ (15) | | |
| IX | $\text{poll}() - G^2(13)$ | | |



only heuristics Best First Search (Informed)



[path = $S \rightarrow B \rightarrow C \rightarrow G2$]

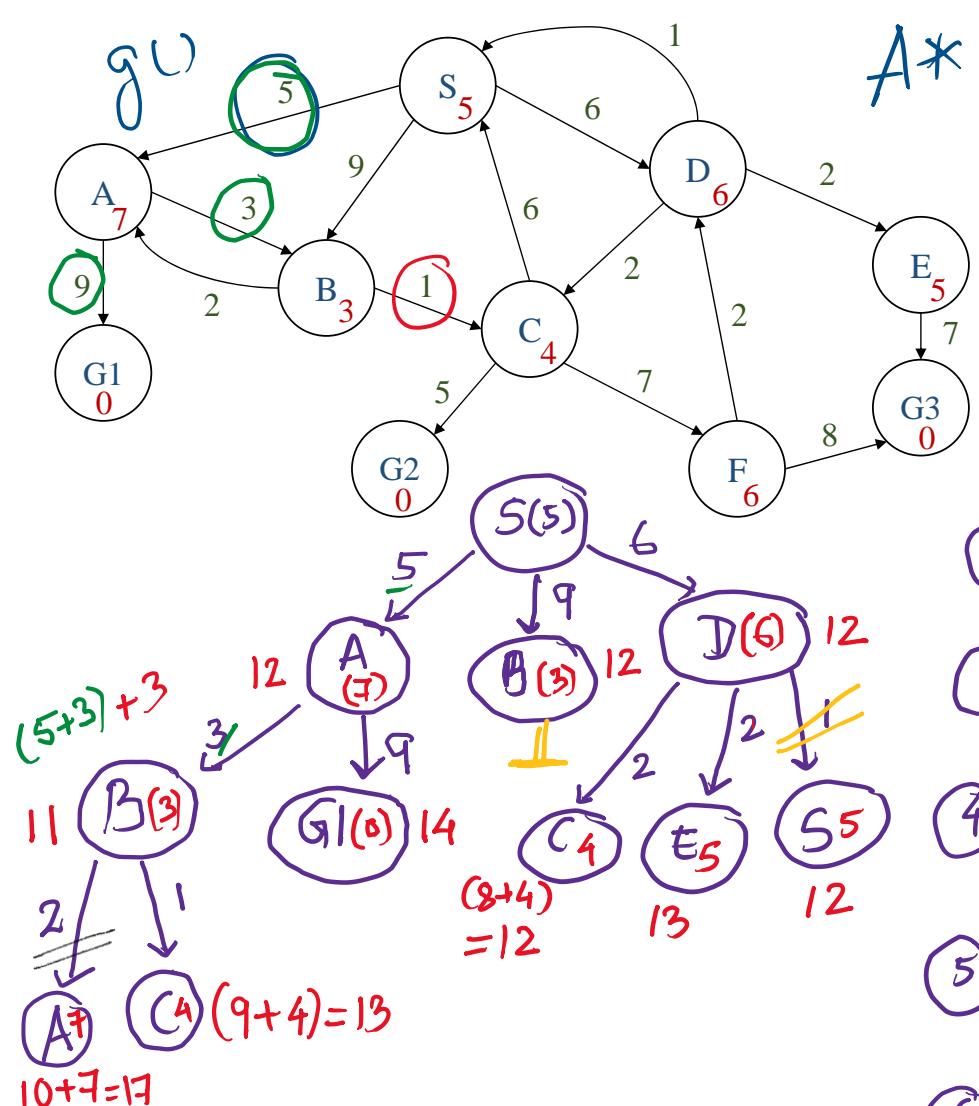


#disadvantage → failure case. heuristics under estimated

Beam Search (widely used in NLP)

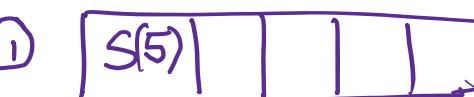
- # Variant of Best First Search
- # B (beam), it denotes the maximum number of nodes (values) the priority queue can/should maintain.
- # Note:- here we need to sort the elements again and again.

A* Search – based on heuristics + path cost

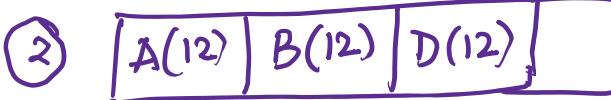


$\frac{g(\cdot)}{\uparrow} + h(\cdot)$ → red
acc. cost from S to that node

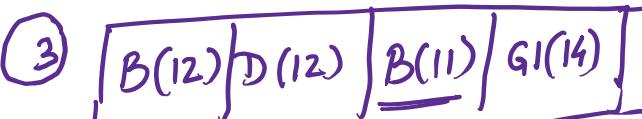
5



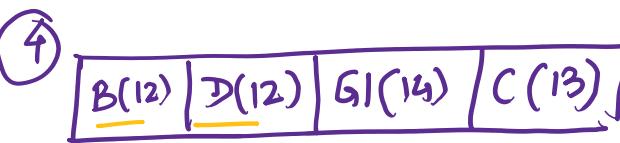
$$V = \emptyset = \{ \}$$



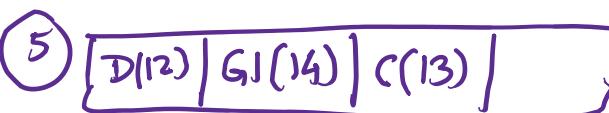
$$V = \{ S(5) \}$$



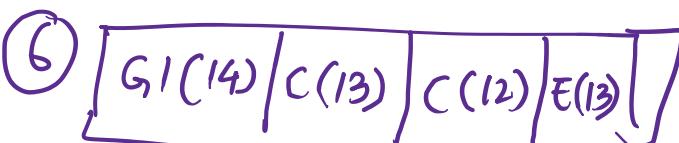
$$V = \{ S(5), A(12) \}$$



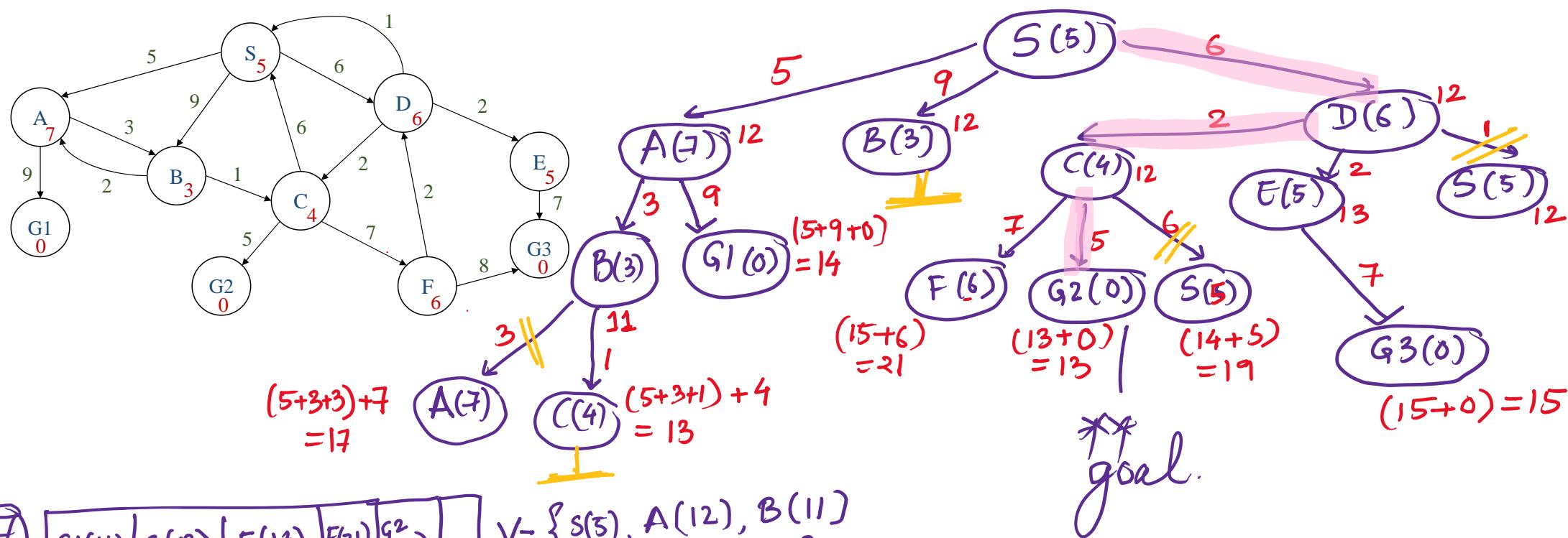
$$V = \{ S(5), \underline{A(12)}, \underline{B(11)} \}$$



$$V = \{ S(5), A(12), B(11) \}$$



$$V = \{ S(5), A(12), B(11), D(12) \}$$



⑦ $\boxed{G1(14) | C(13) | E(13) | F(21) | G2(13) | }$ $V = \{S(5), A(12), B(11)$
 $D(12), C(12)\}$

⑧ $C(13) \times \boxed{G1(14) | E(13) | F(21) | G2(13) | }$

⑨ $\boxed{G1(14) | F(21) | G2(13) | E(13) | G3(15) | }$

$V = \{S(5), A(12), B(11), D(12), C(12), E(13)\}$

⑩ $G2(13) = \text{goal. stop}$ $S \xrightarrow{6} D \xrightarrow{2} C \xrightarrow{5} G2$ path-cost 13

$$TC = O(V+E) = O(b^d) \quad SC = O(b^d)$$

assignment:- Run A* on the modified graph. (heuristics corrected).

$\left\{ \begin{array}{l} S \rightarrow h(S) = 5 \\ g(S - G2) = \underline{13} \quad 5 < 13 \end{array} \right\}$
underestimating cost

① Diyas = {500} overestimation

S1 ✓
(200)

S2
(100)

S3 X
(300)

② Diyas = {150} underestimation

S1 X
(200)

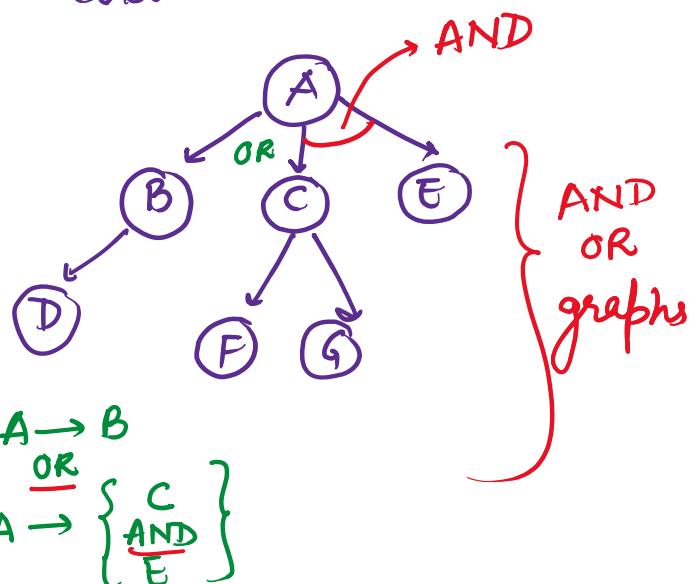
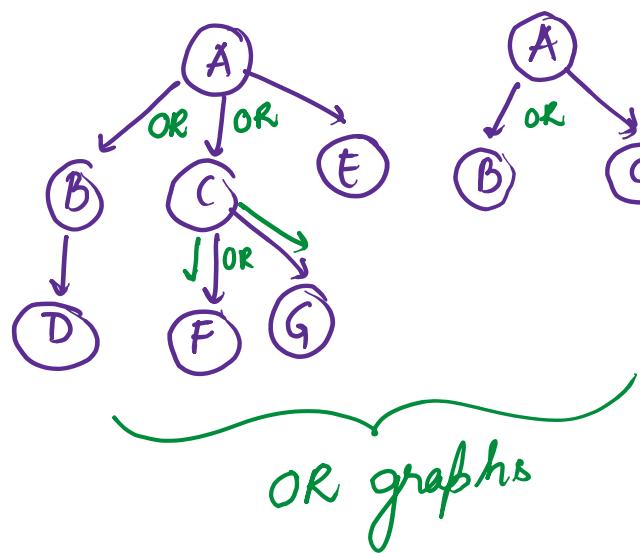
S2 ✓
(100)

S3 ✓
(300)

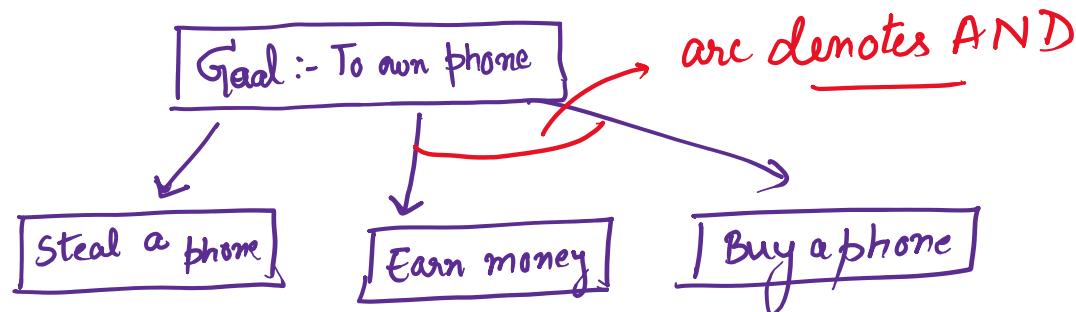
AO* search algorithm

cost $f(n) = g(n) + h(n)$ } \rightarrow heuristic cost
actual/edge cost

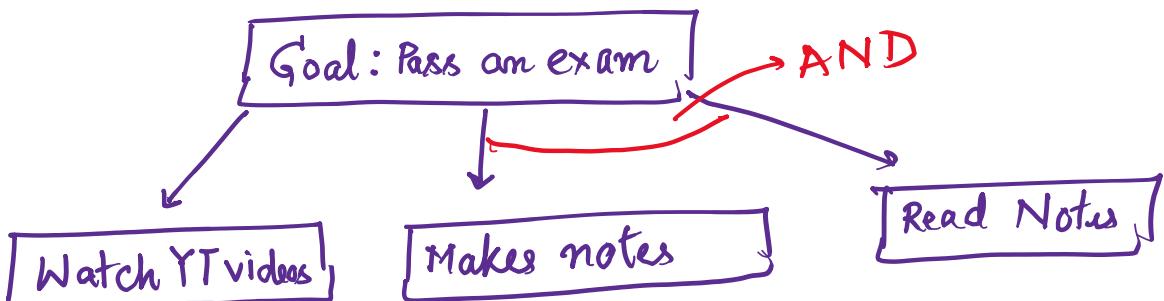
AND-OR graphs



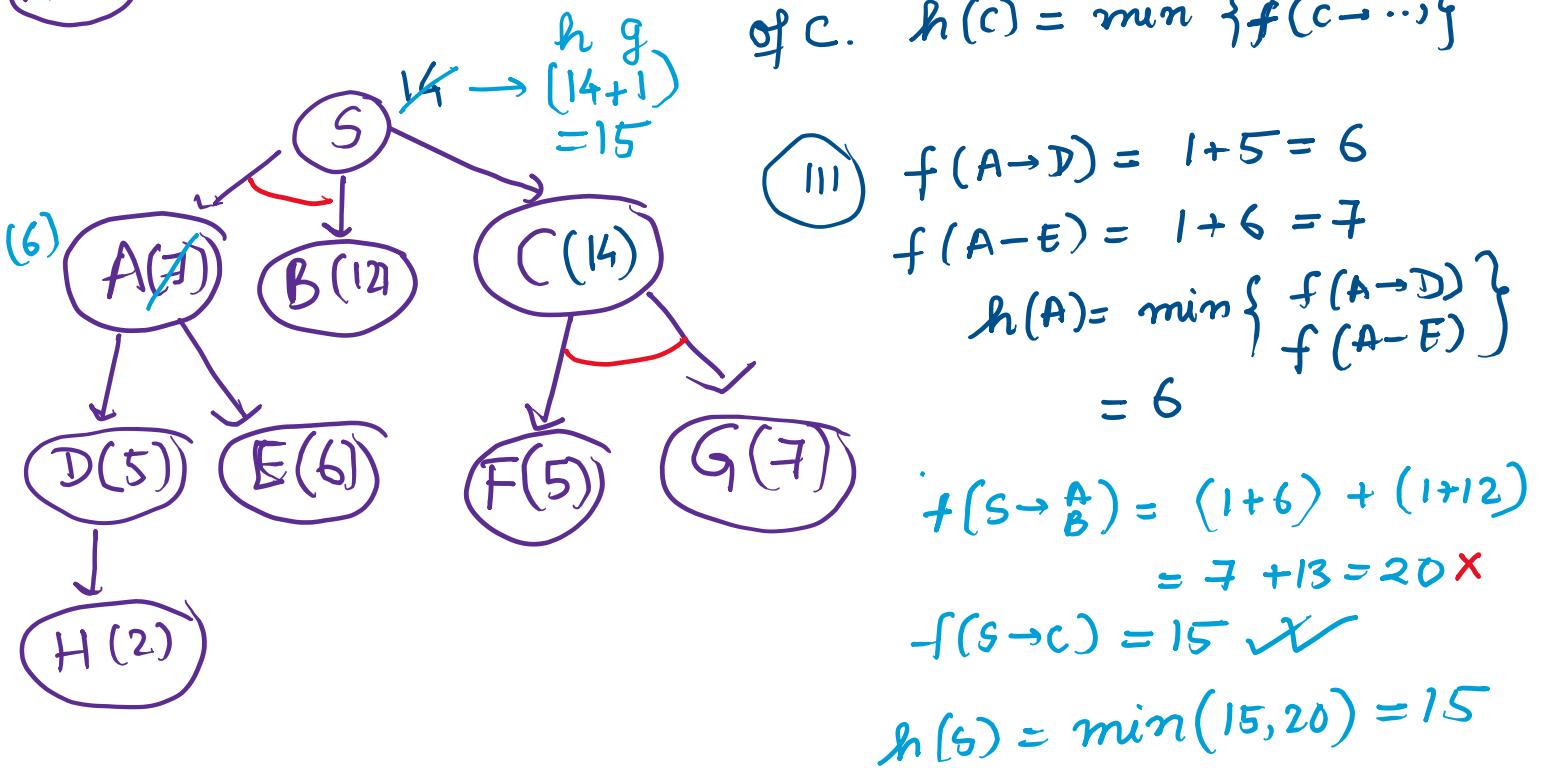
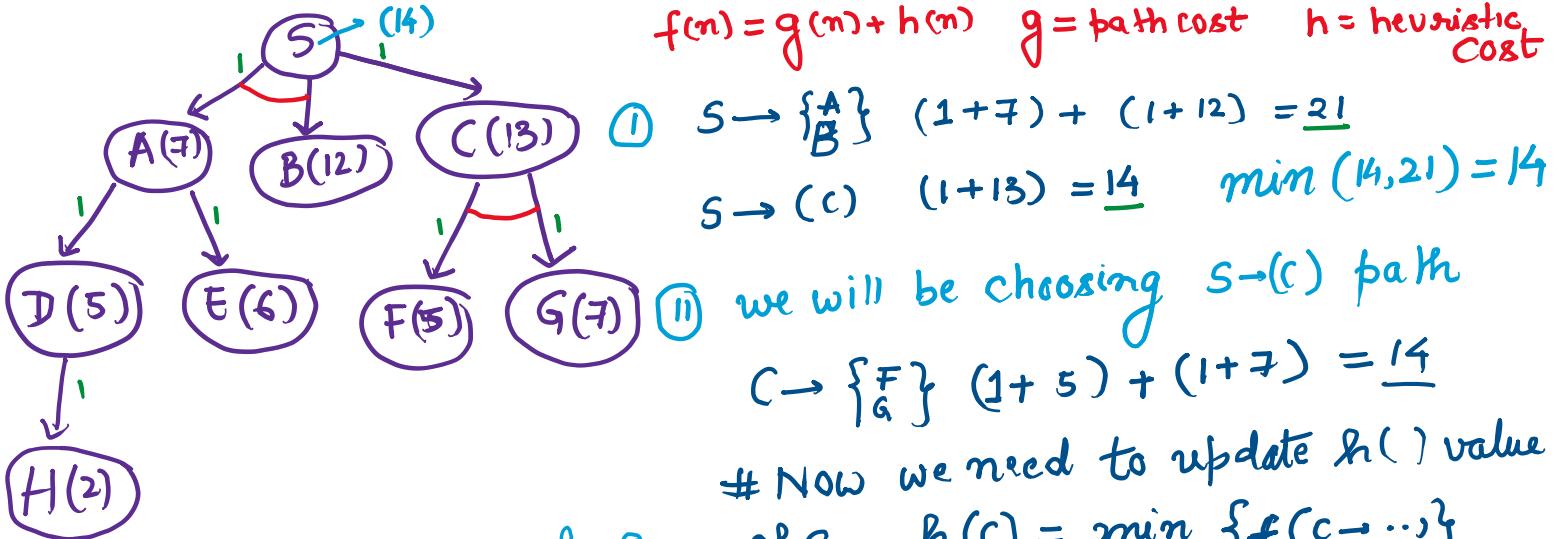
①

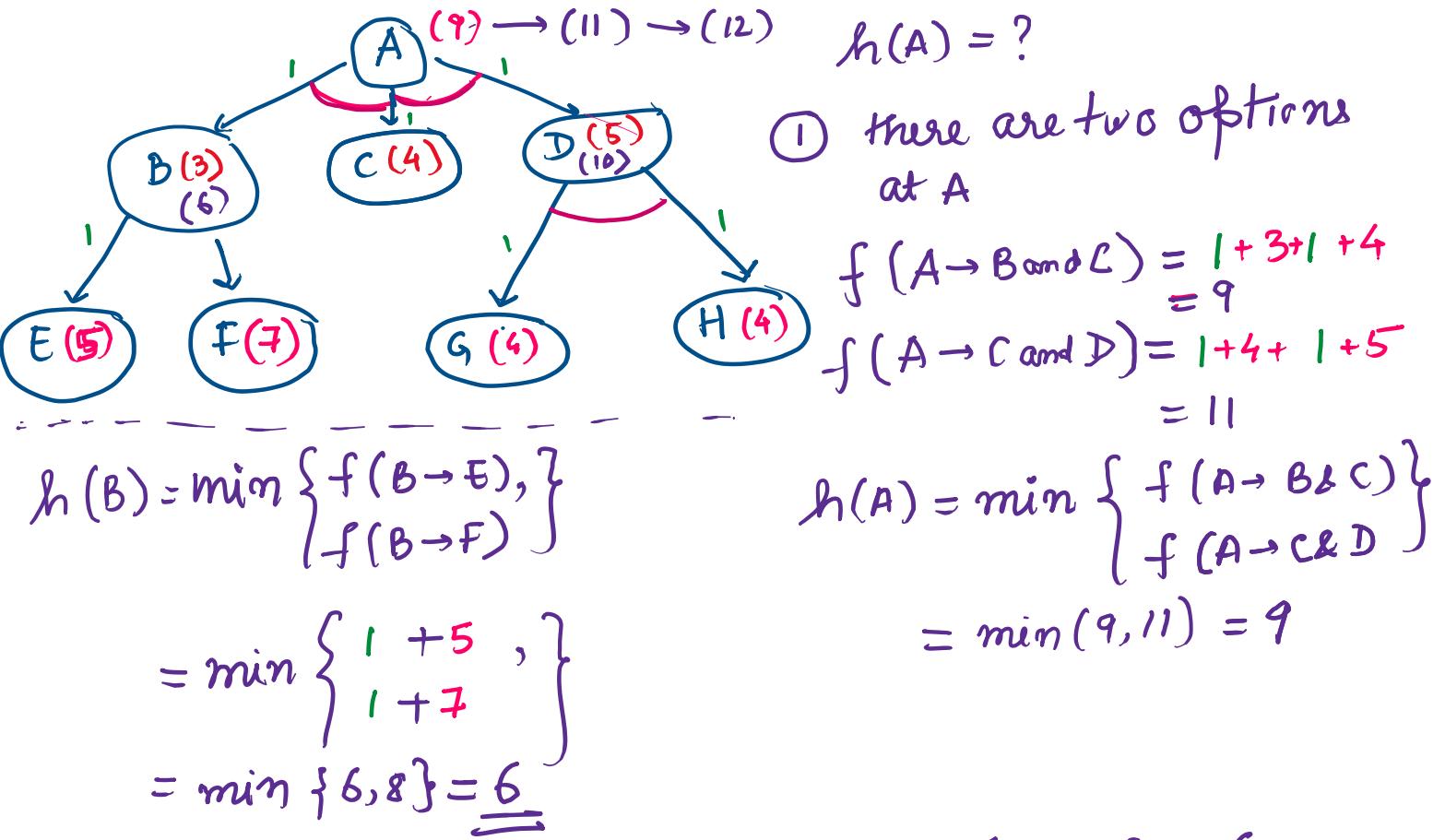


②



AND-OR graphs



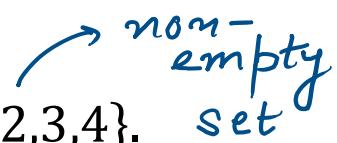


we will update heuristic value of B from 3 → 6
 $h(B) = 6$. and hence we need to re-calculate $h(A)$.

$$h(A) = \min \left\{ \begin{array}{l} f(A \rightarrow B \& C), \\ f(A \rightarrow C \& D) \end{array} \right\} = \min \left(\begin{array}{l} 12, \\ 11 \end{array} \right) = 11$$

now since path $A \rightarrow C \& D$ is more efficient
 we choose that path

Constraint Satisfaction Problems (CSPs)

- A set of variables, $X_1, X_2, X_3, \dots, X_n$.
 - Each variable X_i has a non-empty domain of possible values, e.g. $X_1 = \{1, 2, 3, 4\}$. *[Aim]* 
 - A set of constraints, e.g. $X_1 \neq 3$ (unary) or $X_3 > X_4$ (binary).
 - Solution: assign a value to each variable such that none of the constraints are broken.
 - We will discuss two types of algorithms
 - Simplest algorithm (*not the best*): backtracking
 - AC-3 (Arc Consistency #3) algorithm
- We have a system, where we need to make decisions in parallel.
and these decisions can be expressed/modeled as variables.
◦ for every variable – we have certain choices (some possible values)
◦ We have a set of constraints. These are restrictions that
do not allow certain values to be assigned to variables.

vars

$$A = \{1, 2, 3\}$$

$$B = \{1, 2, 3\}$$

$$C = \{1, 2, 3\}$$

domains

$$\left. \begin{array}{l} C_1 : A > B \\ C_2 : B \neq C \\ C_3 : A \neq C \end{array} \right\} \text{constraints}$$

Back Tracking Algorithm (system)

- ① $A=1$,
check whether any constraint is violated
 $C_1, C_2, C_3 \rightarrow$ none of them

- ② $A=1, B=1$
check any constraint is violated.
 $C_1 \rightarrow$ violated

\therefore backtrack the most recent assignment
($B=1$)

- ④ $A=1, B=3$
 $C_1 \rightarrow$ violated
a) B.T. the most recent assign
($B=3$)

no more values of B are left

- b) B.T. the prev assign
($A=1$)

- ⑤ $A=2$,

- ⑥ $A=2, B=1$

- ⑦ $A=2, B=1, C=1$

C_2 is violated

BT $C=1$

- ⑧ $A=2, B=1, C=2$

C_3 is violated

BT $C=2$

- ③ $A=1, B=2$
 $C_1 \rightarrow$ violated

Backtrack the most recent assignment
($B=2$)

- ⑨ $A=2, B=1, C=3$

$C_1 A > B \checkmark$ solution
 $C_2 B \neq C \checkmark$
 $C_3 A \neq C \checkmark$

AC-3 Algorithm

1. Turn each binary constraint into two arcs e.g. : $A > B$ becomes $A > B$ and $B < A$. $A = B$ becomes $A = B$ and $B = A$.
2. Add all the arcs to an agenda. → set of arcs.
3. Repeat until agenda is empty:
 - i. Take an arc (X_i, X_j) off the agenda and check it
 - ii. For every value of X_i , there must be some value of X_j .
 - iii. Remove any inconsistent values from X_i .
 - iv. If X_i has changed, add all arcs of the form (X_k, X_i) to the agenda

(if any values of X_i is removed)

$$A = \{1, 2, 3\}$$

$$B = \{1, 2, 3\}$$

$$C = \{1, 2, 3\}$$

$$A > B$$

$$B = C$$

Agenda

~~$A > B$~~

~~$B < A$~~

~~$B = C$~~

~~$C = B$~~

~~$A > B$~~

~~$B = C$~~

① $A > B$

a) $A = 1$, no values of B are there. ∴ remove 1 from A .

② $B < A$

a) $B = 3$, no values of A are there. ∴ remove 3 from B

③ $B = C$

④ $C = B$

a) $C = 3$, no values of B are present
∴ remove 3 from C

arcs

$$A > B$$

$$B < A$$

$$B = C$$

$$C = B$$

$$A = \{2, 3\}$$

$$B = \{1, 2\}$$

$$C = \{1, 2\}$$

✓ A > B

✗ B = C

reduced problem.

Now, we can use B.T. (or any standard algo) to find a solution.

====

Crypt arithmetic Problems

- ① Type of CSPs.
- ② Constraints:
 - a) No two characters have same values
 - b) sum of digits must satisfy the given problem.
 - c) range of digits (0-9)

Q1

$$\begin{array}{r}
 C_2 \quad C_1 \\
 \text{TO} \\
 + \quad GO \\
 \hline
 O \quad V \quad T
 \end{array}$$

Letter	Digits
T	2
O	1
G	
U	

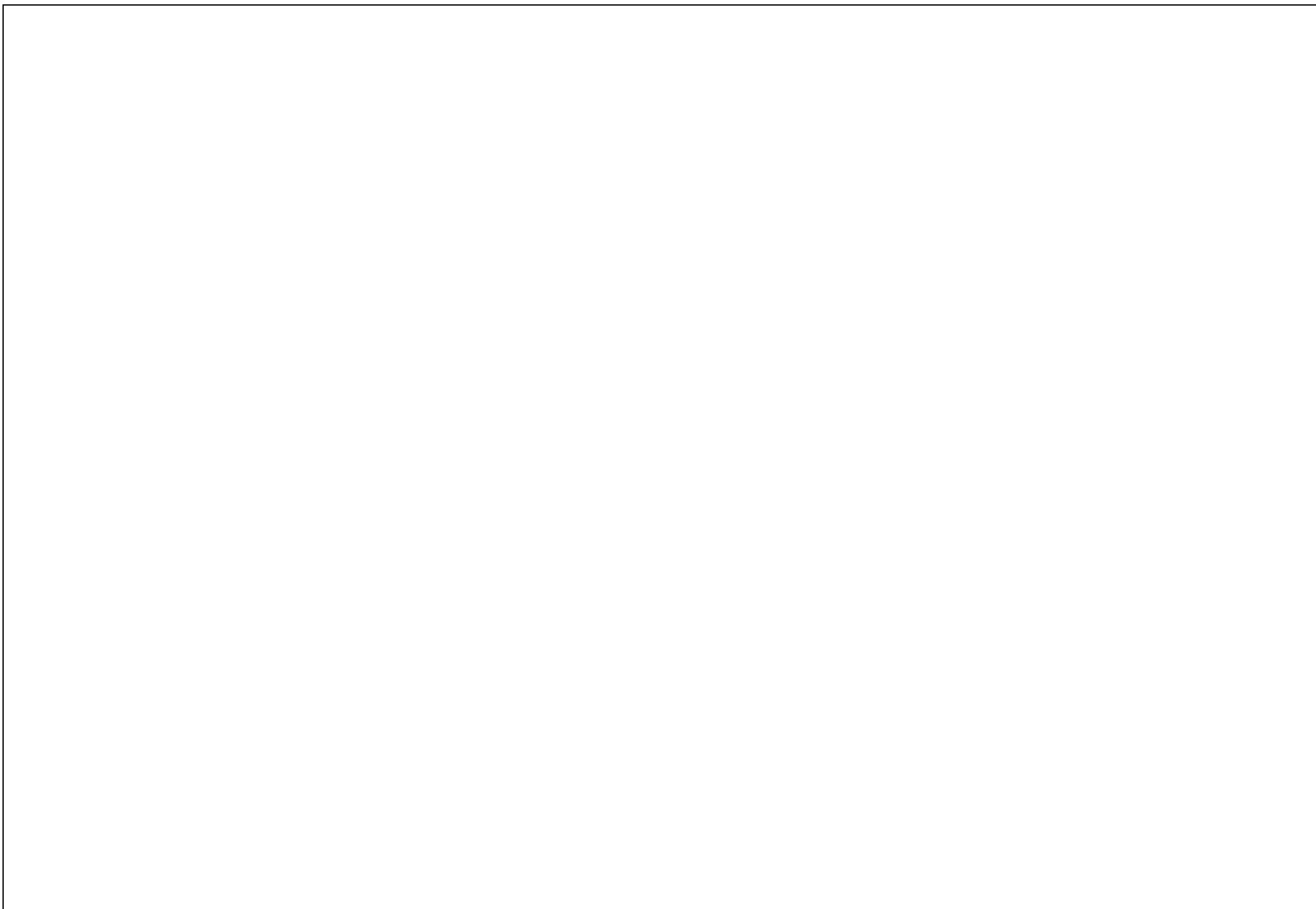
$$\begin{array}{r}
 2 \quad 1 \\
 + \quad 8 \quad 1 \\
 \hline
 1 \quad 0 \quad 2
 \end{array}$$

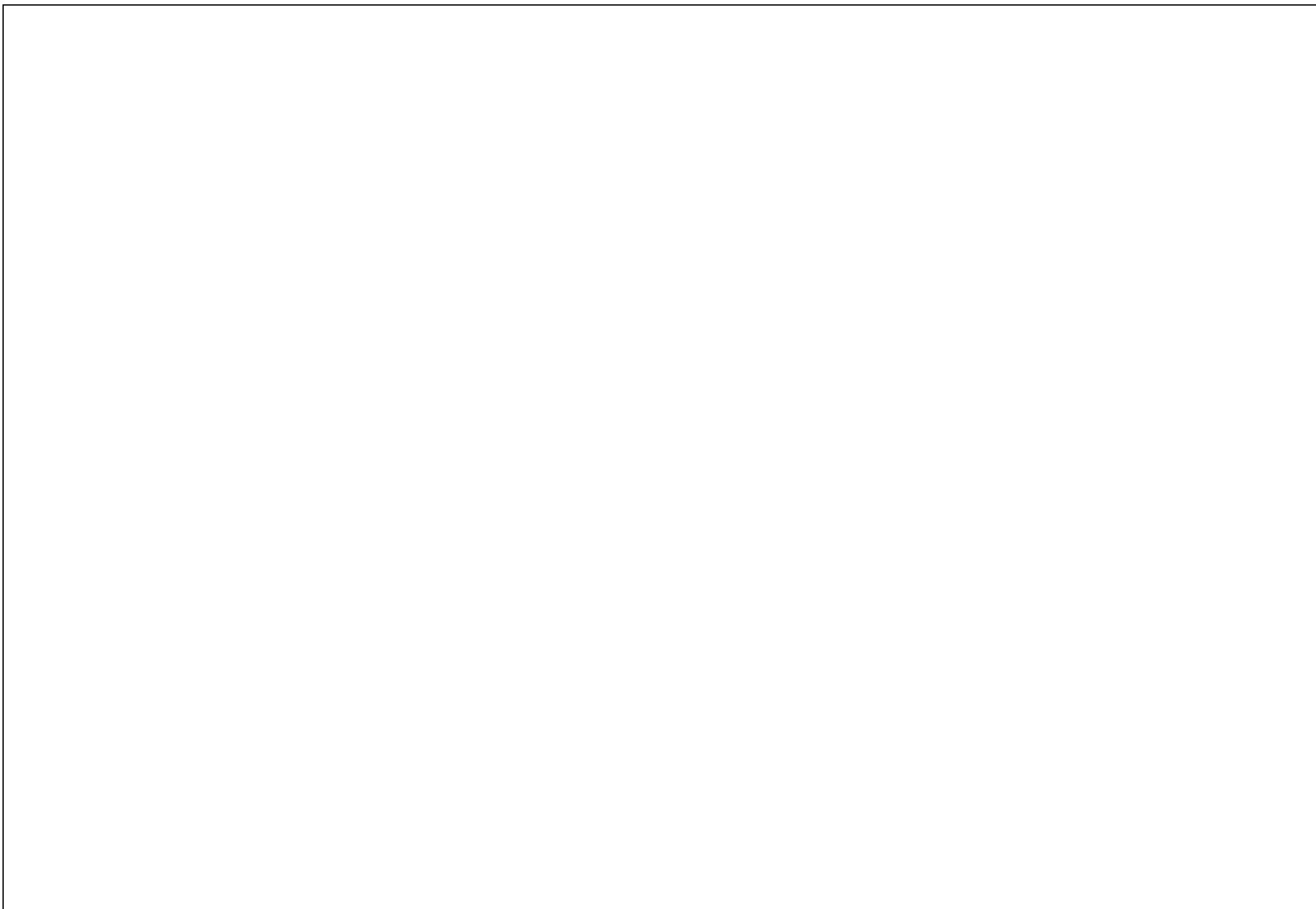
$$9+9=18$$

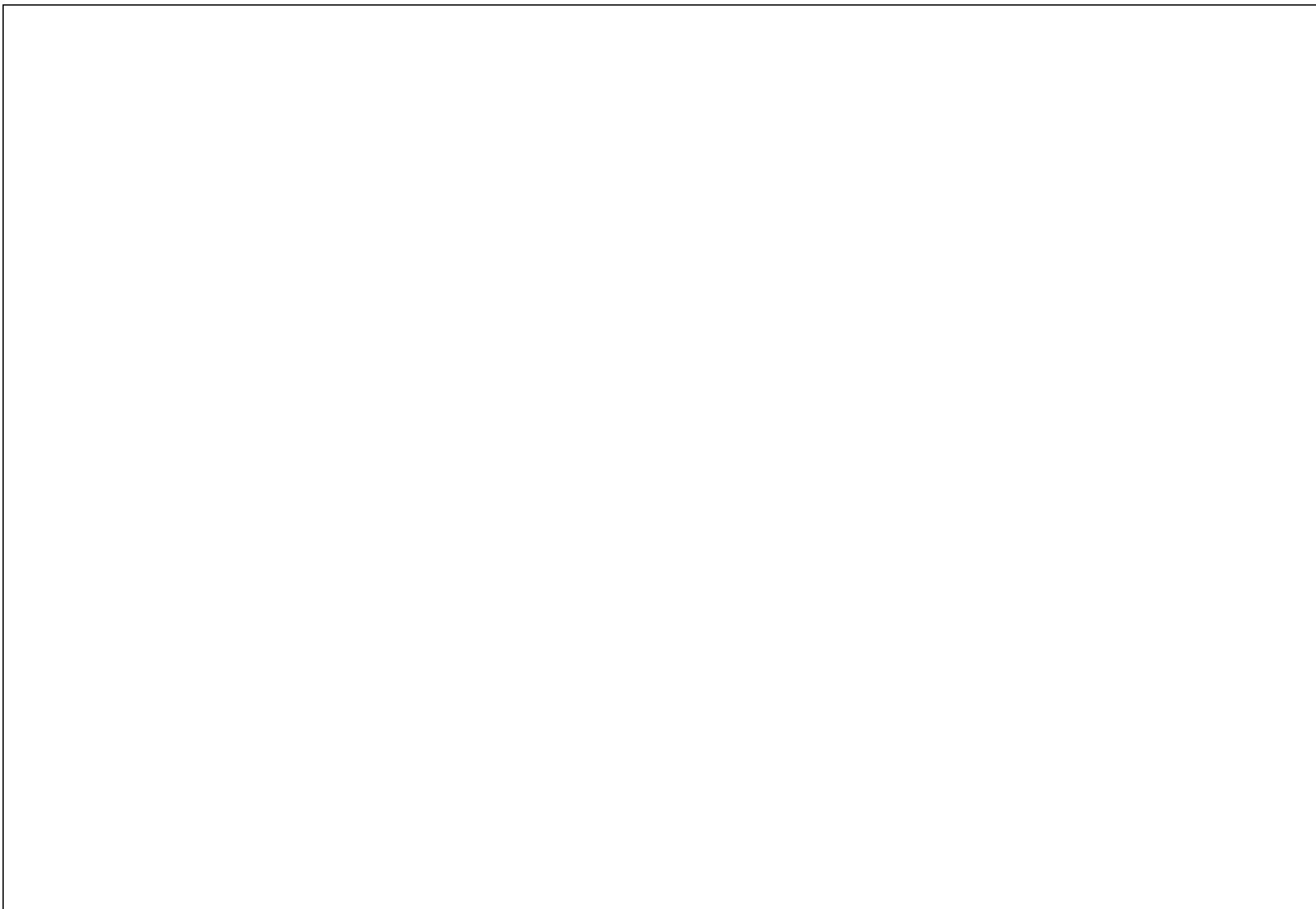
$$2+G = U+10$$

a) $G=9 \ # 2+9 = U+10 \Rightarrow U=1 \text{ but } O=1 \therefore U \neq 1$

b) $G=8 \ # 2+8 = U+10 \Rightarrow \underline{U=0}$







Artificial Intelligence (IT38005)

Constraints Satisfaction Problems

Anup Kumar Gupta

Cryptanalysis or Cryptarithmetic Problems

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

Constraints

- There should be a unique digit to be replaced with a unique alphabet.
- The result should satisfy the predefined arithmetic rules.
- The digits should be from 0-9 only.
- There should be only one carry forward, while performing the addition operation on a problem.

Generic Algorithm: Constraint Satisfaction

1. Propagate available constraints. To do this, first set OPEN to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty:
 - a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.
 - b) If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to OPEN all objects that share any constraints with OB.
 - c) Remove OB from OPEN.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, then return failure.
4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:
 - a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
 - b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

Algorithm: Constraint Satisfaction

- The following problem is given to us, where we need to assign unique values to each unique character, such that they satisfy the predefined arithmetic rules.

$$\begin{array}{r} \text{T} \quad \text{O} \\ + \quad \text{G} \quad \text{O} \\ \hline \text{O} \quad \text{U} \quad \text{T} \end{array}$$

Let us begin solving the problem:

- Assume the carry forwards to be $C1, C2$
- $O = 1$, since two single-digit numbers plus a carry cannot total more than 1.
- $C2 = 1, C1 \in \{0,1\}$.
- Now $T = 1 + 1 = 2$

$$\begin{array}{r} C2 \quad C1 \\ \text{T} \quad \text{O} \\ + \quad \text{G} \quad \text{O} \\ \hline \text{O} \quad \text{U} \quad \text{T} \end{array}$$

$$\begin{array}{r} 1 \quad C1 \\ \text{T} \quad 1 \\ + \quad \text{G} \quad 1 \\ \hline 1 \quad \text{U} \quad \text{T} \end{array}$$

Algorithm: Constraint Satisfaction

- The following problem is given to us, where we need to assign unique values to each unique character, such that they satisfy the predefined arithmetic rules.

$$\begin{array}{r} \text{T} \quad \text{O} \\ + \quad \text{G} \quad \text{O} \\ \hline \text{O} \quad \text{U} \quad \text{T} \end{array}$$

Let us begin solving the problem:

- Assume the carry forwards to be $C1, C2$
- $O = 1$, since two single-digit numbers plus a carry cannot total more than 1.
- $C2 = 1, C1 \in \{0,1\}$.
- Now $T = 1 + 1 = 2$
- Also note that $C1 = 0$.

$$\begin{array}{r} C2 \quad C1 \\ \text{T} \quad \text{O} \\ + \quad \text{G} \quad \text{O} \\ \hline \text{O} \quad \text{U} \quad \text{T} \end{array}$$

$$\begin{array}{r} 1 \quad C1 \\ 2 \quad 1 \\ + \quad \text{G} \quad 1 \\ \hline 1 \quad \text{U} \quad 2 \end{array}$$

Algorithm: Constraint Satisfaction

- The following problem is given to us, where we need to assign unique values to each unique character, such that they satisfy the predefined arithmetic rules.

$$\begin{array}{r} \text{T} \quad \text{O} \\ + \quad \text{G} \quad \text{O} \\ \hline \text{O} \quad \text{U} \quad \text{T} \end{array}$$

Let us begin solving the problem:

- Assume the carry forwards to be $C1, C2$
- $O = 1$, since two single-digit numbers plus a carry cannot total more than 1.
- $C2 = 1, C1 \in \{0,1\}$.
- Now $T = 1 + 1 = 2$
- Also note that $C1 = 0$.
- Now propagate the constraints, using the additional constraints:
 - $2 + G = U + 10$ (Since, there is one carry $C2 = 1$)
 - or $G - U = 8$

$$\begin{array}{r} \text{C2} \quad \text{C1} \\ \text{T} \quad \text{O} \\ + \quad \text{G} \quad \text{O} \\ \hline \text{O} \quad \text{U} \quad \text{T} \end{array}$$

$$\begin{array}{r} 1 \quad 0 \\ \quad \quad \quad 2 \quad 1 \\ + \quad \quad \quad \text{G} \quad 1 \\ \hline 1 \quad \text{U} \quad 2 \end{array}$$

Algorithm: Constraint Satisfaction

- While propagating the constraints, we encountered the following additional constraints: $G - U = 8$
- The feasible values can be:
 - $G = 9, U = 1$
 - $G = 8, U = 0$
- Now if we select $G = 9, U = 1$, then the constraint that every value should be unique is broken.
- Selecting $G = 8, U = 0$ satisfies every constraint, hence it is the solution.

$$\begin{array}{r} & T & O \\ & + & G & O \\ \hline O & U & T \\ \hline \end{array}$$
$$\begin{array}{r} 1 & 0 \\ & 2 & 1 \\ & + & G & 1 \\ \hline 1 & U & 2 \\ \hline \end{array}$$

$$\begin{array}{r} 2 & 1 \\ + & 8 & 1 \\ \hline 1 & 0 & 2 \\ \hline \end{array}$$

Algorithm: Constraint Satisfaction

- $M = 1$
- Since $C_3 + S + M > 9$ (in order to generate carry).
 - $C_3 \in \{0,1\}$ and $M = 1$.
 - Thus, $S \in \{8,9\}$.
- Since $C_3 + S + M > 9$ (in order to generate carry),
 - $S \in \{8,9\}$, $M=1$, $C_3 \in \{0,1\}$
 - $C_3 + S + M \leq 11$
 - $O \in \{0,1\}$
 - $O \neq 1$, since $M = 1$.
 - $O = 0$.
- Consider the 3rd column ($C_2 + E + O = N$) i.e ($C_2 + E = N$)
 - Since $C_2 \in \{0,1\}$, $E = N$ or $E = N+1$
 - Since $E \neq N$, Hence $E = N + 1$ and $C_2 = 1$.

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 C_3 & C_2 & C_1 \\
 S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

Algorithm: Constraint Satisfaction

- $M = 1$
- $S \in \{8,9\}$
- $C_3 \in \{0,1\}$
- $O = 0$
- $E = N + 1$ and $C_2 = 1$.
- In order for $C_2 = 1$, i.e. to have a carry, $C_1 + N + R > 9$.
 - Since $C_1 \in \{0,1\}$, hence $N + R \in \{8, 9, 10, \dots, 17\}$
- The highest value of $C_1 + N + R$ is 18, hence $E \neq 9$.

$$\begin{array}{r} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

$$\begin{array}{r} C_3 & C_2 & C_1 \\ S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Algorithm: Constraint Satisfaction

The derived additional constraints are:

- $M = 1$,
- $S \in \{8,9\}$
- $C3 \in \{0,1\}$, $C1 \in \{0,1\}$ and $C2 = 1$.
- $O = 0$ and $E = N + 1$
- $N + R \in \{8, 9, 10, \dots, 17\}$ and $E \neq 9$.

$$\begin{array}{r} & S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

$$\begin{array}{r} C3 & C2 & C1 \\ S & E & N & D \\ + & M & O & R & E \\ \hline M & O & N & E & Y \end{array}$$

Since we cannot go forward, we assume some values and move forward. If we find any violations, we will backtrack and make a new assignment.

Algorithm: Constraint Satisfaction

The derived additional constraints are:

- $S \in \{8,9\}$, $C3 \in \{0,1\}$, $C1 \in \{0,1\}$, $O = 0$, $E = N + 1$
- $N + R \in \{8, 9, 10, \dots 17\}$, $E \neq 9$.

Since we cannot go forward, we assume some values and move forward. If we find any violations, we will backtrack and make a new assignment.

Assume $E = 2$, (E is most recurring character, and we start from lowest value)

- $N = 3$

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 C3 & 1 & C1 \\
 S & E & N & D \\
 + & 1 & 0 & R & E \\
 \hline
 1 & 0 & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 C3 & 1 & C1 \\
 S & 2 & N & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & N & 2 & Y
 \end{array}$$

Algorithm: Constraint Satisfaction

The derived additional constraints are:

- $S \in \{8,9\}$, $C3 \in \{0,1\}$, $C1 \in \{0,1\}$, $O = 0$, $E = N + 1$
- $N + R \in \{8, 9, 10, \dots 17\}$, $E \neq 9$.

Since we cannot go forward, we assume some values and move forward. If we find any violations, we will backtrack and make a new assignment.

Assume $E = 2$, (E is most recurring character, and we start from lowest value)

- $N = 3$

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 C3 & 1 & C1 \\
 S & E & N & D \\
 + & 1 & 0 & R & E \\
 \hline
 1 & 0 & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 C3 & 1 & C1 \\
 S & 2 & N & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & N & 2 & Y
 \end{array}$$

$$\begin{array}{r}
 C3 & 1 & C1 \\
 S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

Algorithm: Constraint Satisfaction

The derived additional constraints are:

- $S \in \{8,9\}$, $C3 \in \{0,1\}$, $C1 \in \{0,1\}$, $O = 0$, $E = N + 1$
- $N + R \in \{8, 9, 10, \dots 17\}$, $E \neq 9$.

Since we cannot go forward, we assume some values and move forward. If we find any violations, we will backtrack and make a new assignment.

Assume $E = 2$, (E is most recurring character, and we start from lowest value)

- $N = 3$
- $C1 + 3 + R = 2$ or $C1 + 3 + R = 12$, and since all digits are non-negative.
- $C1 + 3 + R = 12$ i.e., $C1 + R = 9$ and hence $R \in \{8, 9\}$.

Again, we cannot move forward.

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & C1 \\
 & S & E & N & D \\
 + & 1 & 0 & R & E \\
 \hline
 1 & 0 & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & C1 \\
 & S & 2 & N & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & N & 2 & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & C1 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

Algorithm: Constraint Satisfaction

The derived additional constraints are:

- $M = 1, S \in \{8,9\}, C3 \in \{0,1\}, C1 \in \{0,1\}, O = 0, E = N + 1$
- $N + R \in \{8, 9, 10, \dots 17\}, E \neq 9.$
- $E = 2, N = 3$ and $R \in \{8, 9\}.$

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & C1 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

Again, we cannot move forward. We will assume value for another character.

- $C1 = 1$

$D + 2 = Y + 10$ or $D = Y + 8$ or $D = 8$ or $9.$

$$\begin{array}{r}
 & C3 & 1 & 1 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

But both results in conflict,

- if $D = 8, Y = 0$ [Conflict, $O = 0]$
- if $D = 9, Y = 1$ [Conflict, $M = 1]$. Therefore, we backtrack $C1 = 1$

$$\begin{array}{r}
 & C3 & 1 & 1 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

Algorithm: Constraint Satisfaction

The derived additional constraints are:

- $M = 1, S \in \{8,9\}, C3 \in \{0,1\}, C1 \in \{0,1\}, O = 0, E = N + 1$
- $N + R \in \{8, 9, 10, \dots 17\}, E \neq 9.$
- $E = 2, N = 3$ and $R \in \{8, 9\}.$

We backtrack $C1 = 1$, and assign

- $C1 = 0$
- $D + 2 = Y$ and $N + R = E + 10.$
- $R = 9$

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 C3 & 1 & C1 \\
 S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

$$\begin{array}{r}
 C3 & 1 & 0 \\
 S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

$$\begin{array}{r}
 C3 & 1 & 0 \\
 S & 2 & 3 & D \\
 + & 1 & 0 & 9 & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

Algorithm: Constraint Satisfaction

The derived additional constraints are:

- $M = 1, S \in \{8,9\}, C3 \in \{0,1\}, C1 \in \{0,1\}, O = 0, E = N + 1$
- $N + R \in \{8, 9, 10, \dots 17\}, E \neq 9.$
- $E = 2, N = 3$ and $R \in \{8, 9\}.$

We backtrack $C1 = 1$, and assign

- $C1 = 0$
- $D + 2 = Y$ and $N + R = E + 10.$
- $R = 9$
- $S \in \{8,9\}$ and $R = 9$, hence $S = 8$

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & C1 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & 0 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & 0 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & 9 & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

Algorithm: Constraint Satisfaction

The derived additional constraints are:

- $M = 1, S = 8, C3 \in \{0,1\}, C1 \in \{0,1\}, O = 0, E = N + 1$
- $N + R \in \{8, 9, 10, \dots 17\}, E \neq 9.$
- $E = 2, N = 3$ and $R \in \{8, 9\}.$

We backtrack $C1 = 1$, and assign

- $C1 = 0$
- $D + 2 = Y$ and $N + R = E + 10.$
- $R = 9$
- $S \in \{8,9\}$ and $R = 9$, hence $S = 8$ and $C3 = 0$

$$\begin{array}{r}
 & S & E & N & D \\
 + & M & O & R & E \\
 \hline
 M & O & N & E & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & C1 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

$$\begin{array}{r}
 & C3 & 1 & 0 \\
 & S & 2 & 3 & D \\
 + & 1 & 0 & R & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

$$\begin{array}{r}
 & 0 & 1 & 0 \\
 & 8 & 2 & 3 & D \\
 + & 1 & 0 & 9 & 2 \\
 \hline
 1 & 0 & 3 & 2 & Y
 \end{array}$$

Contact Details

- Anup Kumar Gupta
- PhD scholar, IIT Indore
- Email: msrphd2105101002@iiti.ac.in
- Website: <https://anupkumargupta.github.io/>



ICS141: Discrete Mathematics for Computer Science I

Dept. Information & Computer Sci., University of Hawaii

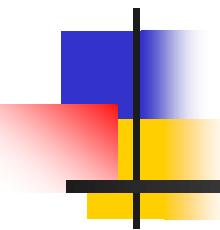
Originals slides by Dr. Baek and Dr. Still, adapted by J. Stelovsky

Based on slides Dr. M. P. Frank and Dr. J.L. Gross

Provided by McGraw-Hill



Lecture 1



Course Overview

Chapter 1. The Foundations

1.1 Propositional Logic



1.1 Propositional Logic

- Logic
 - Study of reasoning.
 - Specifically concerned with whether reasoning is correct.
 - Focuses on the relationship among statements, not on the content of any particular statement.
 - Gives precise meaning to mathematical statements.
- ***Propositional Logic*** is the logic that deals with statements (propositions) and compound statements built from simpler statements using so-called *Boolean connectives*.
- Some applications in computer science:
 - Design of digital electronic circuits.
 - Expressing conditions in programs.
 - Queries to databases & search engines.



Definition of a *Proposition*

Definition: A *proposition* (denoted p, q, r, \dots) is simply:

- a *statement* (i.e., a declarative sentence)
 - *with some definite meaning*,
(not vague or ambiguous)
- having a *truth value* that's either *true* (**T**) or *false* (**F**)
 - it is **never** both, neither, or somewhere "in between!"
 - However, you might not *know* the actual truth value,
 - and, the truth value might *depend* on the situation or context.
- Later, we will study *probability theory*, in which we assign *degrees of certainty* ("between" **T** and **F**) to propositions.
 - But for now: think True/False only! (or in terms of **1** and **0**)



Examples of Propositions

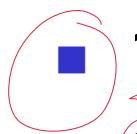
- It is raining. (In a given situation)
- Beijing is the capital of China. (T)
- $2 + 2 = 5$. (F)
- $1 + 2 = 3$. (T)
- A fact-based declaration is a proposition, even if no one knows whether it is true
 - 11213 is prime.
 - There exists an odd perfect number.



Examples of Non-Propositions

The following are **NOT** propositions:

- Who's there? (interrogative, question)
- Just do it! (imperative, command)
- La la la la la. (meaningless interjection)
- Yeah, I sorta dunno, whatever... (vague)
- $1 + 2$ (expression with a non-true/false value)
- $x + 2 = 5$ (declaration about semantic tokens of non-constant value)



$$x + 2 = 5, \text{ given } x = 4$$

P False



Truth Tables

- An *operator* or *connective* combines one or more *operand* expressions into a larger expression. (e.g., “+” in numeric expressions.)
- **Unary** operators take one operand (e.g., -3); **Binary** operators take two operands (e.g. 3×4).
- **Propositional** or **Boolean operators** operate on propositions (or their truth values) instead of on numbers.
- The **Boolean domain** is the set $\{T, F\}$. Either of its elements is called a **Boolean value**.
An n -tuple (p_1, \dots, p_n) of Boolean values is called a **Boolean n -tuple**.
- An n -operand truth table is a table that assigns a Boolean value to the set of all Boolean n -tuples.



Some Popular Boolean Operators

<u>Formal Name</u>	<u>Nickname</u>	<u>Arity</u>	<u>Symbol</u>
Negation operator	NOT	Unary	\neg
Conjunction operator	AND	Binary	\wedge
Disjunction operator	OR	Binary	\vee
Exclusive-OR operator	XOR	Binary	\oplus
Implication operator	IMPLIES	Binary	\rightarrow
Biconditional operator	IFF	Binary	\leftrightarrow



The Negation Operator

- The unary negation operator “ \neg ” (NOT) transforms a proposition into its logical *negation*.
- E.g. If p = “I have brown hair.”
then $\neg p$ = “It is not the case that I have brown hair” or “I do **not** have brown hair.”
- The *truth table* for NOT:

p	$\neg p$
T	F
F	T

Operand column **Result column**



The Conjunction Operator

- The binary **conjunction** operator “ \wedge ” (AND) combines two propositions to form their logical *conjunction*.
- *E.g.* If p = “I will have salad for lunch.” and q = “I will have steak for dinner.”
then, $p \wedge q$ = “I will have salad for lunch and I will have steak for dinner.”



Conjunction Truth Table

Operand columns

AND

p	q	$p \wedge q$
T	T	T
T	F	F
F	T	F
F	F	F

- Note that a conjunction $p_1 \wedge p_2 \wedge \dots \wedge p_n$ of n propositions will have 2^n rows in its truth table

but only 1 row will be true (T)



The Disjunction Operator

- The binary ***disjunction operator*** “ \vee ” (**OR**) combines two propositions to form their logical *disjunction*.
- *E.g.* If p = “My car has a bad engine.” and q = “My car has a bad carburetor.”
then, $p \vee q$ = “My car has a bad engine, or my car has a bad carburetor.”

Meaning is like “and/or” in informal English.



Disjunction Truth Table

p	q	$p \vee q$
T	T	T
T	F	T
F	T	T
F	F	F

Note difference
from AND

- Note that $p \vee q$ means that p is true, or q is true, or **both** are true!
- So, this operation is also called ***inclusive or***, because it **includes** the possibility that both p and q are true.



The Exclusive-Or Operator

- The binary exclusive-or operator “ \oplus ” (XOR) combines two propositions to form their logical “exclusive or”
- *E.g. If p = “I will earn an A in this course.” and q = “I will drop this course.”, then*
 $p \oplus q$ = “I will **either** earn an A in this course, **or** I will drop it (**but not both!**)”



Exclusive-Or Truth Table

p	q	$p \oplus q$
T	T	F
T	F	T
F	T	T
F	F	F

Note difference from OR.

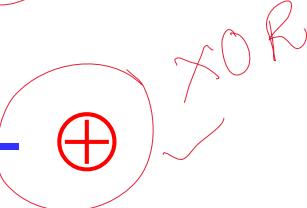
- Note that $p \oplus q$ means that p is true, or q is true, but **not both!**
- This operation is called **exclusive or**, because it **excludes** the possibility that both p and q are true.



Natural Language is Ambiguous

- Note that the English “or” can be ambiguous regarding the “both” case!

“Pat is a singer or
Pat is a writer.” - 

“Pat is a man or
Pat is a woman.” - 

- Need context to disambiguate the meaning!
- For this class, assume “or” means inclusive (\vee).

p	q	$p \text{ "or" } q$
T	T	?
T	F	T
F	T	T
F	F	F



The Implication Operator

if then operator

→ also known as

- The conditional statement (aka implication)
 $p \rightarrow q$ states that p implies q .
- I.e., If p is true, then q is true; but if p is not true, then q could be either true or false.
- E.g., let p = “You study hard.”
 q = “You will get a good grade.”
 $p \rightarrow q$ = “If you study hard, then you will get a good grade.” (else, it could go either way)
- p : hypothesis or antecedent or premise
- q : conclusion or consequence

Implication Truth Table

p	q	$p \rightarrow q$
T	T	T
T	F	F
F	T	T
F	F	T

The only
False case!

- $p \rightarrow q$ is **false** only when p is true but q is **not** true.
- $p \rightarrow q$ does **not** require that p or q are ever true!
- E.g. “ $(1=0) \rightarrow$ pigs can fly” is TRUE!



Lecture 2

Chapter 1. The Foundations

1.1 Propositional Logic



Review: The Implication Operator

- The conditional statement (a.k.a. *implication*)
 $p \rightarrow q$ states that p implies q .
- *I.e., If p is true, then q is true; but if p is not true, then q could be either true or false.*
- *E.g., let p = “You study hard.”
 q = “You will get a good grade.”
 $p \rightarrow q$ = “If you study hard, then you will get a good grade.” (else, it could go either way)*
 - p : *hypothesis* or *antecedent* or *premise*
 - q : *conclusion* or *consequence*



Review: Implication Truth Table

p	q	$p \rightarrow q$
T	T	T
T	F	F }
F	T	T
F	F	T

The only
False case!

- $p \rightarrow q$ is **false** only when p is true but q is **not** true.
- $p \rightarrow q$ does **not** require that p or q are ever true!
 - E.g. “ $(1=0) \rightarrow$ pigs can fly” is TRUE!



Examples of Implications

- “If this lecture ever ends, then the sun will rise tomorrow.” *True or False?* ($T \rightarrow T$)
- “If $1+1=6$, then Obama is president.”
True or False? ($F \rightarrow T$)
- “If the moon is made of green cheese, then I am richer than Bill Gates.” *True or False?* ($F \rightarrow F$)
- “If Tuesday is a day of the week, then I am a penguin.” *True or False?* ($T \rightarrow F$)



English Phrases Meaning $p \rightarrow q$

- “ p implies q ”
- “if p , then q ”
- “if p , q ”
- “when p , q ”
- “whenever p , q ”
- “ q if p ”
- “ q when p ”
- “ q whenever p ”

- “ p only if q ”
- “ p is sufficient for q ”
- “ q is necessary for p ”
- “ q follows from p ”
- “ q is implied by p ”

We will see some equivalent logic expressions later.

roads wet $\leftarrow p \rightarrow q \rightarrow$ rain

Converse, Inverse, Contrapositive

- Some terminology, for an implication $p \rightarrow q$:
- Its **converse** is: $\underline{q \rightarrow p}$.
- Its **inverse** is: $\underline{\neg p \rightarrow \neg q}$.
- Its **contrapositive**: $\underline{\neg q \rightarrow \neg p}$.

p	q	$p \rightarrow q$	$q \rightarrow p$	$\neg p \rightarrow \neg q$	$\neg q \rightarrow \neg p$
T	T	T ✓	T ✓	T	T
T	F	F ✗	F ✗	T	F
{ F	T	T	F ✗	F	T
{ F	F	T	T	T	T

- One of these three has the *same meaning* (same truth table) as $p \rightarrow q$. Can you figure out which?

Contrapositive



Examples

- p : Today is Easter

(Happens only on Sunday)
q: Tomorrow is Monday

- $p \rightarrow q$:

If today is Easter then tomorrow is Monday.

- **Converse**: $q \rightarrow p$

If tomorrow is Monday then today is Easter.

- **Inverse**: $\neg p \rightarrow \neg q$

If today is not Easter then tomorrow is not Monday.

- **Contrapositive**: $\neg q \rightarrow \neg p$

If tomorrow is not Monday then today is not Easter.



The Biconditional Operator

- The **biconditional** statement $p \leftrightarrow q$ states that p **if and only if** (iff) q .
- p = “It is below freezing.”
 q = “It is snowing.”
 $p \leftrightarrow q$ = “It is below freezing if and only if it is snowing.”

$p \rightarrow q$ and
 $q \rightarrow p$

or

= “That it is below freezing is necessary and sufficient for it to be snowing”

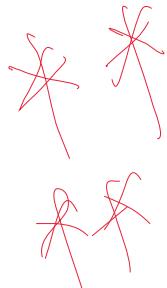


Biconditional Truth Table

p	q	$p \leftrightarrow q$
T	T	T
T	F	F
F	T	F
F	F	T

- p is necessary and sufficient for q
- If p then q , and conversely
- p iff q

- $p \leftrightarrow q$ is equivalent to $(p \rightarrow q) \wedge (q \rightarrow p)$.
- $p \leftrightarrow q$ means that p and q have the **same** truth value.
- $p \leftrightarrow q$ does **not** imply that p and q are true.
- Note this truth table is the exact **opposite** of \oplus 's! Thus, $p \leftrightarrow q$ means $\neg(p \oplus q)$.





Boolean Operations Summary

- Conjunction: $p \wedge q$, (read p and q), “discrete math is a required course **and** I am a computer science major”.
- Disjunction: , $p \vee q$, (read p or q), “discrete math is a required course **or** I am a computer science major”.
- Exclusive or: $p \oplus q$, “discrete math is a required course **or** I am a computer science major **but not both**”.
- Implication: $p \rightarrow q$, “**if** discrete math is a required course **then** I am a computer science major”.
- Biconditional: $p \leftrightarrow q$, “discrete math is a required course **if and only if** I am a computer science major”.



Boolean Operations Summary

- We have seen 1 unary operator and 5 binary operators. What are they? Their truth tables are below.

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

- For an implication $p \rightarrow q$
- Its **converse** is: $q \rightarrow p$
- Its **inverse** is: $\neg p \rightarrow \neg q$
- Its **contrapositive**: $\neg q \rightarrow \neg p$



Compound Propositions

- A ***propositional variable*** is a variable such as p , q , r (possibly subscripted, e.g. p_j) over the Boolean domain.
- An ***atomic proposition*** is either Boolean constant or a propositional variable: e.g. T , F , p
- A ***compound proposition*** is derived from atomic propositions by application of propositional operators:
e.g. $\neg p$, $p \vee q$, $(p \vee \neg q) \rightarrow q$
- Precedence of logical operators: \neg , \wedge , \vee , \rightarrow , \leftrightarrow
- Precedence also can be indicated by parentheses.
 - e.g. $\neg p \wedge q$ means $(\neg p) \wedge q$, not $\neg(p \wedge q)$



An Exercise

- Any compound proposition can be evaluated by a truth table
- $(p \vee \neg q) \rightarrow q$

p	q	$\neg q$	$p \vee \neg q$	$(p \vee \neg q) \rightarrow q$
T	T	F	T	T ✓
T	F	T	T	F ✓
F	T	F	F	T ✓
F	F	T	T	F ✓



Translating English Sentences

- Let p = “It rained last night”,
 q = “The sprinklers came on last night,”
 r = “The lawn was wet this morning.”

Translate each of the following into English:

$\neg p$

= “It didn’t rain last night.”

$r \wedge \neg p$

= “The lawn was wet this morning,
and it didn’t rain last night.”

$\neg r \vee p \vee q =$

“The lawn wasn’t wet this
morning, or it rained last night, or
the sprinklers came on last night.”



Another Example

- Find the converse of the following statement.
 - “Raining tomorrow is a sufficient condition for my not going to town.”
 - **Step 1:** Assign propositional variables to component propositions.
 - p : It will rain tomorrow
 - q : I will not go to town
 - **Step 2:** Symbolize the assertion: $p \rightarrow q$
 - **Step 3:** Symbolize the converse: $q \rightarrow p$
 - **Step 4:** Convert the symbols back into words.
 - “If I don’t go to town then it will rain tomorrow” or
 - “Raining tomorrow is a *necessary condition* for my not going to town.”
- p* → *q*
me
} *S*
a is a sufficient
for *b*.
 $a \rightarrow b$ ✓
 $b \rightarrow a$ ✗



Logic and Bit Operations

- A ***bit*** is a **binary** (base 2) **digit**: 0 or 1.
- Bits may be used to represent truth values.
 - By convention:
0 represents “False”; **1** represents “True”.
- A ***bit string of length n*** is an ordered sequence of $n \geq 0$ bits.
- By convention, bit strings are (sometimes) written left to right:
 - e.g. the “first” bit of the bit string “1001101010” is 1.
 - What is the length of the above bit string?



Bitwise Operations

- Boolean operations can be extended to operate on bit strings as well as single bits.
- Example:

01 1011 0110	
11 0001 1101	
11 1011 1111	Bit-wise <u>OR</u>
01 0001 0100	Bit-wise <u>AND</u>
10 1010 1011	Bit-wise <u>XOR</u>



Lecture 3

Chapter 1. The Foundations

1.2 Propositional Equivalences

1.3 Predicates and Quantifiers



1.2 Propositional Equivalence

- A **tautology** is a compound proposition that is **true** no matter what the truth values of its atomic propositions are! (T)
 - e.g. $p \vee \neg p$ (“Today the sun will shine or today the sun will not shine.”) [What is its truth table?] (F)
- A **contradiction** is a compound proposition that is **false** no matter what! (F)
 - e.g. $\underline{p} \wedge \underline{\neg p}$ (“Today is Wednesday and today is not Wednesday.”) [Truth table?]
- A **contingency** is a compound proposition that is neither a tautology nor a contradiction.
 - e.g. $(p \vee q) \rightarrow \neg r$



Logical Equivalence



- Compound proposition p is **logically equivalent** to compound proposition q , written $p \equiv q$ or $p \Leftrightarrow q$, iff the compound proposition $p \leftrightarrow q$ is a tautology.
- Compound propositions p and q are logically equivalent to each other iff p and q contain the same truth values as each other in all corresponding rows of their truth tables.



Proving Equivalence via Truth Tables

- Prove that $\neg(p \wedge q) \equiv \neg p \vee \neg q$. (De Morgan's law)

p	q	$p \wedge q$	$\neg p$	$\neg q$	$\neg p \vee \neg q$	$\neg(p \wedge q)$
T	T	T	F	F	F	F
T	F	F	F	T	T	T
F	T	F	T	F	T	T
F	F	F	T	T	T	T

Handwritten annotations: A pink oval encloses the columns for $\neg p \vee \neg q$ and $\neg(p \wedge q)$. A red oval encloses the values for $\neg p \vee \neg q$ in the last four rows. A pink arrow points from the label 'n' to the first column. A pink arrow points from the label 'e' to the second column. A pink arrow points from the label 'n' to the third column.

- Show that Check out the solution in the textbook!
- $\neg(p \vee q) \equiv \neg p \wedge \neg q$ (De Morgan's law)
- $p \rightarrow q \equiv \neg p \vee q$
- $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$ (distributive law)



Equivalence Laws

- These are similar to the arithmetic identities you may have learned in algebra, but for propositional equivalences instead.
- They provide a pattern or template that can be used to match part of a much more complicated proposition and to find an equivalence for it and possibly simplify it.



Equivalence Laws

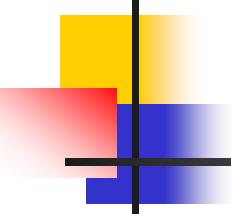
- *Identity:* $p \wedge T \equiv p$ $p \vee F \equiv p$
- *Domination:* $p \vee T \equiv T$ $p \wedge F \equiv F$
- *Idempotent:* $p \vee p \equiv p$ $p \wedge p \equiv p$
- *Double negation:* $\neg\neg p \equiv p$
- *Commutative:* $p \vee q \equiv q \vee p$ $p \wedge q \equiv q \wedge p$
- *Associative:* $(p \vee q) \vee r \equiv p \vee (q \vee r)$
 $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$



More Equivalence Laws

- *Distributive:* $p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
 $p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
- *De Morgan's:*
 $\neg(p \wedge q) \equiv \neg p \vee \neg q$
 $\neg(p \vee q) \equiv \neg p \wedge \neg q$
- *Absorption*
 $p \vee (p \wedge q) \equiv p$ $p \wedge (p \vee q) \equiv p$
- *Trivial tautology/contradiction:*
 $p \vee \neg p \equiv T$ $p \wedge \neg p \equiv F$

See Table 6, 7, and 8 of Section 1.2



Defining Operators via Equivalences



University of Hawaii

Using equivalences, we can *define* operators in terms of other operators.

- Exclusive or: $p \oplus q \equiv (p \wedge \neg q) \vee (\neg p \wedge q)$
 $p \oplus q \equiv (p \vee q) \wedge \neg(p \wedge q)$
- Implies: $p \rightarrow q \equiv \neg p \vee q$
- Biconditional: $p \leftrightarrow q \equiv (p \rightarrow q) \wedge (q \rightarrow p)$
 $p \leftrightarrow q \equiv \neg(p \oplus q)$

This way we can “normalize” propositions



An Example Problem

- Show that $\neg(p \rightarrow q)$ and $p \wedge \neg q$ are logically equivalent.

$\neg(p \rightarrow q)$ [Expand definition of \rightarrow]

$\equiv \neg(\neg p \vee q)$ [DeMorgan's Law]

$\equiv \neg(\neg p) \wedge \neg q$ [Double Negation]

$\equiv p \wedge \neg q$



Another Example Problem

- Check using a symbolic derivation whether

$$(p \wedge \neg q) \rightarrow (p \oplus r) \equiv \neg p \vee q \vee \neg r$$

$$\begin{aligned}(p \wedge \neg q) \rightarrow (p \oplus r) & \quad [\text{Expand definition of } \rightarrow] \\ \equiv \neg(p \wedge \neg q) \vee (p \oplus r) & \quad [\text{Expand definition of } \oplus] \\ \equiv \neg(p \wedge \neg q) \vee ((p \vee r) \wedge \neg(p \wedge r)) & \\ & \quad [\text{DeMorgan's Law}] \\ \equiv (\neg p \vee q) \vee ((p \vee r) \wedge \neg(p \wedge r)) &\end{aligned}$$

cont.



Example Continued...

$$(p \wedge \neg q) \rightarrow (p \oplus r) \equiv \neg p \vee q \vee \neg r$$

$$\begin{aligned} & (\neg p \vee q) \vee ((p \vee r) \wedge \neg(p \wedge r)) \quad [\vee \text{ Commutative}] \\ & \equiv (q \vee \neg p) \vee ((p \vee r) \wedge \neg(p \wedge r)) \quad [\vee \text{ Associative}] \\ & \equiv q \vee (\neg p \vee ((p \vee r) \wedge \neg(p \wedge r))) \quad [\text{Distribute } \vee \text{ over } \wedge] \\ & \equiv q \vee ((\neg p \vee (p \vee r)) \wedge (\neg p \vee \neg(p \wedge r))) \quad [\vee \text{ Assoc.}] \\ & \equiv q \vee ((\neg p \vee p) \vee r) \wedge (\neg p \vee \neg(p \wedge r)) \quad [\text{Trivial taut.}] \\ & \equiv q \vee ((\mathbf{T} \vee r) \wedge (\neg p \vee \neg(p \wedge r))) \quad [\text{Domination}] \\ & \equiv q \vee (\mathbf{T} \wedge (\neg p \vee \neg(p \wedge r))) \quad [\text{Identity}] \\ & \equiv q \vee (\neg p \vee \neg(p \wedge r)) \end{aligned}$$

cont.



End of Long Example

$$(p \wedge \neg q) \rightarrow (p \oplus r) \equiv \neg p \vee q \vee \neg r$$

$$q \vee (\neg p \vee \neg(p \wedge r)) \quad [\text{DeMorgan's Law}]$$

$$\equiv q \vee (\neg p \vee (\neg p \vee \neg r)) \quad [\vee \text{ Associative}]$$

$$\equiv q \vee ((\neg p \vee \neg p) \vee \neg r) \quad [\text{Idempotent}]$$

$$\equiv q \vee (\neg p \vee \neg r) \quad [\text{Associative}]$$

$$\equiv (q \vee \neg p) \vee \neg r \quad [\vee \text{ Commutative}]$$

$$\equiv \neg p \vee q \vee \neg r \quad \blacksquare$$



Review: Propositional Logic (1.1-1.2)

- Atomic propositions: p, q, r, \dots
- Boolean operators: $\neg \wedge \vee \oplus \rightarrow \leftrightarrow$
- Compound propositions: $(p \wedge \neg q) \vee r$
- Equivalences: $p \wedge \neg q \Leftrightarrow \equiv \neg(p \rightarrow q)$
- Proving equivalences using:
 - Truth tables
 - Symbolic derivations (series of logical equivalences) $p \equiv q \equiv r \equiv \dots$



1.3 Predicate Logic

- Consider the sentence

“For every x , $x > 0$ ”

If this were a true statement about the positive integers, it could not be adequately symbolized using only statement letters, parentheses and logical connectives.

*The sentence contains two new features: a **predicate** and a **quantifier***



Subjects and Predicates

- In the sentence “The dog is sleeping”:
 - The phrase “the dog” denotes the **subject** – the *object* or *entity* that the sentence is about.
 - The phrase “is sleeping” denotes the **predicate** – a property that the subject of the statement can have.
- In predicate logic, a **predicate** is modeled as a ***propositional function* $P(\cdot)$** from subjects to propositions.
 - $P(x) = \text{“}x \text{ is sleeping}\text{”}$ (where x is any subject).
 - $P(\text{The cat}) = \text{“}The \ cat \ is \ sleeping\text{”}$ (proposition!)



More About Predicates

- Convention: Lowercase variables $x, y, z\dots$ denote subjects; uppercase variables $P, Q, R\dots$ denote propositional functions (or predicates).
- Keep in mind that *the result of applying a predicate P to a value of subject x is the proposition*. But the predicate P , or the statement $P(x)$ **itself** (e.g. $P =$ “is sleeping” or $P(x) =$ “ x is sleeping”) is **not** a proposition.
 - e.g. if $P(x) =$ “ x is a prime number”,
 $P(3)$ is the *proposition* “3 is a prime number.”



Propositional Functions

- Predicate logic *generalizes* the grammatical notion of a predicate to also include propositional functions of **any** number of arguments, each of which may take **any** grammatical role that a noun can take.
 - e.g.:

let $P(x,y,z)$ = “ x gave y the grade z ”

then if

x = “Mike”, y = “Mary”, z = “A”,

then

$P(x,y,z)$ = “**Mike** gave **Mary** the grade **A**.”



Examples

- Let $P(x)$: $x > 3$. Then
 - $P(4)$ is TRUE/FALSE
 - $P(2)$ is TRUE/FALSE
- Let $Q(x, y)$: x is the capital of y . Then
 - $Q(\text{Washington D.C., U.S.A.})$ is TRUE
 - $Q(\text{Hilo, Hawaii})$ is FALSE
 - $Q(\text{Massachusetts, Boston})$ is FALSE
 - $Q(\text{Denver, Colorado})$ is TRUE
 - $Q(\text{New York, New York})$ is FALSE
- Read EXAMPLE 6 (pp.33)
 - If $x > 0$ then $x := x + 1$ (in a computer program)

$4 > 3$
$2 > 3$



Lecture 4

Chapter 1. The Foundations

1.3 Predicates and Quantifiers



Universe of Discourse (U.D.)

- The power of distinguishing subjects from predicates is that it lets you state things about *many* objects at once.
- e.g., let $P(x) = "x + 1 > x"$. We can then say, “For **any** number x , $P(x)$ is true” instead of $(0 + 1 > 0) \wedge (1 + 1 > 1) \wedge (2 + 1 > 2) \wedge \dots$
- The collection of values that a variable x can take is called x ’s ***universe of discourse*** or the ***domain of discourse*** (often just referred to as the ***domain***).



Quantifier Expressions

- **Quantifiers** provide a notation that allows us to *quantify* (count) *how many* objects in the universe of discourse satisfy the given predicate.
- “ \forall ” is the FOR \forall LL or *universal* quantifier.
 $\forall x P(x)$ means *for all* x in the domain, $P(x)$.
- “ \exists ” is the \exists XISTS or *existential* quantifier.
 $\exists x P(x)$ means *there exists* an x in the domain (that is, 1 or more) *such that* $P(x)$.



The Universal Quantifier \forall

- $\forall x P(x)$: **For all x in the domain, $P(x)$.**
- $\forall x P(x)$ is
 - **true** if $P(x)$ is true for every x in D (D : domain of discourse)
 - **false** if $P(x)$ is false for at least one x in D
 - For every real number x , $x^2 \geq 0$ TRUE
 - For every real number x , $x^2 - 1 > 0$ FALSE
- A **counterexample** to the statement $\forall x P(x)$ is a value x in the domain D that makes $P(x)$ false
- What is the truth value of $\forall x P(x)$ when the domain is empty? TRUE



The Universal Quantifier \forall

- If all the elements in the domain can be listed as x_1, x_2, \dots, x_n then, $\forall x P(x)$ is the same as the conjunction:

$$P(x_1) \wedge P(x_2) \wedge \cdots \wedge P(x_n)$$

- Example: Let the domain of x be parking spaces at UH. Let $P(x)$ be the statement “ x is full.” Then the ***universal quantification*** of $P(x)$, $\forall x P(x)$, is the *proposition*:
 - “All parking spaces at UH are full.”
 - or “Every parking space at UH is full.”
 - or “For each parking space at UH, that space is full.”



The Existential Quantifier \exists

- $\exists x P(x)$: *There exists an x in the domain (that is, 1 or more) such that $P(x)$.*
- $\exists x P(x)$ is
 - *true* if $P(x)$ is true for at least one x in the domain
 - *false* if $P(x)$ is false for every x in the domain
- What is the truth value of $\exists x P(x)$ when the domain is empty? FALSE
- If all the elements in the domain can be listed as x_1, x_2, \dots, x_n then, $\exists x P(x)$ is the same as the disjunction:

$$P(x_1) \vee P(x_2) \vee \cdots \vee P(x_n)$$



The Existential Quantifier \exists

- Example:

Let the domain of x be parking spaces at UH.

Let $P(x)$ be the statement “ x is full.”

Then the ***existential quantification*** of $P(x)$,
 $\exists x P(x)$, is the *proposition*:

- “Some parking spaces at UH are full.”
- or “There is a parking space at UH that is full.”
- or “At least one parking space at UH is full.”



Free and Bound Variables

- An expression like $P(x)$ is said to have a ***free variable*** x (meaning, x is undefined).
- A quantifier (either \forall or \exists) *operates* on an expression having one or more free variables, and ***binds*** one or more of those variables, to produce an expression having one or more ***bound variables***.



Example of Binding

- $P(x,y)$ has 2 free variables, x and y .
- $\forall x P(x,y)$ has 1 free variable , and one bound variable . [Which is which?]
- “ $P(x)$, where $x = 3$ ” is another way to bind x .
- An expression with zero free variables is a bona-fide (actual) proposition.
- An expression with one or more free variables is not a proposition:

e.g. $\forall x P(x,y) = Q(y)$



Quantifiers with Restricted Domains

- Sometimes the universe of discourse is restricted within the quantification, e.g.,

- $\forall x > 0 P(x)$ is shorthand for

“For all x that are greater than zero, $P(x)$.[”]

$$= \forall x (x > 0 \rightarrow P(x))$$

False $\longrightarrow P(x)$
True

- $\exists x > 0 P(x)$ is shorthand for

“There is an x greater than zero such that

$P(x)$.[”]

$$= \exists x (x > 0 \wedge P(x))$$



domain

Translating from English

- Express the statement “Every student in this class has studied calculus” using predicates and quantifiers.
 - Let $C(x)$ be the statement: “ x has studied calculus.”
 - If domain for x consists of the students in this class, then
 - it can be translated as $\forall x C(x)$

or

- If domain for x consists of all people
- Let $S(x)$ be the predicate: “ x is in this class”
- Translation: $\forall x (S(x) \rightarrow C(x))$



Translating from English

- Express the statement “*Some students in this class has visited Mexico*” using predicates and quantifiers.
 - Let $M(x)$ be the statement: “*x has visited Mexico*”
 - If domain for x consists of the students in this class, then
 - it can be translated as $\exists x M(x)$
 - or
 - If domain for x consists of all people
 - Let $S(x)$ be the statement: “*x is in this class*”
 - Then, the translation is $\exists x (S(x) \wedge M(x))$



Translating from English

- Express the statement “*Every student in this class has visited either Canada or Mexico*” using predicates and quantifiers.
 - Let $C(x)$ be the statement: “*x has visited Canada*” and $M(x)$ be the statement: “*x has visited Mexico*”
 - If domain for x consists of the students in this class, then
 - it can be translated as $\forall x (C(x) \vee M(x))$



Negations of Quantifiers

- $\forall x P(x)$: “Every student in the class has taken a course in calculus” ($P(x)$: “x has taken a course in calculus”)
 - “Not every student in the class ... calculus”

$$\neg \underline{\forall x P(x)} \equiv \exists x \underline{\neg P(x)}$$

- Consider $\exists x P(x)$: “There is a student in the class who has taken a course in calculus”
 - “There is no student in the class who has taken a course in calculus”

$$\neg \exists x P(x) \equiv \forall x \underline{\neg P(x)}$$



Negations of Quantifiers

- Definitions of quantifiers: If the domain = $\{a, b, c, \dots\}$

- $\forall x P(x) \equiv \underline{P(a) \wedge P(b) \wedge P(c) \wedge \dots}$

- $\exists x P(x) \equiv \underline{P(a) \vee P(b) \vee P(c) \vee \dots}$

- From those, we can prove the laws:

- $\neg \forall x P(x) \equiv \neg (\underline{P(a) \wedge P(b) \wedge P(c) \wedge \dots})$

$$\equiv \neg P(a) \vee \neg P(b) \vee \neg P(c) \vee \dots$$

$$\equiv \exists x \neg P(x)$$

- $\neg \exists x P(x) \equiv \neg (\underline{P(a) \vee P(b) \vee P(c) \vee \dots})$

$$\equiv \neg P(a) \wedge \neg P(b) \wedge \neg P(c) \wedge \dots$$

$$\equiv \forall x \neg P(x)$$

- Which *propositional* equivalence law was used to prove this?

DeMorgan's



Negations of Quantifiers

Theorem:

- Generalized De Morgan's laws for logic

$$1. \neg \forall x P(x) \equiv \exists x \neg P(x)$$

$$2. \neg \exists x P(x) \equiv \forall x \neg P(x)$$



Negations: Examples

- What are the negations of the statements $\forall x (x^2 > x)$ and $\exists x (x^2 = 2)$?
 - $\neg \forall x (x^2 > x) \equiv \exists x \neg(x^2 > x) \equiv \exists x (x^2 \leq x)$
 - $\neg \exists x (x^2 = 2) \equiv \forall x \neg(x^2 = 2) \equiv \forall x (x^2 \neq 2)$
- Show that $\neg \forall x(P(x) \rightarrow Q(x))$ and $\exists x(P(x) \wedge \neg Q(x))$ are logically equivalent.
 - $\neg \forall x(P(x) \rightarrow Q(x)) \equiv \exists x \neg(P(x) \rightarrow Q(x))$
 $\equiv \exists x \neg(\neg P(x) \vee Q(x))$
 $\equiv \exists x (P(x) \wedge \neg Q(x))$



Summary

© The McGraw-Hill Companies, Inc. all rights reserved.

TABLE 1 Quantifiers.

<i>Statement</i>	<i>When True?</i>	<i>When False?</i>
$\forall x P(x)$	$P(x)$ is true for every x .	There is an x for which $P(x)$ is false.
$\exists x P(x)$	There is an x for which $P(x)$ is true.	$P(x)$ is false for every x .

© The McGraw-Hill Companies, Inc. all rights reserved.

TABLE 2 De Morgan's Laws for Quantifiers.

<i>Negation</i>	<i>Equivalent Statement</i>	<i>When Is Negation True?</i>	<i>When False?</i>
$\neg \exists x P(x)$	$\forall x \neg P(x)$	For every x , $P(x)$ is false.	There is an x for which $P(x)$ is true.
$\neg \forall x P(x)$	$\exists x \neg P(x)$	There is an x for which $P(x)$ is false.	$P(x)$ is true for every x .



Lecture 5

Chapter 1. The Foundations

1.4 Nested Quantifiers

1.5 Rules of Inference

Nesting of Quantifiers

2 or more
quantifiers

University of Hawaii

- Example:

Let the domain of \underline{x} and \underline{y} be people.

Let $L(\underline{x}, \underline{y})$ = “ x likes y ” (A statement with 2 free variables – not a proposition)

- Then $\exists \underline{y} L(\underline{x}, \underline{y})$ = “There is someone whom x likes.” (A statement with 1 free variable x – not a proposition)

$\underline{y} \rightarrow$ bounded

- Then $\forall \underline{x} (\exists \underline{y} L(\underline{x}, \underline{y}))$ =
“Everyone has someone whom they like.”
(A Proposition with 0 free variables.)

Nested Quantifiers

- Nested quantifiers are quantifiers that occur within the scope of other quantifiers.
- The order of the quantifiers is **important**, unless all the quantifiers are universal quantifiers or all are existential quantifiers.

© The McGraw-Hill Companies, Inc. all rights reserved.

TABLE 1 Quantifications of Two Variables.

<i>Statement</i>	<i>When True?</i>	<i>When False?</i>
$\{ \forall x \forall y P(x, y) \}$ $\forall y \forall x P(x, y)$	$P(x, y)$ is true for every pair x, y . ✓	There is a pair x, y for which $P(x, y)$ is false.
$\forall x \exists y P(x, y)$	For every x there is a y for which $P(x, y)$ is true.	There is an x such that $P(x, y)$ is false for every y .
$\exists x \forall y P(x, y)$	There is an x for which $P(x, y)$ is true for every y .	For every x there is a y for which $P(x, y)$ is false.
$\exists x \exists y P(x, y)$ $\exists y \exists x P(x, y)$	There is a pair x, y for which $P(x, y)$ is true.	$P(x, y)$ is false for every pair x, y .

*x: people
y: subjects
P(x,y)
x has passed
y*



Nested Quantifiers

- Let the domain of x and y is \mathbf{R} , and $P(x,y)$: $xy = 0$.
Find the truth value of the following propositions.
 - $\forall x \forall y P(x, y)$ (F)
 - $\forall x \exists y P(x, y)$ (T) $y=0$
 - $\exists x \forall y P(x, y)$ (T) $x=0$
 - $\exists x \exists y P(x, y)$ (T) ✓
- $\forall x \exists y P(x, y) \neq \exists y \forall x P(x, y)$
 - For every x , there exists y such that $x + y = 0$. (T)
 - There exists y such that, for every x , $x + y = 0$. (F)

R: set of real numbers



Nested Quantifiers: Example

- Let the domain = $\{1, 2, 3\}$. Find an expression equivalent to $\forall x \exists y P(x,y)$ where the variables are bound by substitution instead:
 - Expand from inside out or outside in.
 - Outside in:

$$\begin{aligned} & \text{Let } x = \{1, 2, 3\} \quad \forall x R(x) \\ & \quad R(x_1) \wedge R(x_2) \\ & \text{Original expression: } \forall x \exists y P(x,y) \\ & \quad \text{Substituted: } \exists y P(1,y) \wedge \exists y P(2,y) \wedge \exists y P(3,y) \\ & \quad \text{Expanded: } [P(1,1) \vee P(1,2) \vee P(1,3)] \wedge \\ & \quad \quad [P(2,1) \vee P(2,2) \vee P(2,3)] \wedge \\ & \quad \quad [P(3,1) \vee P(3,2) \vee P(3,3)] \end{aligned}$$



Quantifier Exercise



- If $R(x,y)$ = “ x relies upon y ,” express the following in unambiguous English when the domain is all people

$$\forall x(\exists y R(x,y)) = \text{Everyone has } \textit{someone} \text{ to rely on.}$$

$$\exists y(\forall x R(x,y)) =$$

There's a poor overburdened soul whom *everyone* relies upon (including himself)!

$$\exists x(\forall y R(x,y)) =$$

There's some needy person who relies upon *everybody* (including himself).

$$\forall y(\exists x R(x,y)) =$$

Everyone has *someone* who relies upon them.

$$\forall x(\forall y R(x,y)) =$$

Everyone relies upon *everybody*, (including themselves)!

$$\neg [\forall x P(x)] \equiv \exists x \neg P(x) \quad \neg \exists x P(x) = \forall x \neg P(x)$$

Negating Nested Quantifiers

University of Hawaii

- Successively apply the rules for negating statements involving a single quantifier
- Example: Express the negation of the statement $\neg (\forall x \exists y (P(x,y) \wedge \exists z R(x,y,z)))$ so that all negation symbols immediately precede predicates.

$$\begin{aligned}
 & \neg \forall x \exists y (P(x,y) \wedge \exists z R(x,y,z)) \\
 & \equiv \exists x \neg \exists y (P(x,y) \wedge \exists z R(x,y,z)) \\
 & \equiv \exists x \forall y \neg (P(x,y) \wedge \exists z R(x,y,z)) \\
 & \equiv \exists x \forall y (\neg P(x,y) \vee \neg \exists z R(x,y,z)) \\
 & \equiv \exists x \forall y (\neg P(x,y) \vee \forall z \neg R(x,y,z))
 \end{aligned}$$



Equivalence Laws

- $\forall x \forall y P(x,y) \equiv \forall y \forall x P(x,y)$ ✓
 $\exists x \exists y P(x,y) \equiv \exists y \exists x P(x,y)$

- $\forall x (P(x) \wedge Q(x)) \equiv (\forall x P(x)) \wedge (\forall x Q(x))$
 $\exists x (P(x) \vee Q(x)) \equiv (\exists x P(x)) \vee (\exists x Q(x))$

- Exercise:
See if you can prove these yourself.



Notational Conventions



- Quantifiers have higher precedence than all logical operators from propositional logic:

$$(\forall x P(x)) \wedge Q(x)$$

- Consecutive quantifiers of the same type can be combined:

$$\forall x \forall y \forall z P(x,y,z) \equiv \underline{\forall x, y, z} P(x,y,z)$$

$$\text{or even } \underline{\forall xyz} P(x,y,z)$$



1.5 Rules of Inference

- **An argument:** a sequence of statements that end with a conclusion
- Some forms of argument (“valid”) never lead from correct statements to an incorrect conclusion. Some other forms of argument (“fallacies”) can lead from true statements to an incorrect conclusion.
- **A logical argument** consists of a list of (possibly compound) propositions called premises/hypotheses and a single proposition called the conclusion.
- **Logical rules of inference:** methods that depend on logic alone for deriving a new statement from a set of other statements. (Templates for constructing valid arguments.)

Valid Arguments (I)

- Example: A logical argument

- If I dance all night, then I get tired.

- I danced all night.

Therefore I got tired.

- Logical representation of underlying variables:

p: I dance all night.

q: I get tired.

- Logical analysis of argument:

$$p \rightarrow q$$

premise 1

$$\underline{p}$$

premise 2

$$\therefore q$$

conclusion

premise
hypothesis
conclusion

p₁ ✓
p₂ ✓
∴ *c₂* ✓



Valid Arguments (II)

- A form of logical argument is *valid* if whenever every premise is true, the conclusion is also true. A form of argument that is not valid is called a *fallacy*.



Inference Rules: General Form

- An *Inference Rule* is
 - A pattern establishing that if we know that a set of *premise* statements of certain forms are all true, then we can validly deduce that a certain related *conclusion* statement is true.

✓ premise 1
✓ premise 2
...

∴ conclusion

“∴” means “therefore”



Inference Rules & Implications

- Each valid logical inference rule corresponds to an implication that is a tautology.

premise 1

premise 2

...

\therefore *conclusion*

Inference rule

- Corresponding tautology:
 $((\text{premise 1}) \wedge (\text{premise 2}) \wedge \dots) \rightarrow \text{conclusion}$

Modus Ponens

 $\{P_1 \wedge P_2 \wedge P_3\} \rightarrow C$

- Rule of ***Modus ponens***
(a.k.a. *law of detachment*)
- $(p \wedge (p \rightarrow q)) \rightarrow q$ is a tautology

“the mode of affirming”

p	q	$p \rightarrow q$	$p \wedge (p \rightarrow q)$	$(p \wedge (p \rightarrow q)) \rightarrow q$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

- Notice that the first row is the only one where premises are all true



Modus Ponens: Example

If $\left\{ \begin{array}{l} p \rightarrow q : \text{"If it snows today} \\ \quad \text{then we will go skiing"} \\ \hline p \end{array} \right. \right\}$ assumed TRUE
Then $\therefore q$: "It is snowing today" is TRUE

If $\left\{ \begin{array}{l} p \rightarrow q : \text{"If } n \text{ is divisible by 3} \\ \quad \text{then } n^2 \text{ is divisible by 3"} \\ \hline p \end{array} \right. \right\}$ assumed TRUE
Then $\therefore q$: "n is divisible by 3" is TRUE



$$\neg p \rightarrow q$$

Modus Tollens

- $$\begin{array}{c} \neg q \\ p \rightarrow q \\ \hline \therefore \neg p \end{array}$$
 Rule of ***Modus tollens***
- $$(\neg q \wedge (p \rightarrow q)) \rightarrow \neg p$$
 is a tautology

“the mode of denying”

If $\left\{ \begin{array}{l} p \rightarrow q : \text{“If this jewel is really a diamond} \\ \quad \text{then it will scratch glass”} \\ \neg q : \text{“The jewel doesn’t scratch glass”} \end{array} \right\}$ assumed TRUE

Then $\frac{\neg q}{\therefore \neg p} : \text{“The jewel is not a diamond”}$ is TRUE



More Inference Rules

- $$\begin{array}{c} p \\ \hline \therefore p \vee q \end{array}$$

Rule of **Addition**

Tautology: $p \rightarrow (p \vee q)$

- $$\begin{array}{c} p \wedge q \\ \hline \therefore p \end{array}$$

Rule of **Simplification**

Tautology: $(p \wedge q) \rightarrow p$

- $$\begin{array}{c} p \checkmark \\ q \checkmark \\ \hline \therefore p \wedge q \end{array}$$

Rule of **Conjunction**

Tautology: $[(p) \wedge (q)] \rightarrow p \wedge q$

Hypothetical Syllogism

- $$\begin{array}{c} p \rightarrow q \\ q \rightarrow r \\ \hline \therefore p \rightarrow r \end{array}$$
 Rule of *Hypothetical syllogism*
Tautology:
$$[(p \rightarrow q) \wedge (q \rightarrow r)] \rightarrow (p \rightarrow r)$$
- Example: State the rule of inference used in the argument:

“If it rains today, then we will not have a barbecue today. If we do not have a barbecue today, then we will have a barbecue tomorrow. Therefore, if it rains today, then we will have a barbecue tomorrow.”

Disjunctive Syllogism

- $$\begin{array}{c} p \vee q \\ \neg p \\ \hline \therefore q \end{array}$$
 Rule of ***Disjunctive syllogism***
- Tautology: $[(p \vee q) \wedge (\neg p)] \rightarrow q$
- Example
 - Ed's wallet is in his back pocket or it is on his desk. $(p \vee q)$ p q
 - Ed's wallet is not in his back pocket. $(\neg p)$
 - Therefore, Ed's wallet is on his desk. (q)



Lecture 6

Chapter 1. The Foundations

1.5 Rules of Inference



Previously...

- Rules of inference
 - Modus ponens
 - Modus tollens
 - Hypothetical syllogism
 - Disjunctive syllogism
 - Resolution
 - Addition
 - Simplification
 - Conjunction

Table 1 in pp.66



Resolution

$$\frac{p \vee q}{\neg p \vee r} \quad \frac{\neg p \vee r}{q \vee r}$$

$$\begin{array}{c} p \vee q \\ \neg p \vee r \\ \hline \therefore q \vee r \end{array}$$

Rule of **Resolution**

Tautology:

$$[(p \vee q) \wedge (\neg p \vee r)] \rightarrow (q \vee r)$$

$$\frac{p \vee q}{q \vee r}$$

When $q = r$:

$$[(p \vee q) \wedge (\neg p \vee q)] \rightarrow q$$

When $r = F$:

$$[(p \vee q) \wedge (\neg p)] \rightarrow q \quad (\text{Disjunctive syllogism})$$

Resolution: Example

$$\begin{array}{c} p \vee q \\ \neg p \vee r \\ \hline \therefore q \vee r \end{array}$$

- Example: Use resolution to show that the hypotheses “Jasmine is skiing or it is not snowing” and “It is snowing or Bart is playing hockey” imply that “Jasmine is skiing or Bart is playing hockey”

$$(p \vee q) \wedge (\neg p \vee r) \rightarrow (q \vee r)$$



Artificial Intelligence (IT38005)

Fuzzy Sets and Operations

Anup Kumar Gupta

Fuzzy Computing

- In the real world there exists much fuzzy knowledge, that is, knowledge which is vague, imprecise, uncertain, ambiguous, inexact, or probabilistic in nature.
- Human can use such information because the human thinking and reasoning frequently involve fuzzy information, possibly originating from inherently inexact human concepts and matching of similar rather than identical experiences.
- The computing systems, based upon classical set theory and two-valued logic, can not answer to some questions, as human does, because they do not have completely true answers.
- We want, the computing systems should not only give human like answers but also describe their reality levels.
- These levels need to be calculated using imprecision and the uncertainty of facts and rules that were applied.

Fuzzy Sets

- Introduced by Lotfi Zadeh in 1965, the fuzzy set theory is an extension of classical set theory where elements have degrees of membership.

Classical Set Theory

- Sets are defined by a simple statement describing whether an element having a certain property belongs to a particular set.
- When set A is contained in a universal space X , then we can state explicitly whether each element x of space X "is or is not" an element of A .
- Set A is well described by a function called characteristic function, $\mu_A(x)$.
- This function, defined on the universal space X , assumes
 - value 1 for those elements x that belong to set A , and
 - value 0 for those elements x that do not belong to set A .

Crisp and Non-crisp Sets

- In classical set theory, the characteristic function $\mu_A(x)$ has only the values 0 ('False') and 1 ('True'). Such sets are crisp sets.
- For non-crisp sets the a more generalized characteristic function $\mu_A(x)$ can be defined.
- This generalized characteristic function $\mu_A(x)$ in case of non-crisp sets is called membership function.
- Such non-crisp sets are also called as Fuzzy Sets.
- Crisp set theory is not capable of representing descriptions and classifications in many cases. It does not provide adequate representation for most cases.
- The proposition of Fuzzy Sets are motivated by the need to capture and represent real world data with uncertainty due to imprecise measurement.

Membership function, $\mu_A(\cdot)$

- A fuzzy set is described by a membership function $\mu_A(x)$ of A .
- This membership function associates to each element $x \in X$ a number as $\mu_A(x)$, in the closed unit interval $[0, 1]$.
- The number $\mu_A(x)$ represents the degree of membership of x in A .

In the case of crisp sets, the members of a set are:

- either out of the set, with membership of degree "0",
- or in the set, with membership of degree "1".

Therefore, Crisp Sets \subseteq Fuzzy Sets

In other words, Crisp Sets are Special cases of Fuzzy Sets.

Fuzzy Set: Definition

- A fuzzy set A defined in the universal space X is a function defined on domain X which assumes values in the range $[0, 1]$.
- A fuzzy set A is written as a set of pairs $\{x, \mu_A(x)\}$ as $A = \{\{x, \mu_A(x)\}\}, \forall x \in X$
- The $\mu_A(x)$ represents the degree of membership of x in A .

Fuzzy Sets

Set A is well described by a function called characteristic function, $\mu_A(x)$.

- This function, defined on the universal space X , assumes
 - value 1 for those elements x that belong to set A , and
 - value 0 for those elements x that do not belong to set A .
- Mathematically,

$$\mu_A(x) = \begin{cases} 1 & \text{if } x \in X \\ 0 & \text{otherwise} \end{cases}$$

- Thus, in classical set theory $\mu_A(x)$ has only the values 0 ('False') and 1 ('True'). Such sets are called **crisp sets**.

Fuzzy Set: Example

- A fuzzy set A is written as a set of pairs $\{x, \mu_A(x)\}$ as $A = \{\{x, \mu_A(x)\}\}, \forall x \in X$

Example: A set *Small* defined on the set $X = \{x: x \in N \text{ and } x \leq 12\}$. Assume that:

$$\begin{array}{ll} \mu_S(1) = 1 & \mu_S(2) = 1 \\ \mu_S(3) = 0.9 & \mu_S(4) = 0.6 \\ \mu_S(5) = 0.4 & \mu_S(6) = 0.3 \\ \mu_S(7) = 0.2 & \mu_S(8) = 0.1 \\ \mu_S(x) = 0 & \forall x \geq 9 \end{array}$$

- Then, following the notations described in the definition above:
- $Small = \{\{1, 1\}, \{2, 1\}, \{3, 0.9\}, \{4, 0.6\}, \{5, 0.4\}, \{6, 0.3\}, \{7, 0.2\}, \{8, 0.1\}, \{9, 0\}, \{10, 0\}, \{11, 0\}, \{12, 0\}\}$
- Note that a fuzzy set can be defined by associating with each x , its value of membership in *Small*.

Example 1 : Heap Paradox

This example represents a situation where vagueness and uncertainty are inevitable.

- If we remove one grain from a heap of grains, we will still have a heap.
- However, if we keep removing one-by-one grain from a heap of grains, there will be a time when we do not have a heap anymore.
- The question is:

“At what time does the heap turn into a countable collection of grains that do not form a heap?”

- There is no one correct answer to this question.

Universal Space: Definition

- The universal space for fuzzy sets and fuzzy relations is defined with three numbers.
- The first two numbers specify the start and end of the universal space, and the third argument specifies the increment between elements.
- This gives the user more flexibility in choosing the universal space.

Example : The fuzzy set of numbers, defined in the universal space

$X = \{x_i\} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$ is presented as:

Universal Space, $X = \{0, 12, 1\}$

- 0 – Starting point
- 12 – Ending Point
- 1 – Increment

Finite and Infinite Universal Space

- Universal sets can be finite or infinite.
 - Any universal set is finite if it consists of a specific number of different elements, that is, if in counting the different elements of the set, the counting can come to an end, else the set is infinite.
1. Let N be the universal space of the days of the week. $N = \{\text{Mo, Tu, We, Th, Fr, Sa, Su}\}$. N is finite.
 2. Let $M = \{1, 3, 5, 7, 9, \dots\}$. M is infinite.
 3. Let $L = \{u \mid u \text{ is a lake in a city}\}$. L is finite.
(Although it may be difficult to count the number of lakes in a city, but L is still a finite universal set.)

Empty Sets

- An **Empty set** is a set that contains only elements with a grade of membership equal to 0.
- Example: Let *Older* be a set of people, in Indore, older than 200.
- The **Empty set** is also called the **Null set**.

Example:

- $Empty = \{\{1, 0\}, \{2, 0\}, \{3, 0\}, \{4, 0\}, \{5, 0\}, \{6, 0\}, \{7, 0\}, \{8, 0\}, \{9, 0\}, \{10, 0\}, \{11, 0\}, \{12, 0\}\}$
- Note that the membership value of each element is 0.

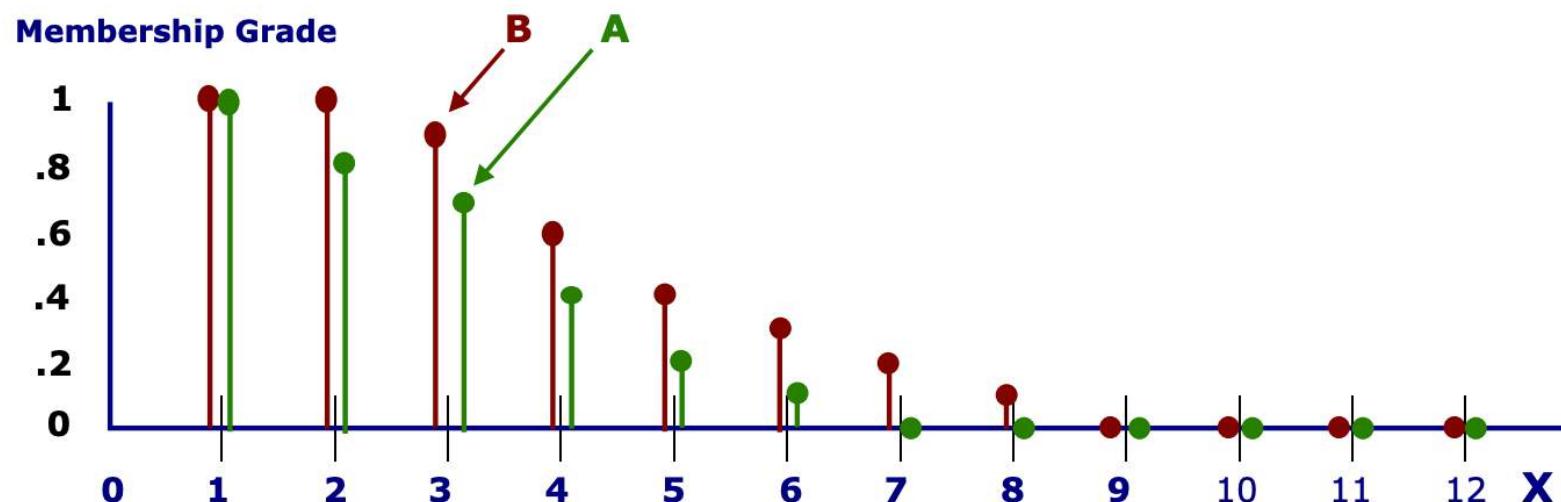
Fuzzy Sets Operations

- Fuzzy set operations are the operations on fuzzy sets.
- The fuzzy set operations are generalization of crisp set operations.
- Zadeh [1965] formulated the fuzzy set theory in the terms of standard operations: Complement, Union, Intersection, and Difference.
- We will look into the following operations:
 1. Inclusion
 2. Equality
 3. Complement
 4. Union
 5. Intersection

Inclusion

- Let A and B be fuzzy sets defined in the same universal space X .
- The fuzzy set A is said to be included in the fuzzy set B if and only if for every x in the set X we have $\mu_A(x) \leq \mu_B(x)$.

$$B = \{\{1, 1\}, \{2, 1\}, \{3, 0.9\}, \{4, 0.6\}, \{5, 0.4\}, \{6, 0.3\}, \{7, 0.2\}, \{8, 0.1\}, \{9, 0\}, \{10, 0\}, \{11, 0\}, \{12, 0\}\}$$
$$A = \{\{1, 1\}, \{2, 0.8\}, \{3, 0.7\}, \{4, 0.4\}, \{5, 0.2\}, \{6, 0.1\}, \{7, 0\}, \{8, 0\}, \{9, 0\}, \{10, 0\}, \{11, 0\}, \{12, 0\}\}$$



Comparability

- Let A and B be fuzzy sets defined in the same universal space X .
- Two fuzzy sets A and B are comparable if the condition $A \subset B$ or $B \subset A$ holds, i.e.,
- if one of the fuzzy sets is a subset of the other set, they are comparable.
- Two fuzzy sets A and B are incomparable if the condition $A \not\subset B$ or $B \not\subset A$ holds.

Examples

- Let $A = \{\{a, 1\}, \{b, 1\}, \{c, 0\}\}$ and $B = \{\{a, 1\}, \{b, 1\}, \{c, 1\}\}$.

Then A is comparable to B , since A is a subset of B .

- Let $C = \{\{a, 1\}, \{b, 1\}, \{c, 0.5\}\}$ and $D = \{\{a, 1\}, \{b, 0.9\}, \{c, 0.6\}\}$.

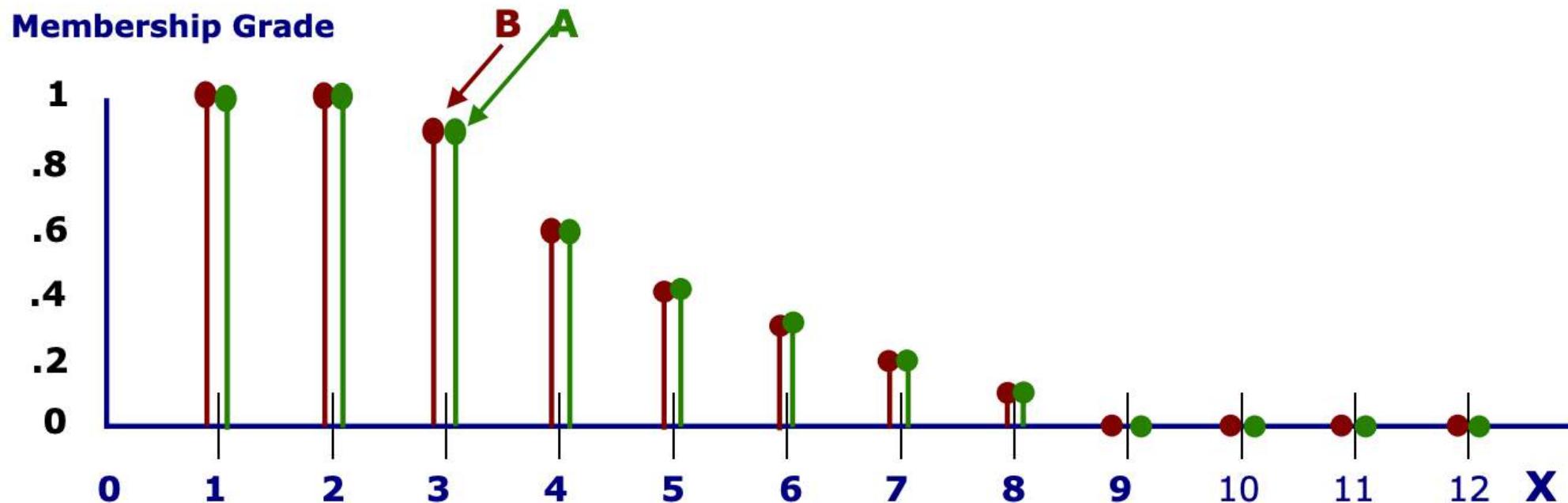
Then C and D are not comparable since C is not a subset of D and D is not a subset of C .

Note:

- If $A(x) \subset B(x) \subset C(x)$, then accordingly $A(x) \subset C(x)$.

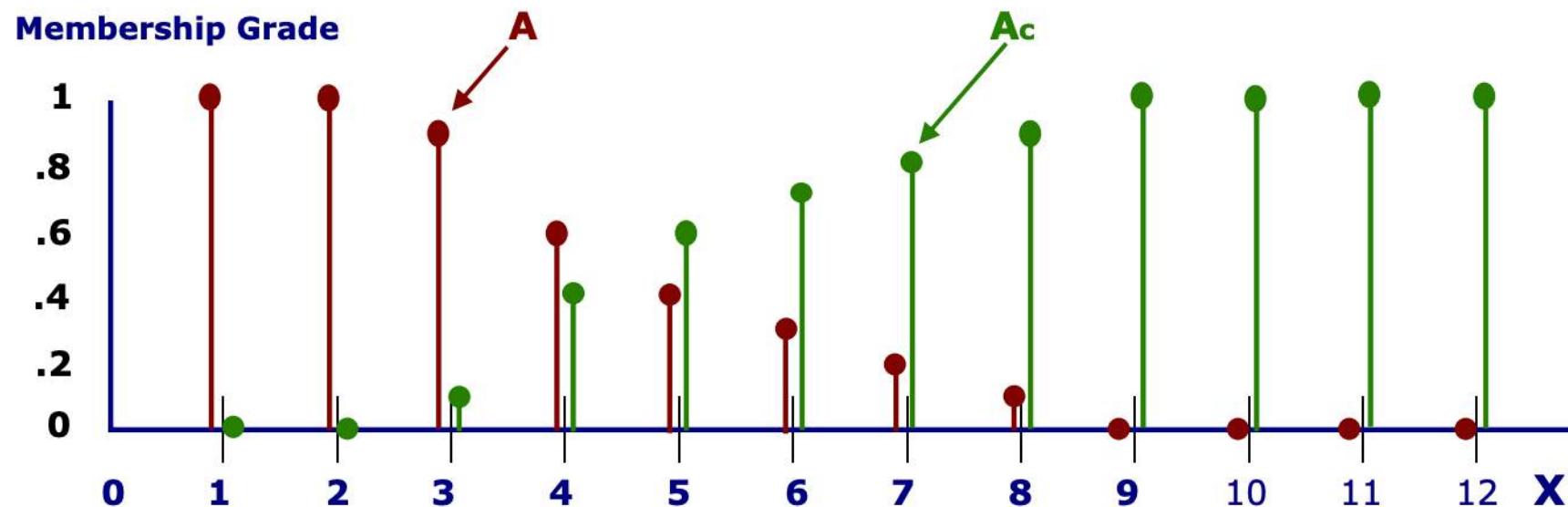
Equality

- Let A and B be fuzzy sets defined in the same universal space X .
- Two fuzzy sets A and B are equal if the condition $\mu_A(x) = \mu_B(x) \forall x \in X$ holds.
- Note : If equality $A(x) = B(x)$ is not satisfied even for one element x in the set X , then we say that A is not equal to B .



Complement

- Let A be a fuzzy set defined in the universal space X .
- Then the fuzzy set B is a complement of the fuzzy set A , if and only if, $\mu_A(x) = 1 - \mu_B(x) \forall x \in X$
- The complement of the fuzzy set A is often denoted by \bar{A} , A' or A^c .
- Note : If $\bar{A} = B$ then $\bar{B} = A$.



Union

- Let A and B be fuzzy sets defined in the same universal space X .
- The union is defined as the smallest fuzzy set that contains both A and B .
- The union of A and B is denoted by $A \cup B$.
- The following relation must be satisfied for the union operation:

$$\forall x \in X, \mu_{A \cup B}(x) = \max(\mu_A(x), \mu_B(x))$$

Example

$$A = \{\{1, 1\}, \{2, 1\}, \{3, 0.9\}, \{4, 0.6\}, \{5, 0.4\}, \{6, 0.3\}, \{7, 0.2\}, \{8, 0.1\}, \{9, 0\}, \{10, 0\}, \{11, 0\}, \{12, 0\}\}$$

$$B = \{\{1, 0\}, \{2, 0\}, \{3, 0\}, \{4, 0.2\}, \{5, 0.5\}, \{6, 0.8\}, \{7, 1\}, \{8, 1\}, \{9, 0.7\}, \{10, 0.4\}, \{11, 0.1\}, \{12, 0\}\}$$

$$A \cup B = \{\{1, 1\}, \{2, 1\}, \{3, 0.9\}, \{4, 0.6\}, \{5, 0.5\}, \{6, 0.8\}, \{7, 1\}, \{8, 1\}, \{9, 0.7\}, \{10, 0.4\}, \{11, 0.1\}, \{12, 0\}\}$$

Union

- The notion of the union is closely related to that of the connective "or".
- Let A is a set of students that take the course Soft Computing, and B is a set of students that take the course Machine Learning.
- If “ $Ashu$ takes the course Soft Computing” or “ $Ashu$ takes the course Machine Learning” then $Ashu$ is associated with the union of A and B .
- Implies $Ashu$ is a member of $A \cup B$.
- If $Trishna$ is not taking any of the courses, then $Trishna$ is **not** a member of $A \cup B$.

Properties Related to Union

The properties related to union are :

Identity, Idempotence, Commutativity and Associativity.

Identity:

$$\bullet A \cup \phi = A$$

$$\bullet A \cup X = X$$

Idempotence:

$$\bullet A \cup A = A$$

Commutativity:

$$\bullet A \cup B = B \cup A$$

Associativity:

$$\bullet A \cup (B \cup C) = (A \cup B) \cup C$$

Intersection

- Let A and B be fuzzy sets defined in the same universal space X .
- The intersection is defined as the largest fuzzy set included by both A and B .
- The intersection of A and B is denoted by $A \cap B$.
- The following relation must be satisfied for the intersection operation:

$$\forall x \in X, \quad \mu_{A \cap B}(x) = \min(\mu_A(x), \mu_B(x))$$

Example

$$A = \{\{1, 1\}, \{2, 1\}, \{3, 0.9\}, \{4, 0.6\}, \{5, 0.4\}, \{6, 0.3\}, \{7, 0.2\}, \{8, 0.1\}, \{9, 0\}, \{10, 0\}, \{11, 0\}, \{12, 0\}\}$$

$$B = \{\{1, 0\}, \{2, 0\}, \{3, 0\}, \{4, 0.2\}, \{5, 0.5\}, \{6, 0.8\}, \{7, 1\}, \{8, 1\}, \{9, 0.7\}, \{10, 0.4\}, \{11, 0.1\}, \{12, 0\}\}$$

$$A \cap B = \{\{1, 0\}, \{2, 0\}, \{3, 0\}, \{4, 0.2\}, \{5, 0.4\}, \{6, 0.3\}, \{7, 0.2\}, \{8, 0.1\}, \{9, 0\}, \{10, 0\}, \{11, 0\}, \{12, 0\}\}$$

Cartesian Product

Cartesian Product of two Crisp sets

- Let A and B be crisp sets defined in the universe of discourse X and Y .
- The Cartesian product of A and B is denoted by $A \times B$
- It is denoted as : $A \times B = \{(a, b) \mid a \in A, b \in B\}$
- Note: Generally, $A \times B \neq B \times A$

Example

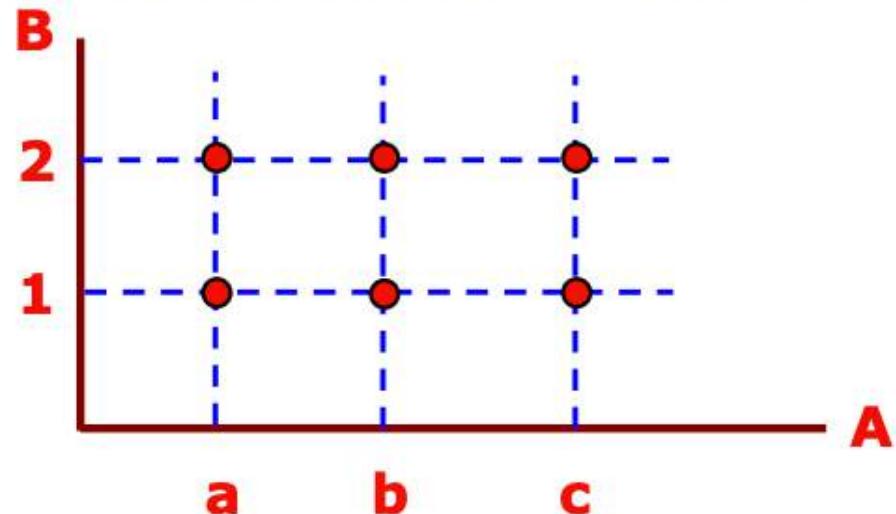
Let $A = \{a, b, c\}$ and $B = \{1, 2\}$, then

$$A \times B = \{(a, 1), (a, 2), (b, 1), (b, 2), (c, 1), (c, 2)\}$$

$$B \times A = \{(1, a), (1, b), (1, c), (2, a), (2, b), (2, c)\}$$

Note that $A \times B \neq B \times A$

Graphic representation of $A \times B$



Cartesian Product

Cartesian Product of two Fuzzy sets

- Let A and B be two fuzzy sets defined in the universe of discourse X and Y .
- The Cartesian product of A and B is denoted by $A \times B$
- It is defined as :

$$\mu_{A \times B}(x, y) = \min(\mu_A(x), \mu_B(y)), \quad \forall x \in X, y \in Y$$

Thus, the Cartesian product $A \times B$ is a fuzzy set of ordered pair (x, y) , $\forall x \in X, y \in Y$, with membership values of (x, y) , given by $\min(\mu_A(x), \mu_B(y))$.

In a sense Cartesian product of two Fuzzy sets is a *Fuzzy relation*.

Fuzzy Relations

- Fuzzy relations describe the degree of association of the elements.
- Example: “ x is approximately equal to y ”
- Fuzzy relations offer the capability to capture the uncertainty and vagueness in relations between sets and elements of a set.
- Fuzzy relations make the description of a concept possible.
- Fuzzy relations were introduced to supersede classical crisp relations.

Fuzzy Relations – Definition

- Fuzzy relation is a generalization of the definition of fuzzy set from 2-D space to 3-D space.

Fuzzy relation definition

- Consider a Cartesian product $A \times B = \{(x, y) \mid x \in A, y \in B\}$, where A and B are subsets of universal sets U_1 and U_2 .
- Fuzzy relation on $A \times B$ is denoted by R or $R(x, y)$ is defined as the set:

$$R = \{((x, y), \mu_R(x, y)) \mid (x, y) \in A \times B, \mu_R(x, y) \in [0, 1]\}$$

where $\mu_R(x, y)$ is a function in two variables called membership function.

Fuzzy Relations – Definition

- It gives the degree of membership of the ordered pair (x, y) in R associating with each pair (x, y) in $A \times B$, a real number in the interval $[0, 1]$.
- The degree of membership $\mu_R(x, y)$ indicates the degree to which x is in relation to y .

Note :

- Definition of fuzzy relation is a generalization of the definition of fuzzy set from the 2D space $(x, \mu_A(x, y))$ to 3-D space $((x, y), \mu_R(x, y))$.

Fuzzy Relations – Example

$$R = \left\{ \begin{array}{l} ((x_1, y_1), 0.0), ((x_1, y_2), 0.1), ((x_1, y_3), 0.2), \\ ((x_2, y_1), 0.7), ((x_2, y_2), 0.2), ((x_2, y_3), 0.3), \\ ((x_3, y_1), 1.0), ((x_3, y_2), 0.6), ((x_3, y_3), 0.2) \end{array} \right\}$$

- This relation can be written in matrix form as:

$$R \triangleq \begin{array}{|c|c|c|c|c|} \hline & y & y_1 & y_2 & y_3 \\ \hline x & & & & \\ \hline x_1 & 0.0 & 0.1 & 0.2 & \\ \hline x_2 & 0.7 & 0.2 & 0.3 & \\ \hline x_3 & 1.0 & 0.6 & 0.2 & \\ \hline \end{array}$$

This symbol \triangleq denotes the operation ‘*is defined as*’, and the values in the matrix are the values of the membership function.

Fuzzy Relations – Example

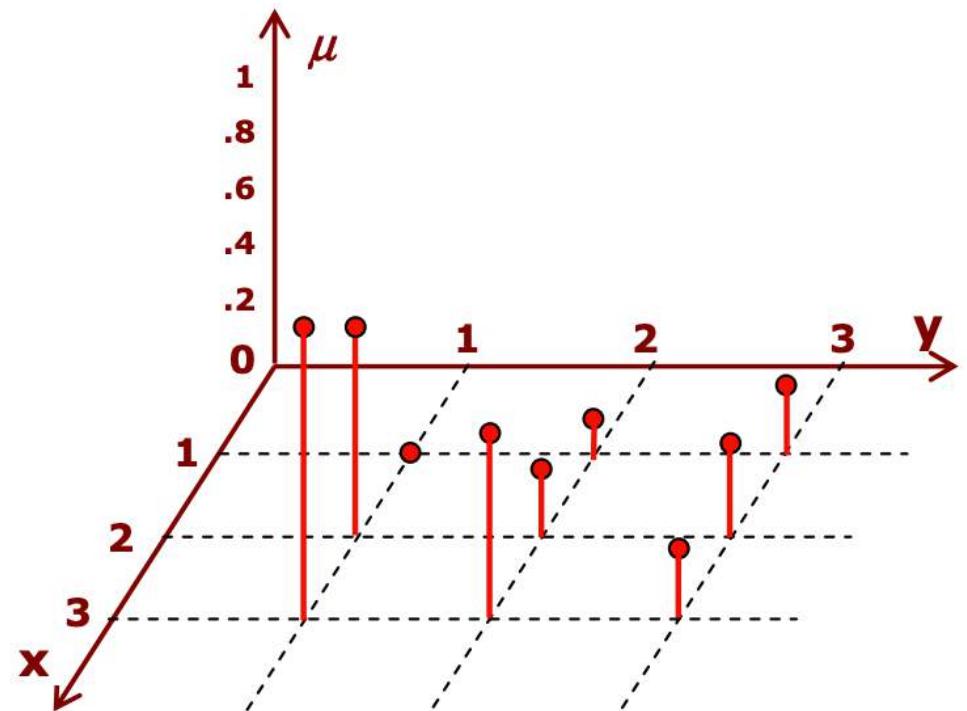
$$R \triangleq$$

x	y	y_1	y_2	y_3
x				
x_1	0.0	0.1	0.2	
x_2	0.7	0.2	0.3	
x_3	1.0	0.6	0.2	

Since the values of the membership function **(0.7, 1.0, 0.6)** are **in the direction of x** below the major diagonal **(0, 0.2, 0.2)** in the matrix are greater than those **(0.1, 0.2, 0.3)** in the direction of **y** , we therefore say that the relation R describes x is greater than y .

Assuming,

$x_1 = 1, x_2 = 2, x_3 = 3$ and $y_1 = 1, y_2 = 2, y_3 = 3$, the relation can be graphically represented by points in 3D space (X, Y, μ) as figure given below.



Projections of Fuzzy Relations – Definition

- Given a fuzzy relation on $A \times B$, R or $R(x, y)$, with membership value $\mu_R(x, y)$
- Formally $R = \{(x, y), \mu_R(x, y) \mid (x, y) \in A \times B, \mu_R(x, y) \in [0,1]\}$

Then the first, the second and the total projections of fuzzy relations are :

- First Projection of R:

$$R^{(1)} = \left\{ \left((x), \mu_{R^{(1)}}(x, y) \right) \right\} = \left\{ \left((x), \max_y \mu_R(x, y) \right) \mid (x, y) \in A \times B \right\}$$

- Second Projection of R:

$$R^{(2)} = \left\{ \left((y), \mu_{R^{(2)}}(x, y) \right) \right\} = \left\{ \left((y), \max_x \mu_R(x, y) \right) \mid (x, y) \in A \times B \right\}$$

- Total or Global Projection of R:

$$R^{(T)} = \max_x \max_y \{ \mu_R(x, y) \mid (x, y) \in A \times B \}$$

Note: $\max(\cdot)$ means max with respect to y while x is considered fixed

First Projection of Fuzzy Relations

- First Projection of R:

$$R^{(1)} = \left\{ \left((x), \mu_{R^{(1)}}(x, y) \right) \right\} = \left\{ \left((x), \max_y \mu_R(x, y) \right) \mid (x, y) \in A \times B \right\}$$

$R \triangleq$

x	y	y_1	y_2	y_3	$R^{(1)}$
x_1		0.0	0.1	0.2	0.2
x_2		0.7	0.2	0.3	0.7
x_3		1.0	0.6	0.2	1.0

$$R^{(1)} = \{(x_1, 0.2), (x_2, 0.7), (x_3, 1.0)\}$$

Second Projection of Fuzzy Relations

- Second Projection of R:

$$R^{(2)} = \left\{ \left((y), \mu_{R^{(2)}}(x, y) \right) \right\} = \left\{ \left((y), \max_x \mu_R(x, y) \right) \mid (x, y) \in A \times B \right\}$$

	y	y_1	y_2	y_3
x				
x_1	0.0	0.1	0.2	
x_2	0.7	0.2	0.3	
x_3	1.0	0.6	0.2	
$R^{(2)}$	1.0	0.6	0.3	

$$R^{(2)} = \{(y_1, 1.0), (y_2, 0.6), (y_3, 0.3)\}$$

Total Projection of Fuzzy Relations

- Total Projection of R:

$$R^{(T)} = \max_x \max_y \{ \mu_R(x, y) | (x, y) \in A \times B \}$$

$R \triangleq$

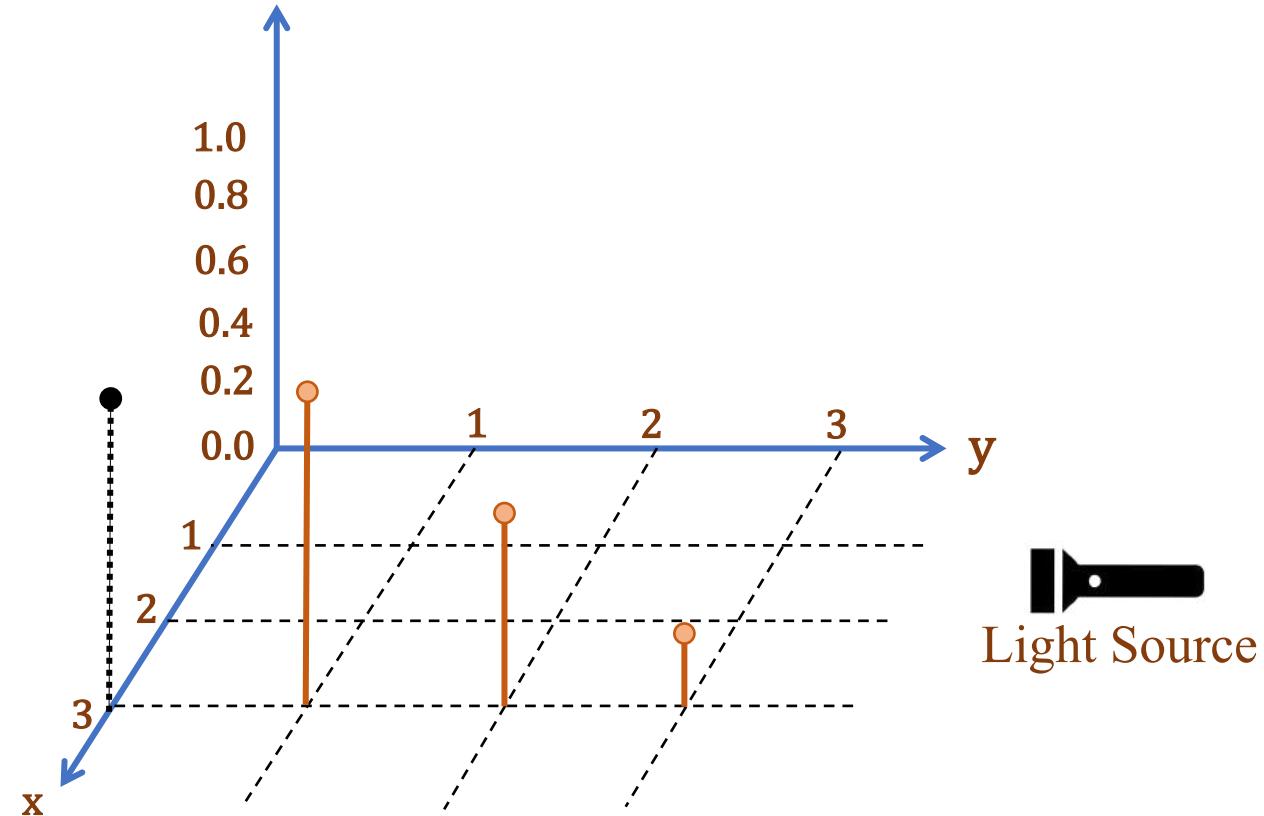
x	y	y_1	y_2	y_3	$R^{(1)}$
x_1		0.0	0.1	0.2	0.2
x_2		0.7	0.2	0.3	0.7
x_3		1.0	0.6	0.2	1.0
$R^{(2)}$		1.0	0.6	0.3	1.0

$$R^{(T)} = 1.0$$

- Note: Select max with respect to $R^{(1)}$ and $R^{(2)}$

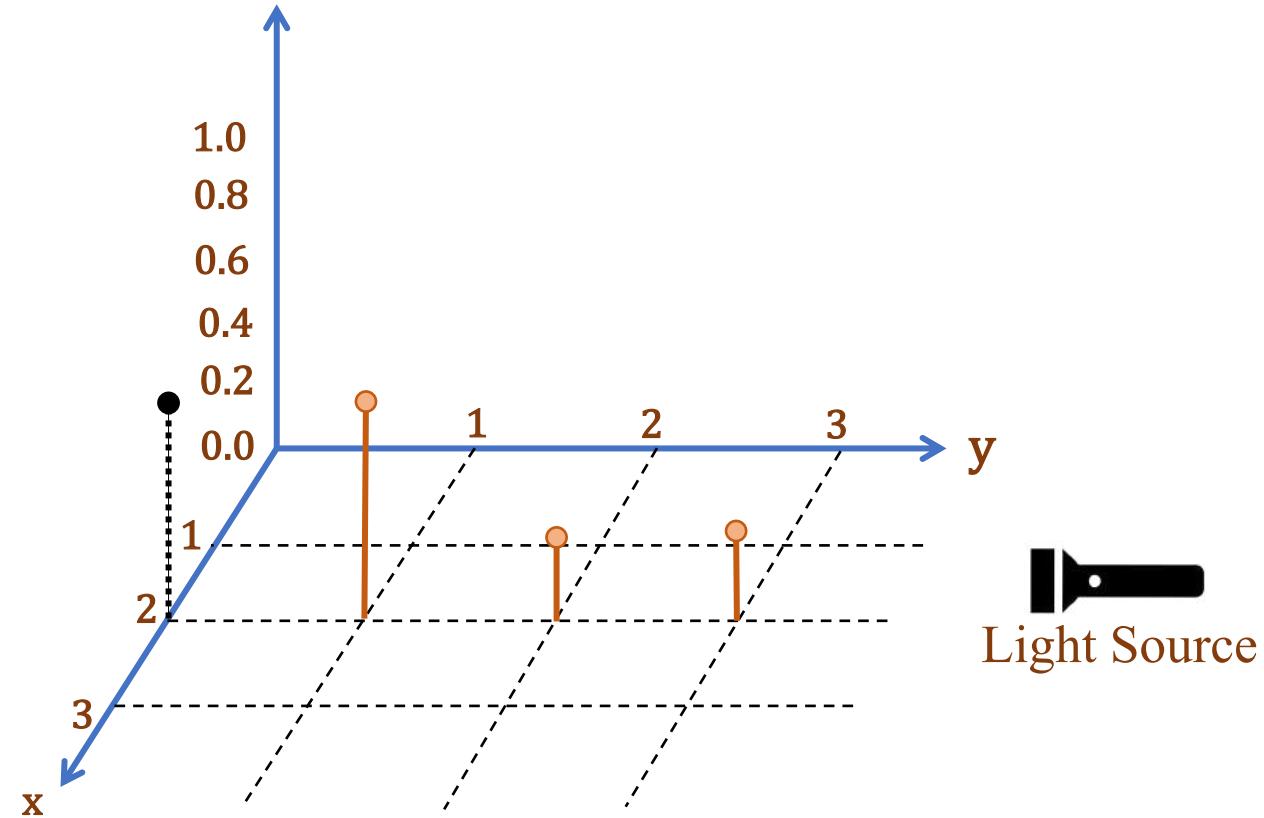
First Projection – Physical significance

	y	y_1	y_2	y_3	$R^{(1)}$
x					
x_1	0.0	0.1	0.2	0.2	0.2
x_2	0.7	0.2	0.3	0.3	0.7
x_3	1.0	0.6	0.2	0.2	1.0



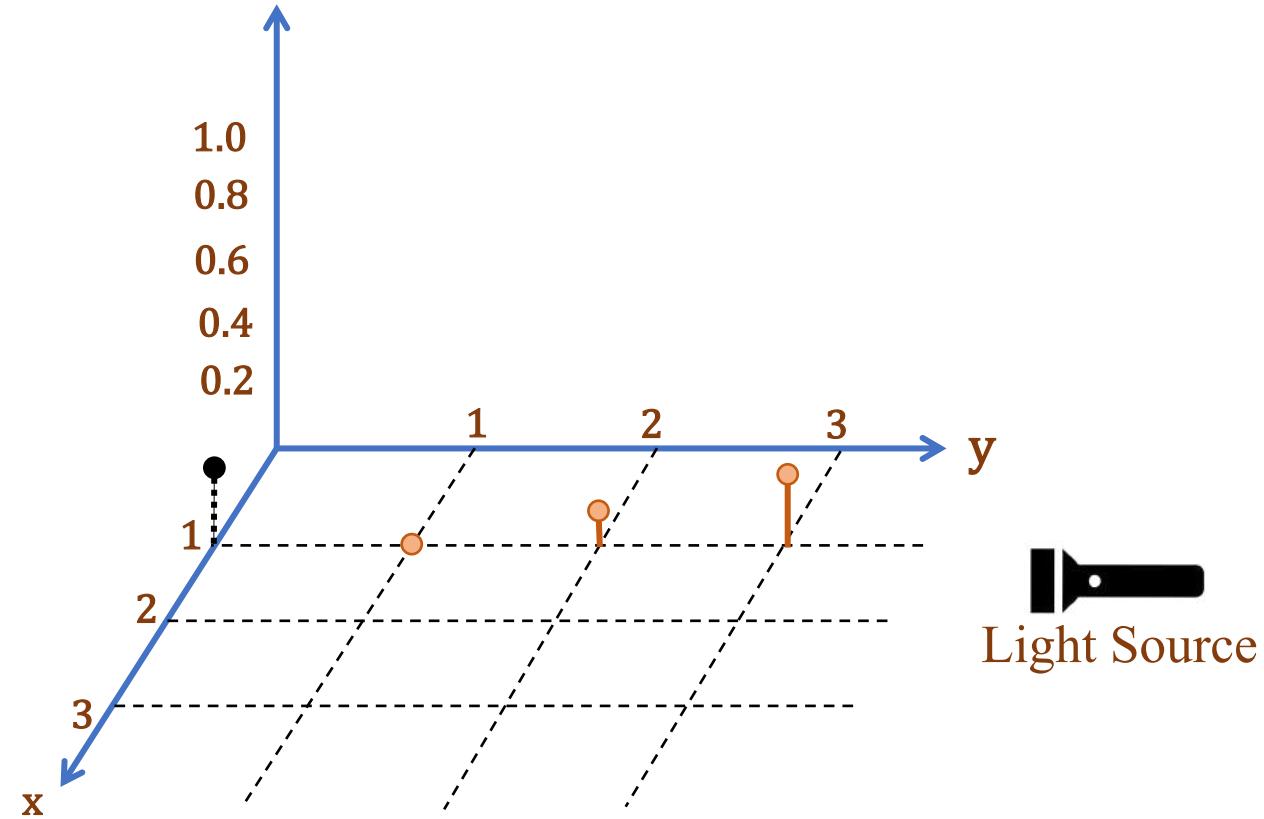
First Projection – Physical significance

	y	y_1	y_2	y_3	$R^{(1)}$
x					
x_1	0.0	0.1	0.2	0.2	0.2
x_2	0.7	0.2	0.3	0.3	0.7
x_3	1.0	0.6	0.2	0.2	1.0



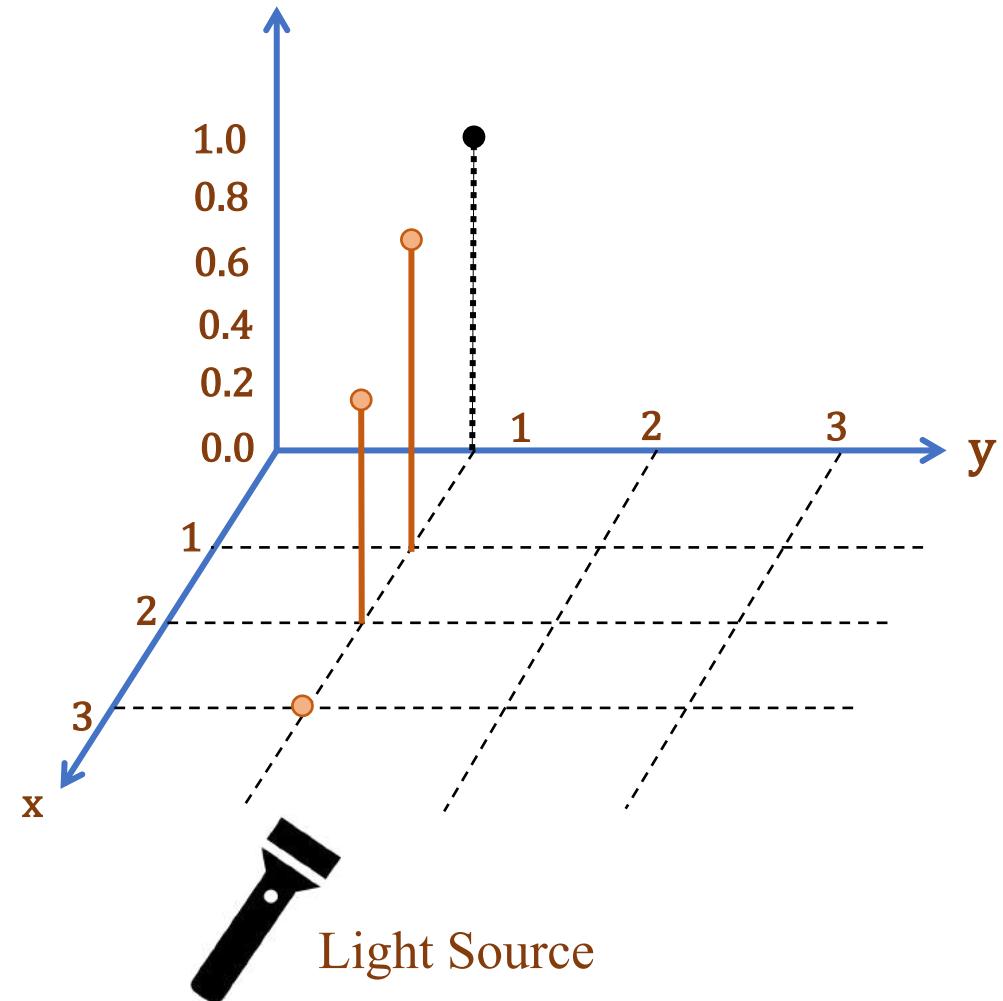
First Projection – Physical significance

	y	y_1	y_2	y_3	$R^{(1)}$
x					
x_1	0.0	0.1	0.2	0.2	0.2
x_2	0.7	0.2	0.3	0.3	0.7
x_3	1.0	0.6	0.2	0.2	1.0



Second Projection – Physical significance

x	y	y_1	y_2	y_3
x_1	0.0	0.1	0.2	
x_2	0.7	0.2	0.3	
x_3	1.0	0.6	0.2	
$R^{(2)}$	1.0	0.6	0.3	



Compositions of Fuzzy Relations

- Fuzzy relation in different product space can be combined with each other by the operation called “Composition”.
- There are many composition methods in use , e.g.,
 - max-product method
 - max-average method
 - max-min method
 - min-max method.
- But max-min and min-max composition methods are widely used in fuzzy logic applications.
- The selection of composition depends on the field of work. This will be out-of-scope for current syllabus.
- We will focus only two techniques: max-min and min-max composition methods.

Max-Min Composition

Let $R_1(x, y)$ and $R_2(y, z)$ be two relations, such that

$$R_1(x, y) = \left\{ \left((x, y), \mu_{R_1}(x, y) \right) \mid (x, y) \in A \times B \right\}$$

$$R_2(y, z) = \left\{ \left((y, z), \mu_{R_2}(y, z) \right) \mid (y, z) \in B \times C \right\}$$

- The max- min composition is then the fuzzy set, $R_1 \circ R_2$, defined as:

$$R_1 \circ R_2 = \left\{ \left((x, z), \max_y (\min \{ \mu_{R_1}(x, y), \mu_{R_2}(y, z) \}) \right) \mid x \in A, y \in B, z \in C \right\}$$

- $(x, z) \in A \times C$, hence the domain of $R_1 \circ R_2$ is $A \times C$

Max-Min Composition - Example

$$R_1 \circ R_2 = \left\{ \left((x, z), \max_y (\min \{ \mu_{R_1}(x, y), \mu_{R_2}(y, z) \}) \right) \mid x \in A, y \in A, z \in A \right\}$$

$$R_1 \triangleq \begin{array}{c|c|c|c|c} & & y & & \\ \hline & x & & & \\ \hline x_1 & & 0.1 & 0.3 & 0.0 \\ \hline x_2 & & 0.8 & 1.0 & 0.3 \end{array}$$

$$R_2 \triangleq \begin{array}{c|c|c|c|c} & & z & & \\ \hline & y & & & \\ \hline y_1 & & 0.8 & 0.2 & 0.0 \\ \hline y_2 & & 0.2 & 1.0 & 0.6 \\ \hline y_3 & & 0.5 & 0.0 & 0.4 \end{array}$$

Step 1: Compute $\min \{ \mu_{R_1}(x, y), \mu_{R_2}(y, z) \}$

Consider the row x_1 and the column z_1 . For $y_j, j = 1, 2, 3$ compute

$$\min \{ \mu_{R_1}(x, y_j), \mu_{R_2}(y_j, z) \}$$

Max-Min Composition - Steps

Step 1: Compute $\min\{\mu_{R_1}(x, y), \mu_{R_2}(y, z)\}$

Consider the row x_1 and the column z_1 .

For all $y_j j = 1, 2, 3$ compute $\min\{\mu_{R_1}(x_1, y_j), \mu_{R_2}(y_j, z_1)\}$.

- for $j = 1, \min\{\mu_{R_1}(x_1, y_1), \mu_{R_2}(y_1, z_1)\} = \min(0.1, 0.8) = 0.1$
- for $j = 2, \min\{\mu_{R_1}(x_1, y_2), \mu_{R_2}(y_2, z_1)\} = \min(0.3, 0.2) = 0.2$
- for $j = 3, \min\{\mu_{R_1}(x_1, y_3), \mu_{R_2}(y_3, z_1)\} = \min(0.0, 0.8) = 0.0$

x	y	y_1	y_2	y_3
x_1	0.1	0.3	0.0	
x_2	0.8	1.0	0.3	

y	z	z_1	z_2	z_3
y_1	0.8	0.2	0.0	
y_2	0.2	1.0	0.6	
y_3	0.5	0.0	0.4	

Max-Min Composition - Steps

Step 1: Compute $\min\{\mu_{R_1}(x_1, y_j), \mu_{R_2}(y_j, z_1)\}$ for all $y_j, j = 1, 2, 3$.

- for $j = 1, \min\{\mu_{R_1}(x_1, y_1), \mu_{R_2}(y_1, z_1)\} = \min(0.1, 0.8) = 0.1$
- for $j = 2, \min\{\mu_{R_1}(x_1, y_2), \mu_{R_2}(y_2, z_1)\} = \min(0.3, 0.2) = 0.2$
- for $j = 3, \min\{\mu_{R_1}(x_1, y_3), \mu_{R_2}(y_3, z_1)\} = \min(0.0, 0.8) = 0.0$

Step 2: To compute $\mu_{R_1 \circ R_2}(x_1, z_1)$ take **max** of the the above value.

$$\text{Hence, } \mu_{R_1 \circ R_2}(x_1, z_1) = \max \left(\begin{array}{l} \min\{\mu_{R_1}(x_1, y_1), \mu_{R_2}(y_1, z_1)\}, \\ \min\{\mu_{R_1}(x_1, y_2), \mu_{R_2}(y_2, z_1)\}, \\ \min\{\mu_{R_1}(x_1, y_3), \mu_{R_2}(y_3, z_1)\} \end{array} \right)$$

$$= \max(0.1, 0.2, 0.0) = 0.2$$

	y	y_1	y_2	y_3
x				
x_1	0.1	0.3	0.0	
x_2	0.8	1.0	0.3	

	z	z_1	z_2	z_3
y				
y_1	0.8	0.2	0.0	
y_2	0.2	1.0	0.6	
y_3	0.5	0.0	0.4	

Max-Min Composition - Steps

Now we have $\mu_{R_1 \circ R_2}(x_1, z_1) = 0.2$

Similarly, find all the membership values, which are:

$$\begin{aligned} & \mu_{R_1 \circ R_2}(x_1, z_1), \mu_{R_1 \circ R_2}(x_1, z_2), \mu_{R_1 \circ R_2}(x_1, z_3) \\ & \mu_{R_1 \circ R_2}(x_2, z_1), \mu_{R_1 \circ R_2}(x_2, z_2), \mu_{R_1 \circ R_2}(x_2, z_3) \end{aligned}$$

Fill the membership table with the following values

x	y	y_1	y_2	y_3
x_1		$\mu_{R_1 \circ R_2}(x_1, z_1)$	$\mu_{R_1 \circ R_2}(x_1, z_2)$	$\mu_{R_1 \circ R_2}(x_1, z_3)$
x_2		$\mu_{R_1 \circ R_2}(x_2, z_1)$	$\mu_{R_1 \circ R_2}(x_2, z_2)$	$\mu_{R_1 \circ R_2}(x_2, z_3)$

Max-Min Composition - Steps

$$R_1 \circ R_2 \triangleq$$

x	y	y_1	y_2	y_3
x_1	0.2	0.3	0.3	
x_2	0.8	1.0	0.6	

Min-Max Composition

Let $R_1(x, y)$ and $R_2(y, z)$ be two relations, such that

$$R_1(x, y) = \left\{ \left((x, y), \mu_{R_1}(x, y) \right) \mid (x, y) \in A \times B \right\}$$

$$R_2(y, z) = \left\{ \left((y, z), \mu_{R_2}(y, z) \right) \mid (y, z) \in B \times C \right\}$$

- The min-max composition is then the fuzzy set, $R_1 \circ R_2$, defined as:

$$R_1 \square R_2 = \left\{ \left((x, z), \min_y (\max \{ \mu_{R_1}(x, y), \mu_{R_2}(y, z) \}) \right) \mid x \in A, y \in B, z \in C \right\}$$

- $(x, z) \in A \times C$, hence the domain of $R_1 \square R_2$ is $A \times C$

Artificial Intelligence (IT38005)

Genetic Algorithms and Modeling

Anup Kumar Gupta

Genetic Algorithms

- Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics.
- GAs are inspired by Darwin's theory about evolution – “*survival of the fittest*”.
- GAs represent an intelligent exploitation of a random search used to solve optimization problems. GAs, although randomized, exploit historical information to direct the search into the region of better performance within the search space.
- In nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.

Solving Problems

- Solving problems mean looking for solutions, which is the best, among others.

Finding the solution to a problem is often thought:

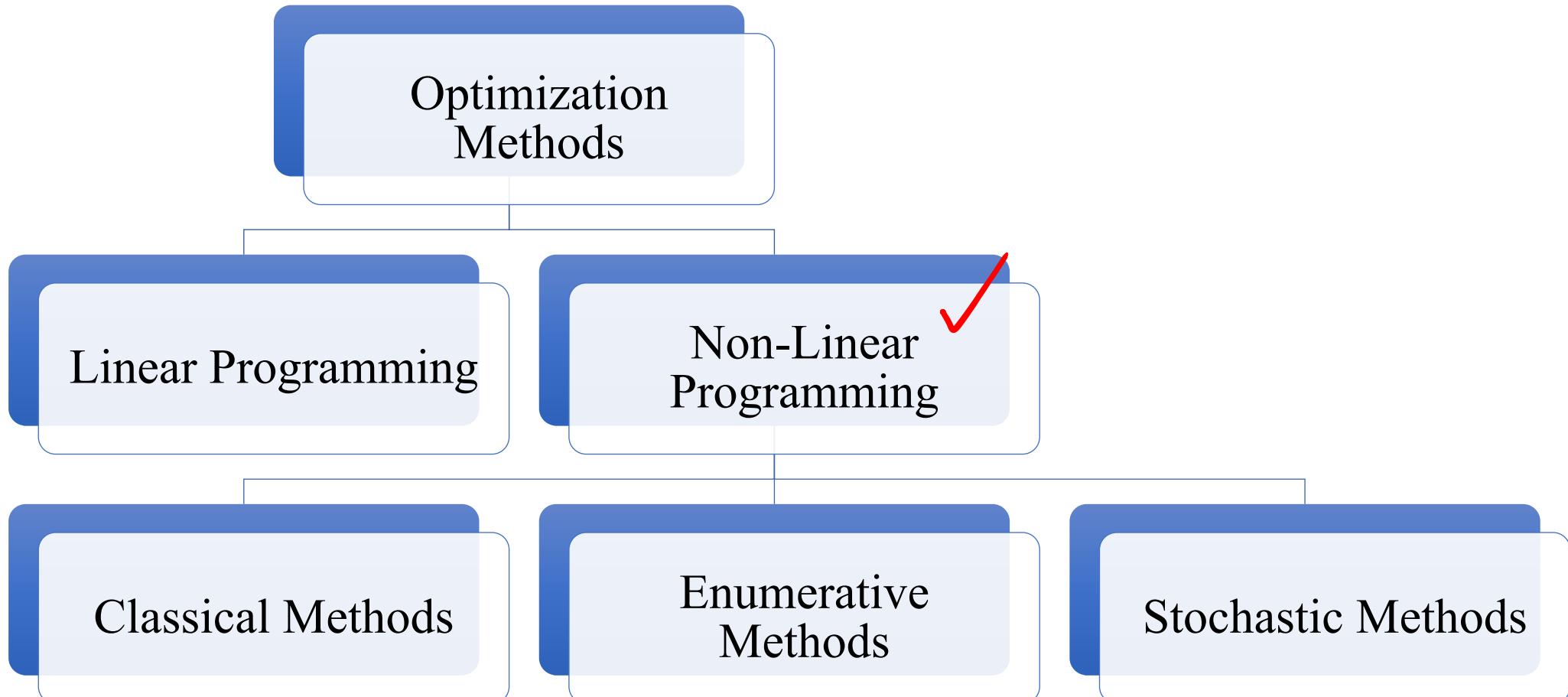
- Computer Science and AI:
 - Finding solution is a process of search through the space of possible solutions.
 - The set of possible solutions defines the search space (also called state space) for a given problem.
 - Solutions or partial solutions are viewed as points in the search space.
- Engineering and mathematics:
 - Finding solution is viewed as a process of optimization.
 - The problems are first formulated as mathematical models expressed in terms of functions.
 - To find a solution, discover the parameters that optimize the model or the function components that provide optimal system performance.

Optimization **

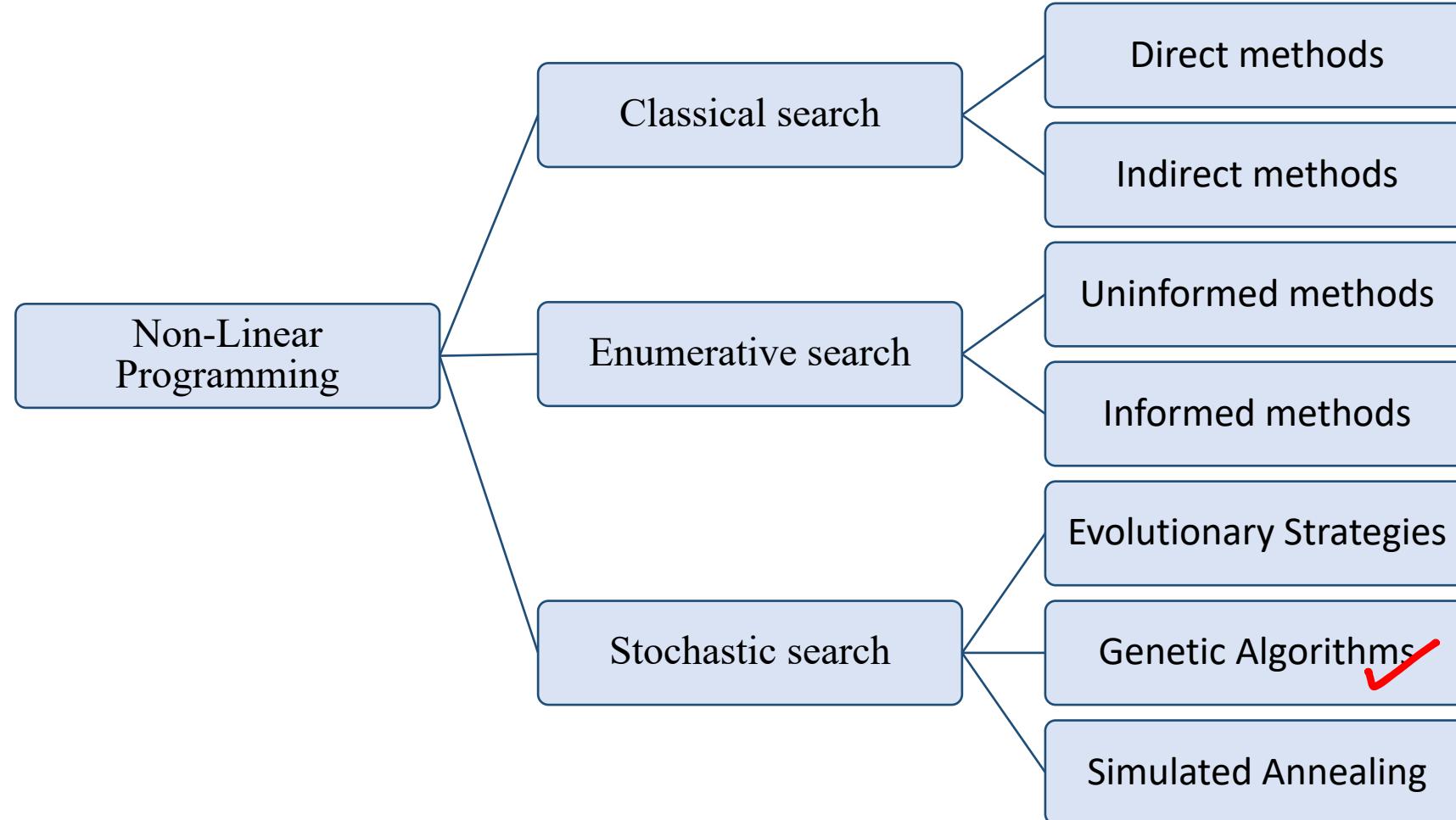
- Optimization is a process that finds a best, or optimal, solution for a problem.
- The optimization problems are centered around three factors :
 1. **An objective function:** which is to be minimized or maximized.
 - In manufacturing, we want to maximize the profit or minimize the cost .
 - In designing an automobile panel, we want to maximize the strength.
 2. **A set of unknowns or variables:** that affect the objective function.
 - In manufacturing, the variables are the amount of resources used or the total time spent.
 - In panel design problem, the variables are shape and dimensions of the panel.
 3. **A set of constraints:** that allow the unknowns to take on certain values but exclude others.
 - 1. In manufacturing, one constrain is, that all “*time*” variables to be non-negative.
 - 2 In the panel design, we want to limit the weight and put constraints on its shape.
- An optimization problem is defined as : Finding values of the variables that minimize or maximize the **objective function** while satisfying the **constraints**.

optimize

Optimization Methods



Non-Linear Programming



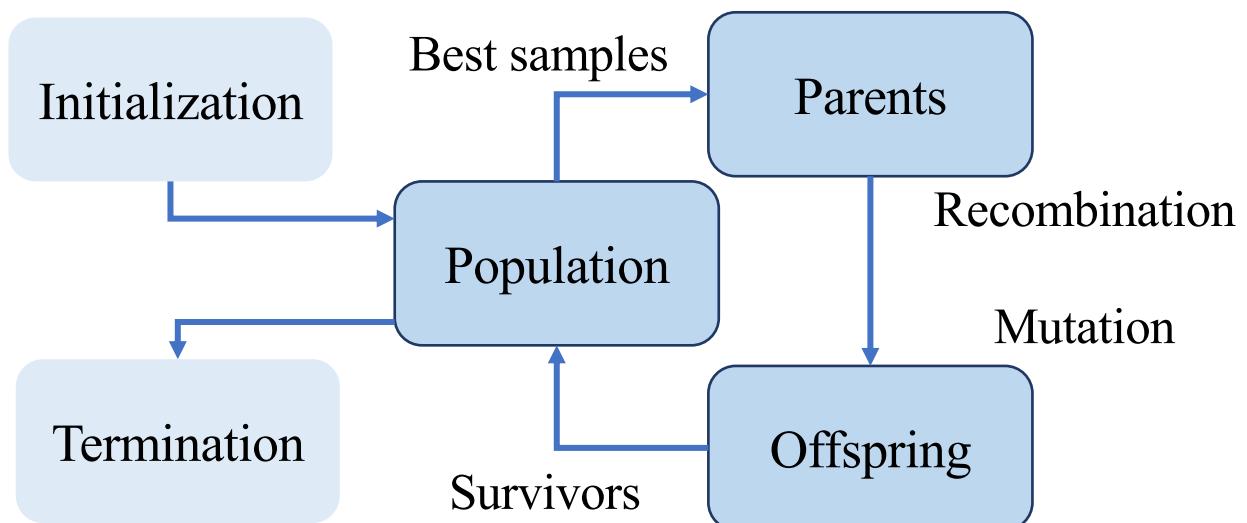
Genetic Algorithms – Basic Concepts

- Genetic algorithms (GAs) are the main paradigm of evolutionary computing.
- GAs are inspired by Darwin's theory about evolution – the “*survival of the fittest*”.
- In nature, competition among individuals for scanty resources results in the fittest individuals dominating over the weaker ones.
- GAs are the ways of solving problems by mimicking processes nature uses; i.e., **Selection**, **Crossover**, **Mutation** and **Accepting**, to evolve a solution to a problem.
- The biological background (basic genetics), the scheme of evolutionary processes, the working principles and the steps involved in GAs are illustrated in next slides.

Biological Background – Basic Genetics

- Every organism has a set of rules, describing how that organism is built. All living organisms consist of cells.
- In each cell there is same set of **chromosomes**. Chromosomes are strings of DNA and serve as a model for the whole organism.
- A chromosome consists of **genes**, blocks of DNA.
- Each gene encodes a particular protein that represents a **trait** (feature), e.g., color of eyes.
- Complete set of genetic material (all chromosomes) is called a **genome**.
- When two organisms mate, they share their genes; the resultant offspring may end up having half the genes from one parent and half from the other. This process is called **recombination (cross over)** .
- The new created offspring can then be **mutated**. Mutation means, that the elements of DNA are a bit changed. These changes are mainly caused by **errors in copying genes** from parents.
- The **fitness** of an organism is measured by success of the organism in its life (**survival**).

Biological Background – Basic Genetics



Pseudo Code

Begin

Initialize population with random candidate solution.
Evaluate each candidate;

Repeat until <**termination condition**>

- 1.** **Select** parents;
- 2.** **Recombine** pairs of parents;
- 3.** **Mutate** the resulting offspring;
- 4.** **Select** individuals or the next generation;

End

Search Space

In solving problems, some solution will be the best, among others. The space of all feasible solutions (among which the desired solution resides) is called **search space** (also called **state space**).

- Each point in the search space represents one **possible solution**.
- Each possible solution can be “marked” by its value (or **fitness**) for the problem.
- The GA looks for the **best** solution among several possible solutions.
- Looking for a solution is then equal to looking for some extreme value (minimum or maximum) in the search space.
- At times, the search space may be well defined, but usually only a few points in the search space are known.

In using GA, the process of finding solutions generates other points (possible solutions) as evolution proceeds.

Analogy to Biological Terms

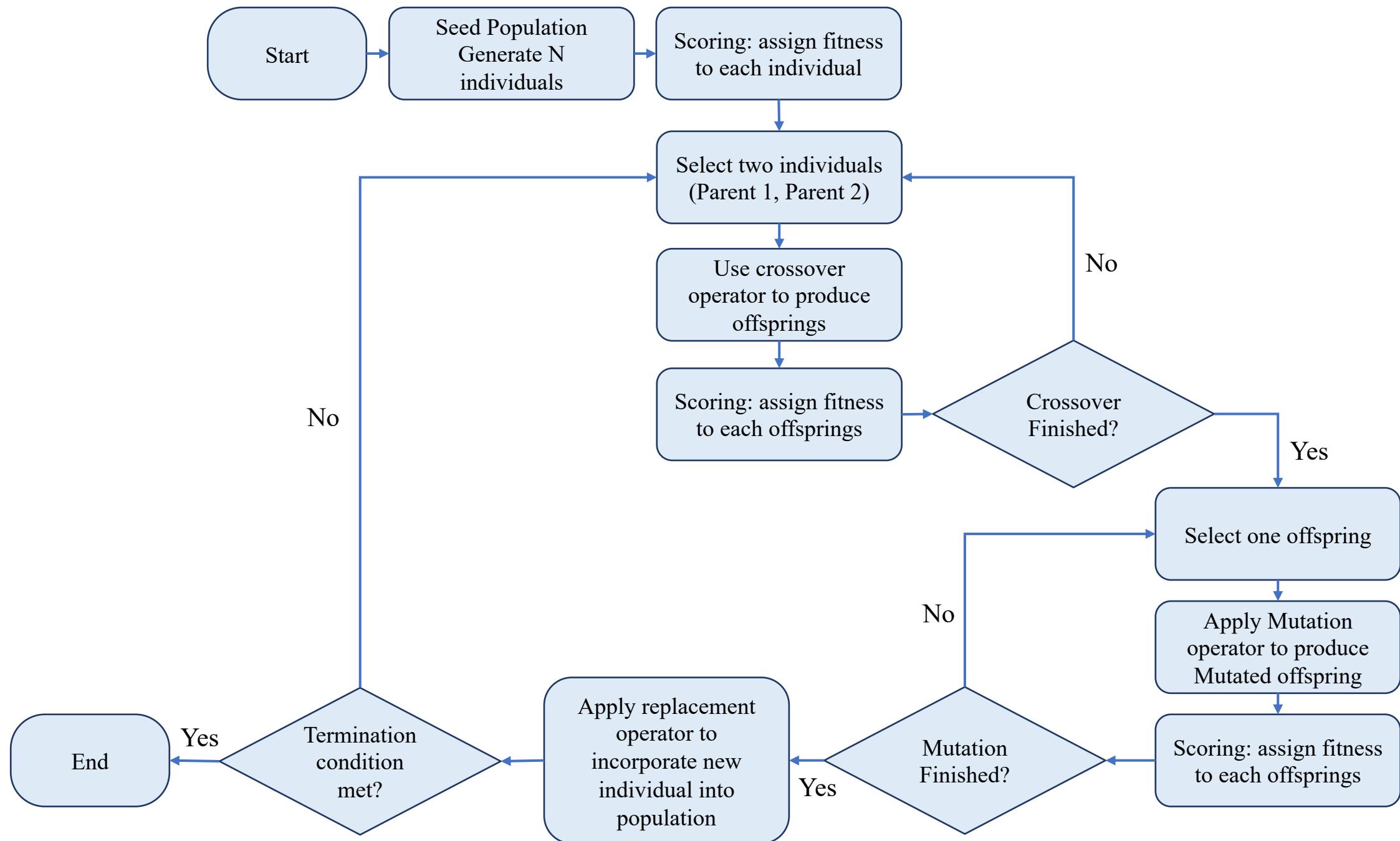
- **Chromosome:** a set of genes; a chromosome contains the solution in form of genes.
- **Gene :** a part of chromosome; a gene contains a part of solution. It determines the solution. e.g., 16743 is a chromosome and 1, 6, 7, 4 and 3 are its genes.
- **Individual:** same as chromosome.
- **Population:** number of individuals present with same length of chromosome.
- **Fitness:** the value assigned to an individual based on how far or close an individual is from the solution; greater the fitness value better the solution it contains.
- **Fitness function:** a function that assigns fitness value to the individual. It is problem specific.
- **Breeding:** taking two fit individuals and then intermingling their chromosome to create new two individuals.
- **Mutation:** changing a random gene in an individual.
- **Selection:** selecting individuals for creating the next generation.

Working principles

- Genetic algorithm begins with a set of solutions (represented as chromosomes). This set of solutions is called the population.
- Solutions from one population are taken and used to form a new population. This is motivated by the possibility that the new population will be better than the old one.
- Solutions are selected according to their fitness to form new solutions (offspring); more suitable they are, more chances they have to reproduce.
- This is repeated until some condition is satisfied. The termination condition can be for example: number of populations or improvement of the best solution (number of iterations).

Outline of basic Genetic Algorithms

(Explained in the next slides)



1. [Start] Generate random population of n chromosomes (suitable solutions for the problem).
2. [Fitness] Evaluate the fitness $f(x)$ of each chromosome x in the population.
3. [New population] Create a new population by repeating following steps until the new population is complete.
 - a) [Selection] Select two parent chromosomes from a population according to their fitness (better the fitness, bigger the chance to be selected)
 - b) [Crossover] With a crossover probability, cross over the parents to form new offspring (children). If no crossover was performed, offspring is the exact copy of parents.
 - c) [Mutation] With a mutation probability, mutate new offspring at each locus (position in chromosome).
 - d) [Accepting] Place new offspring in the new population
4. [Replace] Use new generated population for a further run of the algorithm
5. [Test] If the end condition is satisfied, stop, and return the best solution in current population
6. [Loop] Go to step 2

Note: The genetic algorithm's performance is largely influenced by two operators called **crossover** and **mutation**. These two operators are the most important parts of GA.

Encoding

- Before a genetic algorithm can be put to work on any problem, a method is needed to encode potential solutions to that problem in a form so that a computer can process.
- One common approach is to encode solutions as binary strings: sequences of 1's and 0's, where the digit at each position represents the value of some aspect of the solution.

Example :

- In binary form a gene looks like: (11100010)
- A chromosome is an array of genes. A chromosome looks like:

Gene 1	Gene 2	Gene 3	Gene 4
11000010	00001110	001111010	10100011

Encoding

- A chromosome should in some way contain information about solution which it represents; it thus requires encoding. The most popular way of encoding is a binary string like :

Chromosome 1 : 1101100100110110

Chromosome 2 : 1101111000011110

- Each bit in the string represent some characteristics of the solution.
- There are many other ways of encoding, e.g., encoding values as integer or real numbers or some permutations and so on.
- The virtue of these encoding method depends on the problem to work on.

Binary Encoding

- Binary encoding is the most common to represent information contained. In genetic algorithms, it was first used because of its relative simplicity.
- In binary encoding, every chromosome is a string of bits : 0 or 1, like

Chromosome 1: 1 0 1 1 0 0 1 0 1 1 0 0 1 0 1 0 1 1 1 0 0 1 0 1

Chromosome 2: 1 1 1 1 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1 1

- Binary encoding gives many possible chromosomes even with a smaller possible settings for a trait (feature).
- This encoding is often not natural for many problems and sometimes corrections must be made after crossover and/or mutation.

Binary Encoding - Examples

Example 1:

- One variable function, say 0 to 15 numbers, numeric values, represented by 4-bit binary string.

Numeric value	4-bit string						
0	0000	4	0100	8	1000	12	1100
1	0001	5	0101	9	1001	13	1101
2	0010	6	0110	10	1010	14	1110
3	0011	7	0111	11	1011	15	1111

Value Encoding

- The value encoding can be used in problems where values such as real numbers are used.
- Use of binary encoding for this type of problems would be difficult.
- In value encoding, every chromosome is a sequence of some values.
- The values can be anything connected to the problem, such as: real numbers, characters, or objects.

Examples:

Chromosome A 1.2324 5.3243 0.4556 2.3293 2.4545

Chromosome B ABDJE IFJDH DIERJ FDLDF LFEGT

Chromosome C (back), (back), (right), (forward), (left)

- Value encoding is often necessary to develop some new types of crossovers and mutations specific for the problem.

Permutation Encoding

- Permutation encoding can be used in ordering problems, such as traveling salesman problem or task ordering problem.
- In permutation encoding, every chromosome is a string of numbers that represent a position in a sequence.

Examples:

Chromosome A 1 5 3 2 4 8 6 7 9

Chromosome B 7 6 3 4 1 9 8 2 4

- Permutation encoding is useful for ordering problems. For some problems, crossover and mutation corrections must be made to leave the chromosome consistent.

Permutation Encoding – Problems related to Sequences

- Examples:

The Traveling Salesman problem:

- There are cities and given distances between them. Traveling salesman must visit all of them, but he does not want to travel more than necessary. Find a sequence of cities with a minimal traveled distance. Here, encoded chromosomes describe the order of cities the salesman visits.

The Eight Queens problem:

- There are eight queens. Find a way to place them on a chess board so that no two queens attack each other. Here, encoding describes the position of a queen on each row.

Tree Encoding

- Tree encoding is used mainly for evolving programs or expressions.

For genetic programming:

- In tree encoding, every chromosome is a tree of some objects, such as functions or commands in programming language.
- Tree encoding is useful for evolving programs or any other structures that can be encoded in trees.
- The crossover and mutation can be done relatively easy way.

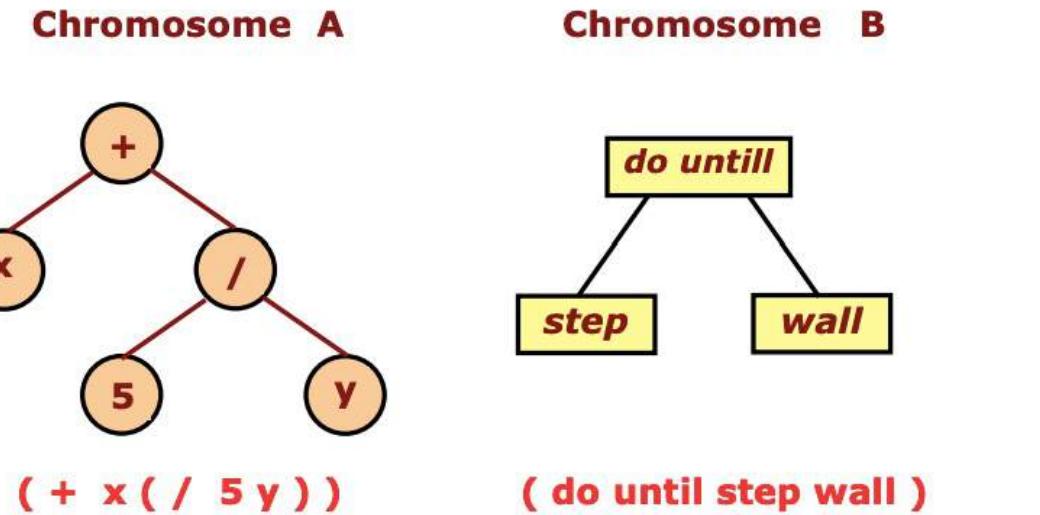


Fig. Example of Chromosomes with tree encoding

Note: Tree encoding is good for evolving programs. The programming language LISP is often used. Programs in LISP can be easily parsed as a tree, so the crossover and mutation are relatively easy.

Operations on Genetic Algorithm

- Genetic operators used in genetic algorithms maintain genetic diversity.
- Genetic diversity or variation is a necessity for the process of evolution.
- Genetic operators are analogous to those which occur in the natural world:
 - Reproduction (or Selection)
 - Crossover (or Recombination)
 - Mutation.

Parameters of Genetic Algorithm

- In addition to these operators, there are some parameters of GA. One important parameter is Population size.
- Population size says how many chromosomes are in population (in one generation).
- If there are only few chromosomes, then GA would have a few possibilities to perform crossover and only a small part of search space is explored.
- If there are many chromosomes, then GA slows down.
- Research shows that after some limit, it is not useful to increase population size, because it does not help in solving the problem faster. The population size depends on the type of encoding and the problem.

Reproduction or Selection

- Reproduction is usually the first operator applied on population.
- From the population, the chromosomes are selected to be parents to crossover and produce offspring.

The problem is how to select these chromosomes?

- According to Darwin's evolution theory 'survival of the fittest' – the best ones should survive and create new offspring.
- In selection we extract a subset of genes from an existing population, according to any definition of quality.
- Every gene has a meaning, so one can derive from the gene a kind of quality measurement called fitness function. Following this quality (fitness value), selection can be performed.
- Fitness function quantifies the optimality of a solution (chromosome) so that a particular solution may be ranked against all the other solutions. The function depicts the closeness of a given 'solution' to the desired result.

Roulette-wheel selection (Fitness-Proportionate Selection)

- Roulette-wheel selection, also known as Fitness Proportionate Selection, is a genetic operator, used for selecting potentially useful solutions for recombination.

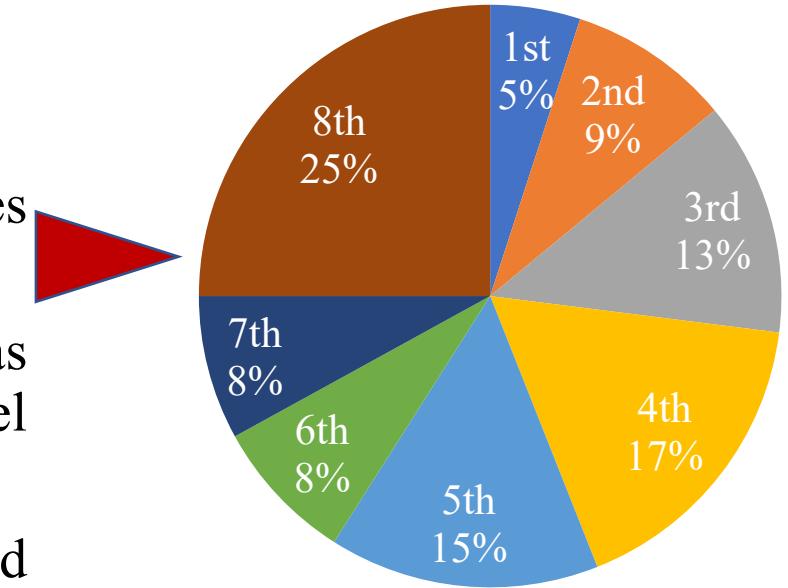
In fitness-proportionate selection:

- the chance of an individual's being selected is proportional to its fitness, greater or less than its competitors' fitness.
- conceptually, this can be thought as a game of Roulette.



Roulette-wheel selection – Example

- The Roulette-wheel simulates 8 individuals with fitness values F_i , marked at its circumference.
- In a roulette wheel selection, the circular wheel is divided as described before. A fixed point is chosen on the wheel circumference as shown and the wheel is rotated.
- The region of the wheel which comes in front of the fixed point is chosen as the parent. For the second parent, the same process is repeated.
- For instance, the 8^{th} individual has a higher fitness than others, so the wheel would choose the 8^{th} individual more than other individuals.
- A fitter individual has a greater pie on the wheel and therefore a greater chance of landing in front of the fixed point when the wheel is rotated. Therefore, the probability of choosing an individual depends directly on its fitness.



Roulette-wheel selection (Fitness-Proportionate Selection)

- Probability of i^{th} string is $p_i = \frac{F_i}{\sum_{j=1}^N F_j}$, where
- n = number of individuals which is also called population size;
- F_i = fitness of i^{th} string in the population.
- Because the circumference of the wheel is marked according to a string's fitness, the Roulette-wheel mechanism is expected to make $\frac{F_i}{\bar{F}}$ copies of the i^{th} string.
- Average fitness $\bar{F} = \frac{\sum_{j=1}^N F_j}{n}$
- Expected count $E[i] = N \times p_i$

Example of Selection

The aim of this evolutionary algorithms is to maximize the function $f(x) = x^2$ with x in the integer interval $[0, 31]$, i.e., $x = 0, 1, 2, \dots, 31$.

1. The first step is encoding of chromosomes; use binary representation for integers; 5-bits are used to represent integers up to 31.
2. Assume that the population size is 4.
3. Generate initial population at random. They are chromosomes or genotypes, e.g., 01101, 11000, 01000, 10011
4. Calculate fitness value for everyone:
 - a) Decode the individual into an integer (called phenotypes)
 $01101 \rightarrow 13, 11000 \rightarrow 24, 01000 \rightarrow 8, 10011 \rightarrow 19$
 - b) Evaluate the fitness according to $f(x) = x^2$,
 $13 \rightarrow 169, 24 \rightarrow 576, 8 \rightarrow 64, 19 \rightarrow 361$

Example of Selection

5. Select parents (two individuals) for crossover based on their fitness in p_i . We use roulette-wheel selection algorithm.

String #	Initial Population	x value	Fitness F_i , $f(x) = x^2$	p_i	Expected count $N \times p_i$
1	01101				
2	11000				
3	01000				
4	10011				
Sum					
Average					
Max (selected)					

Example of Selection

String #	Initial Population	x value	Fitness F_i , $f(x) = x^2$	p_i	Expected count $N \times p_i$
1	01101	13			
2	11000	24			
3	01000	8			
4	10011	19			
Sum					
Average					
Max (selected)					

Example of Selection

String #	Initial Population	x value	Fitness F_i , $f(x) = x^2$	p_i	Expected count $N \times p_i$
1	01101	13	169		
2	11000	24	576		
3	01000	8	64		
4	10011	19	361		
Sum					
Average					
Max (selected)					

Example of Selection

String #	Initial Population	x value	Fitness F_i , $f(x) = x^2$	p_i	Expected count $N \times p_i$
1	01101	13	169		
2	11000	24	576		
3	01000	8	64		
4	10011	19	361		
Sum			1170		
Average			293		
Max (selected)			576		

Example of Selection

String #	Initial Population	x value	Fitness F_i , $f(x) = x^2$	p_i	Expected count $N \times p_i$
1	01101	13	169	0.14	
2	11000	24	576	0.49	
3	01000	8	64	0.06	
4	10011	19	361	0.31	
Sum			1170		
Average			293		
Max (selected)			576		

Example of Selection

String #	Initial Population	x value	Fitness F_i , $f(x) = x^2$	p_i	Expected count $N \times p_i$
1	01101	13	169	0.14	0.56
2	11000	24	576	0.49	1.96
3	01000	8	64	0.06	0.24
4	10011	19	361	0.31	1.24
Sum			1170		
Average			293		
Max (selected)			576		

Example of Selection

The second string has maximum probability

String #	Initial Population	x value	Fitness F_i , $f(x) = x^2$	p_i	Expected count $N \times p_i$
1	01101	13	169	0.14	0.56
2	11000	24	576	0.49	1.96
3	01000	8	64	0.06	0.24
4	10011	19	361	0.31	1.24
Sum			1170		
Average			293		
Max (selected)			576		

Example of Selection

The second string has maximum chances of selection.

String #	Initial Population	x value	Fitness F_i , $f(x) = x^2$	p_i	Expected count $N \times p_i$
1	01101	13	169	0.14	0.56
2	11000	24	576	0.49	1.96
3	01000	8	64	0.06	0.24
4	10011	19	361	0.31	1.24
Sum			1170		
Average			293		
Max (selected)			576		

Crossover

- Crossover is a genetic operator that combines (mates) two chromosomes (parents) to produce a new chromosome (offspring).
- The idea behind crossover is that the new chromosome may be better than both parents if it takes the best characteristics from each of the parents.
- Crossover occurs during evolution according to a user-definable crossover probability.
- Crossover selects genes from parent chromosomes and creates a new offspring.
- Some of the crossover operators are:
 - one-point, two-point, uniform, arithmetic, and heuristic crossovers.
- The operators are selected based on the way chromosomes are encoded.

One-Point Crossover

- One-Point crossover operator randomly selects one crossover point and then copy everything before this point from the first parent and then everything after the crossover point copy from the second parent. The crossover would then look as shown below.

- Consider the two parents selected for crossover:

Parent 1: 1011010100111010

Parent 2: 1101001001110101

- We choose a crossover point.

Parent 1: 1011|010100111010

Parent 2: 1101|001001110101

- Interchange the parents' chromosomes after the crossover points. The offspring thus produced are:

Offspring 1: 1011|001001110101

Offspring 2: 1101|010100111010

Two-Point Crossover

- Two-Point crossover operator randomly selects two crossover points within a chromosome then interchanges the two parent chromosomes between these points to produce two new offspring. The crossover would then look as shown below.

- Consider the two parents selected for crossover:

Parent 1: 1011010100111010

Parent 2: 1101001001110101

- We choose two crossover points.

Parent 1: 1011|01010011|1010

Parent 2: 1101|00100111|0101

- Interchange the parents' chromosomes after the crossover points. The offspring thus produced are:

Offspring 1: 1011|00100111|1010

Offspring 2: 1101|01010011|0101

Uniform Crossover

- Uniform crossover operator decides (with some probability) which parent will contribute how the gene values in the offspring chromosomes. The crossover operator allows the parent chromosomes to be mixed at the gene level rather than the segment level (as with one- and two-point crossovers).
- Consider the two parents selected for crossover:

Parent 1: **1101100100110110**

Parent 2: **1101111000011110**

- If the mixing ratio is 0.5, then half of the genes in the offspring will come from parent 1 and the other half will come from parent 2. One of the possible set of offspring after uniform crossover would be:

Offspring 1: **1₁ 1₂ 0₂ 1₁ 1₁ 1₂ 1₂ 0₂ 0₁ 0₁ 0₂ 1₁ 1₂ 1₁ 1₁ 0₂**

Offspring 2: **1₂ 1₁ 0₁ 1₂ 1₂ 0₁ 0₁ 1₁ 0₂ 0₂ 1₁ 1₂ 0₁ 1₂ 1₂ 0₁**

- Note: The subscripts indicate which parent the gene came from.

Arithmetic Crossover

- Arithmetic crossover operator linearly combines two parent chromosome vectors to produce two new offspring according to the following equations, where a is a weight parameter.

$$\begin{aligned}\text{Offspring}_1 &= a \times \text{Parent}_1 + (1 - a) \times \text{Parent}_2 \\ \text{Offspring}_2 &= (1 - a) \times \text{Parent}_1 + a \times \text{Parent}_2\end{aligned}$$

- Consider the two parents selected for crossover:

Parent 1: (0.3) (0.4) (0.2) (7.4)

Parent 2: (0.5) (4.5) (0.1) (5.6)

- If the weight parameter is 0.7, then the offspring after arithmetic crossover would be:

Offspring 1: (0.360) (2.330) (0.170) (6.87)

Offspring 2: (0.402) (2.981) (0.149) (5.842)

Mutation

- After a crossover is performed, mutation takes place.
- Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of chromosomes to the next.
- Mutation occurs during evolution according to a user-definable mutation probability, usually set to low value, say 0.01 a good first choice.
- Mutation alters one or more gene values in a chromosome from its initial state.
- This can result in entirely new gene values being added to the gene pool. With the new gene values, the genetic algorithm may be able to arrive at better solution than was previously possible.

Mutation

- Mutation is an important part of the genetic search, helps to prevent the population from stagnating at any local optima.
- Mutation is intended to prevent the search falling into a local optimum of the state space.
- The Mutation operators are of many types: Flip Bit, Boundary, Non-Uniform, Uniform, and Gaussian.
- The operators are selected based on the way chromosomes are encoded.

Flip Bit

- The mutation operator simply inverts the value of the chosen gene, that is 0 is changed to 1, and 1 is changed to 0.
- This mutation operator can only be used for binary genes.
- Consider the two original offspring selected for mutation.

Offspring 1: 1011001001111010

Offspring 2: 1101010100110101

- Randomly pick the bits to be mutated.

Offspring 1: 1011001001111010

Offspring 2: 1101010100110101

- Invert the selected bits.

Mutated offspring 1: 1011011001101010

Mutated offspring 2: 1100010110110101

Mutation Techniques for Integer and Float genes

Boundary Mutation

- The mutation operator replaces the value of the chosen gene with either the upper or lower bound for that gene (chosen randomly).

Non-uniform Mutation

- The mutation operator increases the probability such that the amount of the mutation will be close to 0 as the generation number increases.
- This mutation operator prevents the population from stagnating in the early stages of the evolution then allows the genetic algorithm to fine tune the solution in the later stages of evolution.

Uniform Mutation

- The mutation operator replaces the value of the chosen gene with a uniform random value selected between the user-specified upper and lower bounds for that gene.

Gaussian Mutation

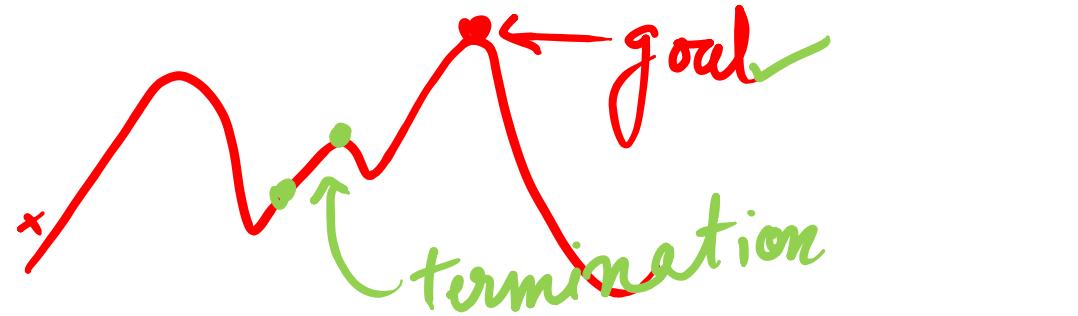
- The mutation operator adds a unit Gaussian distributed random value to the chosen gene. The new gene value is clipped if it falls outside of the user-specified lower or upper bounds for that gene.

Artificial Intelligence (IT38005)

Hill Climbing

Anup Kumar Gupta

Hill Climbing Algorithm



- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem.
- It terminates when it reaches a peak value where no neighbour has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems.
- It is also called greedy local search as it only looks to its good immediate neighbour state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

no previous states are required.

Features of Hill Climbing Algorithm

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

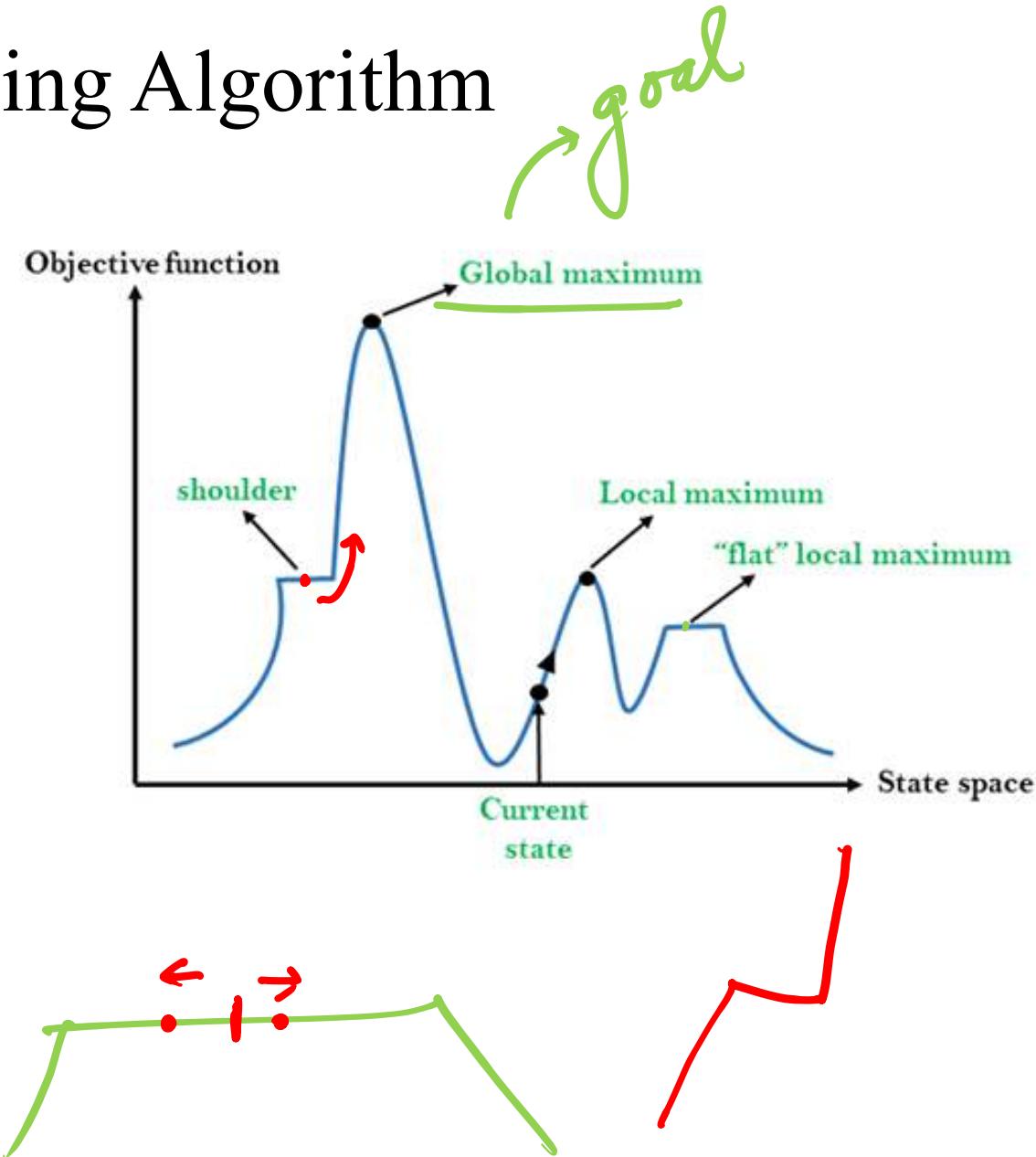




State-space Diagram for Climbing Algorithm

Hill

- **Local Maximum:** Local maximum is a state which is better than its neighbour states, but there may be another state which is higher than it.
- **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum:** It is a flat space in the landscape where all the neighbour states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.



Contact Details

- Anup Kumar Gupta
- PhD scholar, IIT Indore
- Email: msrphd2105101002@iiti.ac.in
- Website: <https://anupkumargupta.github.io/>

References

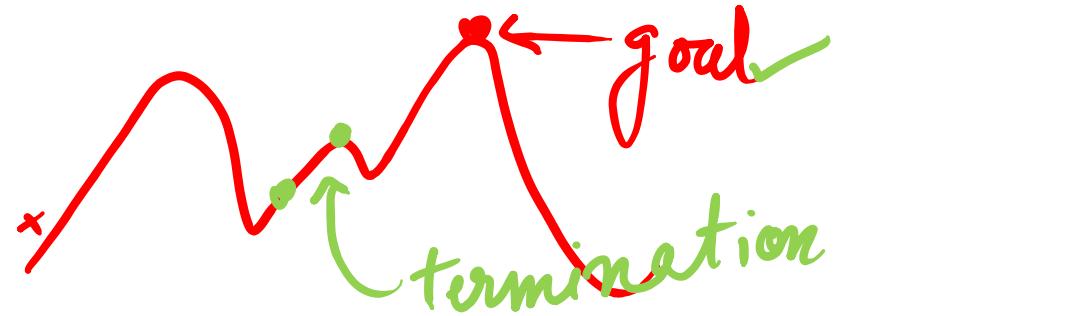
- <https://www.javatpoint.com/hill-climbing-algorithm-in-ai>

Artificial Intelligence (IT38005)

Hill Climbing

Anup Kumar Gupta

Hill Climbing Algorithm



- Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem.
- It terminates when it reaches a peak value where no neighbour has a higher value.
- Hill climbing algorithm is a technique which is used for optimizing the mathematical problems.
- It is also called greedy local search as it only looks to its good immediate neighbour state and not beyond that.
- A node of hill climbing algorithm has two components which are state and value.
- Hill climbing is mostly used when a good heuristic is available.
- In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

no previous states are required.

Features of Hill Climbing Algorithm

- **Generate and Test variant:** Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- **Greedy approach:** Hill-climbing algorithm search moves in the direction which optimizes the cost.
- **No backtracking:** It does not backtrack the search space, as it does not remember the previous states.

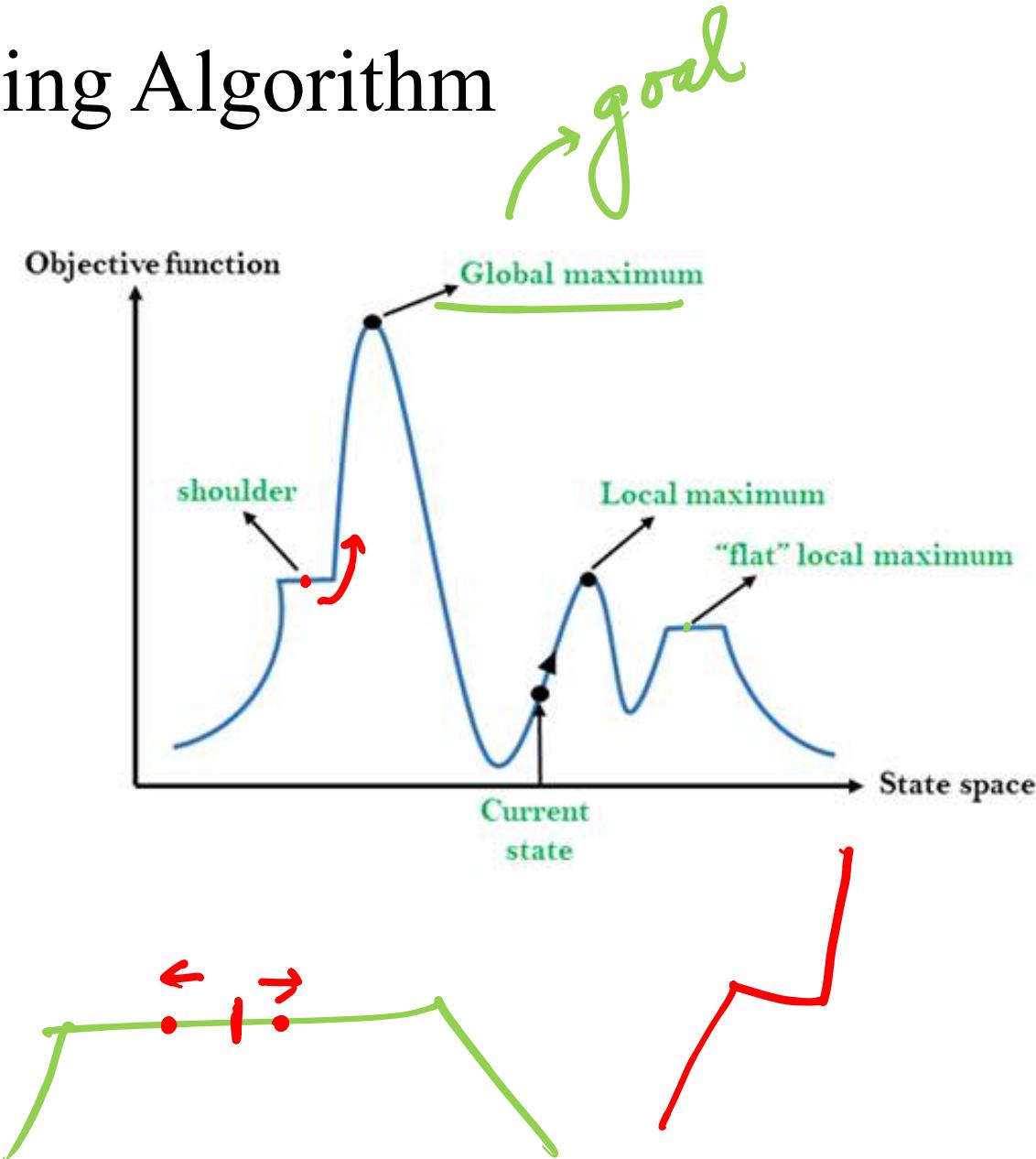




State-space Diagram for Climbing Algorithm

Hill

- **Local Maximum:** Local maximum is a state which is better than its neighbour states, but there may be another state which is higher than it.
- **Global Maximum:** Global maximum is the best possible state of state space landscape. It has the highest value of objective function.
- **Current state:** It is a state in a landscape diagram where an agent is currently present.
- **Flat local maximum:** It is a flat space in the landscape where all the neighbour states of current states have the same value.
- **Shoulder:** It is a plateau region which has an uphill edge.



Contact Details

- Anup Kumar Gupta
- PhD scholar, IIT Indore
- Email: msrphd2105101002@iiti.ac.in
- Website: <https://anupkumargupta.github.io/>

References

- <https://www.javatpoint.com/hill-climbing-algorithm-in-ai>

IAI : Semantic Networks and Frames

© John A. Bullinaria, 2005

1. Components of a Semantic Network
2. AND/OR Trees
3. IS-A and IS-PART Hierarchies
4. Representing Events and Language
5. Intersection Search
6. Inheritance and Defaults
7. Tangled Hierarchies and Inferential Distance
8. The Relation between Semantic Networks and Frames
9. Components of Frame Based Systems
10. Slots as Fully-Fledged Objects

Components of a Semantic Network

→ used for Knowledge Representation.

We can define a **Semantic Network** by specifying its fundamental components:

{ Lexical part

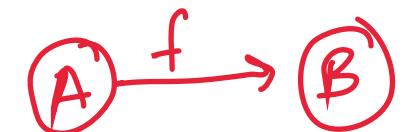
nodes – denoting objects / entities
links – denoting relations between objects
labels – denoting particular objects and relations

(edges)

$f: A \rightarrow B$

{ Structural part

the links and nodes form directed graphs
the labels are placed on the links and nodes



{ Semantic part

meanings are associated with the link and node labels
(the details will depend on the application domain)

• domain-specific

{ Procedural part

Scripts

constructors allow creation of new links and nodes
destructors allow the deletion of links and nodes
writers allow the creation and alteration of labels
readers can extract answers to questions

• user defined.

Clearly we are left with plenty of flexibility in creating these representations.

OOPs →

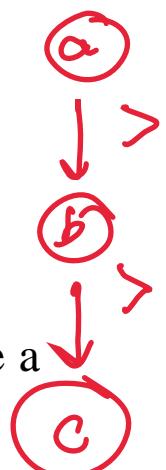
$\forall x A(x) \rightarrow E(x)$ } \rightarrow sentences

Semantic Networks as Knowledge Representations

Using Semantic Networks for representing knowledge has particular advantages:

1. They allow us to structure the knowledge to reflect the structure of that part of the world which is being represented. \rightarrow Domain or Universe
2. The semantics, i.e. real world meanings, are clearly identifiable. [Easy to understand]
3. There are very powerful representational possibilities as a result of "is a" and "is a part of" inheritance hierarchies. [Visually]
4. They can accommodate a hierarchy of default values (for example, we can assume the height of an adult male to be 178cm, but if we know he is a baseball player we should take it to be 195cm).
5. They can be used to represent events and natural language sentences.

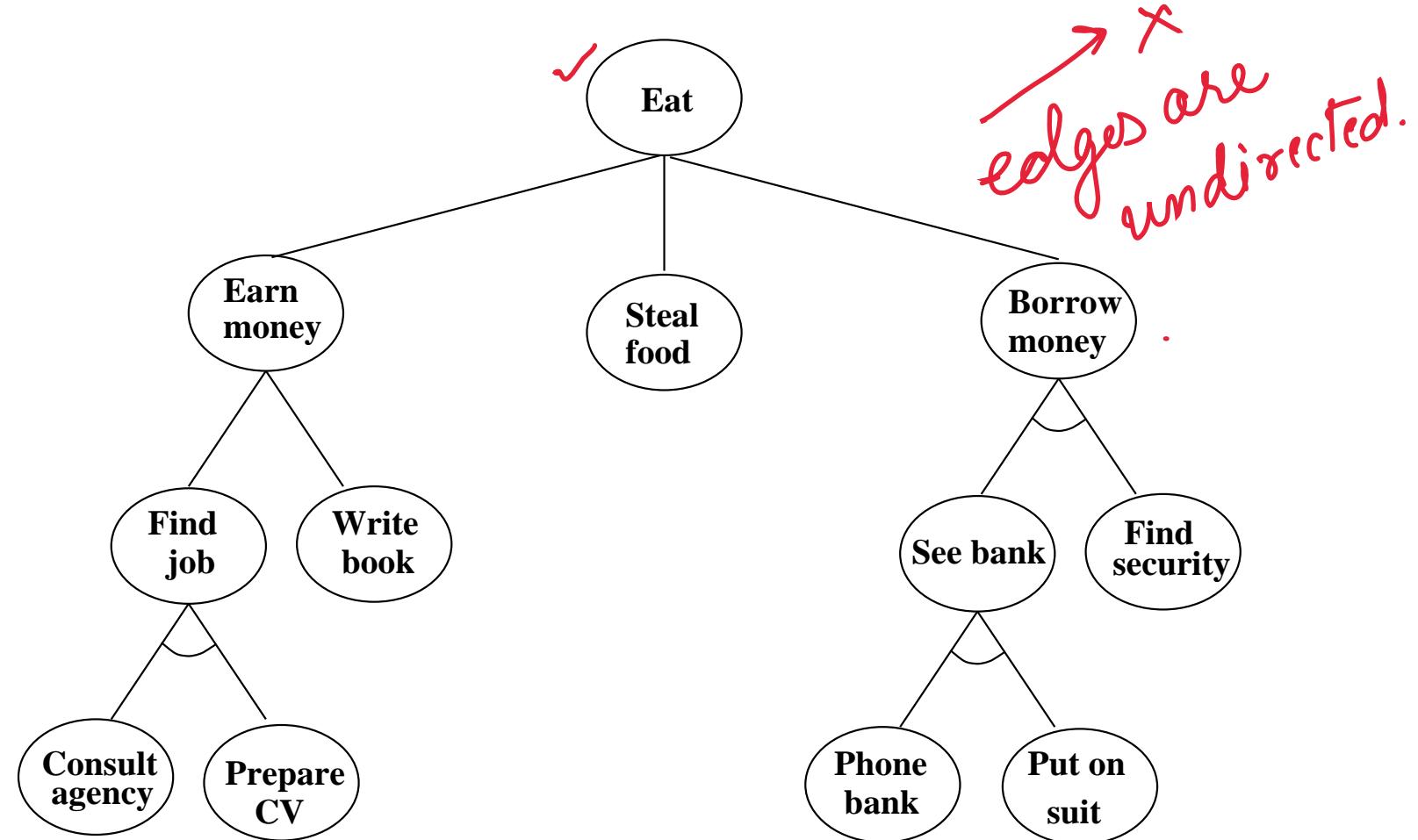
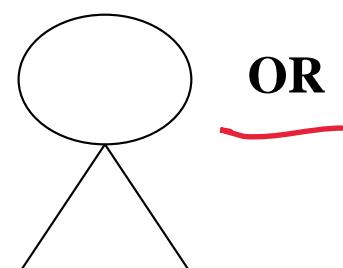
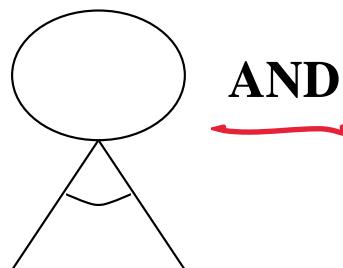
~~* They may be ambiguous.~~
Clearly, the notion of a semantic network is extremely general. However, that can be a problem, unless we are clear about the syntax and semantics in each case.



{ AND / OR Trees }

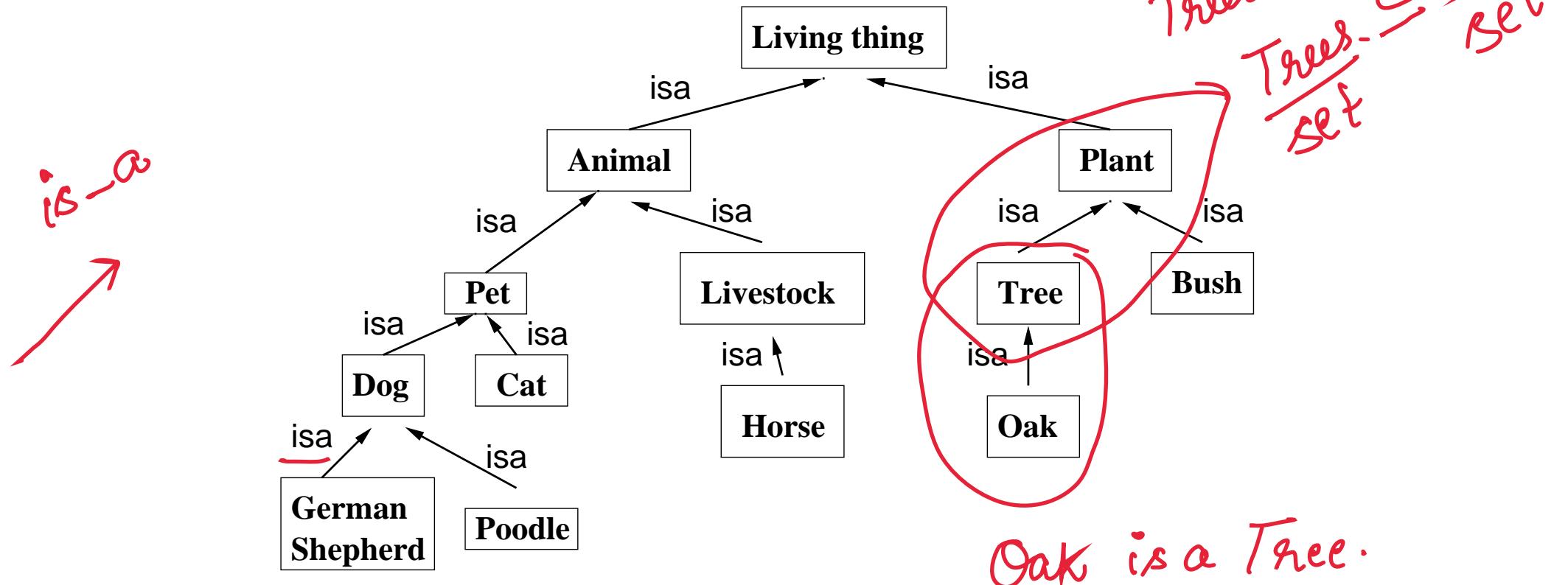
One particularly simple form of semantic network is an **AND/OR Tree**. For example:

Two node types:



An IS-A Hierarchy

Another simple form of semantic network is an *is-a hierarchy*. For example:

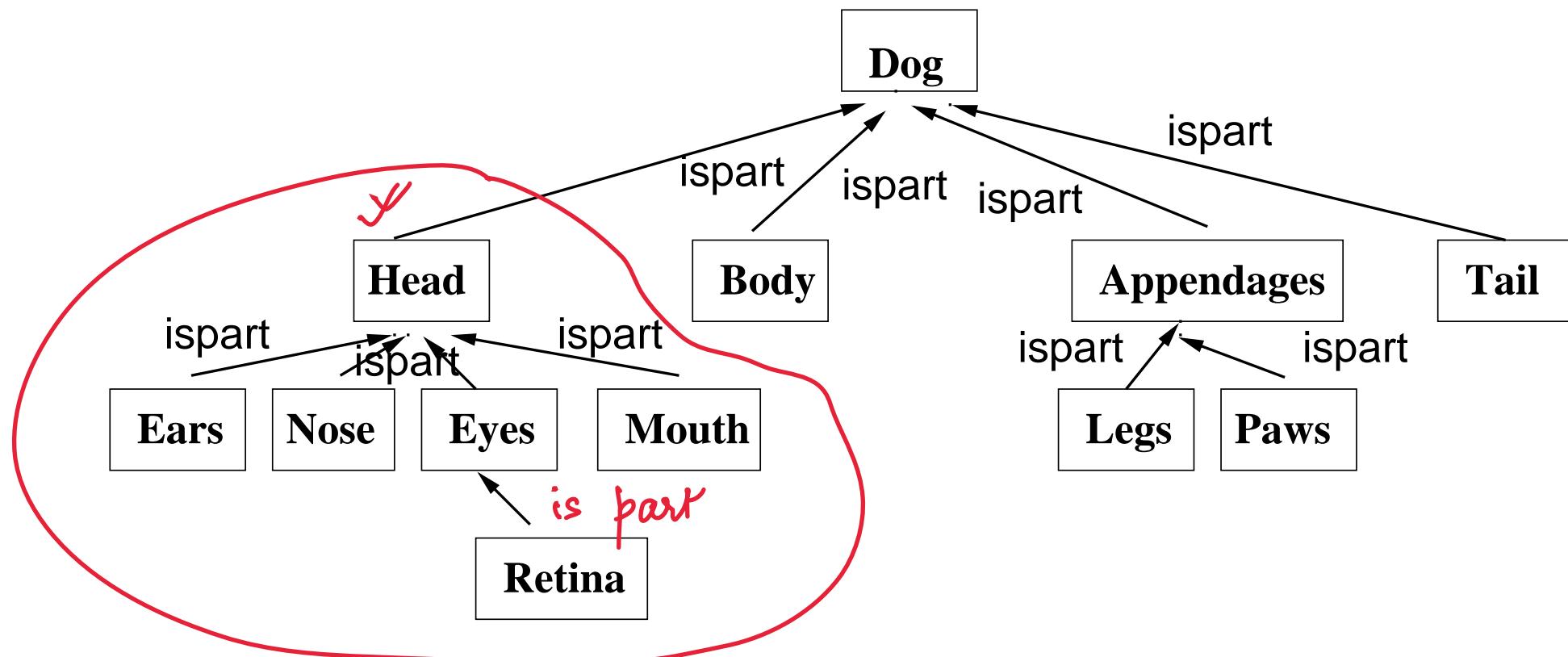


In set-theory terms, *is-a* corresponds to the sub-set relation \subseteq , and *instance* corresponds to the membership relation \in .

$$\frac{\text{Element}}{\text{Set}} = \frac{\text{Oak}}{\text{Tree.}}$$

An IS-PART Hierarchy

If necessary, we can take the hierarchy all the way down to the molecular or atomic level with an is-part hierarchy. For example:



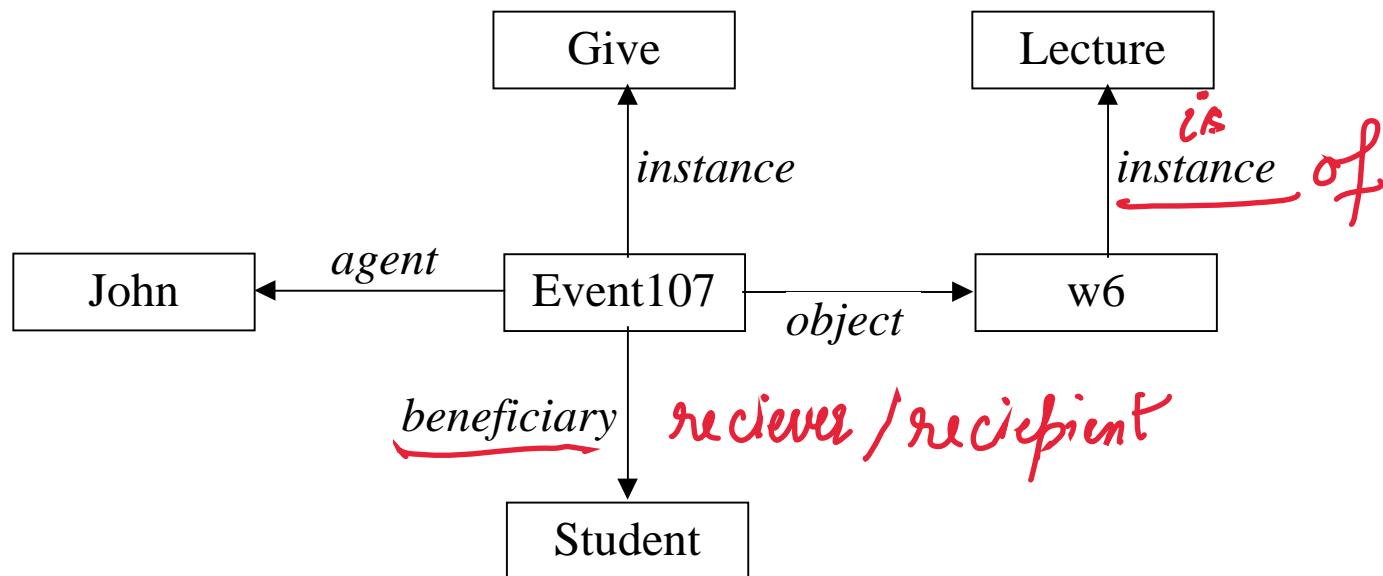
Naturally, where we choose to stop the hierarchy depends on what we want to represent.

Natural language or sentences.

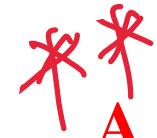
Representing Events and Language

Semantic networks are also very good at representing events, and simple declarative sentences, by basing them round an “event node”. For example:

“John gave lecture w6 to his students”

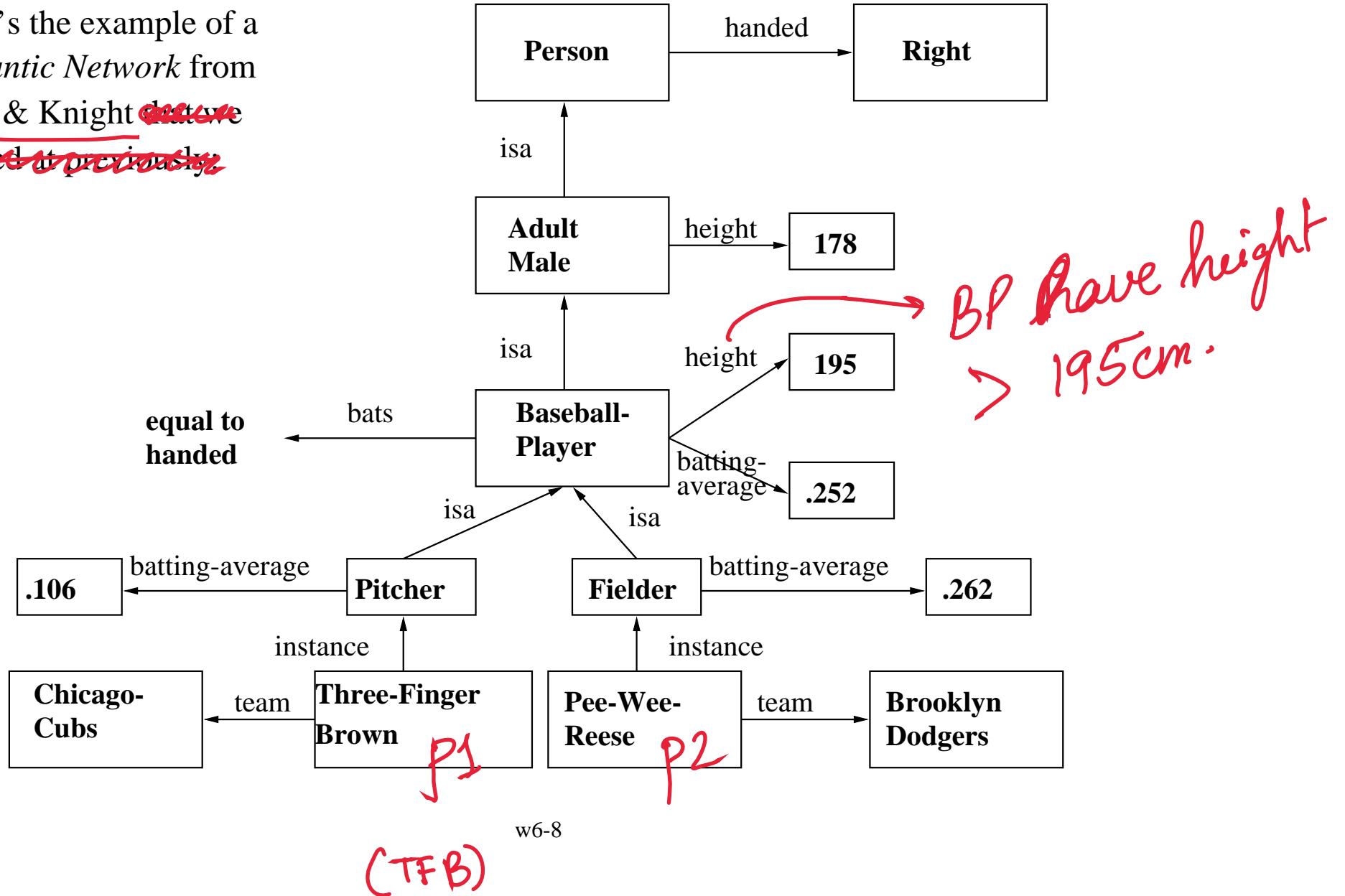


In fact, several of the earliest semantic networks were English-understanding programs.



A Typical Mixed-Type Semantic Network

Here's the example of a *Semantic Network* from Rich & Knight ~~that we looked at previously~~



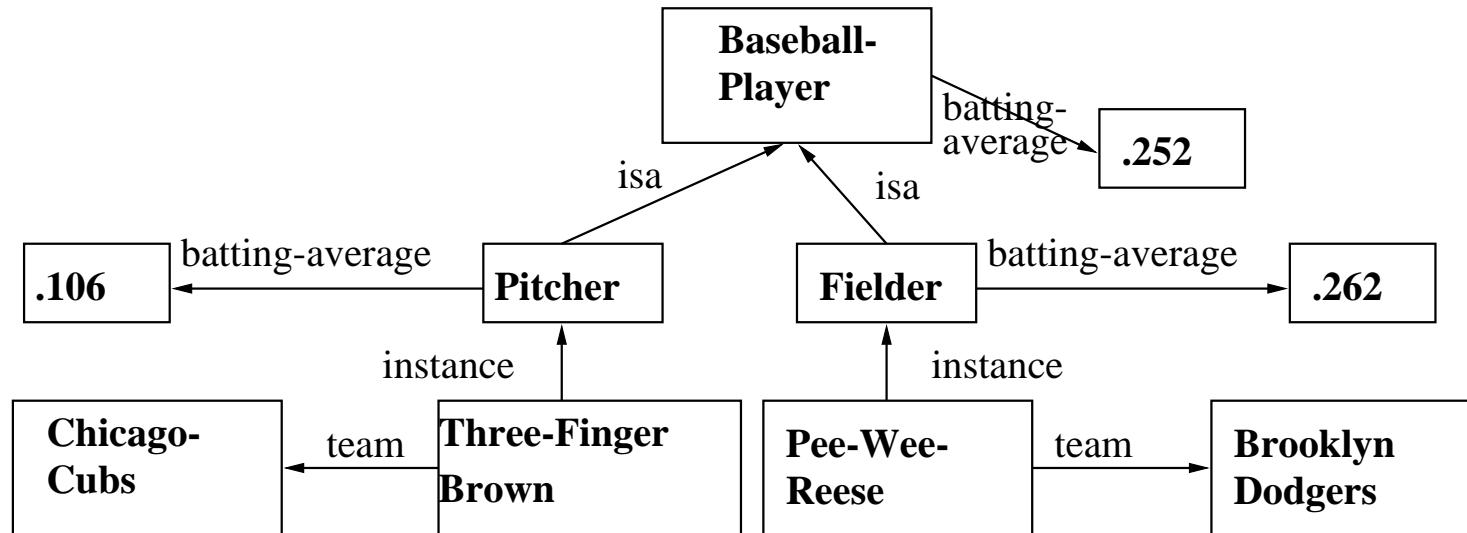
Intersection Search

Why Semantic Nets?

One of the earliest ways that semantic networks were used was to find relationships between objects by spreading *activation* from each of two nodes and seeing where the activations met. This process is called *intersection search*.

Inference.

Question: “What is the relation between Chicago cubs and Brooklyn Dodgers?”



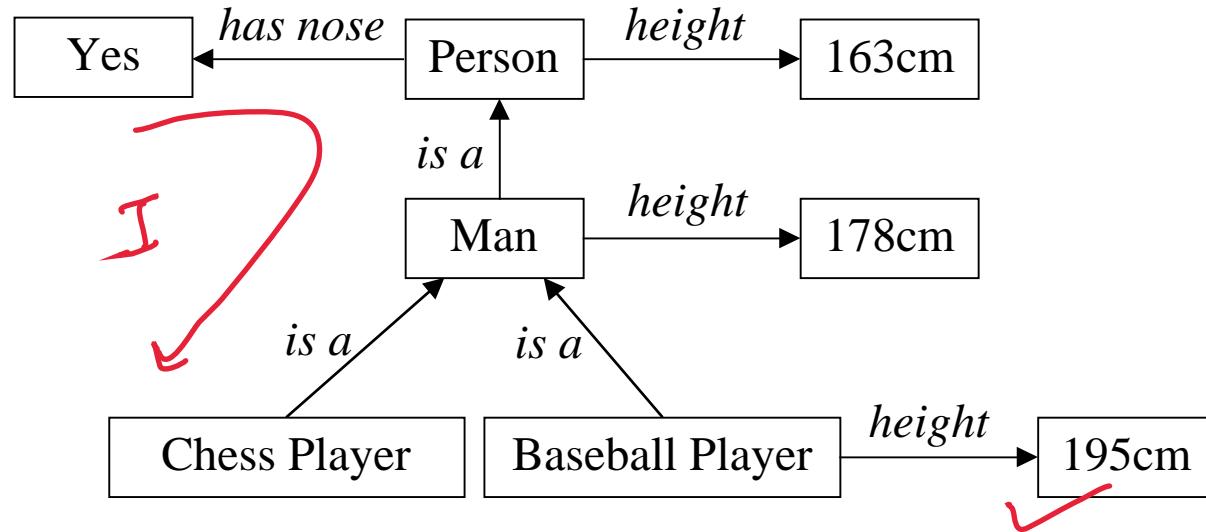
Answer: “They are both teams of baseball players.”

00 ps

. Animals
{#legs 3}
↓

Inheritance and Defaults

Two important features of semantic networks are the ideas of *default* (or typical) values and *inheritance*. Consider the following section of a semantic network:

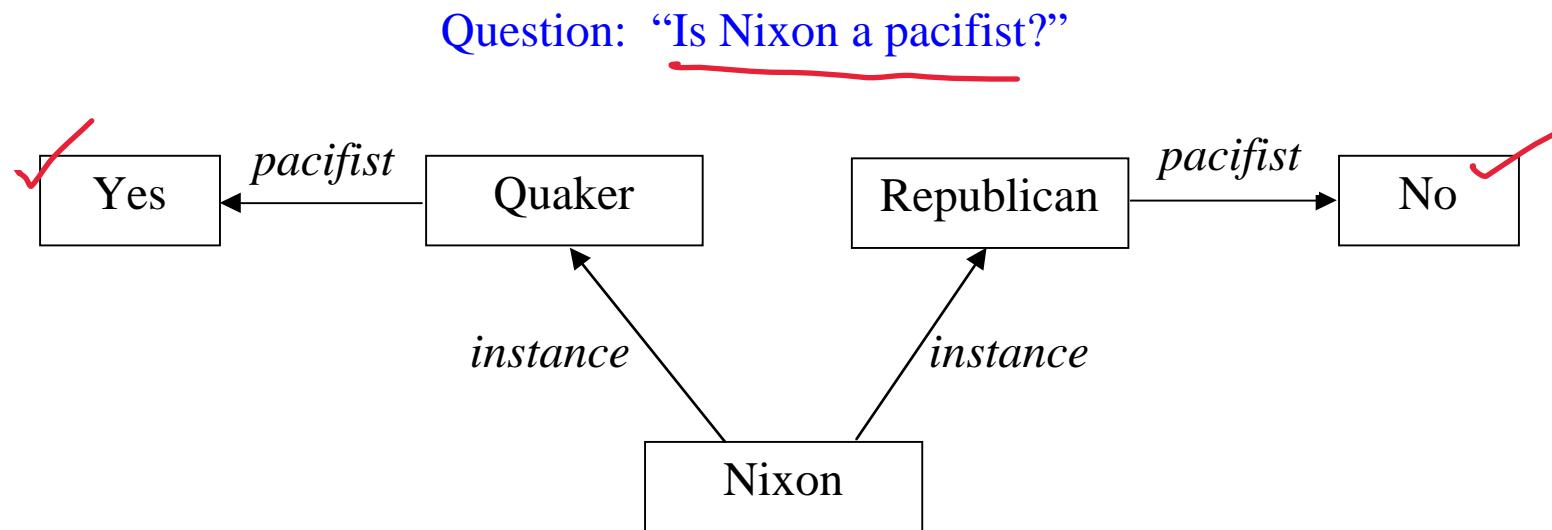


Person
{#Name }
↓
Student.
{#College-
name }

We can assign expected/default values of parameters (e.g. height, has nose) and inherit them from higher up the hierarchy. This is more efficient than listing all the details at each level. We can also *over-ride* the defaults. For example, baseball players are taller than average, so their default height over-rides the default height for men.

Multiple Inheritance

With simple trees, inheritance is straight-forward. However, when multiple inheritance is allowed, problems can occur. For example, consider this famous example:

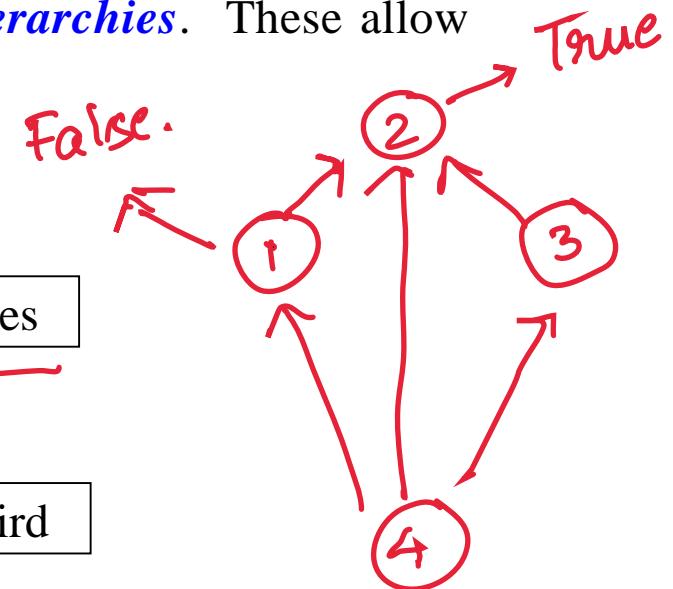
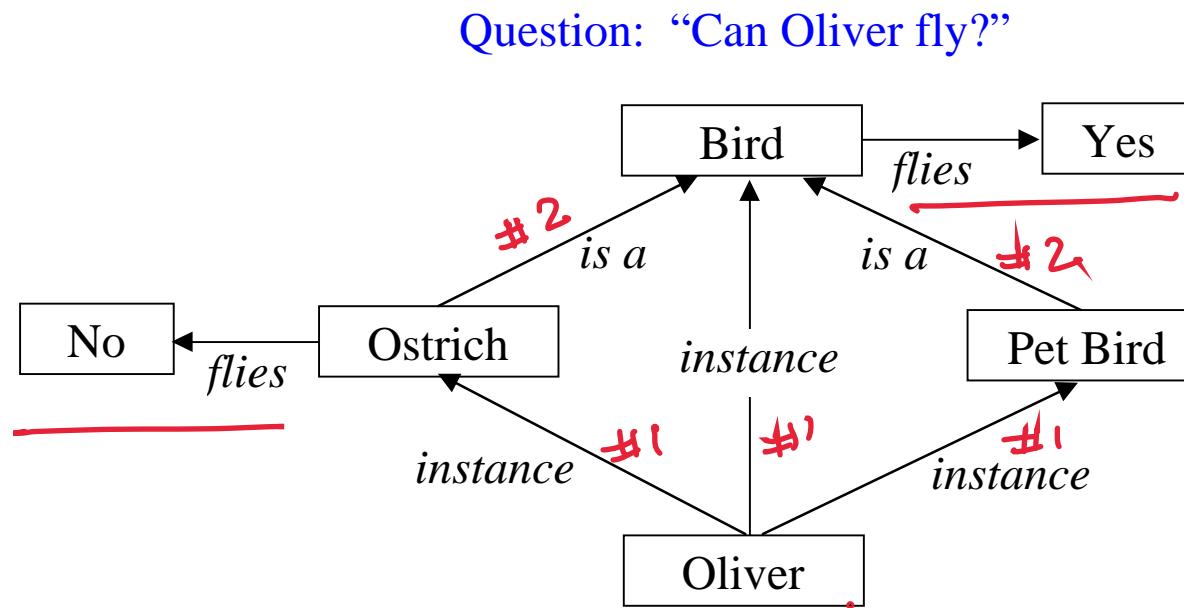


Conflicts like this are common in the real world. It is important that the inheritance algorithm reports the conflict, rather than just traversing the tree and reporting the first answer it finds. In practice, we aim to build semantic networks in which all such conflicts are either over-ridden, or resolved appropriately.

Tangled Hierarchies

Solⁿ:—
assigning priority
to relⁿ.

Hierarchies that are not simple trees are called *tangled hierarchies*. These allow another type of inheritance conflict. For example:



A better solution than having a specific “flies no” for all individual instances of an ostrich, would be to have an algorithm for traversing the algorithm which guarantees that specific knowledge will always dominate over general knowledge. How?

Inferential Distance

Not essential
for exam.

Note that simply counting nodes as a measure of distance will not generally give the required results. Why?

Instead, we can base our inheritance algorithm on the *inferential distance*, which can be used to define the concept of “closer” as follows:

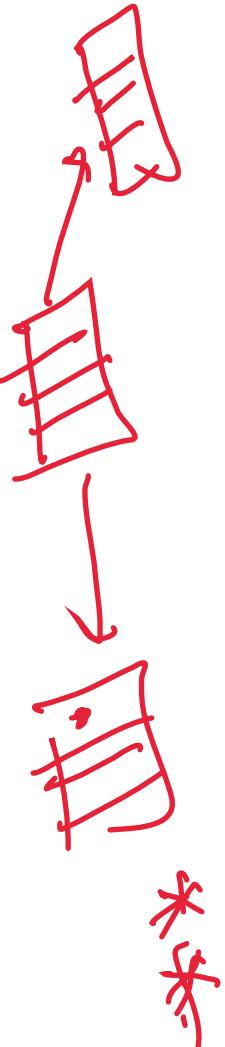
“*Node1* is closer to *Node2* than *Node3* if and only if *Node1* has an inference path through *Node2* to *Node3*, i.e. *Node2* is in between *Node1* and *Node3*. ”

Closer nodes in this sense will be more specific than further nodes, and so we should inherit any defaults from them.

Notice that inferential distance only defines a *partial ordering* – so it won’t be any help with the Nixon example.

In general, the *inferential engine* will be composed of many procedural rules like this to define how the semantic network should be processed.

The Relation between Semantic Networks and Frames



The idea of *semantic networks* started out as a natural way to represent labelled connections between entities. But, as the representations are expected to support increasingly large ranges of problem solving tasks, the representation schemes necessarily become increasingly complex.

In particular, it becomes necessary to assign more structure to nodes, as well as to links. For example, in many cases we need node labels that can be computed, rather than being fixed in advance. It is natural to use database ideas to keep track of everything, and the nodes and their relations begin to look more like *frames*.

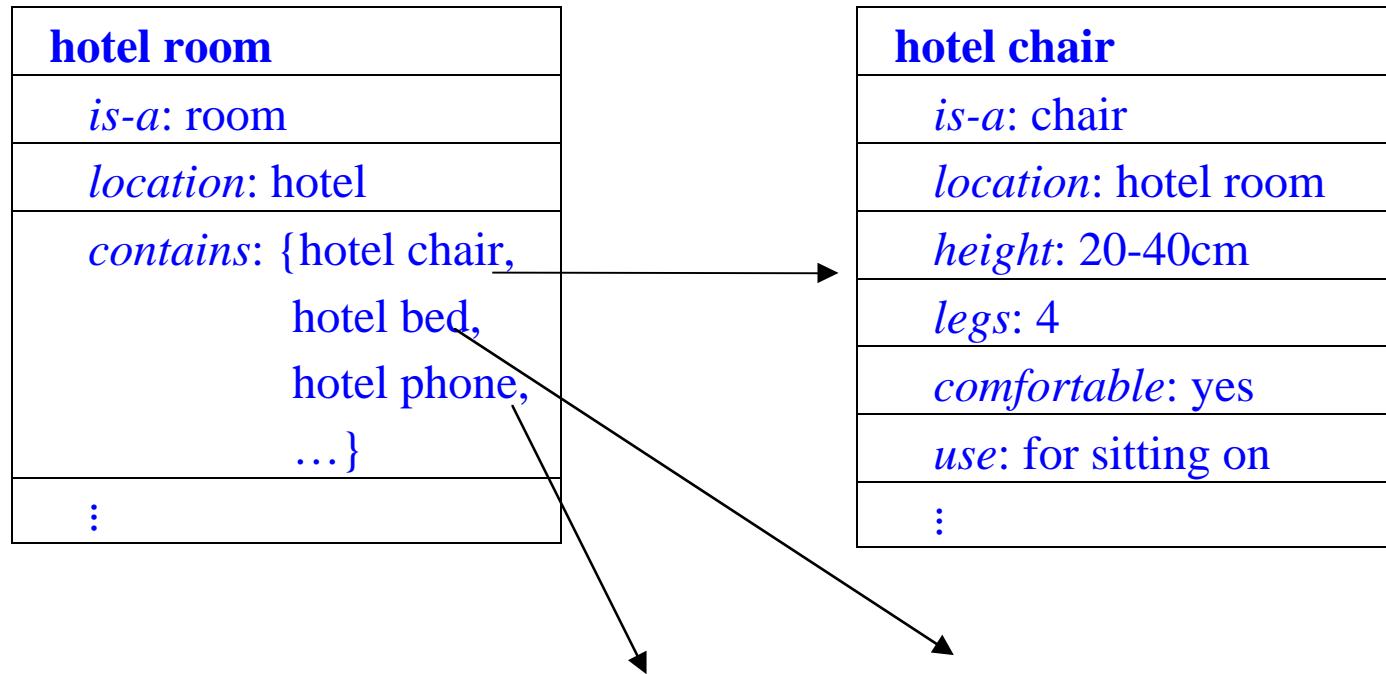
In the literature, the distinction between frames and semantic networks is actually rather blurred. However, the more structure a system has, the more likely it is to be termed a frame system rather than a semantic network.

For our purposes, we shall use the *practical distinction* that semantic networks look like networks, and frames look like frames.

Frame Based Systems – The Basic Idea

A **frame** consists of a selection of slots which can be filled by values, or procedures for calculating values, or pointers to other frames. For example:

inheritance



A complete frame based representation will consist of a whole hierarchy or network of frames connected together by appropriate links/pointers.

Used for dynamic values

Student

$m_1 = 1$

$m_2 = 2$

$m_3 = 3$

avg mark
)

{ $m_1 + m_2 + m_3$

3

Frames as a Knowledge Representation

The simplest type of frame is just a data structure with similar properties and possibilities for knowledge representation as a semantic network, with the same ideas of inheritance and default values.

Frames become much more powerful when their slots can also contain instructions (procedures) for computing things from information in other slots or in other frames.

- The original idea of frames was due to Minsky (1975) who defined them as “data-structures for representing stereotyped situations”, such as going into a hotel room.

(*dynamic comp*)

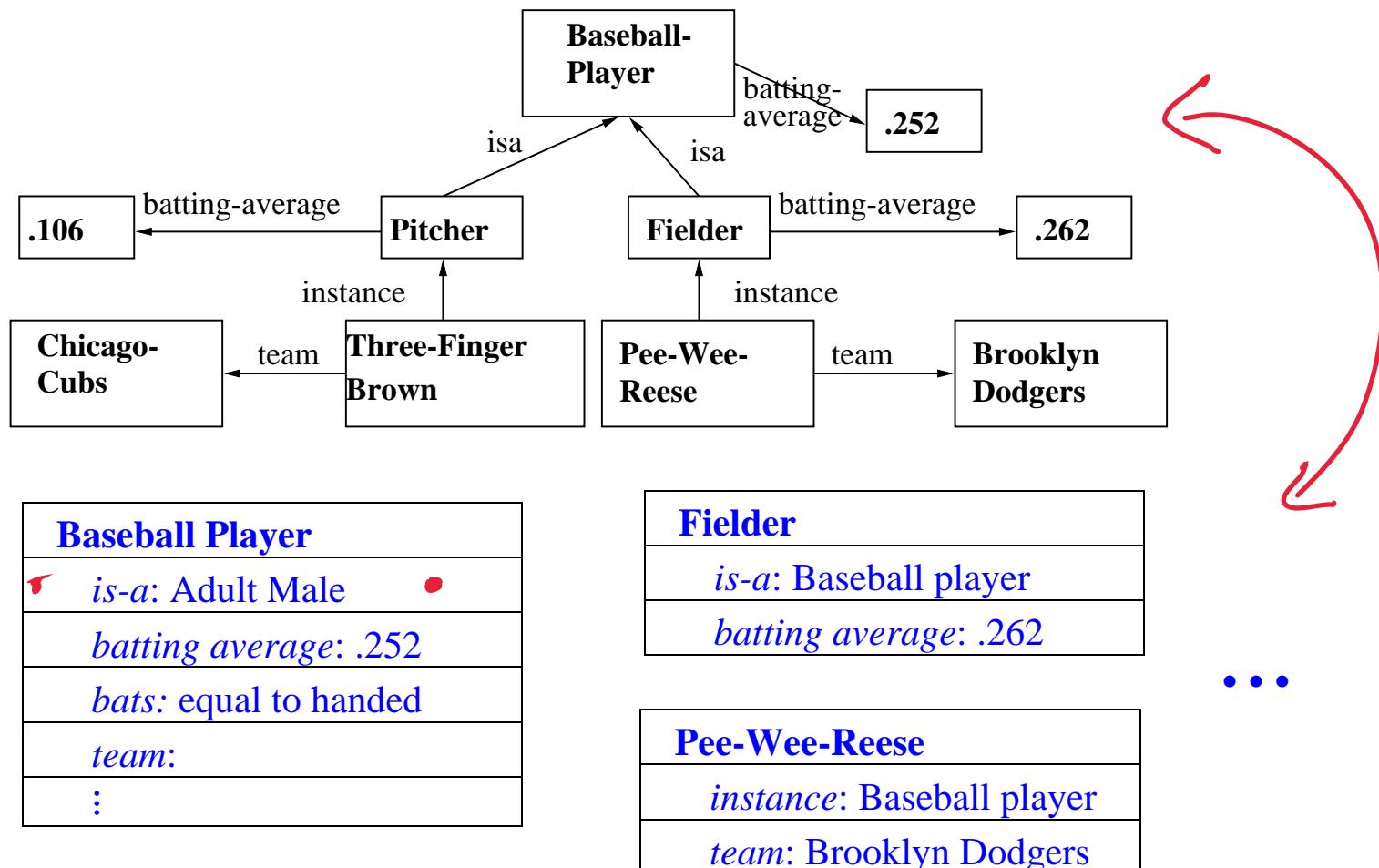
This type of frames are now generally referred to as Scripts. Attached to each frame will then be several kinds of information. Some information can be about how to use the frame. Some can be about what one can expect to happen next, or what one should do next. Some can be about what to do if our expectations are not confirmed. Then, when one encounters a new situation, one can select from memory an appropriate frame and this can be adapted to fit reality by changing particular details as necessary.

Frames + Procedures = Scripts
wo-16

* eg *

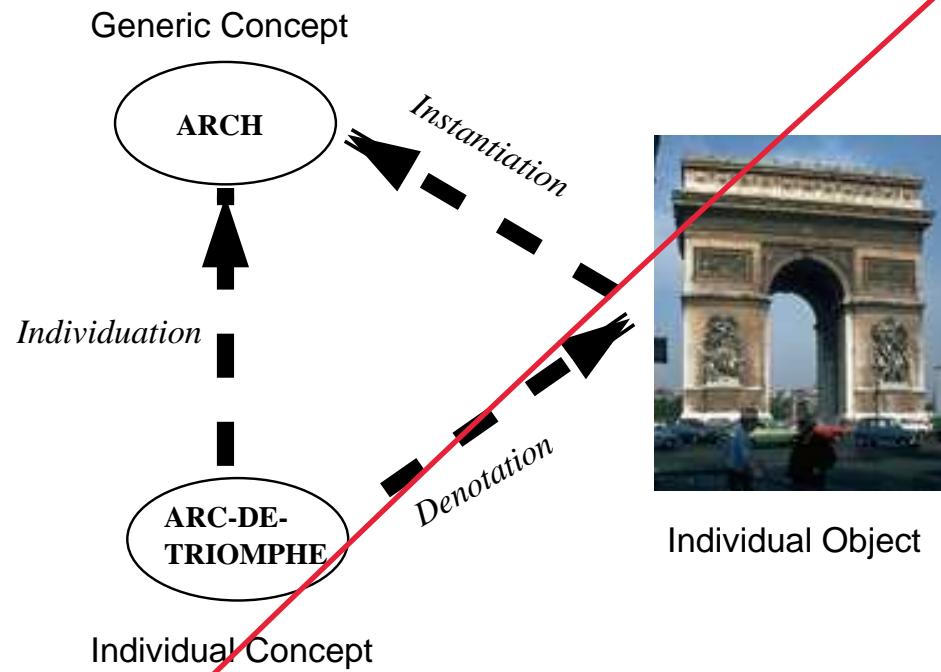
Converting between Semantic Networks and Frames

It is easy to construct frames for each node of a semantic net by reading off the links, e.g.



Set Theory as a Basis For Frame Systems

The relationship between real world instances, the representation of instances, and the representation of sets/classes of instances, is already quite familiar, e.g.



Clearly *is-a* corresponds to subset \subseteq , and *instance* corresponds to element \in . Then set theory concepts such as **transitivity**, **intersection**, etc. apply automatically to our frames.

Slots as Fully-Fledged Objects

We have seen that frame based representations can be made much more powerful by allowing the slot fillers to become more than simple values. This includes being frames in their own right, with a full range of hierarchical arrangements, inheritance, etc. The main filler properties we generally want to represent are:

1. Details about whether the slot is single or multi-valued.
2. Constraints on the ranges of values or types of values.
3. Simple default values for the attribute.
4. Rules for inheriting values for the attribute.
5. Rules for computing values separately from inheritance.
6. The classes/frames to which it can be attached.
7. Inverse attributes.

Naturally, the frame system interpreter must know how to process such frames.

Overview and Reading

1. We began by looking at various common styles of semantic networks: *AND/OR* trees, *IS-A* and *IS-PART* hierarchies, and event/language networks.
2. The important procedural concepts of *Intersection Search*, *Inheritance*, *Defaults*, and *Inferential Conflict* were then covered.
3. We then looked at the relation between Semantic Networks and Frames, and how to convert from one to the other.
4. We ended by considering how slots can be promoted to fully-fledged objects with extremely general filler properties.

Reading

1. Rich & Knight: Chapters 9, 10
2. Winston: Chapter 2, 10
3. Jackson: Chapter 6
4. Russell & Norvig: Section 10.6