

Module 1

Introduction

Version 2 CSE IIT, Kharagpur

1.1 Instructional Objectives

- Understand the definition of artificial intelligence
- Understand the different faculties involved with intelligent behavior
- Examine the different ways of approaching AI
- Look at some example systems that use AI
- Trace briefly the history of 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.

We will introduce the following entities:

- An agent
- An intelligent agent
- A rational agent

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.

On completion of this lesson the student 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

The student will become familiar with different 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

Lesson 1

Introduction to AI

Version 2 CSE IIT, Kharagpur

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?

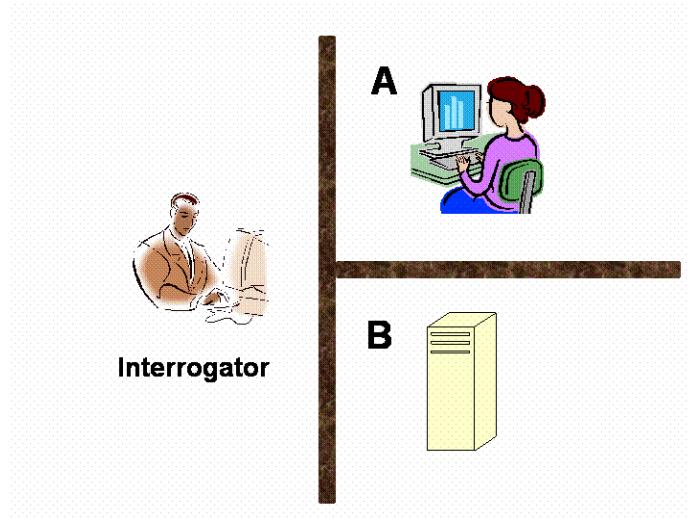
- Is it that which characterize humans? Or is there an absolute standard of judgement?
- Accordingly there are two possibilities:
 - A system with intelligence is expected to behave as intelligently as a human
 - A system with intelligence is expected to behave in the best possible manner
- Secondly 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?

Given this scenario different interpretations have been used by different researchers as defining the scope and view of Artificial Intelligence.

1. 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.

2. 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.



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 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. For more details on the Turing test visit the site <http://cogsci.ucsd.edu/~asaygin/tt/ttest.html>

3. 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.
4. 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

While studying the typical range of tasks that we might 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 factly and 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.

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. 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.

Intelligent behaviour

This discussion brings us back to the question of what constitutes intelligent behaviour. 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

1.1.3 Practical Impact of AI

AI components are embedded in numerous devices e.g. in copy machines for automatic correction of operation for copy quality improvement. AI systems are in everyday use for identifying credit card fraud, for advising doctors, for recognizing speech and in helping complex planning tasks. Then there are intelligent tutoring systems that provide students with personalized attention

Thus AI has increased understanding of the nature of intelligence and found many applications. It has helped in the understanding of human reasoning, and of the nature of intelligence. It has also helped us understand the complexity of modeling human reasoning.

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. Excessive optimism in the 1950s and 1960s concerning strong AI has given way to an appreciation of the extreme difficulty of the problem. 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 recognise 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 recognise faces and other objects, or about how we solve abstract problems.

1.1.5 Limits of AI Today

Today's successful AI systems operate in well-defined domains and employ narrow, specialized knowledge. Common sense knowledge is needed to function in complex, open-ended worlds. Such a system also needs to understand unconstrained natural language. However these capabilities are not yet fully present in today's intelligent systems.

What can AI systems do

Today's AI systems have been able to achieve limited success in some of these tasks.

- In Computer vision, the systems are capable of face recognition
- In Robotics, we have been able to make vehicles that are mostly autonomous.
- In Natural language processing, we have systems that are capable of simple machine translation.
- Today's Expert systems can carry out medical diagnosis in a narrow domain
- Speech understanding systems are capable of recognizing several thousand words continuous speech
- Planning and scheduling systems had been employed in scheduling experiments with

the Hubble Telescope.

- The Learning systems are capable of doing text categorization into about a 1000 topics
- In Games, AI systems can play at the Grand Master level in chess (world champion), checkers, etc.

What can AI systems NOT do yet?

- Understand natural language robustly (e.g., read and understand articles in a newspaper)
- Surf the web
- Interpret an arbitrary visual scene
- Learn a natural language
- Construct plans in dynamic real-time domains
- Exhibit true autonomy and intelligence

1.2 AI History

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.

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.

In the 19th century, George Boole developed a binary algebra representing (some) "laws of thought."

Charles Babbage & Ada Byron worked on programmable mechanical calculating machines.

In the late 19th century and early 20th century, mathematical philosophers like Gottlob Frege, Bertram Russell, Alfred North Whitehead, and Kurt Gödel built on Boole's initial logic concepts to develop mathematical representations of logic problems.

The advent of electronic computers provided a revolutionary advance in the ability to

study intelligence.

In 1943 McCulloch & Pitts developed a Boolean circuit model of brain. They wrote the paper “A Logical Calculus of Ideas Immanent in Nervous Activity”, which explained how it is possible for neural networks to compute.

Marvin Minsky and Dean Edmonds built the SNARC in 1951, which is the first randomly wired neural network learning machine (SNARC stands for Stochastic Neural-Analog Reinforcement Computer). It was a neural network computer that used 3000 vacuum tubes and a network with 40 neurons.

In 1950 Turing wrote an article on “Computing Machinery and Intelligence” which articulated a complete vision of AI. For more on Alan Turing see the site <http://www.turing.org.uk/turing/>

Turing’s paper talked of many things, of solving problems by searching through the space of possible solutions, guided by heuristics. He illustrated his ideas on machine intelligence by reference to chess. He even propounded the possibility of letting the machine alter its own instructions so that machines can learn from experience.

In 1956 a famous conference took place in Dartmouth. The conference brought together the founding fathers of artificial intelligence for the first time. In this meeting the term “Artificial Intelligence” was adopted.

Between 1952 and 1956, Samuel had developed several programs for playing checkers. In 1956, Newell & Simon’s Logic Theorist was published. It is considered by many to be the first AI program. In 1959, Gelernter developed a Geometry Engine. In 1961 James Slagle (PhD dissertation, MIT) wrote a symbolic integration program, SAINT. It was written in LISP and solved calculus problems at the college freshman level. In 1963, Thomas Evan’s program Analogy was developed which could solve IQ test type analogy problems.

In 1963, Edward A. Feigenbaum & Julian Feldman published Computers and Thought, the first collection of articles about artificial intelligence.

In 1965, J. Allen Robinson invented a mechanical proof procedure, the Resolution Method, which allowed programs to work efficiently with formal logic as a representation language. In 1967, the Dendral program (Feigenbaum, Lederberg, Buchanan, Sutherland at Stanford) was demonstrated which could interpret mass spectra on organic chemical compounds. This was the first successful knowledge-based program for scientific reasoning. In 1969 the SRI robot, Shakey, demonstrated combining locomotion, perception and problem solving.

The years from 1969 to 1979 marked the early development of knowledge-based systems. In 1974: MYCIN demonstrated the power of rule-based systems for knowledge representation and inference in medical diagnosis and therapy. Knowledge representation

schemes were developed. These included frames developed by Minsky. Logic based languages like Prolog and Planner were developed.

In the 1980s, Lisp Machines developed and marketed.

Around 1985, neural networks return to popularity

In 1988, there was a resurgence of probabilistic and decision-theoretic methods

The early AI systems used general systems, little knowledge. AI researchers realized that specialized knowledge is required for rich tasks to focus reasoning.

The 1990's saw major advances in all areas of AI including the following:

- machine learning, data mining
- intelligent tutoring,
- case-based reasoning,
- multi-agent planning, scheduling,
- uncertain reasoning,
- natural language understanding and translation,
- vision, virtual reality, games, and other topics.

Rod Brooks' COG Project at MIT, with numerous collaborators, made significant progress in building a humanoid robot

The first official Robo-Cup soccer match featuring table-top matches with 40 teams of interacting robots was held in 1997. For details, see the site <http://murray.newcastle.edu.au/users/students/2002/c3012299/bg.html>

In the late 90s, Web crawlers and other AI-based information extraction programs become essential in widespread use of the world-wide-web.

Interactive robot pets ("smart toys") become commercially available, realizing the vision of the 18th century novelty toy makers.

In 2000, the Nomad robot explores remote regions of Antarctica looking for meteorite samples.

We will now look at a few famous AI system that has been developed over the years.

1. ALVINN:

Autonomous Land Vehicle In a Neural Network

In 1989, Dean Pomerleau at CMU created ALVINN. This is a system which learns to control vehicles by watching a person drive. It contains a neural network whose input is a 30x32 unit two dimensional camera image. The output layer is a representation of the direction the vehicle should travel.

The system drove a car from the East Coast of USA to the west coast, a total of about

2850 miles. Out of this about 50 miles were driven by a human, and the rest solely by the system.

2. Deep Blue

In 1997, the Deep Blue chess program created by IBM, beat the current world chess champion, Gary Kasparov.

3. Machine translation

A system capable of translations between people speaking different languages will be a remarkable achievement of enormous economic and cultural benefit. Machine translation is one of the important fields of endeavour in AI. While some translating systems have been developed, there is a lot of scope for improvement in translation quality.

4. Autonomous agents

In space exploration, robotic space probes autonomously monitor their surroundings, make decisions and act to achieve their goals.

NASA's Mars rovers successfully completed their primary three-month missions in April, 2004. The Spirit rover had been exploring a range of Martian hills that took two months to reach. It is finding curiously eroded rocks that may be new pieces to the puzzle of the region's past. Spirit's twin, Opportunity, had been examining exposed rock layers inside a crater.

5. Internet agents

The explosive growth of the internet has also led to growing interest in internet agents to monitor users' tasks, seek needed information, and to learn which information is most useful

For more information the reader may consult AI in the news:

<http://www.aaai.org/AITopics/html/current.html>

Module

1

Introduction

Version 2 CSE IIT, Kharagpur

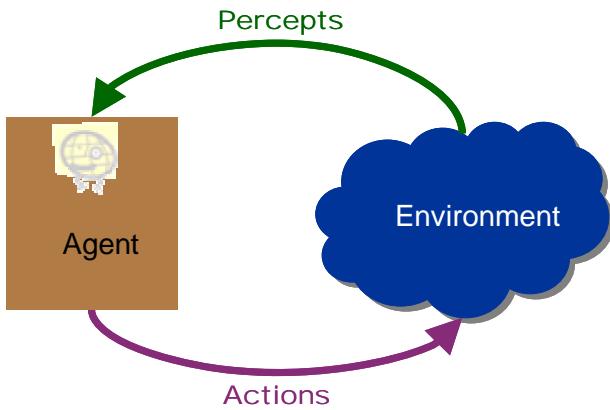
Lesson 2

Introduction to Agent

Version 2 CSE IIT, Kharagpur

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. Actions can be grouped into action sequences. The agent can have goals which it tries to achieve.

Thus, an agent can be looked upon as a system that implements a mapping from percept sequences to actions.

A performance measure has to be used in order to evaluate an agent.

An autonomous agent decides autonomously which action to take in the current situation to maximize progress towards its goals.

1.3.1.1 Agent Performance

An agent function implements a mapping from perception history to action. The behaviour and performance of intelligent agents have to be evaluated in terms of the agent function.

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 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.

Some examples of robots are Xavier from CMU, COG from MIT, etc.



Xavier Robot (CMU)

Then we have the AIBO entertainment robot from SONY.



Aibo from SONY

3. We also have software agents or softbots that have some functions as sensors and some functions as actuators. Askjeeves.com is an example of a softbot.
4. Expert systems like the Cardiologist is an agent.
5. Autonomous spacecrafts.
6. Intelligent buildings.

1.3.1.3 Agent Faculties

The fundamental faculties of intelligence are

- Acting
- Sensing
- Understanding, reasoning, 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.

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 her 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. 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. It is helpful to view the following definitions as referring to the way the environment appears from the point of view of the agent itself.

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. A bridge playing program is an example of a system operating in a partially observable environment.

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

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.

- A **static** environment does not change while the agent is thinking.
- The passage of time as an agent deliberates is irrelevant.
- The agent doesn't need to observe the world during deliberation.

A Dynamic Environment changes over time independent of the actions of the agent -- and thus 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/ 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 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 agent

In percept based agents,

1. information comes from **sensors - percepts**
2. changes the agents current **state of the world**
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.

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.

1.3.3.3 Subsumption Architecture

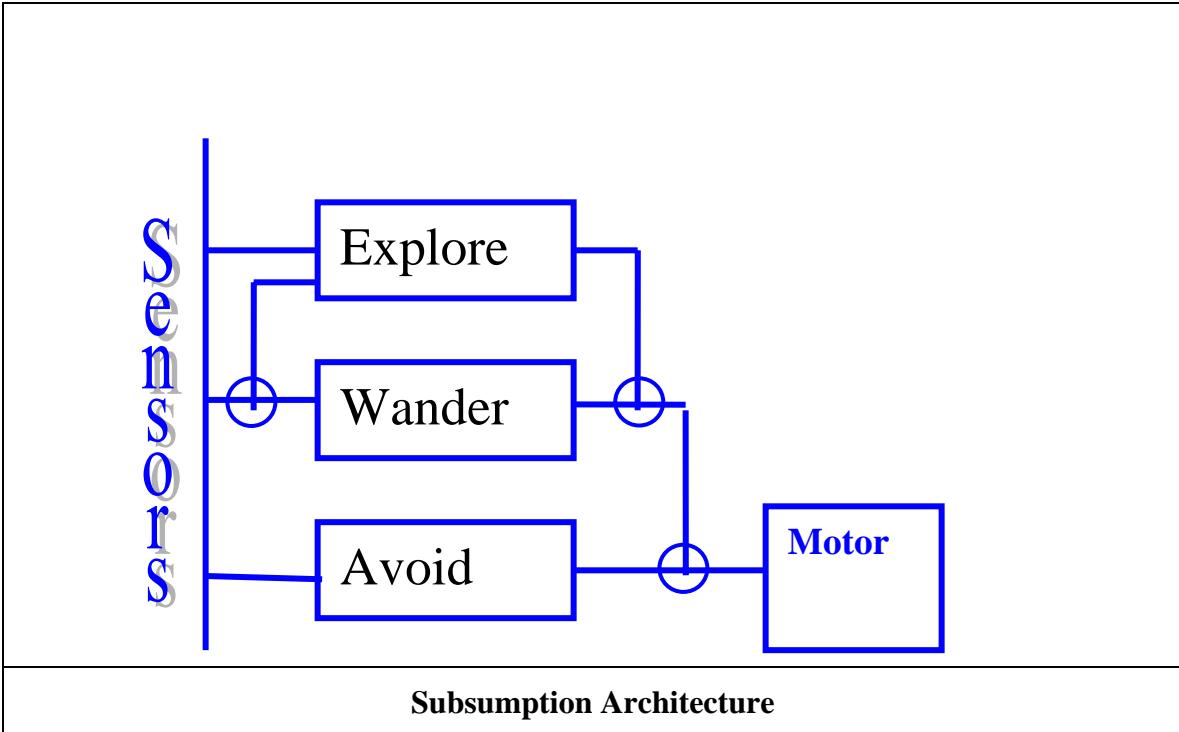
We will now briefly describe the subsumption architecture (Rodney Brooks, 1986). This architecture is based on reactive systems. Brooks notes that in lower animals there is no deliberation and the actions are based on sensory inputs. But even lower animals are capable of many complex tasks. His argument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

The main features of Brooks' architecture are.

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

The Subsumption Architecture built in layers. There are different layers of behaviour. The higher layers can override lower layers. Each activity is modeled by a finite state machine.

The subsumption architecture can be illustrated by Brooks' Mobile Robot example.



The system is built in three layers.

1. Layer 0: Avoid Obstacles
2. Layer1: Wander behaviour
3. Layer 2: Exploration behaviour

Layer 0 (Avoid Obstacles) has the following capabilities:

- Sonar: generate sonar scan
- Collide: send HALT message to forward
- Feel force: signal sent to run-away, turn

Layer1 (Wander behaviour)

- Generates a random heading
- Avoid reads repulsive force, generates new heading, feeds to turn and forward

Layer2 (Exploration behaviour)

- Whenlook notices idle time and looks for an interesting place.
- Pathplan sends new direction to avoid.
- Integrate monitors path and sends them to the path plan.

1.3.3.4 State-based Agent or model-based reflex agent

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 - percepts**
- based on this, the agent changes the current **state of the world**
- based on **state of the world** and **knowledge (memory)**, it triggers **actions** through the **effectors**

1.3.3.5 Goal-based Agent

The goal based agent has some goal which forms a basis of its actions.

Such agents work as follows:

- information comes from **sensors - percepts**
- changes the agents current **state of the world**
- based on **state of the world** and **knowledge (memory)** and **goals/intentions**, it chooses **actions** and does them through the **effectors**.

Goal formulation based on the current situation is a way of solving many problems and search is a universal problem solving mechanism in AI. 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.

1.3.3.6 Utility-based Agent

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 Agent

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

In conclusion AI is a truly fascinating field. It deals with exciting but hard problems. A 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.

- Reflex agents respond immediately to percepts.
- Goal-based agents act in order to achieve their goal(s).
- Utility-based agents maximize their own utility function.
- Representing knowledge is important for successful agent design.
- The most challenging environments are partially observable, stochastic, sequential, dynamic, and continuous, and contain multiple intelligent agents.

Questions

1. Define intelligence.
2. What are the different approaches in defining artificial intelligence?
3. Suppose you design a machine to pass the Turing test. What are the capabilities such a machine must have?
4. Design ten questions to pose to a man/machine that is taking the Turing test.
5. Do you think that building an artificially intelligent computer automatically shed light on the nature of natural intelligence?
6. List 5 tasks that you will like a computer to be able to do within the next 5 years.
7. List 5 tasks that computers are unlikely to be able to do in the next 10 years.
8. Define an agent.
9. What is a rational agent ?
10. What is bounded rationality ?
11. What is an autonomous agent ?
12. Describe the salient features of an agent.
13. Find out about the Mars rover.
 1. What are the percepts for this agent ?
 2. Characterize the operating environment.
 3. What are the actions the agent can take ?
 4. How can one evaluate the performance of the agent ?
 5. What sort of agent architecture do you think is most suitable for this agent ?
14. Answer the same questions as above for an Internet shopping agent.

Answers

1. Intelligence is a rather hard to define term.

Intelligence is often defined in terms of what we understand as intelligence in humans. Allen Newell defines *intelligence* as the *ability to bring all the knowledge a system has at its disposal to bear in the solution of a problem*.

A more practical definition that has been used in the context of building artificial systems with intelligence is *to perform better on tasks that humans currently do better*.

- 2.

- Thinking rationally
- Acting rationally
- Thinking like a human
- Acting like a human

3.

- Natural language processing
- Knowledge representation
- Automated reasoning
- Machine Learning
- Computer vision
- Robotics

4-7 : Use your own imagination

8. An agent is anything that can be viewed as perceiving its environment through sensors and executing actions using actuators.
 9. A rational agent always selects an action based on the percept sequence it has received so as to maximize its (expected) performance measure given the percepts it has received and the knowledge possessed by it.
 10. A rational agent that can use only bounded resources cannot exhibit the optimal behaviour. A bounded rational agent does the best possible job of selecting good actions given its goal, and given its bounded resources.
 11. Autonomous agents are software entities that are capable of independent action in dynamic, unpredictable environments. An autonomous agent can learn and adapt to a new environment.
 12.
 - An agent perceives its environment using sensors
 - An agent takes actions in the environment using actuators
 - A rational agent acts so as to reach its goal, or to maximize its utility
 - Reactive agents decide their action on the basis of their current state and the percepts. Deliberative agents reason about their goals to decide their action.
13. Mars Rover
- a. Spirit's sensor include
 - i. panoramic and microscopic cameras,
 - ii. a radio receiver,
 - iii. spectrometers for studying rock samples including an alpha particle x-ray spectrometer, Mössbauer spectrometer, and miniature thermal emission spectrometer
 - b. The environment (the Martian surface)
 - i. partially observable,
 - ii. non-deterministic,
 - iii. sequential,
 - iv. dynamic,
 - v. continuous, and
 - vi. may be single-agent. If a rover must cooperate with its mother ship or other rovers, or if mischievous Martians tamper with its progress, then the environment gains additional agents

- c. The **rover** Spirit has
 - i. motor-driven wheels for locomotion
 - ii. along with a robotic arm to bring sensors close to interesting rocks and a
 - iii. rock abrasion tool (RAT) capable of efficiently drilling 45mm holes in hard volcanic rock.
 - iv. Spirit also has a radio transmitter for communication.
- d. Performance measure: A **Mars rover** may be tasked with
 - i. maximizing the distance or variety of terrain it traverses,
 - ii. or with collecting as many samples as possible,
 - iii. or with finding life (for which it receives 1 point if it succeeds, and 0 points if it fails).

Criteria such as maximizing lifetime or minimizing power consumption are (at best) derived from more fundamental goals; e.g., if it crashes or runs out of power in the field, then it can't explore.

- e. A model-based reflex agent is suitable for low level navigation.
For route planning, experimentation etc, some combination of goal-based, and utility-based would be needed.

14. Internet book shopping agent

- f. Sensors: Ability to parse Web pages, interface for user requests
- g. Environment: Internet. Partially observable, partly deterministic, sequential, partly static, discrete, single-agent (exception: auctions)
- h. Actuators: Ability to follow links, fill in forms, display info to user
- i. Performance Measure: Obtains requested books, minimizes cost/time
- j. Agent architecture: goal based agent with utilities for open-ended situations

Module 2

Problem Solving using Search- (Single agent search)

2.1 Instructional Objective

- The students should understand the state space representation, and gain familiarity with some common problems formulated as state space search problems.
- The student should be familiar with the following algorithms, and should be able to code the 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
- Be able to apply these search techniques to a given problem whose description is provided.
- The student should understand how implicit state spaces can be unfolded during search.
- Understand how states can be represented by features.

The student will learn the following strategies for uninformed search:

- Breadth first search
- Depth first search
- Iterative deepening search
- Bidirectional search

For each of these they will learn

- The algorithm
- The time and space complexities
- When to select a particular strategy

At the end of this lesson the student should be able to do the following:

- Analyze a given problem and identify the most suitable search strategy for the problem.
- Given a problem, apply one of these strategies to find a solution for the problem.

Lesson 3

Introduction to State Space Search

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

2.2.1 Goal Directed Agent

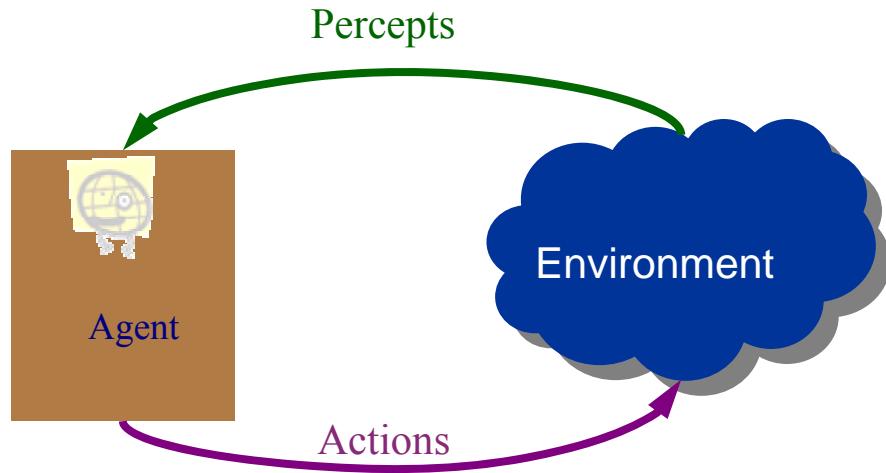


Figure 1

We have earlier discussed about an intelligent agent. Today we will study a type of intelligent agent which we will call a goal directed agent.

A goal directed agent needs to achieve certain goals. Such an agent selects its actions based on the goal it has. Many problems can be represented as a set of states and a set of rules of how one state is transformed to another. Each state is an abstract representation of the agent's environment. It is an abstraction that denotes a configuration of the agent. Initial state : The description of the starting configuration of the agent

An action/ operator takes the agent from one state to another state. A state can have a number of successor states.

A plan is a sequence of actions.

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.

Let us look at a few examples of goal directed agents.

1. 15-puzzle: The goal of an agent working on a 15-puzzle problem may be to reach a configuration which satisfies the condition that the top row has the tiles 1, 2 and 3. The details of this problem will be described later.
2. The goal of an agent may be to navigate a maze and reach the HOME position.

The agent must choose a sequence of actions to achieve the desired goal.

2.2.2 State Space Search Notations

Let us begin by introducing certain terms.

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.

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.

2.2.3 Search Problem

We are now ready to formally describe 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
- G is the set of final states. Note that $G \subseteq S$

These are schematically depicted in Figure 2.

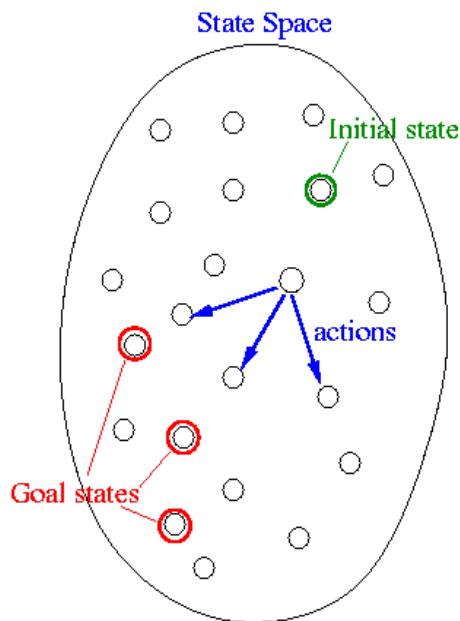


Figure 2

The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state $g \in G$. A search problem is represented by a 4-tuple $\{S, s_0, A, G\}$.

S: set of states

$s_0 \in S$: initial state

A: $S \rightarrow S$ operators/ actions that transform one state to another state

G : goal, a set of states. $G \subseteq S$

This sequence of actions is called a solution plan. It is a path from the initial state to a goal state. A *plan P* is a sequence of actions.

$P = \{a_0, a_1, \dots, a_N\}$ which leads to traversing a number of states $\{s_0, s_1, \dots, s_{N+1} \in G\}$. A sequence of states is called a path. The cost of a path is a positive number. In many cases the path cost is computed by taking the sum of the costs of each action.

Representation of search problems

A search problem is represented using a directed graph.

- The states are represented as nodes.
- The allowed actions are represented as arcs.

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
If it is not, the new state becomes the current state and the process is repeated

2.3 Examples

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 Figure 3.

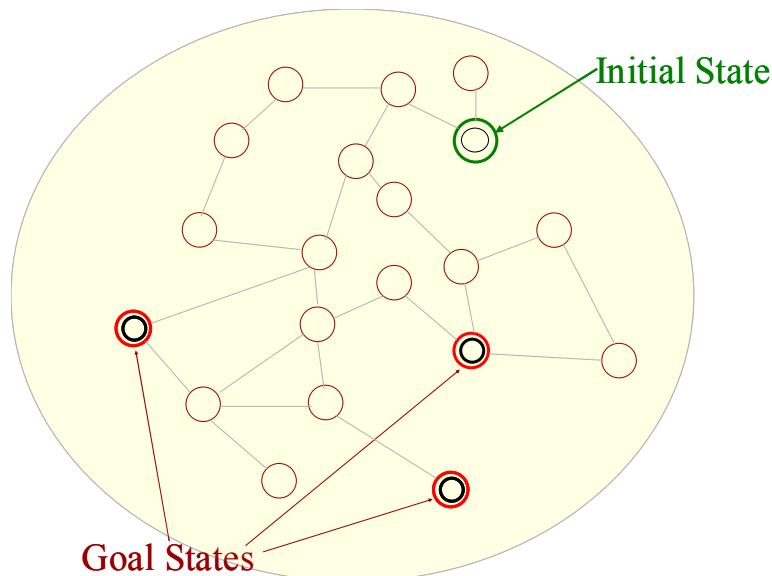


Figure 3

s_0 is the initial state.

The successor states are the adjacent states in the graph.

There are three goal states.

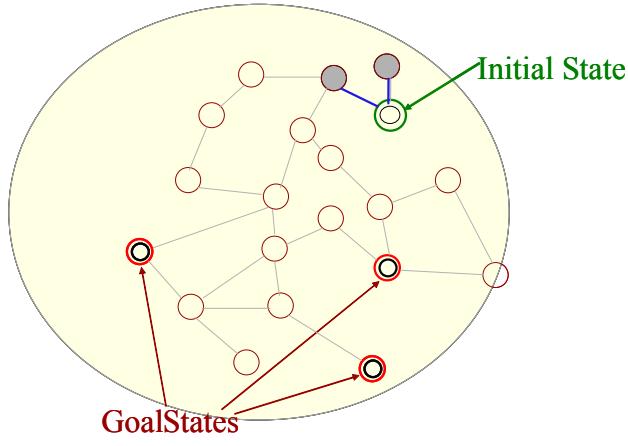


Figure 4

The two successor states of the initial state are generated.

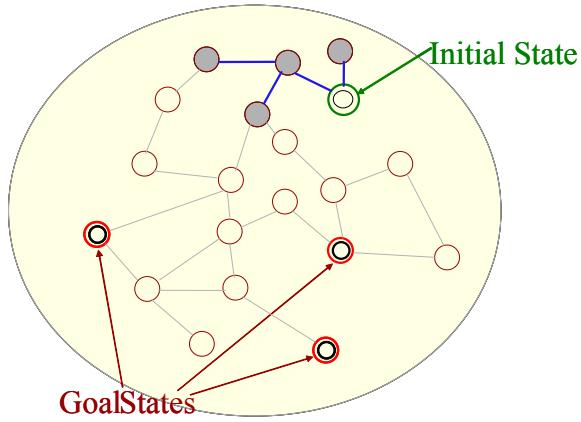


Figure 5

The successors of these states are picked and their successors are generated.

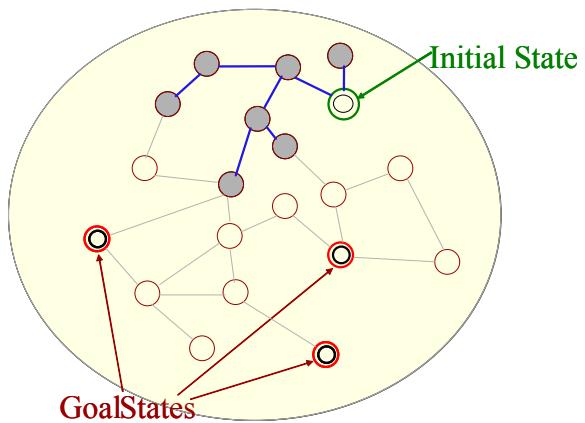


Figure 6

Successors of all these states are generated.

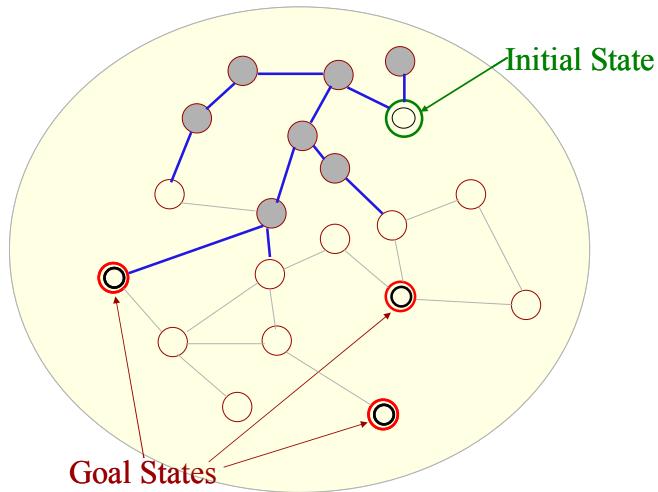


Figure 7

The successors are generated.

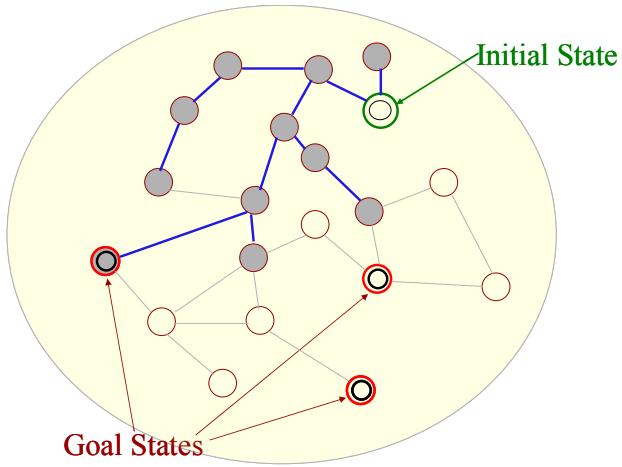


Figure 8

A goal state has been found.

The above 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 grey 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.

We will now illustrate state space search with one more example – the pegs and disks problem. We will illustrate a solution sequence which when applied to the initial state takes us to a goal state.

2.3.2 Example problem: Pegs and Disks problem

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

In the goal state all the pegs are in the needle B as shown in the figure below..

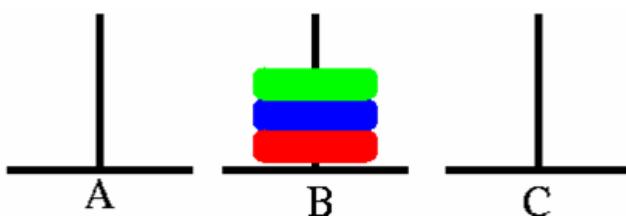


Figure 9

The initial state is illustrated below.

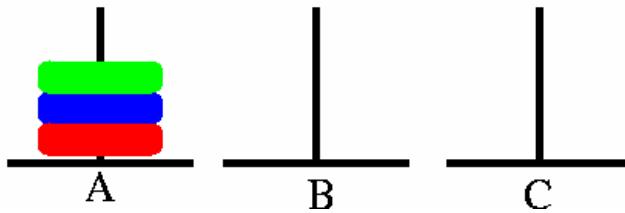


Figure 10

Now we will describe a sequence of actions that can be applied on the initial state.

Step 1: Move A → C

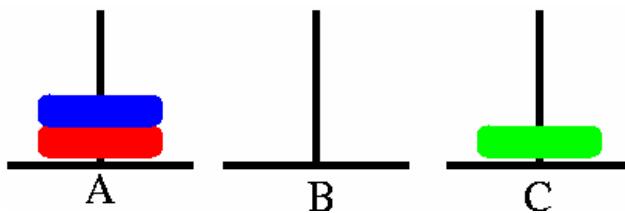


Figure 11

Step 2: Move A → B

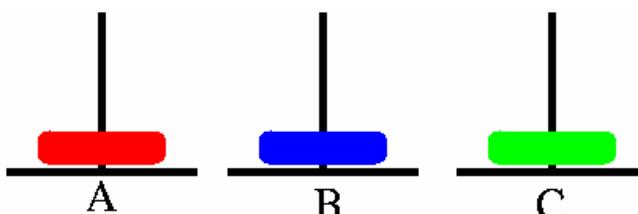


Figure 12

Step 3: Move A → C

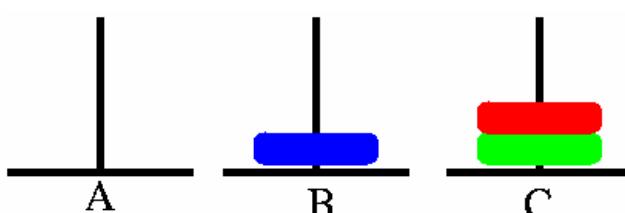


Figure 13

Step 4: Move B→A

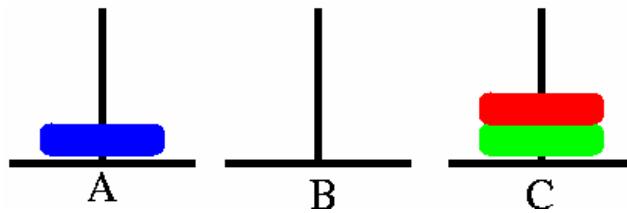


Figure 14

Step 5: Move C → B

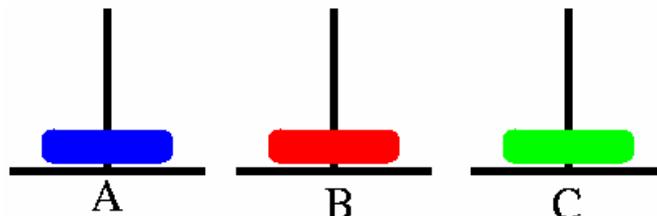


Figure 15

Step 6: Move A → B

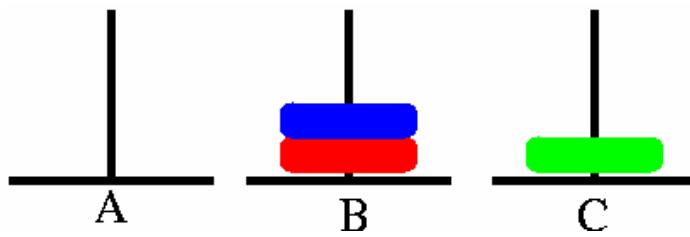


Figure 16

Step 7: Move C→ B

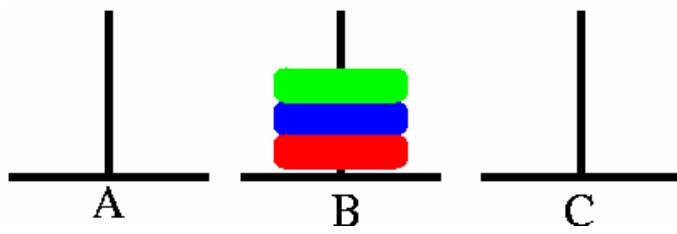


Figure 17

We will now look at another search problem – the 8-queens problem, which can be generalized to the N-queens problem.

2.3.4 8 queens problem

The problem is to place 8 queens on a chessboard so that no two queens are in the same row, column or diagonal

The picture below on the left shows a solution of the 8-queens problem. The picture on the right is not a correct solution, because some of the queens are attacking each other.

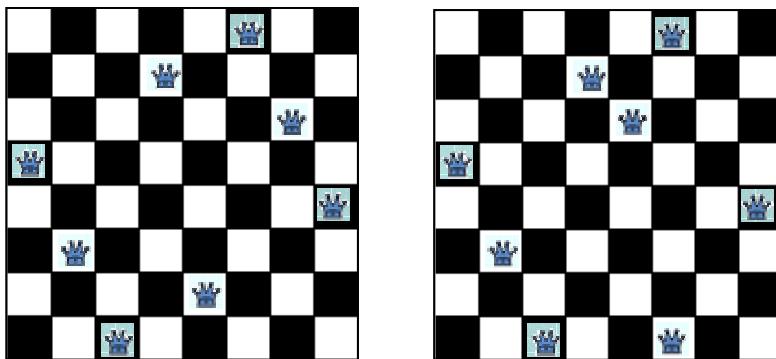


Figure 18

How do we formulate this in terms of a state space search problem? The problem formulation involves deciding the representation of the states, selecting the initial state representation, the description of the operators, and the successor states. We will now show that we can formulate the search problem in several different ways for this problem.

N queens problem formulation 1

- States: Any arrangement of 0 to 8 queens on the board
- Initial state: 0 queens on the board
- Successor function: Add a queen in any square
- Goal test: 8 queens on the board, none are attacked

The initial state has 64 successors. Each of the states at the next level have 63 successors, and so on. We can restrict the search tree somewhat by considering only those successors where no queen is attacking each other. To do that we have to check the new queen against all existing queens on the board. The solutions are found at a depth of 8.

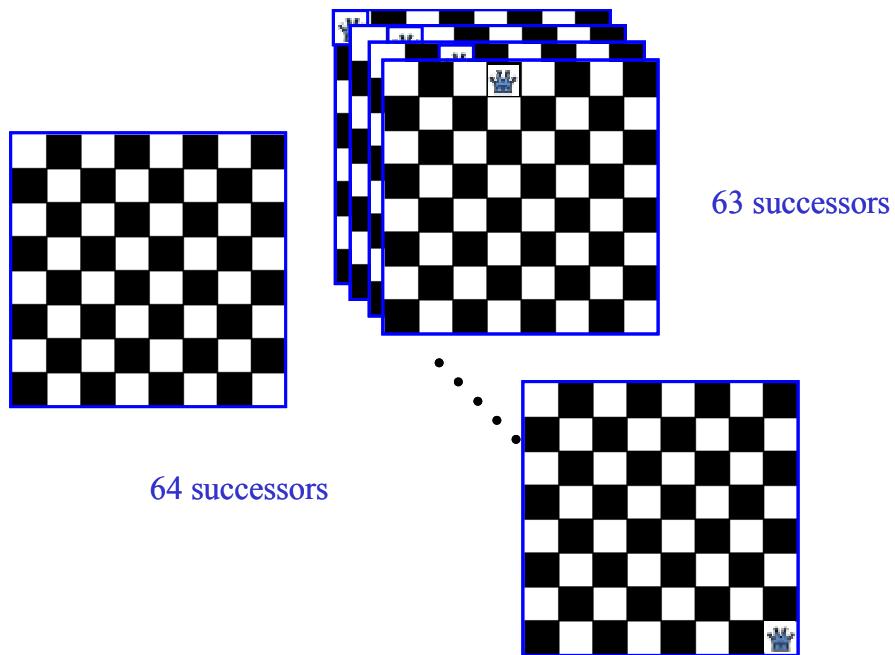


Figure 19

N queens problem formulation 2

- States: Any arrangement of 8 queens on the board
- Initial state: All queens are at column 1
- Successor function: Change the position of any one queen
- Goal test: 8 queens on the board, none are attacked

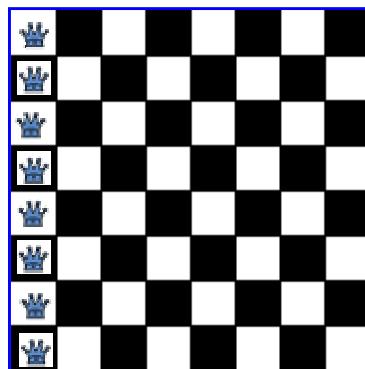


Figure 20

If we consider moving the queen at column 1, it may move to any of the seven remaining columns.

N queens problem formulation 3

- States: Any arrangement of k queens in the first k rows such that none are attacked
- Initial state: 0 queens on the board
- Successor function: Add a queen to the (k+1)th row so that none are attacked.
- Goal test : 8 queens on the board, none are attacked

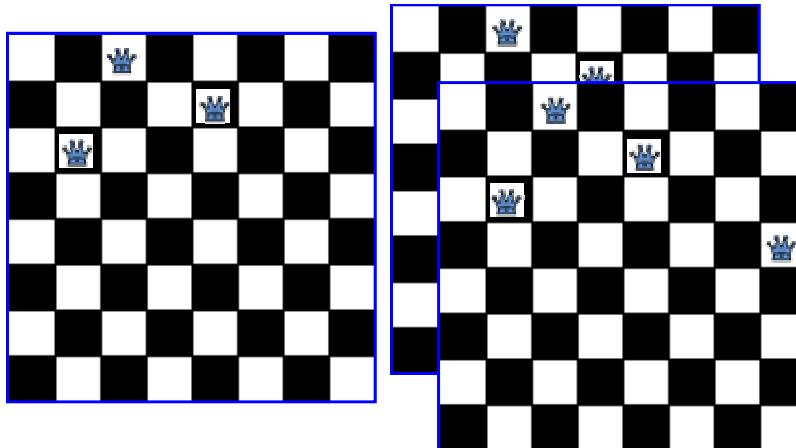


Figure 21

We will now take up yet another search problem, the 8 puzzle.

2.3.5 Problem Definition - Example, 8 puzzle

5	4	
6	1	8
7	3	2

1	4	7
2	5	8
3	6	

Initial State

Goal State

Figure 22

In the 8-puzzle problem we have a 3×3 square board and 8 numbered tiles. The board has one blank position. Bocks can be slid to adjacent blank positions. We can alternatively and equivalently look upon this as the movement of the blank position up, down, left or right. The objective of this puzzle is to move the tiles starting from an initial position and arrive at a given goal configuration.

The 15-puzzle problems is similar to the 8-puzzle. It has a 4×4 square board and 15 numbered tiles

The state space representation for this problem is summarized below:

States: A state is a description of each of the eight tiles in each location that it can occupy.

Operators/Action: The blank moves left, right, up or down

Goal Test: The current state matches a certain state (e.g. one of the ones shown on previous slide)

Path Cost: Each move of the blank costs 1

A small portion of the state space of 8-puzzle is shown below. Note that we do not need to generate all the states before the search begins. The states can be generated when required.

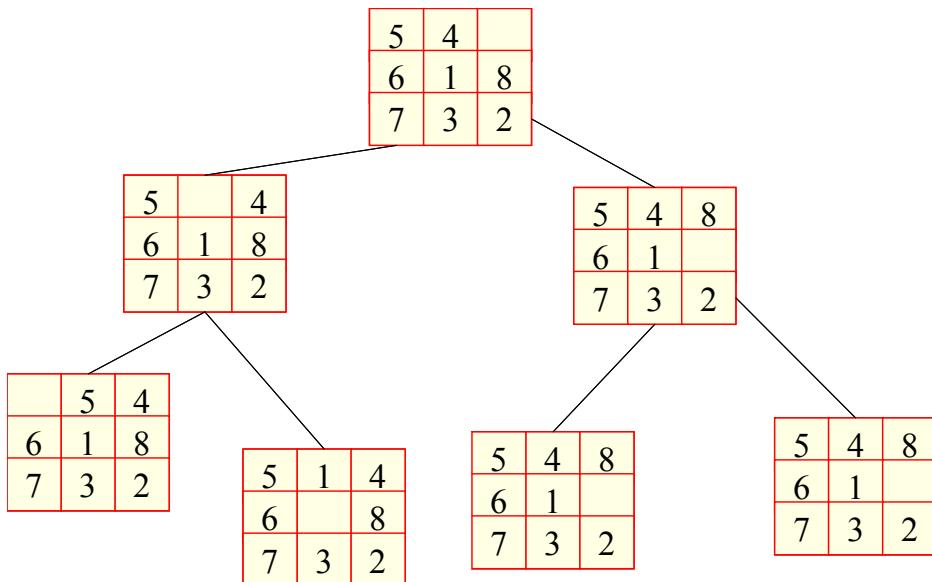


Figure 23

8-puzzle partial state space

2.3.6 Problem Definition - Example, tic-tac-toe

Another example we will consider now is the game of tic-tac-toe. This is a game that involves two players who play alternately. Player one puts a **X** in an empty position. Player 2 places an **O** in an unoccupied position. The player who can first get three of his symbols in the same row, column or diagonal wins. A portion of the state space of tic-tac-toe is depicted below.

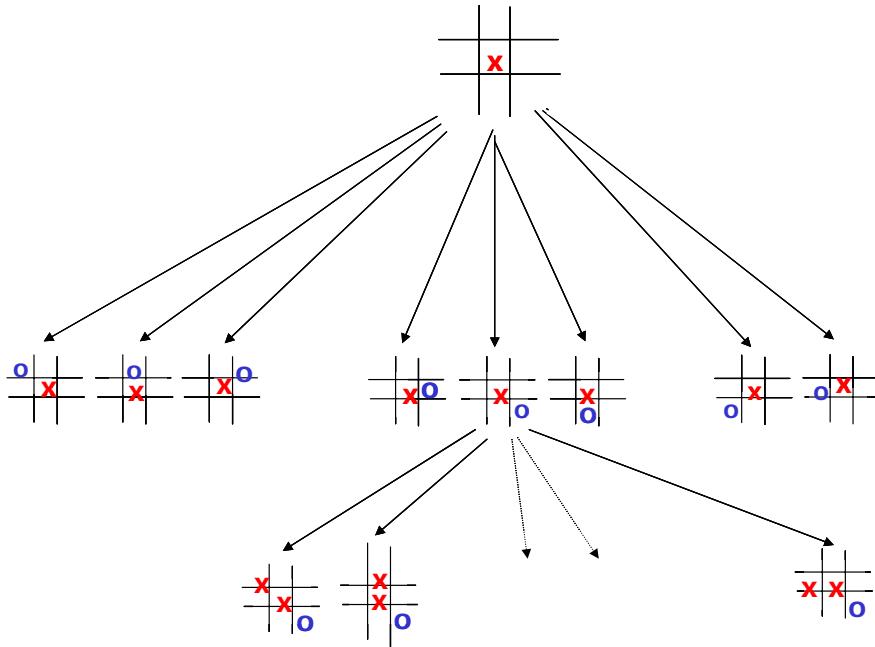


Figure 24

Now that we have looked at the state space representations of various search problems, we will now explore briefly the search algorithms.

2.3.7 Water jug problem

You have three jugs measuring 12 gallons, 8 gallons, and 3 gallons, and a water faucet.

You need to measure out exactly one gallon.

- Initial state: All three jugs are empty
- Goal test: Some jug contains exactly one gallon.
- Successor function: Applying the
 - **action transfer** to jugs i and j with capacities C_i and C_j and containing G_i and G_j gallons of water, respectively, leaves jug i with $\max(0, G_i - (C_j - G_j))$ gallons of water and jug j with $\min(C_j, G_i + G_j)$ gallons of water.
 - Applying the **action fill** to jug i leaves it with C_i gallons of water.
- Cost function: Charge one point for each gallon of water transferred and each gallon of water filled

Explicit vs Implicit state space

The state space may be explicitly represented by a graph. But more typically the state space can be implicitly represented and generated when required. To generate the state space implicitly, the agent needs to know

- the initial state
- The operators and a description of the effects of the operators

An operator is a function which "expands" a node and computes the successor node(s). In the various formulations of the N-queen's problem, note that if we know the effects of the operators, we do not need to keep an explicit representation of all the possible states, which may be quite large.

Module 2

Problem Solving using Search- (Single agent search)

Lesson 4

Uninformed Search

2.4 Search

Searching through a state space involves the following:

- A set of states
- Operators and their costs
- Start state
- A test to check for goal state

We will now outline the basic search algorithm, and then consider various variations of this algorithm.

2.4.1 The basic search algorithm

```
Let L be a list containing the initial state (L= the fringe)
Loop
    if L is empty return failure
    Node ← select (L)
    if Node is a goal
        then return Node
        (the path from initial state to Node)
    else generate all successors of Node, and
        merge the newly generated states into L
End Loop
```

We need to denote the states that have been generated. We will call these as nodes. The data structure for a node will keep track of not only the state, but also the parent state or the operator that was applied to get this state. In addition the search algorithm maintains a list of nodes called the fringe. The fringe keeps track of the nodes that have been generated but are yet to be explored. The fringe represents the frontier of the search tree generated. The basic search algorithm has been described above.

Initially, the fringe contains a single node corresponding to the start state. In this version we use only the OPEN list or fringe. The algorithm always picks the first node from fringe for expansion. If the node contains a goal state, the path to the goal is returned. The path corresponding to a goal node can be found by following the parent pointers. Otherwise all the successor nodes are generated and they are added to the fringe.

The successors of the current expanded node are put in fringe. We will soon see that the order in which the successors are put in fringe will determine the property of the search algorithm.

2.4.2 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. How can we handle loops?

Corresponding to a search algorithm, should we return a path or a node? The answer to this depends on the problem. For problems like N-queens we are only interested in the goal state. For problems like 15-puzzle, we are interested in the solution path.

We see that in the basic search algorithm, we have to select a node for expansion. Which node should we select? Alternatively, how would we place the newly generated nodes in the fringe? We will subsequently explore various search strategies and discuss their properties,

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. Also depending on the problem, our aim may be to find a minimal cost path or any to find path as soon as possible.

Which path to find?

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 we may need to find the shortest path or least cost path.

2.4.3 Evaluating Search strategies

We will look at various search strategies and evaluate their problem solving performance. What are the characteristics of the different search algorithms and what is their efficiency? We will look at the following three factors to measure this.

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 (number of nodes expanded) (worst or average case) to find a solution.
 - b. Space complexity: Space used by the algorithm measured in terms of the maximum size of fringe

The different search strategies that we will consider include the following:

1. Blind Search strategies or Uninformed search
 - a. Depth first search
 - b. Breadth first search
 - c. Iterative deepening search
 - d. Iterative broadening search
2. Informed Search
3. Constraint Satisfaction Search
4. Adversary Search

Blind Search

In this lesson we will talk about blind search or uninformed search that does not use any extra information about the problem domain. The two common methods of blind search are:

- BFS or Breadth First Search
- DFS or Depth First Search

2.4.4 Search Tree

Consider the explicit state space graph shown in the figure.

One may list all possible paths, eliminating cycles from the paths, and we would get the complete search tree from a state space graph. Let us examine certain terminology associated with a search tree. A search tree is a data structure containing a root node, from where the search starts. Every node may have 0 or more children. If a node X is a child of node Y, node Y is said to be a parent of node X.

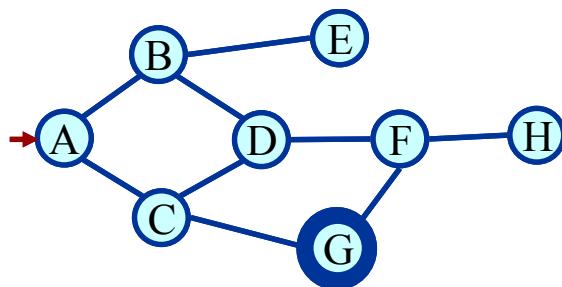


Figure 1: A State Space Graph

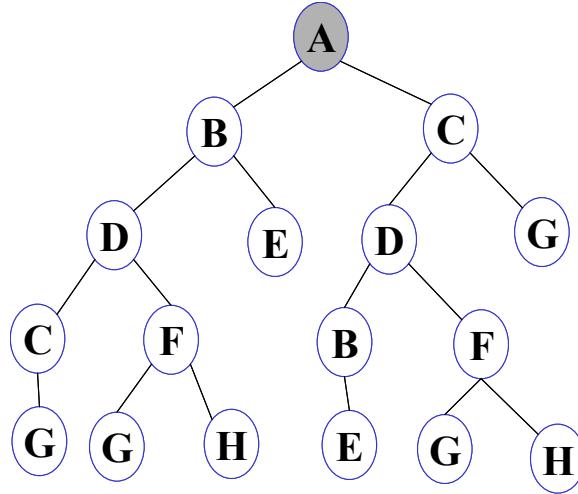


Figure 2: Search tree for the state space graph in Figure 25

Consider the state space given by the graph in Figure 25. Assume that the arcs are bidirectional. Starting the search from state A the search tree obtained is shown in Figure 26.

Search Tree – Terminology

- Root Node: The node from which the search starts.
- Leaf Node: A node in the search tree having no children.
- Ancestor/Descendant: X is an ancestor of Y if either X is Y's parent or X is an ancestor of the parent of Y. If S is an ancestor of Y, Y is said to be a descendant of X.
- Branching factor: the maximum number of children of a non-leaf node in the search tree
- Path: A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.

We also need to introduce some data structures that will be used in the search algorithms.

Node data structure

A node used in the search algorithm is a data structure which contains the following:

1. A state description
2. A pointer to the parent of the node
3. Depth of the node
4. The operator that generated this node
5. Cost of this path (sum of operator costs) from the start state

The nodes that the algorithm has generated are kept in a data structure called OPEN or fringe. Initially only the start node is in OPEN.

The search starts with the root node. The algorithm picks a node from OPEN for expanding and generates all the children of the node. Expanding a node from OPEN results in a closed node. Some search algorithms keep track of the closed nodes in a data structure called CLOSED.

A solution to the search problem is a sequence of operators that is associated with a path from a start node to a goal node. The cost of a solution is the sum of the arc costs on the solution path. For large state spaces, it is not practical to represent the whole space. State space search makes explicit a sufficient portion of an implicit state space graph to find a goal node. Each node represents a partial solution path from the start node to the given node. In general, from this node there are many possible paths that have this partial path as a prefix.

The search process constructs a search tree, where

- **root** is the initial state and
- **leaf nodes** are nodes
 - not yet expanded (i.e., in fringe) or
 - having no successors (i.e., “dead-ends”)

Search tree may be infinite because of loops even if state space is small

The search problem will return as a solution a path to a goal node. Finding a path is important in problems like path finding, solving 15-puzzle, and such other problems. There are also problems like the N-queens problem for which the path to the solution is not important. For such problems the search problem needs to return the goal state only.

2.5 Breadth First Search

2.5.1 Algorithm

Breadth first search

Let *fringe* be a list containing the initial state

Loop

```
    if fringe is empty return failure
    Node ← remove-first (fringe)
    if Node is a goal
        then return the path from initial state to Node
    else generate all successors of Node, and
        (merge the newly generated nodes into fringe)
        add generated nodes to the back of fringe
```

End Loop

Note that in breadth first search the newly generated nodes are put at the back of fringe or the OPEN list. What this implies is that the nodes will be expanded in a FIFO (First In

First Out) order. The node that enters OPEN earlier will be expanded earlier. This amounts to expanding the shallowest nodes first.

2.5.2 BFS illustrated

We will now consider the search space in Figure 1, and show how breadth first search works on this graph.

Step 1: Initially fringe contains only one node corresponding to the source state A.

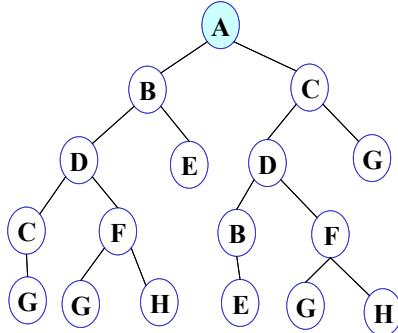


Figure 3

FRINGE: A

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.

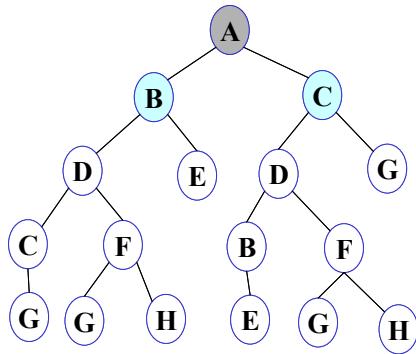


Figure 4

FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.

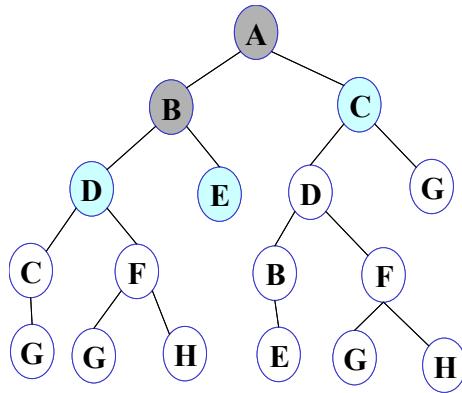


Figure 5

FRINGE: C D E

Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.

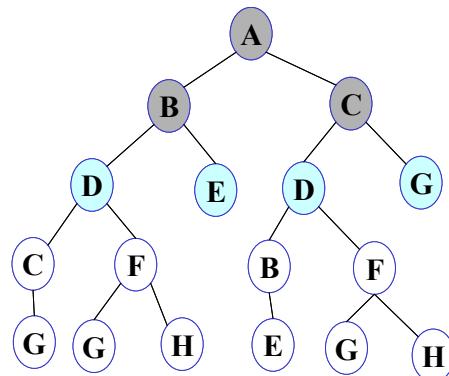


Figure 6

FRINGE: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.

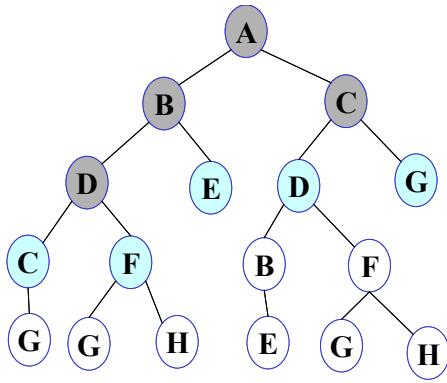
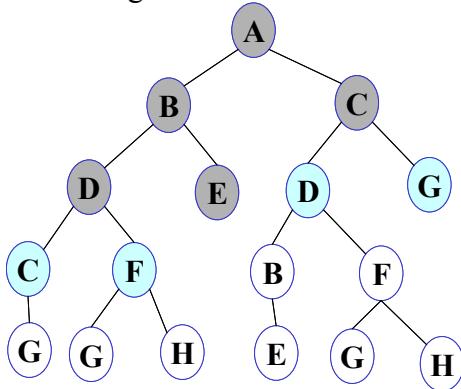


Figure 7

FRINGE: E D G C F

Step 6: Node E is removed from fringe. It has no children.



FRINGE: D G C F

Step 7: D is expanded, B and F are put in OPEN.

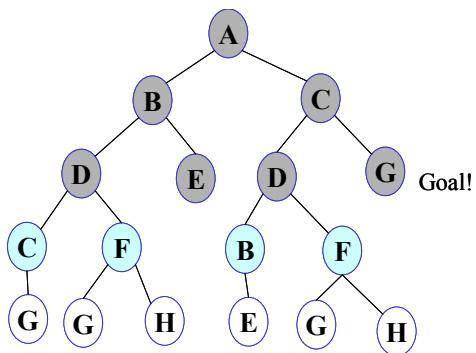


Figure 8

FRINGE: G C F B F

Step 8: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

2.5.3 Properties of Breadth-First Search

We will now explore certain properties of breadth first search. Let us consider a model of the search tree as shown in Figure 3. We assume that every non-leaf node has b children. Suppose that d is the depth of the shallowest goal node, and m is the depth of the node found first.

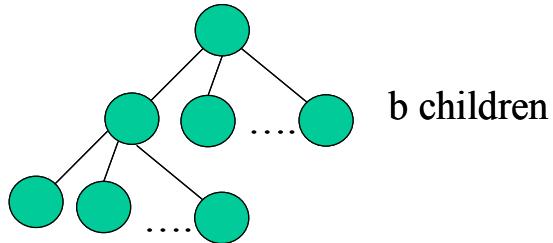


Figure 9: Model of a search tree with uniform branching factor b

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 as shown in Figure 3. Then the time and space complexity of the algorithm is $O(bd)$ where d is the depth of the solution and b is the branching factor (i.e., number of children) at each node.

A complete search tree of depth d where each non-leaf node has b children, has a total of

$$1 + b + b^2 + \dots + b^d = (b^{d+1} - 1)/(b-1) \text{ nodes}$$

Consider a complete search tree of depth 15, where every node at depths 0 to 14 has 10 children and every node at depth 15 is a leaf node. The complete search tree in this case will have $O(10^{15})$ nodes. If BFS expands 10000 nodes per second and each node uses 100 bytes of storage, then BFS will take 3500 years to run in the worst case, and it will use 11100 terabytes of memory. So you can see that the breadth first search algorithm cannot be effectively used unless the search space is quite small. You may also observe that even if you have all the time at your disposal, the search algorithm cannot run because it will run out of memory very soon.

Advantages of Breadth First Search

Finds the path of minimal length to the goal.

Disadvantages of Breadth First Search

Requires the generation and storage of a tree whose size is exponential the the depth of the shallowest goal node

2.6 Uniform-cost search

This algorithm is by Dijkstra [1959]. The algorithm expands nodes in the order of their cost from the source.

We have discussed that operators are associated with costs. The path cost is usually taken to be the sum of the step costs.

In uniform cost search the newly generated nodes are put in OPEN according to their path costs. This ensures that when a node is selected for expansion it is a node with the cheapest cost among the nodes in OPEN.

Let $g(n)$ = cost of the path from the start node to the current node n. Sort nodes by increasing value of g .

Some properties of this search algorithm are:

- Complete
- Optimal/Admissible
- Exponential time and space complexity, $O(b^d)$

2.7 Depth first Search

2.7.1 Algorithm

Depth First Search
Let $fringe$ be a list containing the initial state
Loop
if $fringe$ is empty return failure
Node \leftarrow remove-first ($fringe$)
if Node is a goal
then return the path from initial state to Node
else generate all successors of Node, and
merge the newly generated nodes into $fringe$
add generated nodes to the front of $fringe$
End Loop

The depth first search algorithm puts newly generated nodes in the front of OPEN. This results in expanding the deepest node first. Thus the nodes in OPEN follow a LIFO order (Last In First Out). OPEN is thus implemented using a stack data structure.

2.7.2 DFS illustrated

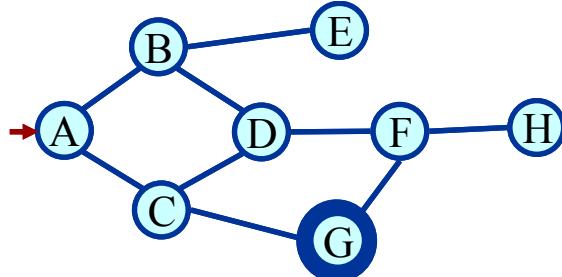


Figure 10

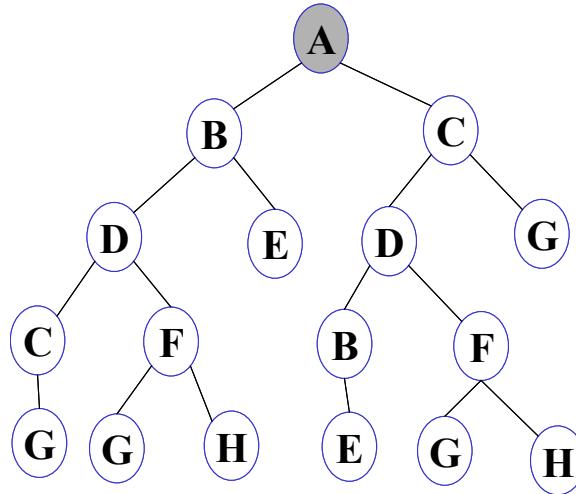
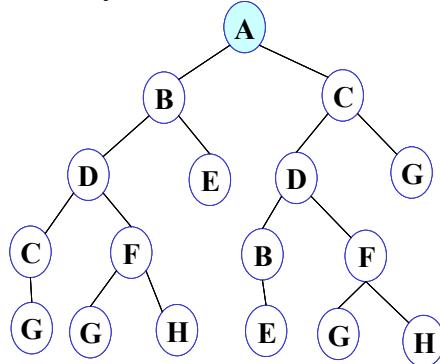


Figure 11: Search tree for the state space graph in Figure 34

Let us now run Depth First Search on the search space given in Figure 34, and trace its progress.

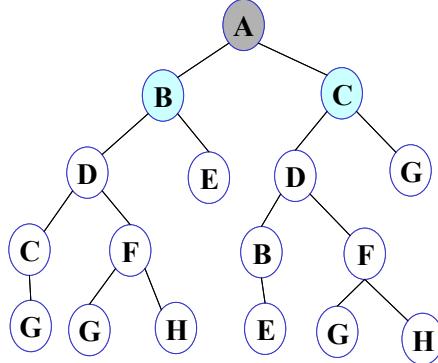
Step 1: Initially fringe contains only the node for A.



FRINGE: A

Figure 12

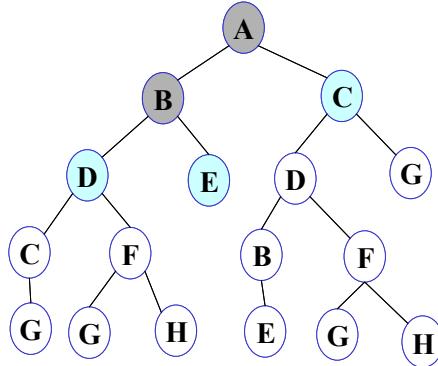
Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.



FRINGE: B C

Figure 13

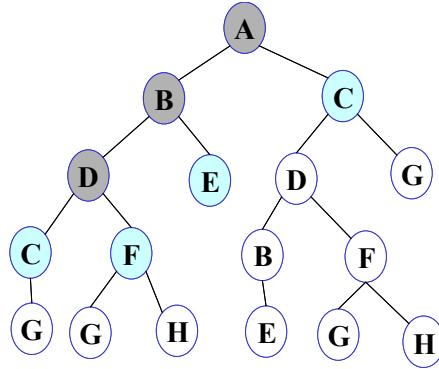
Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.



FRINGE: D E C

Figure 14

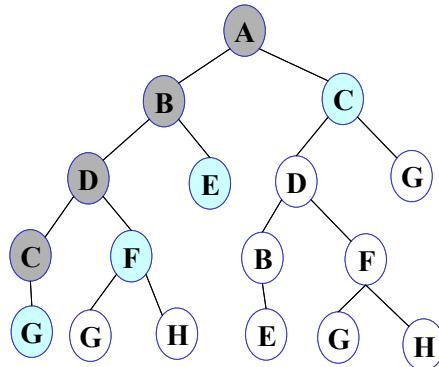
Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.



FRINGE: C F E C

Figure 15

Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.



FRINGE: G F E C

Figure 16

Step 6: Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.

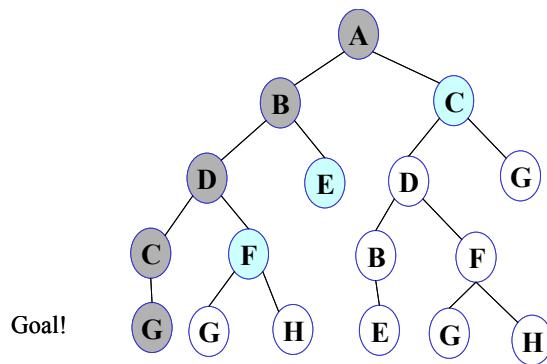


Figure 17

2.7.3 Properties of Depth First Search

Let us now examine some properties of the DFS algorithm. 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, $O(bN)$.

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.

2.7.4 Depth Limited Search

A variation of Depth First Search circumvents the above 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.

Depth limited search (limit)
Let fringe be a list containing the initial state
Loop
if fringe is empty return failure
Node \leftarrow remove-first (fringe)
if Node is a goal
then return the path from initial state to Node
else if depth of Node = limit return cutoff
else add generated nodes to the front of fringe
End Loop

2.7.5 Depth-First Iterative Deepening (DFID)

First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

DFID
<i>until solution found do</i>
<i>DFS with depth cutoff c</i>
<i>c = c+1</i>

Advantage

- Linear memory requirements of depth-first search
- Guarantee for goal node of minimal depth

Procedure

Successive depth-first searches are conducted – each with depth bounds increasing by 1

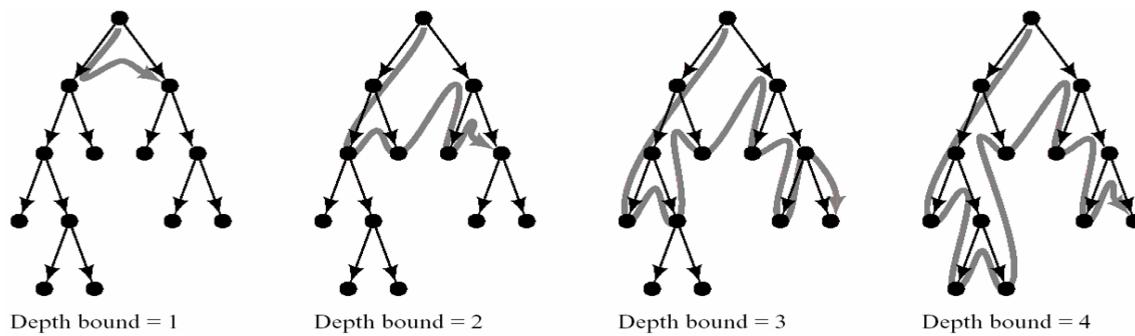


Figure 18: Depth First Iterative Deepening

Properties

For large d the ratio of the number of nodes expanded by DFID compared to that of DFS is given by $b/(b-1)$.

For a branching factor of 10 and deep goals, 11% more nodes expansion in iterative-deepening search than breadth-first search

The algorithm is

- Complete
- Optimal/Admissible if all operators have the same cost. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- Time complexity is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, $O(b^d)$

If branching factor is b and solution is at depth d , then nodes at depth d are generated once, nodes at depth $d-1$ are generated twice, etc.
Hence $b^d + 2b^{(d-1)} + \dots + db \leq b^d / (1 - 1/b)^2 = O(b^d)$.

- **Linear space complexity, $O(bd)$, like DFS**

Depth First Iterative Deepening combines the advantage of BFS (i.e., completeness) with the advantages of DFS (i.e., limited space and finds longer paths more quickly)

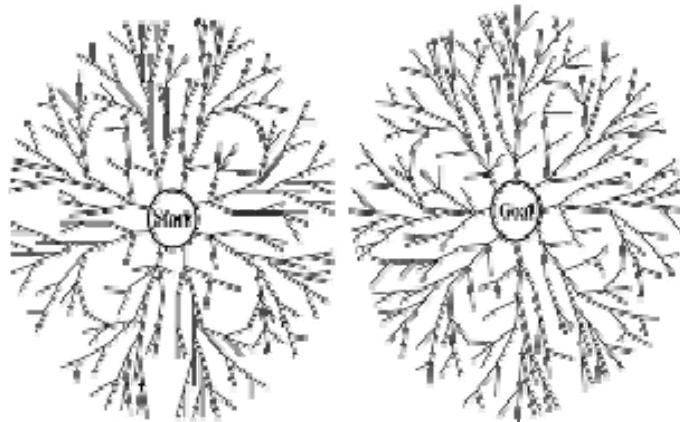
This algorithm is generally preferred for **large state spaces** where the **solution depth is unknown**.

There is a related technique called *iterative broadening* is useful when there are many goal nodes. This algorithm works by first constructing a search tree by expanding only one child per node. In the 2nd iteration, two children are expanded, and in the i th iteration I children are expanded.

Bi-directional search

Suppose that the search problem is such that the arcs are bidirectional. That is, if there is an operator that maps from state A to state B, there is another operator that maps from state B to state A. Many search problems have reversible arcs. 8-puzzle, 15-puzzle, path planning etc are examples of search problems. However there are other state space search formulations which do not have this property. The water jug problem is a problem that does not have this property. But if the arcs are reversible, you can see that instead of starting from the start state and searching for the goal, one may start from a goal state and try reaching the start state. If there is a single state that satisfies the goal property, the search problems are identical.

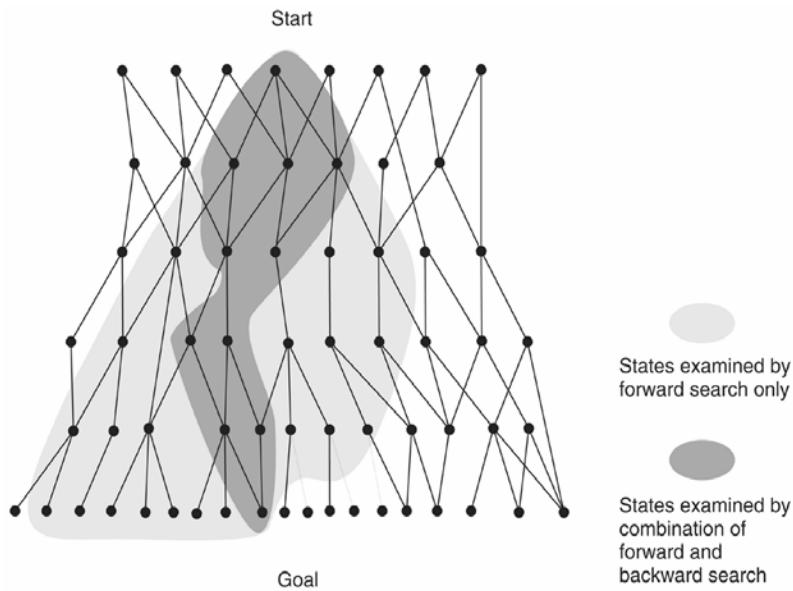
How do we search backwards from goal? One should be able to generate predecessor states. Predecessors of node n are all the nodes that have n as successor. This is the motivation to consider bidirectional search.



Algorithm: Bidirectional search involves alternate searching from the start state toward the goal and from the goal state toward the start. The algorithm stops when the frontiers intersect.

A search algorithm has to be selected for each half. How does the algorithm know when the frontiers of the search tree intersect? For bidirectional search to work well, there must be an efficient way to check whether a given node belongs to the other search tree.

Bidirectional search can sometimes lead to finding a solution more quickly. The reason can be seen from inspecting the following figure.



Also note that the algorithm works well only when there are unique start and goal states. Question: How can you make bidirectional search work if you have 2 possible goal states?

Time and Space Complexities

Consider a search space with branching factor b . Suppose that the goal is d steps away from the start state. Breadth first search will expand $O(b^d)$ nodes.

If we carry out bidirectional search, the frontiers may meet when both the forward and the backward search trees have depth = $d/2$. Suppose we have a good hash function to check for nodes in the fringe. IN this case the time for bidirectional search will be $O((b^{d/2}))$. Also note that for at least one of the searches the frontier has to be stored. So the space complexity is also $O((b^{d/2}))$.

Comparing Search Strategies

	Breadth first	Depth first	Iterative deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^d	$b^{d/2}$
Space	b^d	bd	bd	$b^{d/2}$
Optimum?	Yes	No	Yes	Yes
Complete?	Yes	No	Yes	Yes

Search Graphs

If the search space is not a tree, but a graph, the search tree may contain different nodes corresponding to the same state. It is easy to consider a pathological example to see that the search space size may be exponential in the total number of states.

In many cases we can modify the search algorithm to avoid repeated state expansions. The way to avoid generating the same state again when not required, the search algorithm can be modified to check a node when it is being generated. If another node corresponding to the state is already in OPEN, the new node should be discarded. But what if the state was in OPEN earlier but has been removed and expanded? To keep track of this phenomenon, we use another list called CLOSED, which records all the expanded nodes. The newly generated node is checked with the nodes in CLOSED too, and it is put in OPEN if the corresponding state cannot be found in CLOSED. This algorithm is outlined below:

```

Graph search algorithm
Let fringe be a list containing the initial state
Let closed be initially empty
Loop
    if fringe is empty return failure
    Node  $\leftarrow$  remove-first (fringe)
    if Node is a goal
        then return the path from initial state to Node
    else put Node in closed
        generate all successors of Node S
        for all nodes m in S
            if m is not in fringe or closed
                merge m into fringe
End Loop

```

But this algorithm is quite expensive. Apart from the fact that the CLOSED list has to be maintained, the algorithm is required to check every generated node to see if it is already there in OPEN or CLOSED. Unless there is a very efficient way to index nodes, this will require additional overhead for every node.

In many search problems, we can adopt some less computationally intense strategies. Such strategies do not stop duplicate states from being generated, but are able to reduce many of such cases.

The simplest strategy is to not return to the state the algorithm just came from. This simple strategy avoids many node re-expansions in 15-puzzle like problems.

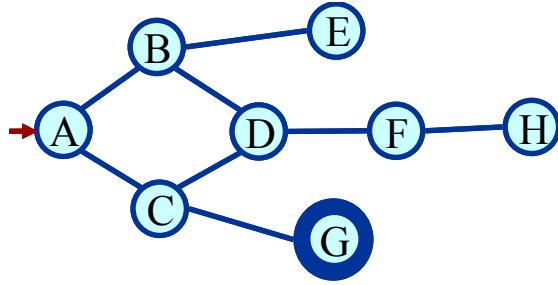
A second strategy is to check that you do not create paths with cycles in them. This algorithm only needs to check the nodes in the current path so is much more efficient than the full checking algorithm. Besides this strategy can be employed successfully with depth first search and not require additional storage.

The third strategy is as outlined in the table. Do not generate any state that was ever created before.

Which strategy one should employ must be determined by considering the frequency of “loops” in the state space.

Questions for Module 2

1. Give the initial state, goal test, successor function, and cost function for each of the following.
Choose a formulation that is precise enough to be implemented.
 - a) You have to colour a planar map using only four colours, in such a way that no two adjacent regions have the same colour.
 - b) In the travelling salesperson problem (TSP) there is a map involving N cities some of which are connected by roads. The aim is to find the shortest tour that starts from a city, visits all the cities exactly once and comes back to the starting city.
 - c) Missionaries & Cannibals problem: 3 missionaries & 3 cannibals are on one side of the river. 1 boat carries 2. Missionaries must never be outnumbered by cannibals. Give a plan for all to cross the river.
2. Given a full 5-gallon jug and an empty 2-gallon jug, the goal is to fill the 2-gallon jug with exactly one gallon of water. You may use the following state space formulation.
State = (x,y) , where x is the number of gallons of water in the 5-gallon jug and y is # of gallons in the 2-gallon jug
Initial State = $(5,0)$
Goal State = $(*,1)$, where * means any amount
Create the search tree. Discuss which search strategy is appropriate for this problem.
3. Consider the following graph.



Starting from state A, execute DFS. The goal node is G. Show the order in which the nodes are expanded. Assume that the alphabetically smaller node is expanded first to break ties.

4. Suppose you have the following search space:

State	next	cost
A	B	4
A	C	1
B	D	3
B	E	8
C	C	0
C	D	2
C	F	6
D	C	2
D	E	4
E	G	2
F	G	8

- a) Draw the state space of this problem.
- b) Assume that the initial state is **A** and the goal state is **G**. Show how each of the following search strategies would create a search tree to find a path from the initial state to the goal state:
 - I. Breadth-first search
 - II. Depth-first search
 - III. Uniform cost search
 - IV. Iterative deepening search

At each step of the search algorithm, show which node is being expanded, and the content of fringe. Also report the eventual solution found by each algorithm, and the

solution cost.

5. Suppose that breadth first search expands N nodes for a particular graph. What will be the maximum number of nodes expanded by Iterative Deepening search ?

Solutions

1. You have to colour a planar map using only four colours, in such a way that no two adjacent regions have the same colour.

The map is represented by a graph. Each region corresponds to a vertex of the graph. If two regions are adjacent, there is an edge connecting the corresponding vertices.

The vertices are named $\langle v_1, v_2, \dots, v_N \rangle$.

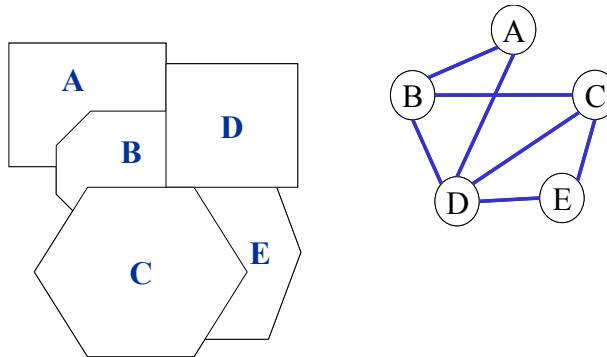
The colours are represented by c_1, c_2, c_3, c_4 .

A state is represented as a N -tuple representing the colours of the vertices. A vertex has colour x if its colour has not yet been assigned. An example state is:

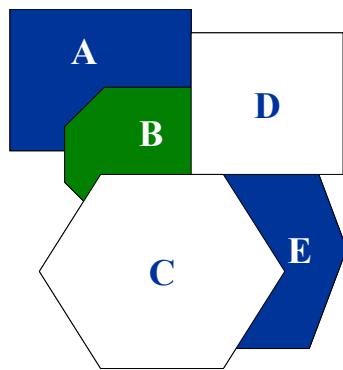
$\{c_1, x, c_1, c_3, x, x, x \dots\}$

colour(i) denotes the colour of v_i .

Consider the map below consisting of 5 regions namely A, B, C, D and E. The adjacency information is represented by the corresponding graph shown.



A state of this problem is shown below.



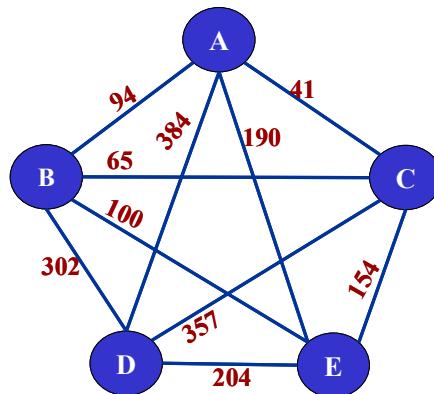
This state is represented as {blue, green, x, x, blue}.

The initial state for this problem is given as {x, x, x, x, x}

The goal test is as follows. For every pair of states s_i and s_j that are adjacent, colour(i) must be different from colour(j).

The successor functions are of the form:

- Change (i, c): Change the colour of a state i to c.
2. In the travelling salesperson problem (TSP) there is a map involving N cities some of which are connected by roads. The aim is to find the shortest tour that starts from a city, visits all the cities exactly once and comes back to the starting city.



Y: set of N cities

$d(x,y)$: distance between cities x and y. $x,y \in Y$

A state is a Hamiltonian path (does not visit any city twice)

X: set of states

X: set of states. $X =$

$\{(x_1, x_2, \dots, x_n) |$

$n=1, \dots, N+1,$

$x_i \in Y \text{ for all } i,$

$x_i \neq x_j$ unless $i=1, j=N+1\}$

Successors of state

(x_1, x_2, \dots, x_n) :

$$\delta \quad (x_1, x_2, \dots, x_n) = \{(x_1, x_2, \dots, x_n, x_{n+1}) \mid x_{n+1} \in Y \\ x_{n+1} \neq x_i \text{ for all } 1 \leq i \leq n\}$$

The set of goal states include all states of length $N+1$

3. **Missionaries & Cannibals** problem: 3 missionaries & 3 cannibals are on one side of the river. 1 boat carries 2. Missionaries must never be outnumbered by cannibals. Give a plan for all to cross the river.

State: $\langle M, C, B \rangle$

- ◆ M: no of missionaries on the left bank
- ◆ C: no of cannibals on the left bank
- ◆ B: position of the boat: L or R

4. Initial state: $\langle 3, 3, L \rangle$

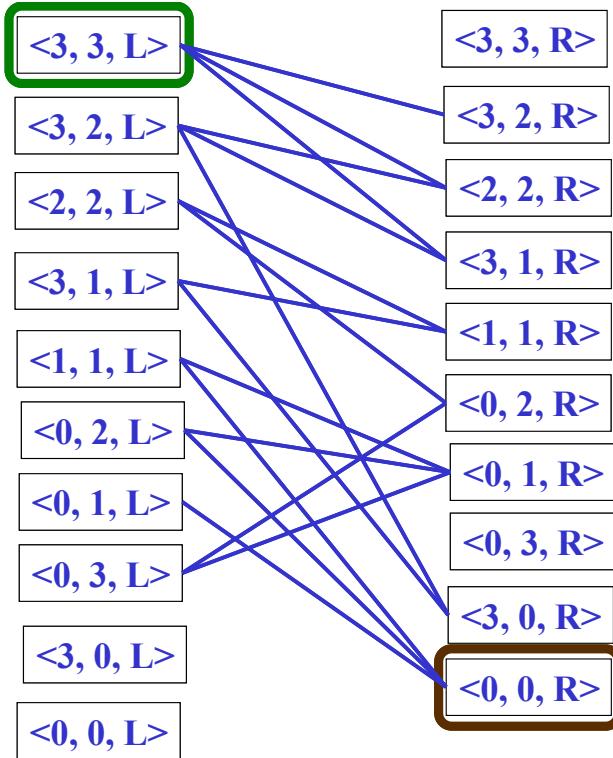
5. Goal state: $\langle 0, 0, R \rangle$

6. Operators: $\langle M, C \rangle$

- M: No of missionaries on the boat
- C: No of cannibals on the boat

Valid operators: $\langle 1, 0 \rangle, \langle 2, 0 \rangle, \langle 1, 1 \rangle, \langle 0, 1 \rangle, \langle 0, 2 \rangle$

$\langle 3, 3, L \rangle$	$\langle 3, 3, R \rangle$
$\langle 2, 3, L \rangle$	$\langle 2, 3, R \rangle$
$\langle 1, 3, L \rangle$	$\langle 1, 3, R \rangle$
$\langle 3, 2, L \rangle$	$\langle 3, 2, R \rangle$
$\langle 2, 2, L \rangle$	$\langle 2, 2, R \rangle$
$\langle 1, 2, L \rangle$	$\langle 1, 2, R \rangle$
$\langle 3, 1, L \rangle$	$\langle 3, 1, R \rangle$
$\langle 2, 1, L \rangle$	$\langle 2, 1, R \rangle$
$\langle 1, 1, L \rangle$	$\langle 1, 1, R \rangle$
$\langle 0, 2, L \rangle$	$\langle 0, 2, R \rangle$
$\langle 0, 1, L \rangle$	$\langle 0, 1, R \rangle$
$\langle 0, 3, L \rangle$	$\langle 0, 3, R \rangle$
$\langle 3, 0, L \rangle$	$\langle 3, 0, R \rangle$
$\langle 2, 0, L \rangle$	$\langle 2, 0, R \rangle$
$\langle 1, 0, L \rangle$	$\langle 1, 0, R \rangle$
$\langle 0, 0, L \rangle$	$\langle 0, 0, R \rangle$



2. Given a full 5-gallon jug and an empty 2-gallon jug, the goal is to fill the 2-gallon jug with exactly one gallon of water. You may use the following state space formulation.

- State = (x,y) , where x is the number of gallons of water in the 5-gallon jug and y is # of gallons in the 2-gallon jug
- Initial State = $(5,0)$
- Goal State = $(*,1)$, where * means any amount

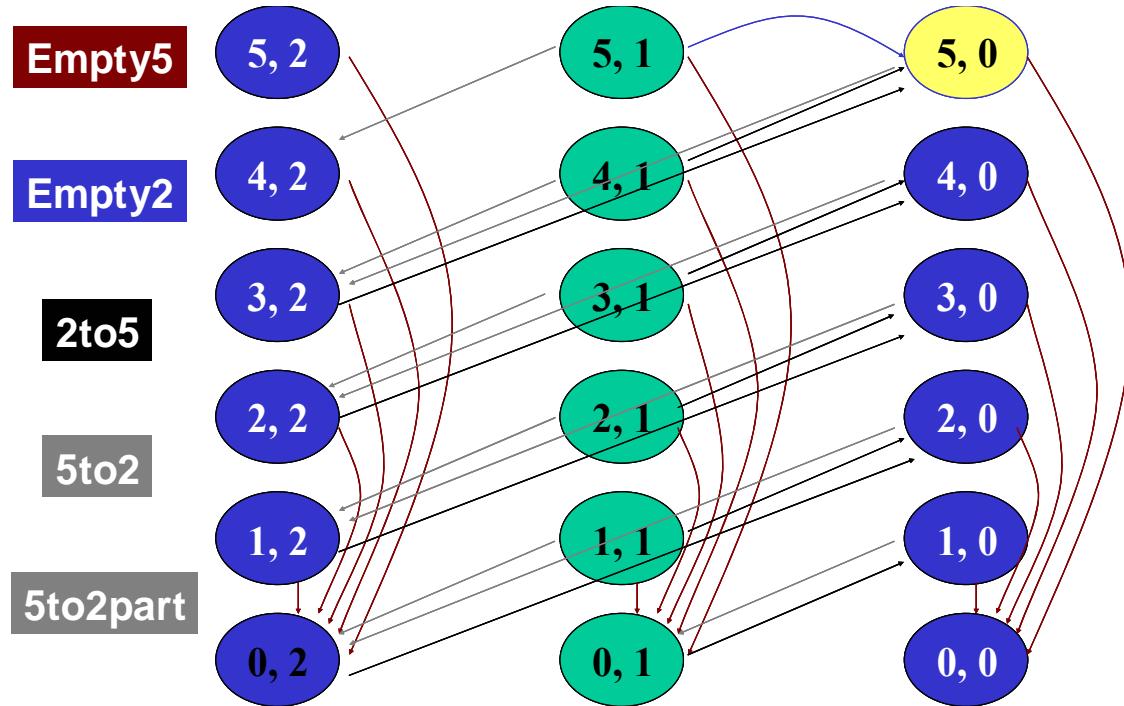
Create the search tree. Discuss which search strategy is appropriate for this problem.

Solution:

The table below shows the different operators and their effects.

Name	Cond.	Transition	Effect
Empty5	-	$(x,y) \rightarrow (0,y)$	Empty 5-gal. jug
Empty2	-	$(x,y) \rightarrow (x,0)$	Empty 2-gal. jug
2to5	$x \leq 3$	$(x,2) \rightarrow (x+2,0)$	Pour 2-gal. into 5-gal.
5to2	$x \geq 2$	$(x,0) \rightarrow (x-2,2)$	Pour 5-gal. into 2-gal.
5to2part	$y < 2$	$(1,y) \rightarrow (0,y+1)$	Pour partial 5-gal. into 2-gal.

The figure below shows the different states and the transitions between the states using the operators above. The transitions corresponding to Empty2 have not been marked to keep the figure clean.



(5,0) is the initial state.

(0,1) is the goal state.

A solution to this problem is given by the path

(5,0) – (3,2) – (3,0) – (1,2) – (1,0) – (0,1).

using the operators

5to2, Empty2, 5to2, Empty2, 5to2.

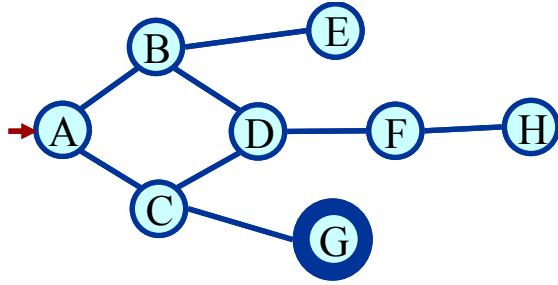
Depth search is not appropriate

The search space is a graph.

The search tree has many repeated states

Breadth first search is appropriate

3. Consider the following graph.



Starting from state A, execute DFS. The goal node is G. Show the order in which the nodes are expanded. Assume that the alphabetically smaller node is expanded first to break ties.

Solution:

Step	Fringe	Node Expanded	Comments
1	A		
2	B C	A	
3	D E C	B	
4	F E C	D	
5	H E C	F	
6	E C	H	
7	C	E	
8	D G	C	
9	F G	D	
10	H G	F	
11	G	H	
12		G	Goal reached!

Module 2

Problem Solving using Search- (Single agent search)

Lesson 5

Informed Search Strategies-I

3.1 Introduction

We have outlined the different types of search strategies. In the earlier chapter we have looked at different blind search strategies. Uninformed search methods lack problem-specific knowledge. Such methods are prohibitively inefficient in many cases. Using problem-specific knowledge can dramatically improve the search speed. In this chapter we will study some informed search algorithms that use problem specific heuristics.

Review of different Search Strategies

1. Blind Search
 - a) Depth first search
 - b) Breadth first search
 - c) Iterative deepening search
 - d) Bidirectional search
2. Informed Search

3.1.1 Graph Search Algorithm

We begin by outlining the general graph search algorithm below.

```
Graph search algorithm
Let fringe be a list containing the initial state
Let closed be initially empty
Loop
    if fringe is empty return failure
    Node ← remove-first (fringe)
        if Node is a goal
            then return the path from initial state to Node
        else put Node in closed
            generate all successors of Node S
            for all nodes m in S
                if m is not in fringe or closed
                    merge m into fringe
End Loop
```

3.1.2 Review of Uniform-Cost Search (UCS)

We will now review a variation of breadth first search we considered before, namely Uniform cost search.

To review, in uniform cost search we enqueue nodes by path cost.

Let $g(n)$ = cost of the path from the start node to the current node n .

The algorithm sorts nodes by increasing value of g , and expands the lowest cost node of the fringe.

Properties of Uniform Cost Search

- Complete
- Optimal/Admissible
- Exponential time and space complexity, $O(b^d)$

The UCS algorithm uses the value of $g(n)$ to select the order of node expansion. We will now introduce informed search or heuristic search that uses problem specific heuristic information. The heuristic will be used to select the order of node expansion.

3.1.3 Informed Search

We have seen that uninformed search methods that systematically explore the state space and find the goal. They are inefficient in most cases. Informed search methods use problem specific knowledge, and may be more efficient. At the heart of such algorithms there is the concept of a heuristic function.

3.1.3.1 Heuristics

Heuristic means “rule of thumb”. To quote Judea Pearl, “Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal”. In heuristic search or informed search, heuristics are used to identify the most promising search path.

Example of Heuristic Function

A heuristic function at a node n is an estimate of the optimum cost from the current node to a goal. It is denoted by $h(n)$.

$h(n)$ = estimated cost of the cheapest path from node n to a goal node

Example 1: We want a path from Kolkata to Guwahati

Heuristic for Guwahati may be straight-line distance between Kolkata and Guwahati

$h(Kolkata) = \text{euclideanDistance}(Kolkata, Guwahati)$

Example 2: 8-puzzle: Misplaced Tiles Heuristics is the number of tiles out of place.

2	8	3
1	6	4
	7	5

Initial State

1	2	3
8		4
7	6	5

Goal state

Figure 1: 8 puzzle

The first picture shows the current state n , and the second picture the goal state.

$$h(n) = 5$$

because the tiles 2, 8, 1, 6 and 7 are out of place.

Manhattan Distance Heuristic: Another heuristic for 8-puzzle is the Manhattan distance heuristic. This heuristic sums the distance that the tiles are out of place. The distance of a tile is measured by the sum of the differences in the x-positions and the y-positions.

For the above example, using the Manhattan distance heuristic,

$$h(n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6$$

We will now study a heuristic search algorithm best-first search.

3.2 Best First Search

Uniform Cost Search is a special case of the best first search algorithm. The algorithm maintains a priority queue of nodes to be explored. A cost function $f(n)$ is applied to each node. The nodes are put in OPEN in the order of their f values. Nodes with smaller $f(n)$ values are expanded earlier. The generic best first search algorithm is outlined below.

Best First Search

```
Let fringe be a priority queue containing the initial state
Loop
    if fringe is empty return failure
    Node  $\leftarrow$  remove-first (fringe)
        if Node is a goal
            then return the path from initial state to Node
        else generate all successors of Node, and
            put the newly generated nodes into fringe
            according to their f values
End Loop
```

We will now consider different ways of defining the function f . This leads to different search algorithms.

3.2.1 Greedy Search

In greedy search, the idea is to expand the node with the smallest estimated cost to reach the goal.

We use a heuristic function

$$f(n) = h(n)$$

$h(n)$ estimates the distance remaining to a goal.

Greedy algorithms often perform very well. They tend to find good solutions quickly, although not always optimal ones.

The resulting algorithm is not optimal. The algorithm is also incomplete, and it may fail to find a solution even if one exists. This can be seen by running greedy search on the following example. A good heuristic for the route-finding problem would be straight-line distance to the goal.

Figure 2 is an example of a route finding problem. S is the starting state, G is the goal state.

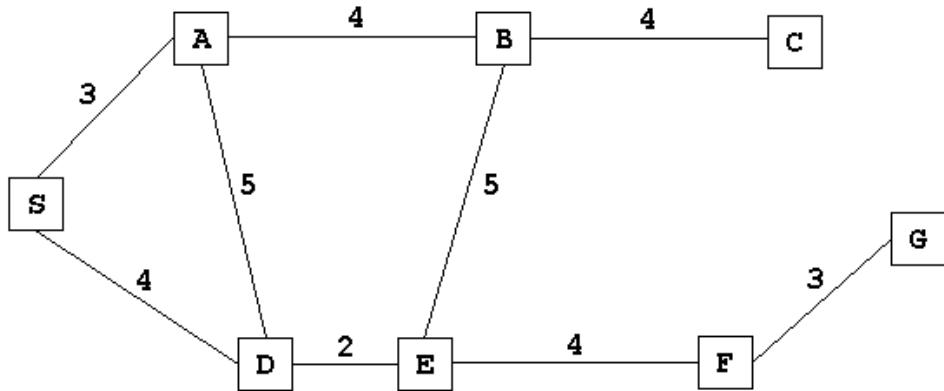


Figure 2

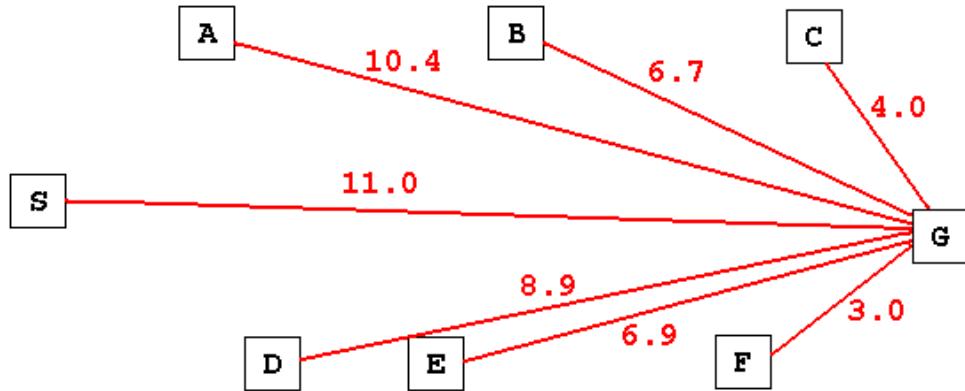
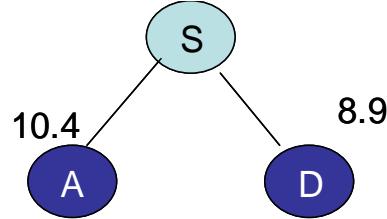


Figure 3

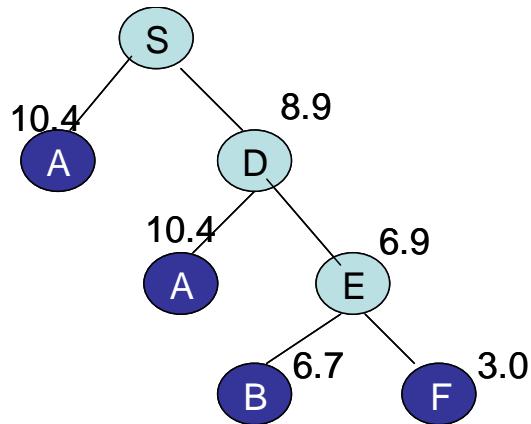
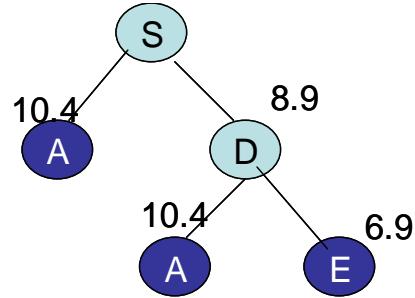
Let us run the greedy search algorithm for the graph given in Figure 2. The straight line distance heuristic estimates for the nodes are shown in Figure 3.



Step 1: S is expanded. Its children are A and D.



Step 2: D has smaller cost and is expanded next.



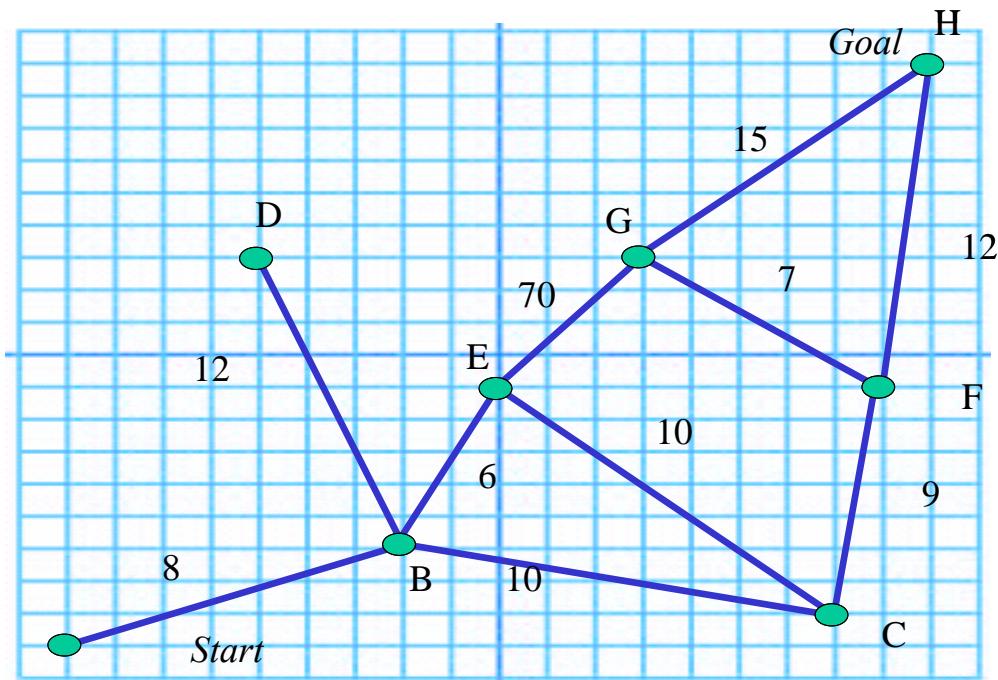
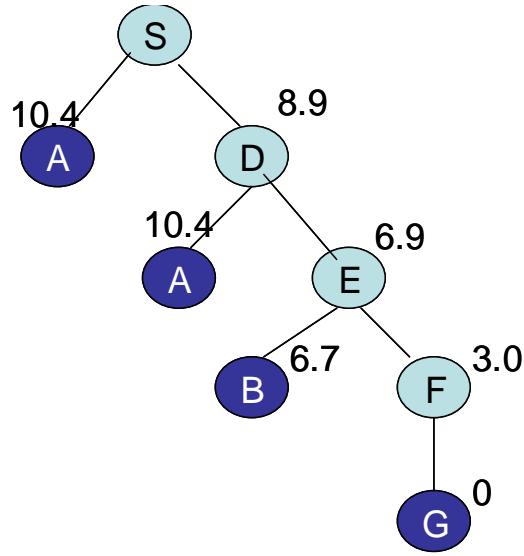


Figure 4

Greedy Best-First Search illustrated

We will run greedy best first search on the problem in Figure 2. We use the straight line heuristic. $h(n)$ is taken to be the straight line distance from n to the goal position.

The nodes will be expanded in the following order:

A
B
E
G
H

The path obtained is A-B-E-G-H and its cost is 99

Clearly this is not an optimum path. The path A-B-C-F-H has a cost of 39.

3.2.2 A* Search

We will next consider the famous A* algorithm. This algorithm was given by Hart, Nilsson & Rafael in 1968.

A* is a best first search algorithm with

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

where

$g(n)$ = sum of edge costs from start to n

$h(n)$ = estimate of lowest cost path from n to goal

$f(n)$ = actual distance so far + estimated distance remaining

$h(n)$ is said to be admissible if it underestimates the cost of any solution that can be reached from n . If $C^*(n)$ is the cost of the cheapest solution path from n to a goal node, and if h is admissible,

$$h(n) \leq C^*(n).$$

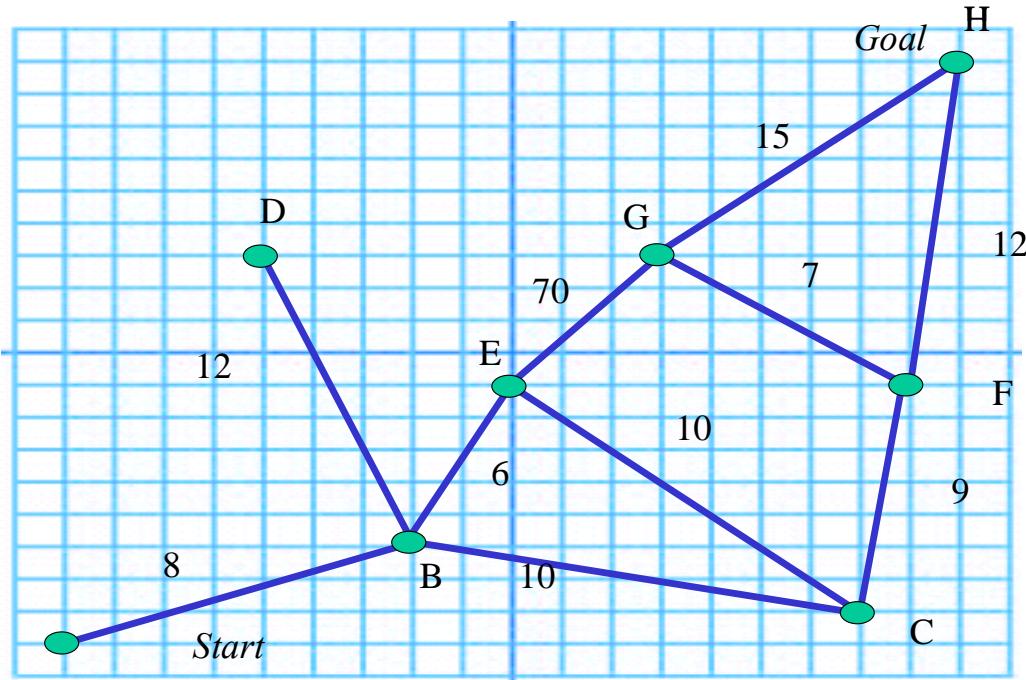
We can prove that if $h(n)$ is admissible, then the search will find an optimal solution.

The algorithm A* is outlined below:

Algorithm A*
OPEN = nodes on frontier. CLOSED = expanded nodes. OPEN = { $<s, \text{nil}>$ } while OPEN is not empty remove from OPEN the node $<n, p>$ with minimum $f(n)$ place $<n, p>$ on CLOSED if n is a goal node, return success (path p) for each edge connecting n & m with cost c if $<m, q>$ is on CLOSED and $\{p e\}$ is cheaper than q then remove n from CLOSED, put $<m, \{p e\}>$ on OPEN else if $<m, q>$ is on OPEN and $\{p e\}$ is cheaper than q then replace q with $\{p e\}$ else if m is not on OPEN then put $<m, \{p e\}>$ on OPEN

```
return failure
```

3.2.1 A* illustrated



The heuristic function used is straight line distance. The order of nodes expanded, and the status of Fringe is shown in the following table.

Steps	Fringe	Node expanded	Comments
1	A		
2	B(26.6)	A	
3	E(27.5), C(35.1), D(35.2)	B	
4	C(35.1), D(35.2), E(41.2) , G(92.5)	E	C is not inserted as there is another C with lower cost.
5	D(35.2), F(37), G(92.5)	C	
6	F(37), G(92.5)	D	
7	H(39), G(42.5)	F	G is replaced with a lower cost node
8	G(42.5)	H	Goal test successful.

The path returned is A-B-C-F-H.

The path cost is 39. This is an optimal path.

3.2.2 A* search: properties

The algorithm A* is admissible. This means that provided a solution exists, the first solution found by A* is an optimal solution. A* is admissible under the following conditions:

- In the state space graph
 - Every node has a finite number of successors
 - Every arc in the graph has a cost greater than some $\varepsilon > 0$
- Heuristic function: for every node n , $h(n) \leq h^*(n)$

A* is also complete under the above conditions.

A* is optimally efficient for a given heuristic – of the optimal search algorithms that expand search paths from the root node, it can be shown that no other optimal algorithm will expand fewer nodes and find a solution

However, the number of nodes searched still exponential in the worst case.

Unfortunately, estimates are usually not good enough for A* to avoid having to expand an exponential number of nodes to find the optimal solution. In addition, A* must keep all nodes it is considering in memory.

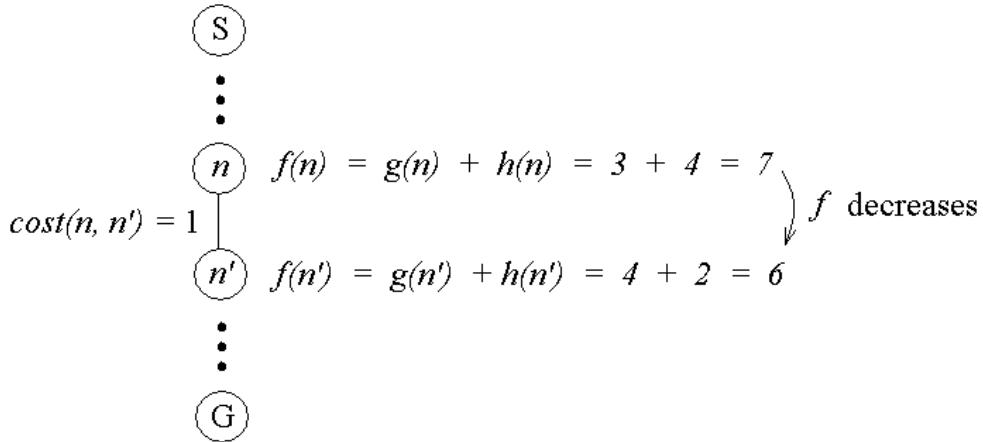
A* is still much more efficient than uninformed methods.

It is always better to use a heuristic function with higher values as long as it does not overestimate.

A heuristic is consistent if:

$$h(n) \leq cost(n, n') + h(n')$$

For example, the heuristic shown below is inconsistent, because $h(n) = 4$, but $cost(n, n') + h(n') = 1 + 2 = 3$, which is less than 4. This makes the value of f decrease from node n to node n':



If a heuristic h is consistent, the f values along any path will be nondecreasing:

$$\begin{aligned}
 f(n') &= \text{estimated distance from start to goal through } n' \\
 &= \text{actual distance from start to } n + \text{step cost from } n \text{ to } n' + \\
 &\quad \text{estimated distance from } n' \text{ to goal} \\
 &= g(n) + \text{cost}(n, n') + h(n') \\
 &\geq g(n) + h(n) \text{ because } \text{cost}(n, n') + h(n') \geq h(n) \text{ by consistency} \\
 &= f(n)
 \end{aligned}$$

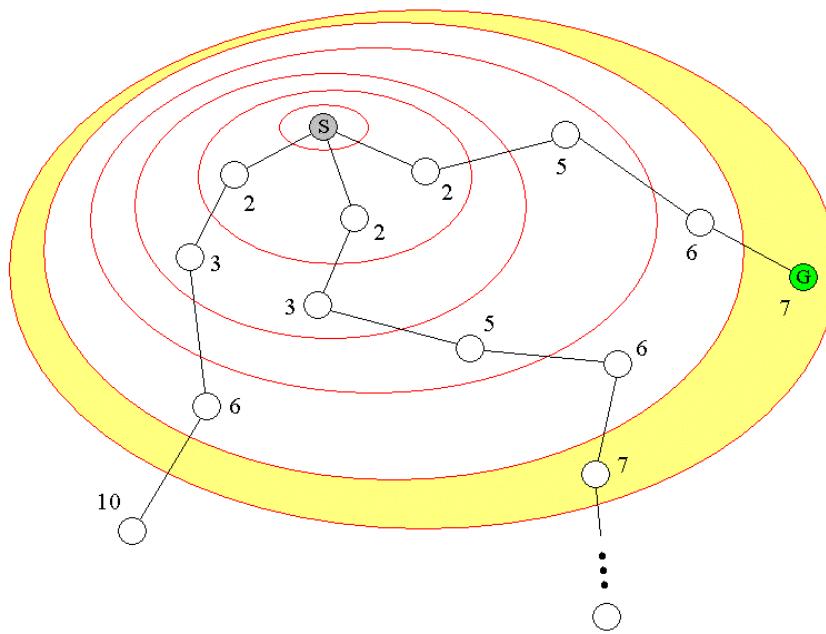
Therefore $f(n') \geq f(n)$, so f never decreases along a path.

If a heuristic h is inconsistent, we can tweak the f values so that they behave as if h were consistent, using the **pathmax equation**:

$$f(n') = \max(f(n), g(n') + h(n'))$$

This ensures that the f values never decrease along a path from the start to a goal.

Given nondecreasing values of f , we can think of A* as searching outward from the start node through successive **contours** of nodes, where all of the nodes in a contour have the same f value:



For any contour, A* examines all of the nodes in the contour before looking at any contours further out. If a solution exists, the goal node in the closest contour to the start node will be found first.

We will now prove the admissibility of A*.

3.2.3 Proof of Admissibility of A*

We will show that A* is admissible if it uses a monotone heuristic.

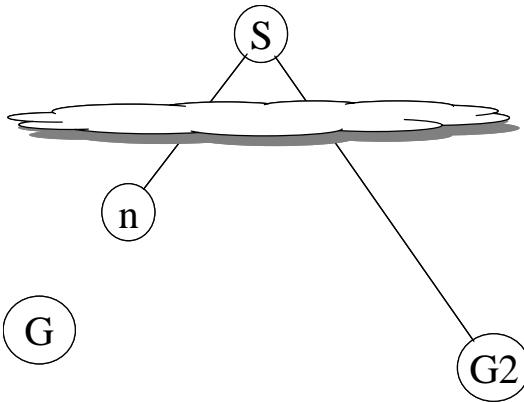
A monotone heuristic is such that along any path the f-cost never decreases.

But if this property does not hold for a given heuristic function, we can make the f value monotone by making use of the following trick (m is a child of n)

$$f(m) = \max(f(n), g(m) + h(m))$$

- o Let G be an optimal goal state
- o C^* is the optimal path cost.
- o G_2 is a suboptimal goal state: $g(G_2) > C^*$

Suppose A* has selected G_2 from OPEN for expansion.



Consider a node n on OPEN on an optimal path to G . Thus $C^* \geq f(n)$

Since n is not chosen for expansion over $G2$, $f(n) \geq f(G2)$

$G2$ is a goal state. $f(G2) = g(G2)$

Hence $C^* \geq g(G2)$.

This is a contradiction. Thus A* could not have selected $G2$ for expansion before reaching the goal by an optimal path.

3.2.4 Proof of Completeness of A*

Let G be an optimal goal state.

A* cannot reach a goal state only if there are infinitely many nodes where $f(n) \leq C^*$.

This can only happen if either happens:

- o There is a node with infinite branching factor. The first condition takes care of this.
- o There is a path with finite cost but infinitely many nodes. But we assumed that Every arc in the graph has a cost greater than some $\epsilon > 0$. Thus if there are infinitely many nodes on a path $g(n) > f^*$, the cost of that path will be infinite.

Lemma: A* expands nodes in increasing order of their f values.

A* is thus **complete** and **optimal**, assuming an admissible and consistent heuristic function (or using the pathmax equation to simulate consistency).

A* is also **optimally efficient**, meaning that it expands only the minimal number of nodes needed to ensure optimality and completeness.

3.2.4 Performance Analysis of A*

Model the search space by a uniform b -ary tree with a unique start state s , and a goal state, g at a distance N from s .

The number of nodes expanded by A* is exponential in N unless the heuristic estimate is logarithmically accurate

$$|h(n) - h^*(n)| \leq O(\log h^*(n))$$

In practice most heuristics have proportional error.

It becomes often difficult to use A* as the OPEN queue grows very large.
A solution is to use algorithms that work with less memory.

3.2.5 Properties of Heuristics

Dominance:

h_2 is said to dominate h_1 iff $h_2(n) \geq h_1(n)$ for any node n .
A* will expand fewer nodes on average using h_2 than h_1 .

Proof:

Every node for which $f(n) < C^*$ will be expanded. Thus n is expanded whenever

$$h(n) < f^* - g(n)$$

Since $h_2(n) \geq h_1(n)$ any node expanded using h_2 will be expanded using h_1 .

3.2.6 Using multiple heuristics

Suppose you have identified a number of non-overestimating heuristics for a problem:
 $h_1(n), h_2(n), \dots, h_k(n)$

Then

$$\max(h_1(n), h_2(n), \dots, h_k(n))$$

is a more powerful non-overestimating heuristic. This follows from the property of dominance

Module 2

Problem Solving using Search- (Single agent search)

Lesson 6

Informed Search Strategies-II

3.3 Iterative-Deepening A*

3.3.1 IDA* Algorithm

Iterative deepening A* or IDA* is similar to iterative-deepening depth-first, but with the following modifications:

The depth bound modified to be an f-limit

1. Start with limit = $h(\text{start})$
2. Prune any node if $f(\text{node}) > f\text{-limit}$
3. Next f-limit = minimum cost of any node pruned

The cut-off for nodes expanded in an iteration is decided by the f-value of the nodes.

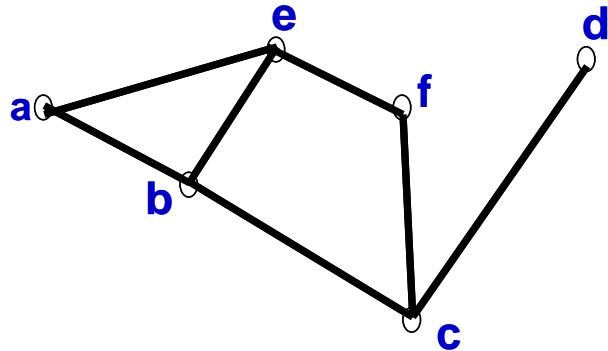


Figure 1

Consider the graph in Figure 3. In the first iteration, only node a is expanded. When a is expanded b and e are generated. The f value of both are found to be 15.

For the next iteration, a f-limit of 15 is selected, and in this iteration, a, b and c are expanded. This is illustrated in Figure 4.

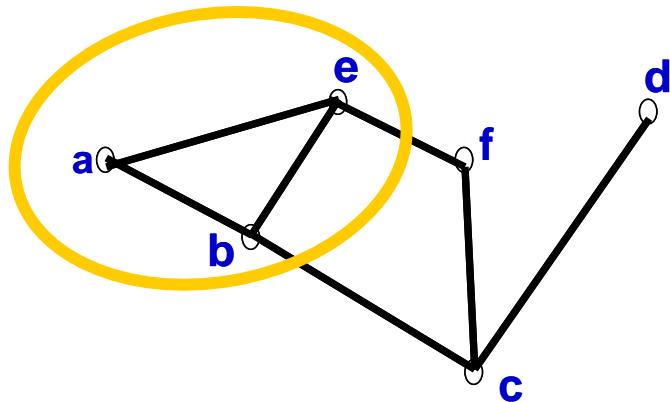


Figure 2: f-limit = 15

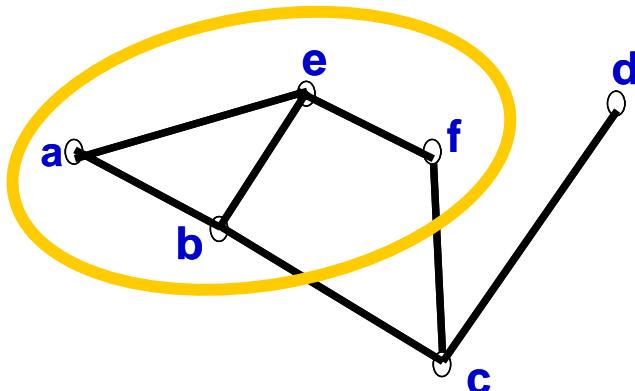


Figure 3: f-limit = 21

3.3.2 IDA* Analysis

IDA* is complete & optimal Space usage is linear in the depth of solution. Each iteration is depth first search, and thus it does not require a priority queue.

The number of nodes expanded relative to A* depends on # unique values of heuristic function. The number of iterations is equal to the number of distinct f values less than or equal to C*.

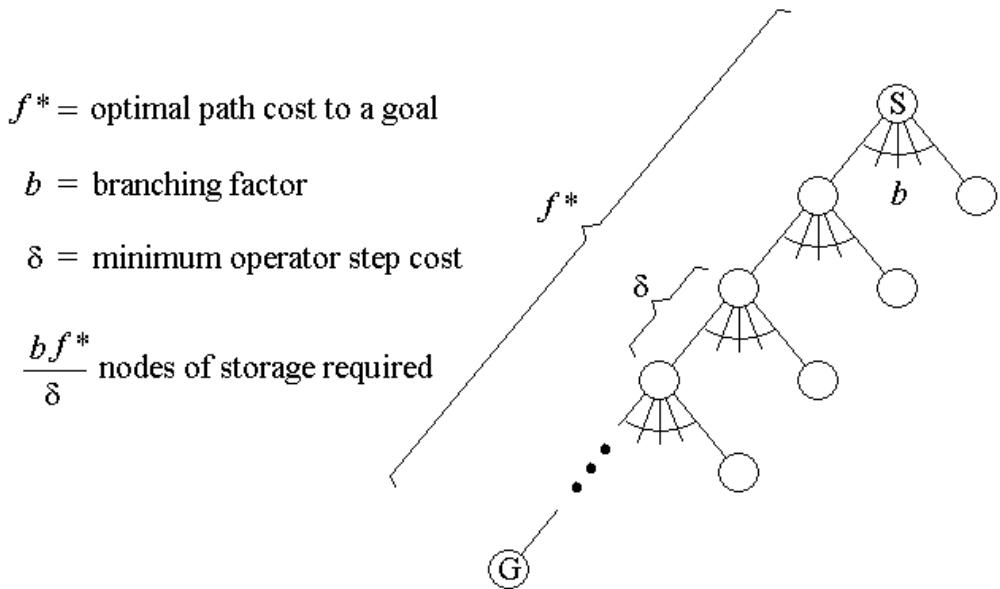
- In problems like 8 puzzle using the Manhattan distance heuristic, there are few possible f values (f values are only integral in this case.). Therefore the number of node expansions in this case is close to the number of nodes A* expands.
- But in problems like traveling salesman (TSP) using real valued costs, each f value may be unique, and many more nodes may need to be expanded. In the worst case, if all f values are distinct, the algorithm will expand only one new node per iteration, and thus if A* expands N nodes, the maximum number of nodes expanded by IDA* is $1+2+\dots+N = O(N^2)$

Why do we use IDA*? In the case of A*, it is usually the case that for slightly larger problems, the algorithm runs out of main memory much earlier than the algorithm runs out of time. IDA* can be used in such cases as the space requirement is linear. In fact 15-puzzle problems can be easily solved by IDA*, and may run out of space on A*.

IDA* is not thus suitable for TSP type of problems. Also IDA* generates duplicate nodes in cyclic graphs. Depth first search strategies are not very suitable for graphs containing too many cycles.

Space required : $O(bd)$

IDA* is complete, optimal, and optimally efficient (assuming a consistent, admissible heuristic), and requires only a polynomial amount of storage in the worst case:



3.4 Other Memory limited heuristic search

IDA* uses very little memory

Other algorithms may use more memory for more efficient search.

3.4.1 RBFS: Recursive Breadth First Search

RBFS uses only linear space.

It mimics best first search.

It keeps track of the f-value of the best alternative path available from any ancestor of the current node.

If the current node exceeds this limit, the alternative path is explored.

RBFS remembers the f-value of the best leaf in the forgotten sub-tree.

```
RBFS (node: N, value: F(N), bound: B)
```

```
IF f(N)>B, RETURN f(N)
IF N is a goal, EXIT algorithm
IF N has no children, RETURN infinity
FOR each child Ni of N,
    IF f(N)<F(N), F[i] := MAX(F(N),f(Ni))
    ELSE F[i] := f(Ni)
sort Ni and F[i] in increasing order of F[i]
IF only one child, F[2] := infinity
WHILE (F[1] <= B and F[1] < infinity)
    F[1] := RBFS(N1, F[1], MIN(B, F[2]))
    insert Ni and F[1] in sorted order
RETURN F[1]
```

3.4.2 MA* and SMA*

MA* and SMA* are restricted memory best first search algorithms that utilize all the memory available.

The algorithm executes best first search while memory is available.

When the memory is full the worst node is dropped but the value of the forgotten node is backed up at the parent.

3.5 Local Search

Local search methods work on complete state formulations. They keep only a small number of nodes in memory.

Local search is useful for solving optimization problems:

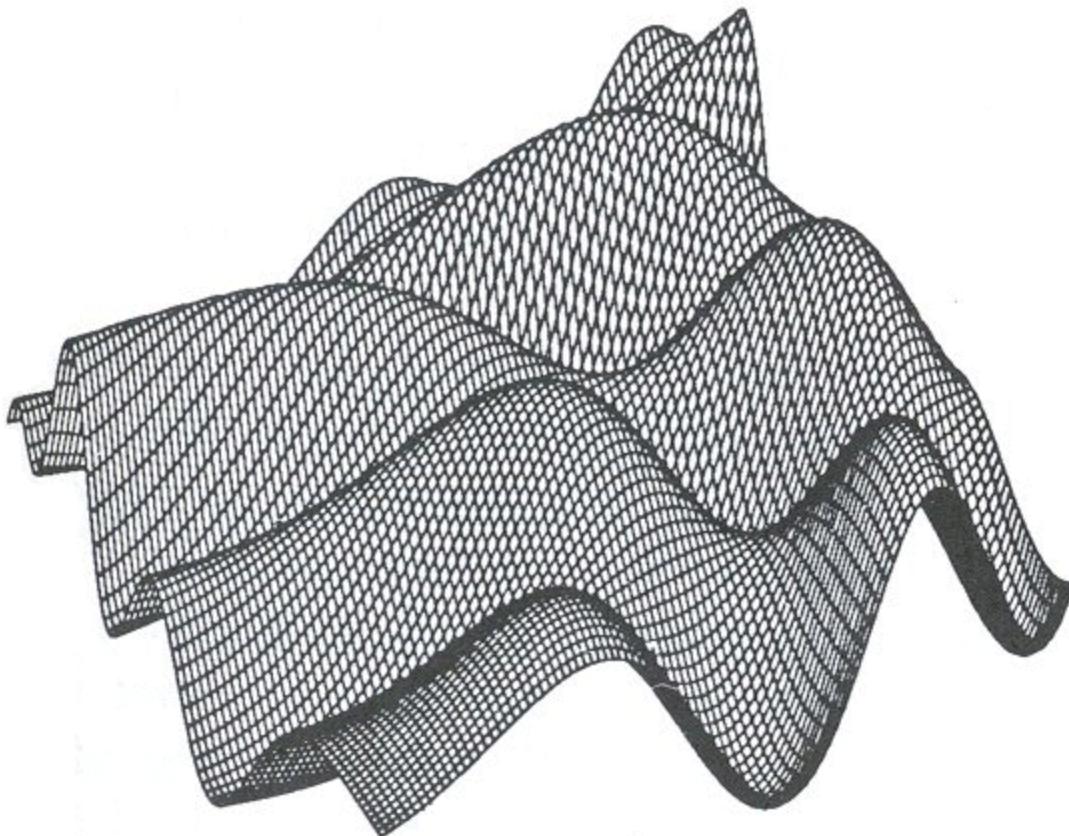
- Often it is easy to find a solution
- But hard to find the best solution

Algorithm goal:

find optimal configuration (e.g., TSP),

- Hill climbing
- Gradient descent
- Simulated annealing
- For some problems the state description contains all of the information relevant for a solution. Path to the solution is unimportant.
- Examples:
 - map coloring
 - 8-queens
 - cryptarithmetic

- Start with a state configuration that violates some of the constraints for being a solution, and make gradual modifications to eliminate the violations.
- One way to visualize iterative improvement algorithms is to imagine every possible state laid out on a landscape with the height of each state corresponding to its goodness. Optimal solutions will appear as the highest points. Iterative improvement works by moving around on the landscape seeking out the peaks by looking only at the local vicinity.



3.5.1 Iterative improvement

In many optimization problems, the path is irrelevant; the goal state itself is the solution. An example of such problem is to find configurations satisfying constraints (e.g., n-queens).

Algorithm:

- Start with a solution
- Improve it towards a good solution

3.5.1.1 Example:

N queens

Goal: Put n chess-game queens on an n x n board, with no two queens on the same row, column, or diagonal.

Example:

Chess board reconfigurations

Here, goal state is initially unknown but is specified by constraints that it must satisfy

Hill climbing (or gradient ascent/descent)

Iteratively maximize “value” of current state, by replacing it by successor state that has highest value, as long as possible.

Note: minimizing a “value” function $v(n)$ is equivalent to maximizing $-v(n)$,

thus both notions are used interchangeably.

Hill climbing – example

Complete state formulation for 8 queens

Successor function: move a single queen to another square in the same column

Cost: number of pairs that are attacking each other.

Minimization problem

Hill climbing (or gradient ascent/descent)

- *Iteratively maximize “value” of current state, by replacing it by successor state that has highest value, as long as possible.*

Note: minimizing a “value” function $v(n)$ is equivalent to maximizing $-v(n)$, thus both notions are used interchangeably.

- Algorithm:
 1. determine successors of current state
 2. choose successor of maximum goodness (break ties randomly)
 3. if goodness of best successor is less than current state's goodness, stop
 4. otherwise make best successor the current state and go to step 1
- No search tree is maintained, only the current state.
- Like greedy search, but only states directly reachable from the current state are considered.
- Problems:

Local maxima

Once the top of a hill is reached the algorithm will halt since every possible step leads down.

Plateaux

If the landscape is flat, meaning many states have the same goodness, algorithm degenerates to a random walk.

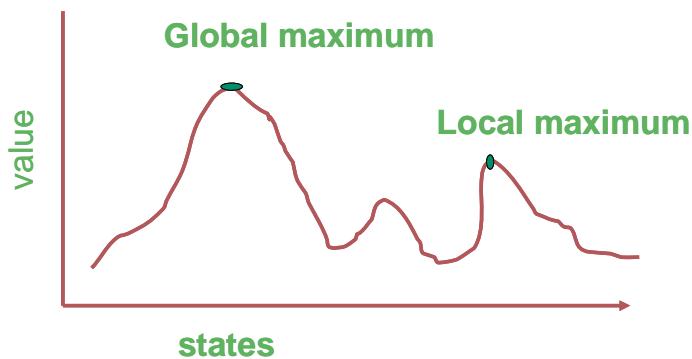
Ridges

If the landscape contains ridges, local improvements may follow a zigzag path up the ridge, slowing down the search.

- Shape of state space landscape strongly influences the success of the search process. A very spiky surface which is flat in between the spikes will be very difficult to solve.
- Can be combined with nondeterministic search to recover from local maxima.
- **Random-restart hill-climbing** is a variant in which reaching a local maximum causes the current state to be saved and the search restarted from a random point. After several restarts, return the best state found. With enough restarts, this method will find the optimal solution.
- **Gradient descent** is an inverted version of hill-climbing in which better states are represented by lower *cost* values. Local *minima* cause problems instead of local maxima.

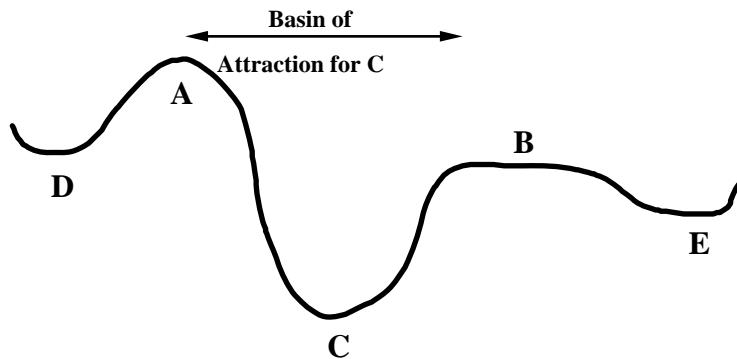
Hill climbing - example

- *Complete state formulation for 8 queens*
 - Successor function: move a single queen to another square in the same column
 - Cost: number of pairs that are attacking each other.
- *Minimization problem*
- *Problem: depending on initial state, may get stuck in local extremum.*



Minimizing energy

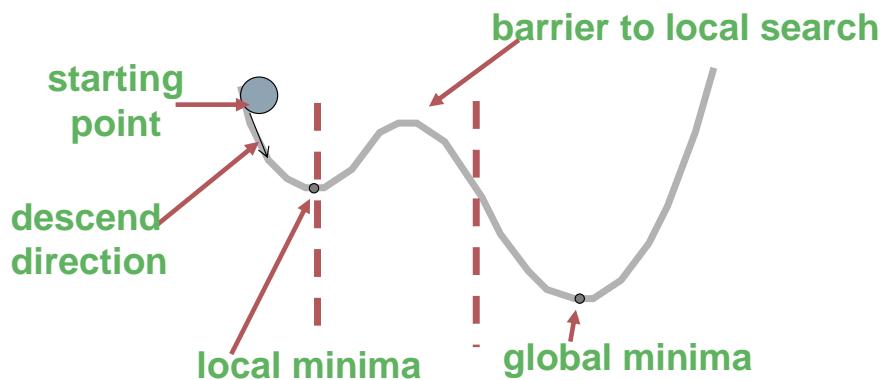
- Compare our state space to that of a physical system that is subject to natural interactions
- Compare our value function to the overall potential energy E of the system.
- On every updating, we have $\Delta E \leq 0$



Hence the dynamics of the system tend to move E toward a minimum.

We stress that there may be different such states — they are *local* minima. Global minimization is not guaranteed.

- *Question: How do you avoid this local minima?*



Consequences of Occasional Ascents

Simulated annealing: basic idea

- *From current state, pick a random successor state;*
- *If it has better value than current state, then “accept the transition,” that is, use successor state as current state;*

Simulated annealing: basic idea

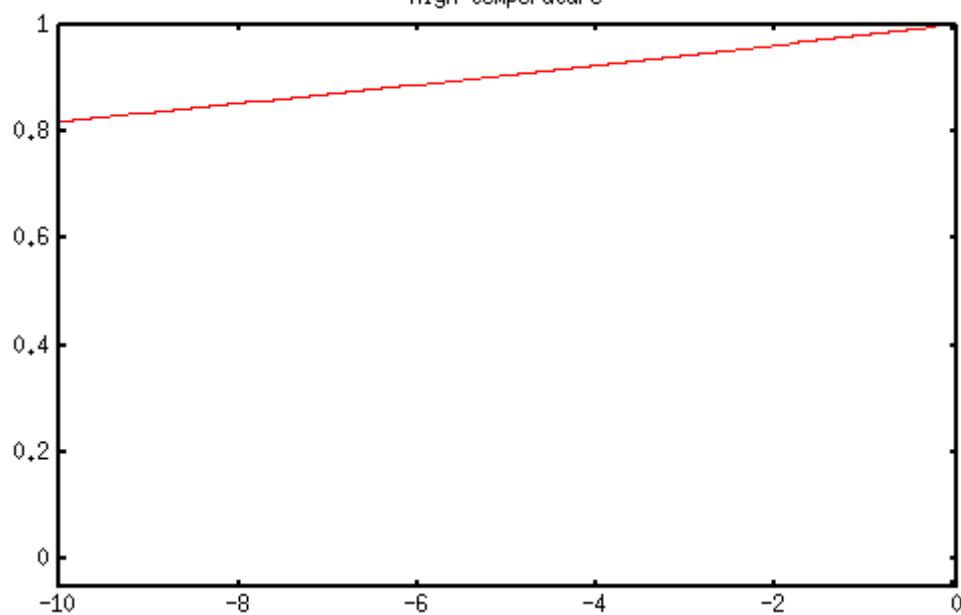
- Otherwise, do not give up, but instead flip a coin and accept the transition with a given probability (that is lower as the successor is worse).
 - So we accept to sometimes “un-optimize” the value function a little with a non-zero probability.
-
- Instead of restarting from a random point, we can allow the search to take some downhill steps to try to escape local maxima.
 - Probability of downward steps is controlled by **temperature** parameter.
 - High temperature implies high chance of trying locally “bad” moves, allowing nondeterministic exploration.
 - Low temperature makes search more deterministic (like hill-climbing).
 - Temperature begins high and gradually decreases according to a predetermined **annealing schedule**.
 - Initially we are willing to try out lots of possible paths, but over time we gradually settle in on the most promising path.
 - If temperature is lowered slowly enough, an optimal solution will be found.
 - In practice, this schedule is often too slow and we have to accept suboptimal solutions.

Algorithm:

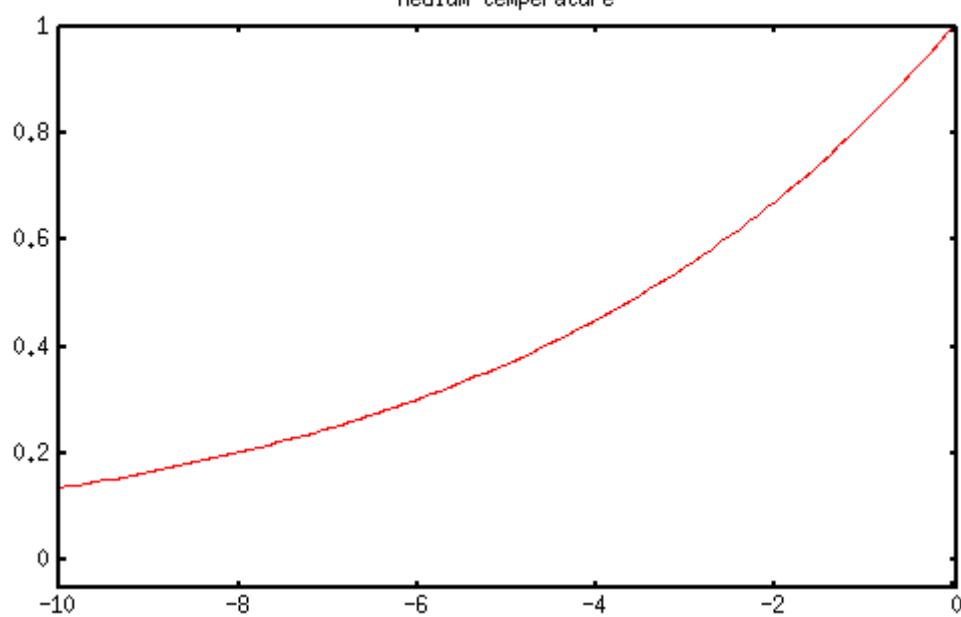
```
set current to start state
for time = 1 to infinity {
    set Temperature to annealing_schedule[time]
    if Temperature = 0 {
        return current
    }
    randomly pick a next state from successors of current
    set ΔE to value(next) - value(current)
    if ΔE > 0 {
        set current to next
    } else {
        set current to next with probability eΔE/Temperature
    }
}
```

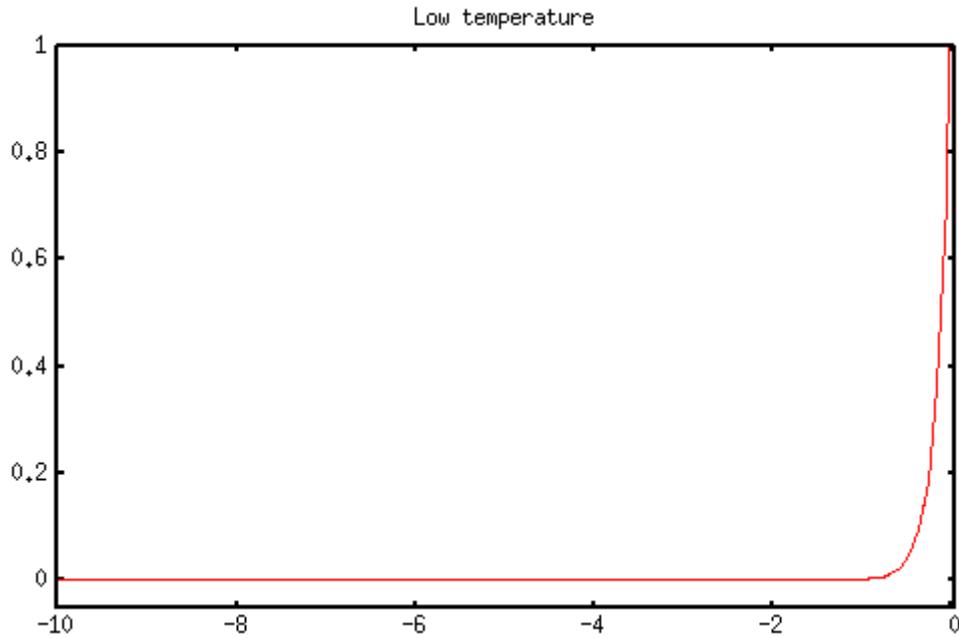
- Probability of moving downhill for negative ΔE values at different temperature ranges:

High temperature



Medium temperature





Other local search methods

- *Genetic Algorithms*

Questions for Lecture 6

1. Compare IDA* with A* in terms of time and space complexity.
 2. Is hill climbing guaranteed to find a solution to the n-queens problem ?
 3. Is simulated annealing guaranteed to find the optimum solution of an optimization problem like TSP ?
1. Suppose you have the following search space:

State	next	cost
A	B	4
A	C	1
B	D	3
B	E	8
C	C	0
C	D	2
C	F	6
D	C	2
D	E	4
E	G	2
F	G	8

- a. Draw the state space of this problem.
- b. Assume that the initial state is **A** and the goal state is **G**. Show how each of the following search strategies would create a search tree to find a path from the initial state to the goal state:
 - i. Uniform cost search
 - ii. Greedy search
 - iii. A* search

At each step of the search algorithm, show which node is being expanded, and the content of fringe. Also report the eventual solution found by each algorithm, and the solution cost.

Module 3

Problem Solving using Search- (Two agent)

3.1 Instructional Objective

- The students should understand the formulation of multi-agent search and in detail two-agent search.
- Students should be familiar with game trees.
- Given a problem description, the student should be able to formulate it in terms of a two-agent search problem.
- The student should be familiar with the minimax algorithms, and should be able to code the algorithm.
- Students should understand heuristic scoring functions and standard strategies for generating heuristic scores.
- Students should understand alpha-beta pruning algorithm, specifically its
 - Computational advantage
 - Optimal node ordering
- Several advanced heuristics used in modern game playing systems like detection of quiescent states, lengthening should be understood.
- A chess playing program will be analyzed in detail.

At the end of this lesson the student should be able to do the following:

- Analyze a given problem and formulate it as a two-agent search problem
- Given a problem, apply possible strategies for two-agent search to design a problem solving agent.

Lesson 7

Adversarial Search

3.2 Adversarial Search

We will set up a framework for formulating a multi-person game as a search problem. We will consider games in which the players alternate making moves and try respectively to maximize and minimize a scoring function (also called utility function). To simplify things a bit, we will only consider games with the following two properties:

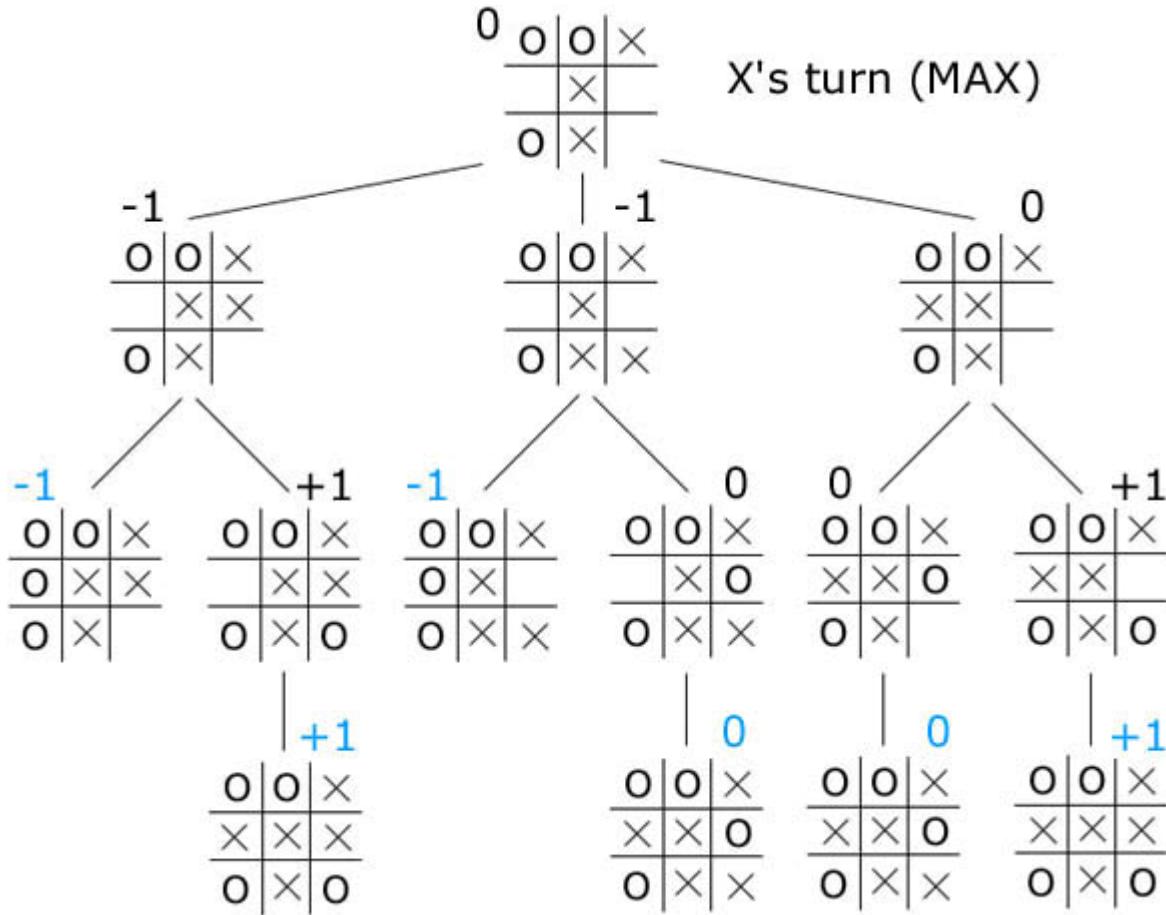
- Two player - we do not deal with coalitions, etc.
- Zero sum - one player's win is the other's loss; there are no cooperative victories

We also consider only perfect information games.

3.3 Game Trees

The above category of games can be represented as a tree where the nodes represent the current state of the game and the arcs represent the moves. The game tree consists of all possible moves for the current players starting at the root and all possible moves for the next player as the children of these nodes, and so forth, as far into the future of the game as desired. Each individual move by one player is called a "ply". The leaves of the game tree represent terminal positions as one where the outcome of the game is clear (a win, a loss, a draw, a payoff). Each terminal position has a score. High scores are good for one of the player, called the MAX player. The other player, called MIN player, tries to minimize the score. For example, we may associate 1 with a win, 0 with a draw and -1 with a loss for MAX.

Example : Game of Tic-Tac-Toe



Above is a section of a game tree for tic tac toe. Each node represents a board position, and the children of each node are the legal moves from that position. To score each position, we will give each position which is favorable for player 1 a positive number (the more positive, the more favorable). Similarly, we will give each position which is favorable for player 2 a negative number (the more negative, the more favorable). In our tic tac toe example, player 1 is 'X', player 2 is 'O', and the only three scores we will have are +1 for a win by 'X', -1 for a win by 'O', and 0 for a draw. Note here that the blue scores are the only ones that can be computed by looking at the current position.

3.4 Minimax Algorithm

Now that we have a way of representing the game in our program, how do we compute our optimal move? We will assume that the opponent is rational; that is, the opponent can compute moves just as well as we can, and the opponent will always choose the optimal move with the assumption that we, too, will play perfectly. One algorithm for computing the best move is the minimax algorithm:

```

minimax(player,board)
    if(game over in current board position)
        return winner
    children = all legal moves for player from this board
    if(max's turn)
        return maximal score of calling minimax on all the children
    else (min's turn)
        return minimal score of calling minimax on all the children

```

If the game is over in the given position, then there is nothing to compute; minimax will simply return the score of the board. Otherwise, minimax will go through each possible child, and (by recursively calling itself) evaluate each possible move. Then, the best possible move will be chosen, where ‘best’ is the move leading to the board with the most positive score for player 1, and the board with the most negative score for player 2.

How long does this algorithm take? For a simple game like tic tac toe, not too long - it is certainly possible to search all possible positions. For a game like Chess or Go however, the running time is prohibitively expensive. In fact, to completely search either of these games, we would first need to develop interstellar travel, as by the time we finish analyzing a move the sun will have gone nova and the earth will no longer exist. Therefore, all real computer games will search, not to the end of the game, but only a few moves ahead. Of course, now the program must determine whether a certain board position is ‘good’ or ‘bad’ for a certain player. This is often done using an *evaluation function*. This function is the key to a strong computer game. The depth bound search may stop just as things get interesting (e.g. in the middle of a piece exchange in chess). For this reason, the depth bound is usually extended to the end of an exchange to an quiescent state. The search may also tend to postpone bad news until after the depth bound leading to the *horizon effect*.

Module 3

Problem Solving using Search- (Two agent)

Lesson 8

Two agent games :
alpha beta pruning

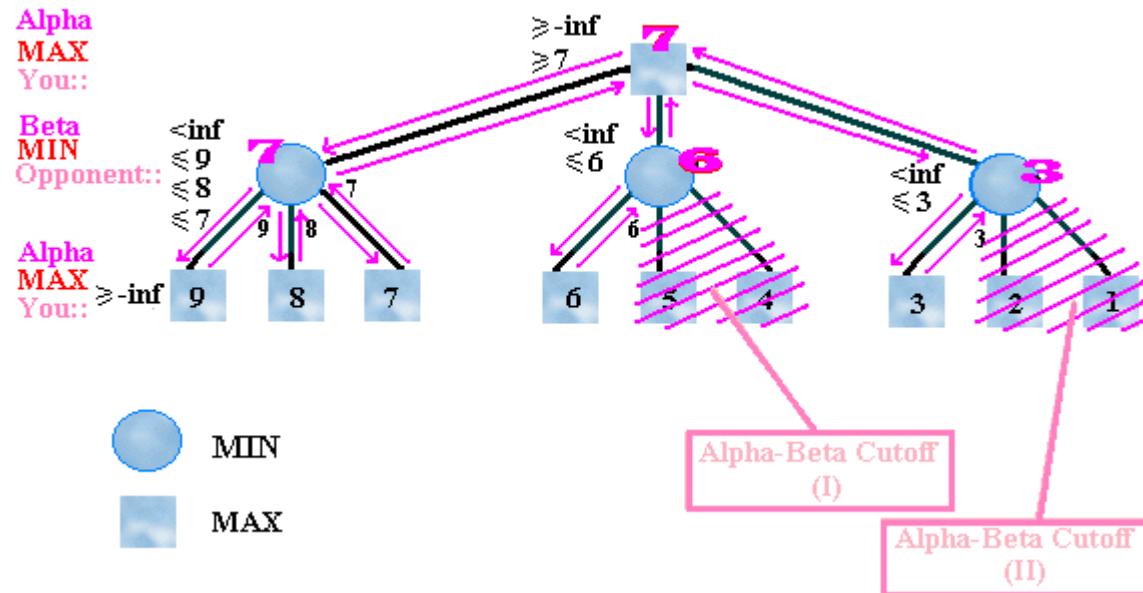
3.5 Alpha-Beta Pruning

ALPHA-BETA pruning is a method that reduces the number of nodes explored in Minimax strategy. It reduces the time required for the search and it must be restricted so that no time is to be wasted searching moves that are obviously bad for the current player. The exact implementation of alpha-beta keeps track of the best move for each side as it moves throughout the tree.

We proceed in the same (preorder) way as for the minimax algorithm. For the **MIN** nodes, the score computed starts with **+infinity** and decreases with time. For **MAX** nodes, scores computed starts with **-infinity** and increase with time.

The efficiency of the *Alpha-Beta* procedure depends on the order in which successors of a node are examined. If we were lucky, at a MIN node we would always consider the nodes in order from low to high score and at a MAX node the nodes in order from high to low score. In general it can be shown that in the most favorable circumstances the alpha-beta search opens as many leaves as minimax on a game tree with double its depth.

Here is an example of Alpha-Beta search:



3.5.1 Alpha-Beta algorithm:

The algorithm maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of respectively. Initially alpha is negative infinity and beta is positive infinity. As the recursion progresses the "window" becomes smaller.

When beta becomes less than alpha, it means that the current position cannot be the result of best play by both players and hence need not be explored further.

Pseudocode for the alpha-beta algorithm is given below.

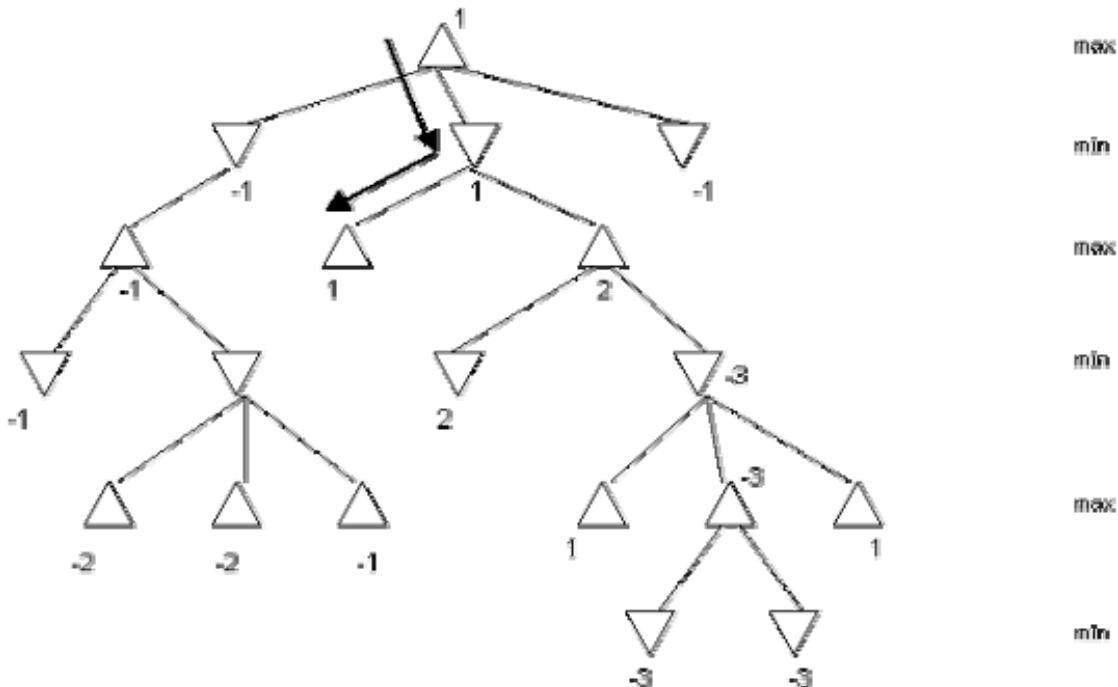
```

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

```

Questions

- Suppose you and a friend of yours are playing a board game. It is your turn to move, and the tree below represents your situation. The values of the evaluation function at the leaf nodes are shown in the tree. Note that in this tree not all leaf nodes are at the same level. Use the minimax procedure to select your next move. Show your work on the tree below.



2. In the above tree use the minimax procedure along with alpha-beta pruning to select the next move. Mark the nodes that don't need to be evaluated.
3. Consider the game of tic-tac-toe. Assume that X is the MAX player. Let the utility of a win for X be 10, a loss for X be -10, and a draw be 0.
- a) Given the game board **board1** below where it is X's turn to play next, show the entire game tree. Mark the utilities of each terminal state and use the minimax algorithm to calculate the optimal move.
- b) Given the game board **board2** below where it is X's turn to play next, show the game tree with a cut-off depth of two ply (i.e., stop after each player makes one move). Use the following evaluation function on all leaf nodes:

$$\text{Eval}(s) = 10X_3(s) + 3X_2(s) + X_1(s) - (10O_3(s) + 3O_2(s) + O_1(s))$$

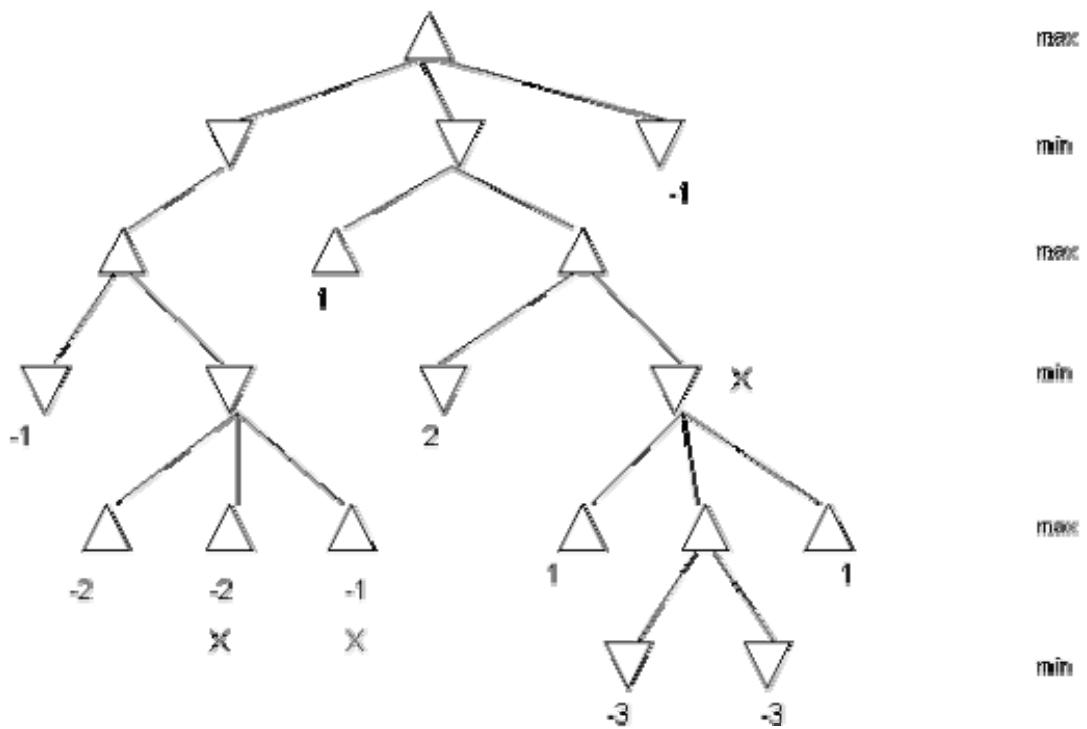
where we define $X_n(s)$ as the number of rows, columns, or diagonals in state s with exactly n X's and no O's, and similarly define $O_n(s)$ as the number of rows, columns, or diagonals in state s with exactly n O's and no X's. Use the minimax algorithm to determine X's best move.

board1		
X	O	X
O		O
	X	

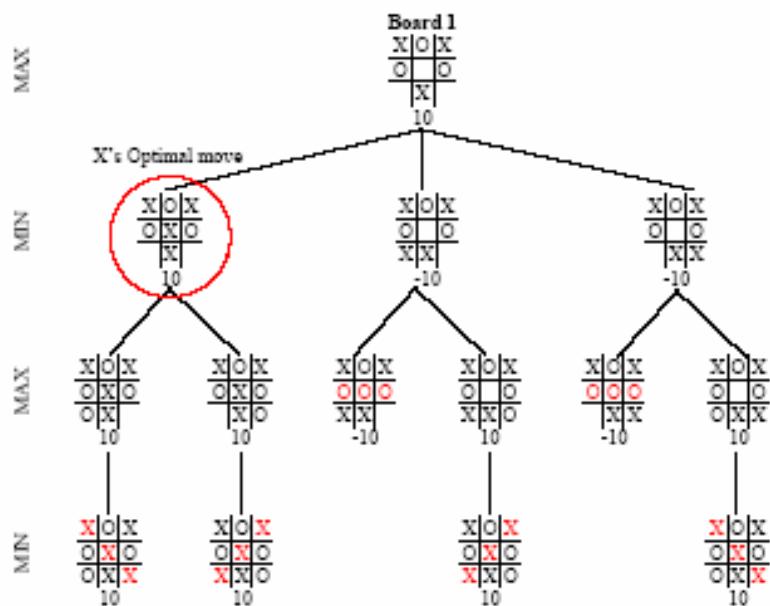
board2		
	X	
O		
X		O

Solution

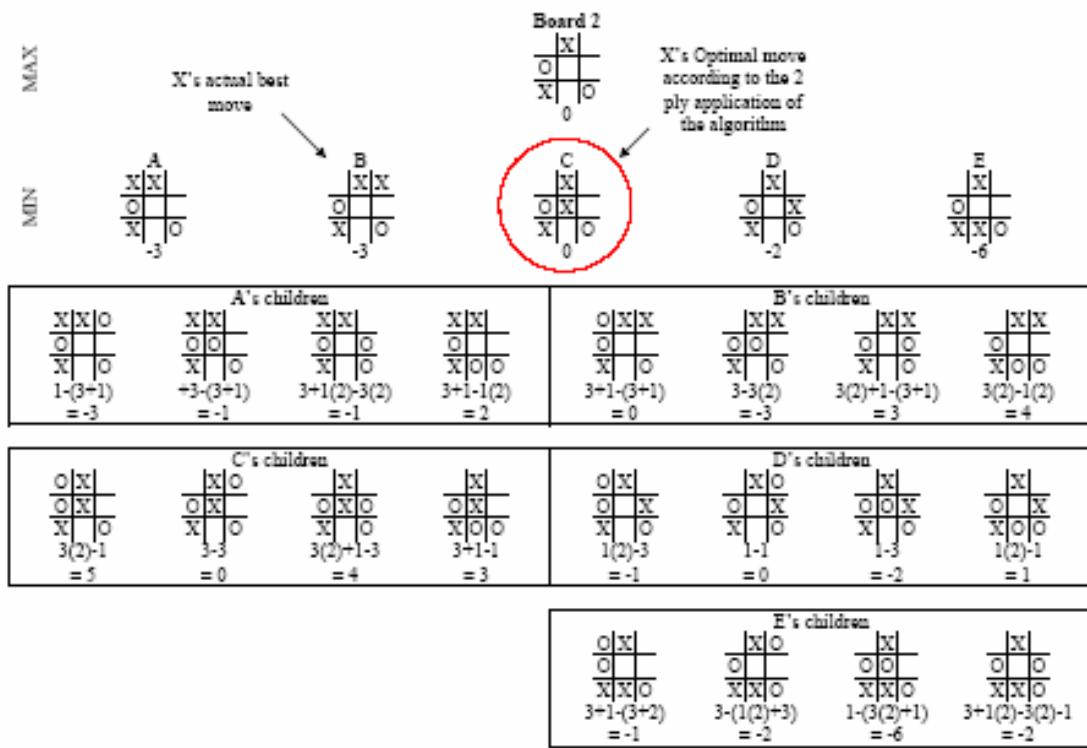
1. The second child of the root node as your next move since that child produces the highest gain.
2. Nodes marked with an "X" are not evaluated when alpha-beta pruning is used. None of the nodes in the subtree rooted at the node marked with an "X" in the third level of the tree are evaluated.



3.a. The game tree is shown below



3.b. The values are shown below



Module 4

Constraint satisfaction problems

4.1 Instructional Objective

- The students should understand the formulation of constraint satisfaction problems
- Given a problem description, the student should be able to formulate it in terms of a constraint satisfaction problem, in terms of constraint graphs.
- Students should be able to solve constraint satisfaction problems using various algorithms.
- The student should be familiar with the following algorithms, and should be able to code the algorithms
 - Backtracking
 - Forward checking
 - Constraint propagation
 - Arc consistency and path consistency
 - Variable and value ordering
 - Hill climbing

The student should be able to understand and analyze the properties of these algorithms in terms of

- time complexity
- space complexity
- termination
- optimality
- Be able to apply these search techniques to a given problem whose description is provided.
- Students should have knowledge about the relation between CSP and SAT

At the end of this lesson the student should be able to do the following:

- Formulate a problem description as a CSP
- Analyze a given problem and identify the most suitable search strategy for the problem.
- Given a problem, apply one of these strategies to find a solution for the problem.

Lesson 9

Constraint satisfaction problems - I

4.2 Constraint Satisfaction Problems

Constraint satisfaction problems or **CSPs** are mathematical problems where one must find states or objects that satisfy a number of *constraints* or criteria. A constraint is a restriction of the feasible solutions in an optimization problem.

Many problems can be stated as constraints satisfaction problems. *Here are some examples:*

Example 1: The n-Queen problem is the problem of putting n chess queens on an $n \times n$ chessboard such that none of them is able to capture any other using the standard chess queen's moves. The colour of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal.

The problem was originally proposed in 1848 by the chess player Max Bazzel, and over the years, many mathematicians, including Gauss have worked on this puzzle. In 1874, S. Gunther proposed a method of finding solutions by using determinants, and J.W.L. Glaisher refined this approach.

The eight queens puzzle has 92 **distinct** solutions. If solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one, the puzzle has 12 **unique** solutions. The following table gives the number of solutions for n queens, both unique and distinct.

$n:$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
unique:	1	0	0	1	2	1	6	12	46	92	341	1,787	9,233	45,752	285,053
distinct:	1	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	2,279,184

Note that the 6 queens puzzle has, interestingly, fewer solutions than the 5 queens puzzle!

Example 2: A crossword puzzle: We are to complete the puzzle

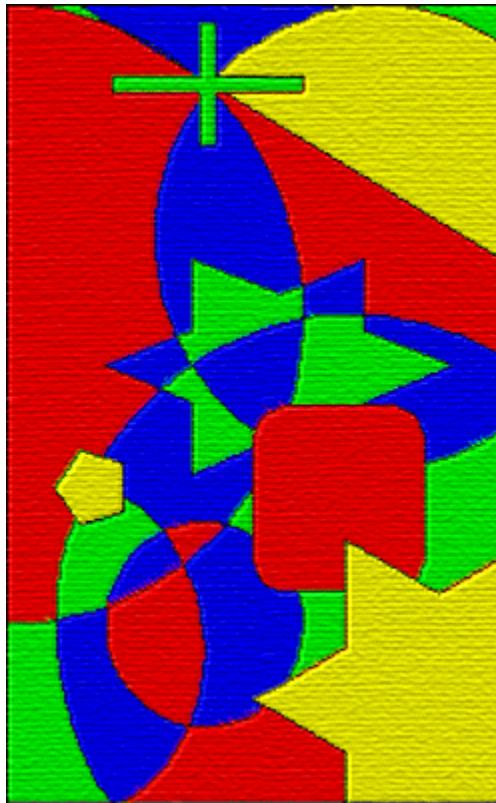
1	2	3	4	5					
1		1		2			3		
2		#		#		#			
3		#		4			5		
4		6		#		7			
5		8							
6			#		#			#	

Given the list of words:

AFT	LASER
ALE	LEE
EEL	LINE
HEEL	SAILS
HIKE	SHEET
HOSES	STEER
KEEL	TIE
KNOT	

The numbers 1,2,3,4,5,6,7,8 in the crossword puzzle correspond to the words that will start at those locations.

Example 3: A map coloring problem: We are given a map, i.e. a planar graph, and we are told to color it using k colors, so that no two neighboring countries have the same color. Example of a four color map is shown below:



The **four color theorem** states that given any plane separated into regions, such as a political map of the countries of a state, the regions may be colored using no more than four colors in such a way that no two adjacent regions receive the same color. Two regions are called *adjacent* if they share a border segment, not just a point. Each region must be contiguous: that is, it may not consist of separate sections like such real countries as Angola, Azerbaijan, and the United States.

It is obvious that three colors are inadequate: this applies already to the map with one region surrounded by three other regions (even though with an even number of surrounding countries three colors are enough) and it is not at all difficult to prove that five colors are sufficient to color a map.

The four color theorem was the first major theorem to be proved using a computer, and the proof is not accepted by all mathematicians because it would be infeasible for a human to verify by hand. Ultimately, one has to have faith in the correctness of the compiler and hardware executing the program used for the proof. The lack of

mathematical elegance was another factor, and to paraphrase comments of the time, "a good mathematical proof is like a poem — this is a telephone directory!"

Example 4: The Boolean satisfiability problem (SAT) is a decision problem considered in complexity theory. An instance of the problem is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true?

In mathematics, a formula of propositional logic is said to be **satisfiable** if truth-values can be assigned to its variables in a way that makes the formula true. The class of satisfiable propositional formulas is NP-complete. The propositional satisfiability problem (SAT), which decides whether or not a given propositional formula is satisfiable, is of central importance in various areas of computer science, including theoretical computer science, algorithmics, artificial intelligence, hardware design and verification.

The problem can be significantly restricted while still remaining NP-complete. By applying De Morgan's laws, we can assume that NOT operators are only applied directly to variables, not expressions; we refer to either a variable or its negation as a *literal*. For example, both x_1 and $\text{not}(x_2)$ are literals, the first a *positive literal* and the second a *negative literal*. If we OR together a group of literals, we get a *clause*, such as $(x_1 \text{ or } \text{not}(x_2))$. Finally, let us consider formulas that are a conjunction (AND) of clauses. We call this form conjunctive normal form. Determining whether a formula in this form is satisfiable is still NP-complete, even if each clause is limited to at most three literals. This last problem is called 3CNFSAT, 3SAT, or 3-satisfiability.

On the other hand, if we restrict each clause to at most two literals, the resulting problem, 2SAT, is in P. The same holds if every clause is a Horn clause; that is, it contains at most one positive literal.

Example 5: A cryptarithmetic problem: In the following pattern

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

we have to replace each letter by a distinct digit so that the resulting sum is correct.

All these examples and other real life problems like time table scheduling, transport scheduling, floor planning etc are instances of the same pattern, captured by the following definition:

A **Constraint Satisfaction Problem** (CSP) is characterized by:

- a set of variables $\{x_1, x_2, \dots, x_n\}$,
- for each variable x_i a domain D_i with the possible values for that variable, and

- a set of *constraints*, i.e. relations, that are assumed to hold between the values of the variables. [These relations can be given intentionally, i.e. as a formula, or extensionally, i.e. as a set, or procedurally, i.e. with an appropriate generating or recognising function.] We will only consider constraints involving one or two variables.

The constraint satisfaction problem is to find, for each i from 1 to n , a value in D_i for x_i so that all constraints are satisfied.

A CSP can easily be stated as a sentence in first order logic, of the form:

$$(\text{exist } x_1) \dots (\text{exist } x_n) (D_1(x_1) \wedge \dots \wedge D_n(x_n) \Rightarrow C_1 \dots C_m)$$

4.3 Representation of CSP

A CSP is usually represented as an undirected graph, called ***Constraint Graph*** where the nodes are the variables and the edges are the binary constraints. Unary constraints can be disposed of by just redefining the domains to contain only the values that satisfy all the unary constraints. Higher order constraints are represented by hyperarcs.

A constraint can affect any number of variables from 1 to n (n is the number of variables in the problem). If all the constraints of a CSP are binary, the variables and constraints can be represented in a constraint graph and the constraint satisfaction algorithm can exploit the graph search techniques.

The conversion of arbitrary CSP to an equivalent binary CSP is based on the idea of introducing a new variable that encapsulates the set of constrained variables. This newly introduced variable, we call it an encapsulated variable, has assigned a domain that is a Cartesian product of the domains of individual variables. Note, that if the domains of individual variables are finite than the Cartesian product of the domains, and thus the resulting domain, is still finite.

Now, arbitrary n -ary constraint can be converted to equivalent unary constraint that constrains the variable which appears as an encapsulation of the original individual variables. As we mentioned above, this unary constraint can be immediately satisfied by reducing the domain of encapsulated variable. Briefly speaking, n -ary constraint can be substituted by an encapsulated variable with the domain corresponding to the constraint.

This is interesting because any constraint of higher arity can be expressed in terms of binary constraints. Hence, binary CSPs are representative of all CSPs.

Example 2 revisited: We introduce a variable to represent each word in the puzzle. So we have the variables:

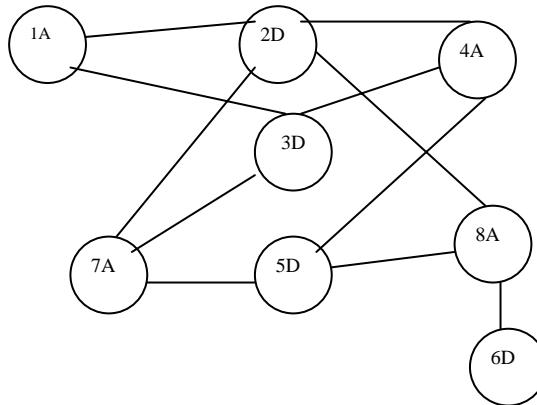
VARIABLE	STARTING CELL	DOMAIN
1ACROSS	1	{HOSES, LASER, SAILS, SHEET, STEER}
4ACROSS	4	{HEEL, HIKE, KEEL, KNOT, LINE}
7ACROSS	7	{AFT, ALE, EEL, LEE, TIE}
8ACROSS	8	{HOSES, LASER, SAILS, SHEET, STEER}
2DOWN	2	{HOSES, LASER, SAILS, SHEET, STEER}
3DOWN	3	{HOSES, LASER, SAILS, SHEET, STEER}
5DOWN	5	{HEEL, HIKE, KEEL, KNOT, LINE}
6DOWN	6	{AFT, ALE, EEL, LEE, TIE}

The domain of each variable is the list of words that may be the value of that variable. So, variable 1ACROSS requires words with five letters, 2DOWN requires words with five letters, 3DOWN requires words with four letters, etc. Note that since each domain has 5 elements and there are 8 variables, the total number of states to consider in a naive approach is $5^8 = 390,625$.

The constraints are all binary constraints:

1ACROSS[3] = 2DOWN[1] i.e. the third letter of 1ACROSS must be equal to the first letter of 2DOWN
 1ACROSS[5] = 3DOWN[1]
 4ACROSS[2] = 2DOWN[3]
 4ACROSS[3] = 5DOWN[1]
 4ACROSS[4] = 3DOWN[3]
 7ACROSS[1] = 2DOWN[4]
 7ACROSS[2] = 5DOWN[2]
 7ACROSS[3] = 3DOWN[4]
 8ACROSS[1] = 6DOWN[2]
 8ACROSS[3] = 2DOWN[5]
 8ACROSS[4] = 5DOWN[3]
 8ACROSS[5] = 3DOWN[5]

The corresponding graph is:



4.4 Solving CSPs

Next we describe four popular solution methods for CSPs, namely, *Generate-and-Test*, *Backtracking*, *Consistency Driven*, and *Forward Checking*.

4.4.1 Generate and Test

We generate one by one all possible complete variable assignments and for each we test if it satisfies all constraints. The corresponding program structure is very simple, just nested loops, one per variable. In the innermost loop we test each constraint. In most situation this method is intolerably slow.

4.4.2 Backtracking

We order the variables in some fashion, trying to place first the variables that are more highly constrained or with smaller ranges. This order has a great impact on the efficiency of solution algorithms and is examined elsewhere. We start assigning values to variables. We check constraint satisfaction at the earliest possible time and extend an assignment if the constraints involving the currently bound variables are satisfied.

Example 2 Revisited: In our crossword puzzle we may order the variables as follows: 1ACROSS, 2DOWN, 3DOWN, 4ACROSS, 7ACROSS, 5DOWN, 8ACROSS, 6DOWN. Then we start the assignments:

```
1ACROSS      <- HOSES
2DOWN        <- HOSES      => failure, 1ACROSS[3] not equal to
2DOWN[1]
                  <- LASER      => failure
                  <- SAILS
3DOWN        <- HOSES      => failure
                  <- LASER      => failure
                  <- SAILS
4ACROSS      <- HEEL       => failure
                  <- HIKE       => failure
                  <- KEEL       => failure
                  <- KNOT       => failure
                  <- LINE       => failure, backtrack
3DOWN        <- SHEET
4ACROSS      <- HEEL
7ACROSS      <- AFT        => failure
....
```

What we have shown is called *Chronological Backtracking*, whereby variables are unbound in the inverse order to the the order used when they were bound. *Dependency Directed Backtracking* instead recognizes the cause of failure and backtracks to one of the causes of failure and skips over the intermediate variables that did not cause the failure.

The following is an easy way to do dependency directed backtracking. We keep track at each variable of the variables that precede it in the backtracking order and to which it is connected directly in the constraint graph. Then, when instantiation fails at a variable, backtracking goes in order to these variables skipping over all other intermediate variables.

Notice then that we will backtrack at a variable up to as many times as there are preceding neighbors. [This number is called the *width* of the variable.] The time complexity of the backtracking algorithm grows when it has to backtrack often. Consequently there is a real gain when the variables are ordered so as to minimize their largest width.

4.4.3 Consistency Driven Techniques

Consistency techniques effectively rule out many inconsistent labeling at a very early stage, and thus cut short the search for consistent labeling. These techniques have since proved to be effective on a wide variety of hard search problems. The consistency techniques are deterministic, as opposed to the search which is non-deterministic. Thus the deterministic computation is performed as soon as possible and non-deterministic computation during search is used only when there is no more propagation to do. Nevertheless, the consistency techniques are rarely used alone to solve constraint satisfaction problem completely (but they could).

In binary CSPs, various consistency techniques for constraint graphs were introduced to prune the search space. The consistency-enforcing algorithm makes any partial solution of a small subnetwork extensible to some surrounding network. Thus, the potential inconsistency is detected as soon as possible.

4.4.3.1 Node Consistency

The simplest consistency technique is referred to as node consistency and we mentioned it in the section on binarization of constraints. The node representing a variable V in constraint graph is node consistent if for every value x in the current domain of V , each unary constraint on V is satisfied.

If the domain D of a variable V contains a value "a" that does not satisfy the unary constraint on V , then the instantiation of V to "a" will always result in immediate failure. Thus, the node inconsistency can be eliminated by simply removing those values from the domain D of each variable V that do not satisfy unary constraint on V .

4.4.3.2 Arc Consistency

If the constraint graph is node consistent then unary constraints can be removed because they all are satisfied. As we are working with the binary CSP, there remains to ensure consistency of binary constraints. In the constraint graph, binary constraint corresponds to arc, therefore this type of consistency is called arc consistency.

Arc (V_i, V_j) is **arc consistent** if for every value x in the current domain of V_i there is some value y in the domain of V_j such that $V_i=x$ and $V_j=y$ is permitted by the binary constraint between V_i and V_j . Note, that the concept of arc-consistency is directional, i.e., if an arc (V_i, V_j) is consistent, than it does not automatically mean that (V_j, V_i) is also consistent.

Clearly, an arc (V_i, V_j) can be made consistent by simply deleting those values from the domain of V_i for which there does not exist corresponding value in the domain of D_j such that the binary constraint between V_i and V_j is satisfied (note, that deleting of such values does not eliminate any solution of the original CSP).

The following algorithm does precisely that.

Algorithm REVISE

```

procedure REVISE(Vi,Vj)
    DELETE <- false;
    for each X in Di do
        if there is no such Y in Dj such that (X,Y) is consistent,
        then
            delete X from Di;
            DELETE <- true;
        endif;
    endfor;
    return DELETE;
end REVISE

```

To make every arc of the constraint graph consistent, it is not sufficient to execute REVISE for each arc just once. Once REVISE reduces the domain of some variable V_i , then each previously revised arc (V_j, V_i) has to be revised again, because some of the members of the domain of V_j may no longer be compatible with any remaining members of the revised domain of V_i . The following algorithm, known as **AC-1**, does precisely that.

Algorithm AC-1

```

procedure AC-1
    Q <- { (Vi,Vj) in arcs(G), i#j };
    repeat
        CHANGE <- false;
        for each (Vi,Vj) in Q do
            CHANGE <- REVISE(Vi,Vj) or CHANGE;
        endfor
    until not(CHANGE)
end AC-1

```

This algorithm is not very efficient because the successful revision of even one arc in some iteration forces all the arcs to be revised again in the next iteration, even though only a small number of them are really affected by this revision. Visibly, the only arcs affected by the reduction of the domain of V_k are the arcs (V_i, V_k) . Also, if we revise the

arc (V_k, V_m) and the domain of V_k is reduced, it is not necessary to re-revise the arc (V_m, V_k) because none of the elements deleted from the domain of V_k provided support for any value in the current domain of V_m . The following variation of arc consistency algorithm, called AC-3, removes this drawback of AC-1 and performs re-revision only for those arcs that are possibly affected by a previous revision.

Algorithm AC-3

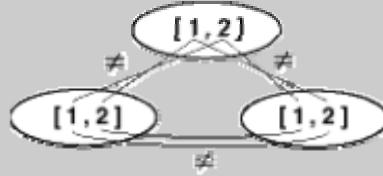
```

procedure AC-3
    Q <- { (Vi,Vj) in arcs(G), i#j };
    while not Q empty
        select and delete any arc  $(V_k, V_m)$  from Q;
        if REVISE( $(V_k, V_m)$ ) then
            Q <- Q union { (Vi, Vj) such that  $(V_i, V_k)$  in
arcs(G), i#k, i#m}
        endif
    endwhile
end AC-3

```

When the algorithm AC-3 revises the edge for the second time it re-tests many pairs of values which are already known (from the previous iteration) to be consistent or inconsistent respectively and which are not affected by the reduction of the domain. As this is a source of potential inefficiency, the algorithm **AC-4** was introduced to refine handling of edges (constraints). The algorithm works with individual pairs of values as the following example shows.

Example:



First, the algorithm AC-4 initializes its internal structures which are used to remember pairs of consistent (inconsistent) values of incidental variables (nodes) - structure $S_{i,a}$. This initialization also counts "supporting" values from the domain of incidental variable - structure $counter_{(i,j),a}$ - and it removes those values which have no support. Once the value is removed from the domain, the algorithm adds the pair $\langle \text{Variable}, \text{Value} \rangle$ to the list Q for re-revision of affected values of corresponding variables.

Algorithm INITIALIZE

```
procedure INITIALIZE
    Q <- {};
    S <- {};  
    % initialize each element of structure S
    for each (Vi,Vj) in arcs(G) do  
        % (Vi,Vj) and (Vj,Vi) are
    same elements
        for each a in Di do
            total <- 0;
            for each b in Dj do
                if (a,b) is consistent according to the constraint
(Vi,Vj) then
                    total <- total+1;
                    Sj,b <- Sj,b union {<i,a>};
                endif
            endfor;
            counter[(i,j),a] <- total;
            if counter[(i,j),a]=0 then
                delete a from Di;
                Q <- Q union {<i,a>};
            endif;
        endfor;
    endfor;
    return Q;
end INITIALIZE
```

After the initialization, the algorithm AC-4 performs re-revision only for those pairs of values of incidental variables that are affected by a previous revision.

Algorithm AC-4

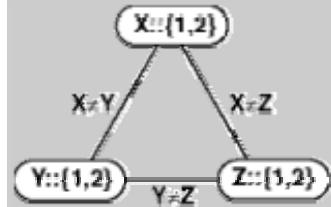
```
procedure AC-4
    Q <- INITIALIZE;
    while not Q empty
        select and delete any pair <j,b> from Q;
        for each <i,a> from Sj,b do
            counter[(i,j),a] <- counter[(i,j),a] - 1;
            if counter[(i,j),a]=0 & a is still in Di then
                delete a from Di;
                Q <- Q union {<i,a>};
            endif
        endfor
    endwhile
end AC-4
```

Both algorithms, AC-3 and AC-4, belong to the most widely used algorithms for maintaining arc consistency. It should be also noted that there exist other algorithms AC-5, AC-6, AC-7 etc. but they are not used as frequently as AC-3 or AC-4.

Maintaining arc consistency removes many inconsistencies from the constraint graph but is any (complete) instantiation of variables from current (reduced) domains a solution to the CSP? If the domain size of each variable becomes one, then the CSP has exactly one solution which is obtained by assigning to each variable the only possible value in its

domain. Otherwise, the answer is no in general. The following example shows such a case where the constraint graph is arc consistent, domains are not empty but there is still no solution satisfying all constraints.

Example:



This constraint graph is arc consistent but there is no solution that satisfies all the constraints.

4.4.3.3 Path Consistency (K-Consistency)

Given that arc consistency is not enough to eliminate the need for backtracking, is there another stronger degree of consistency that may eliminate the need for search? The above example shows that if one extends the consistency test to two or more arcs, more inconsistent values can be removed.

A graph is **K-consistent** if the following is true: Choose values of any K-1 variables that satisfy all the constraints among these variables and choose any Kth variable. Then there exists a value for this Kth variable that satisfies all the constraints among these K variables. A graph is **strongly K-consistent** if it is J-consistent for all J<=K.

Node consistency discussed earlier is equivalent to strong 1-consistency and arc-consistency is equivalent to strong 2-consistency (arc-consistency is usually assumed to include node-consistency as well). Algorithms exist for making a constraint graph strongly K-consistent for K>2 but in practice they are rarely used because of efficiency issues. The exception is the algorithm for making a constraint graph strongly 3-consistent that is usually referred as **path consistency**. Nevertheless, even this algorithm is too hungry and a weak form of path consistency was introduced.

A node representing variable V_i is **restricted path consistent** if it is arc-consistent, i.e., all arcs from this node are arc-consistent, and the following is true: For every value a in the domain D_i of the variable V_i that has *just one supporting value* b from the domain of incidental variable V_j there exists a value c in the domain of other incidental variable V_k such that (a,c) is permitted by the binary constraint between V_i and V_k , and (c,b) is permitted by the binary constraint between V_k and V_j .

The algorithm for making graph restricted path consistent can be naturally based on AC-4 algorithm that counts the number of supporting values. Although this algorithm removes more inconsistent values than any arc-consistency algorithm it does not eliminate the need for search in general. Clearly, if a constraint graph containing n nodes is strongly n-consistent, then a solution to the CSP can be found without any search. But the worst-case complexity of the algorithm for obtaining n-consistency in a n -node constraint graph

is also exponential. If the graph is (strongly) K-consistent for $K < n$, then in general, backtracking cannot be avoided, i.e., there still exist inconsistent values.

4.4.4 Forward Checking

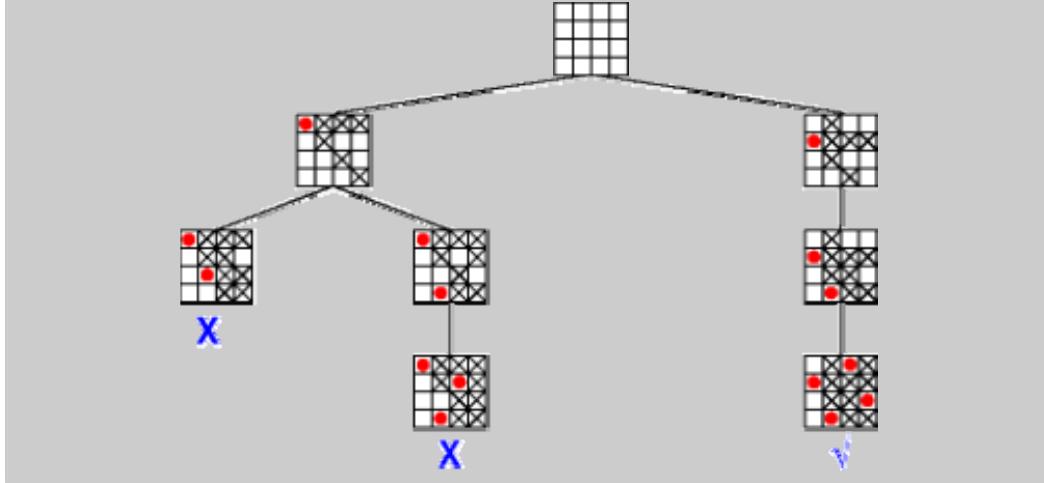
Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency to the instantiated variables, it performs restricted form of arc consistency to the not yet instantiated variables. We speak about restricted arc consistency because forward checking checks only the constraints between the current variable and the future variables. When a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

Algorithm AC-3 for Forward Checking

```
procedure AC3-FC(cv)
    Q <- { (Vi,Vcv) in arcs(G) , i>cv };
    consistent <- true;
    while not Q empty & consistent
        select and delete any arc (vk,Vm) from Q;
        if REVISE(vk,Vm) then
            consistent <- not Dk empty
        endif
    endwhile
    return consistent
end AC3-FC
```

Forward checking detects the inconsistency earlier than simple backtracking and thus it allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. This reduces the search tree and (hopefully) the overall amount of work done. But it should be noted that forward checking does more work when each assignment is added to the current partial solution.

Example: (4-queens problem and FC)



Forward checking is almost always a much better choice than simple backtracking.

4.4.5 Look Ahead

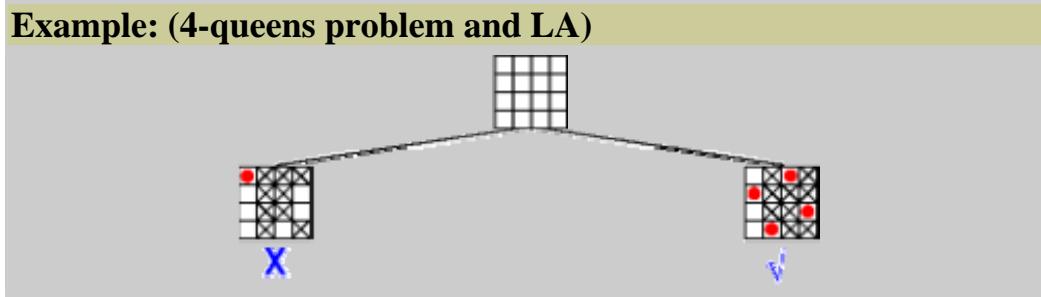
Forward checking checks only the constraints between the current variable and the future variables. So why not to perform full arc consistency that will further reduces the domains and removes possible conflicts? This approach is called (full) look ahead or maintaining arc consistency (MAC).

The advantage of look ahead is that it detects also the conflicts between future variables and therefore allows branches of the search tree that will lead to failure to be pruned earlier than with forward checking. Also as with forward checking, whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no necessary.

Algorithm AC-3 for Look Ahead

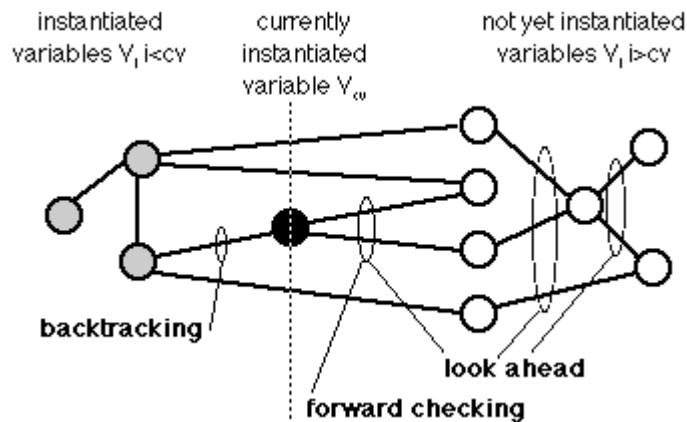
```
procedure AC3-LA(cv)
    Q <- { (Vi, Vcv) in arcs(G), i>cv };
    consistent <- true;
    while not Q empty & consistent
        select and delete any arc (Vk, Vm) from Q;
        if REVISE(Vk, Vm) then
            Q <- Q union { (Vi, Vk) such that (Vi, Vk) in
                arcs(G), i#k, i#m, i>cv }
            consistent <- not Dk empty
        endif
    endwhile
    return consistent
end AC3-LA
```

Look ahead prunes the search tree further more than forward checking but, again, it should be noted that look ahead does even more work when each assignment is added to the current partial solution than forward checking.



4.4.6 Comparison of Propagation Techniques

The following figure shows which constraints are tested when the above described propagation techniques are applied.



More constraint propagation at each node will result in the search tree containing fewer nodes, but the overall cost may be higher, as the processing at each node will be more expensive. In one extreme, obtaining strong n-consistency for the original problem would completely eliminate the need for search, but as mentioned before, this is usually more expensive than simple backtracking. Actually, in some cases even the full look ahead may be more expensive than simple backtracking. That is the reason why forward checking and simple backtracking are still used in applications.

Module 4

Constraint satisfaction problems

Lesson 10

Constraint satisfaction problems - II

4.5 Variable and Value Ordering

A search algorithm for constraint satisfaction requires the order in which variables are to be considered to be specified as well as the order in which the values are assigned to the variable on backtracking. Choosing the right order of variables (and values) can noticeably improve the efficiency of constraint satisfaction.

4.5.1 Variable Ordering

Experiments and analysis of several researchers have shown that the ordering in which variables are chosen for instantiation can have substantial impact on the complexity of backtrack search. The ordering may be either

- a *static ordering*, in which the order of the variables is specified before the search begins, and it is not changed thereafter, or
- a *dynamic ordering*, in which the choice of next variable to be considered at any point depends on the current state of the search.

Dynamic ordering is not feasible for all search algorithms, e.g., with simple backtracking there is no extra information available during the search that could be used to make a different choice of ordering from the initial ordering. However, with forward checking, the current state includes the domains of the variables as they have been pruned by the current set of instantiations, and so it is possible to base the choice of next variable on this information.

Several heuristics have been developed and analyzed for selecting variable ordering. The most common one is based on the "**first-fail**" principle, which can be explained as

"To succeed, try first where you are most likely to fail."

In this method, the variable with the fewest possible remaining alternatives is selected for instantiation. Thus the order of variable instantiations is, in general, different in different branches of the tree, and is determined dynamically. This method is based on assumption that any value is equally likely to participate in a solution, so that the more values there are, the more likely it is that one of them will be a successful one.

The first-fail principle may seem slightly misleading, after all, we do not want to fail. The reason is that if the current partial solution does not lead to a complete solution, then the sooner we discover this the better. Hence encouraging early failure, if failure is inevitable, is beneficial in the long term. On the other end, if the current partial solution can be extended to a complete solution, then every remaining variable must be instantiated and the one with smallest domain is likely to be the most difficult to find a value for (instantiating other variables first may further reduce its domain and lead to a failure). Hence the principle could equally well be stated as:

"Deal with hard cases first: they can only get more difficult if you put them off."

This heuristic should reduce the average depth of branches in the search tree by triggering early failure.

Another heuristic, that is applied when all variables have the same number of values, is to choose the variable which participates in most constraints (in the absence of more specific information on which constraints are likely to be difficult to satisfy, for instance). This heuristic follows also the principle of dealing with hard cases first.

There is also a heuristic for static ordering of variables that is suitable for simple backtracking. This heuristic says: choose the variable which has the largest number of constraints with the past variables. For instance, during solving graph coloring problem, it is reasonable to assign color to the vertex which has common arcs with already colored vertices so the conflict is detected as soon as possible.

4.5.2 Value Ordering

Once the decision is made to instantiate a variable, it may have several values available. Again, the order in which these values are considered can have substantial impact on the time to find the first solution. However, if all solutions are required or there are no solutions, then the value ordering is indifferent.

A different value ordering will rearrange the branches emanating from each node of the search tree. This is an advantage if it ensures that a branch which leads to a solution is searched earlier than branches which lead to dead ends. For example, if the CSP has a solution, and if a correct value is chosen for each variable, then a solution can be found without any backtracking.

Suppose we have selected a variable to instantiate: how should we choose which value to try first? It may be that none of the values will succeed, in that case, every value for the current variable will eventually have to be considered, and the order does not matter. On the other hand, if we can find a complete solution based on the past instantiations, we want to choose a value which is likely to succeed, and unlikely to lead to a conflict. So, we apply the "**succeed first**" principle.

One possible heuristic is to prefer those values that maximize the number of options available. Visibly, the algorithm AC-4 is good for using this heuristic as it counts the supporting values. It is possible to count "promise" of each value, that is the product of the domain sizes of the future variables after choosing this value (this is an upper bound on the number of possible solutions resulting from the assignment). The value with highest promise should be chosen. It is also possible to calculate the percentage of values in future domains which will no longer be usable. The best choice would be the value with lowest cost.

Another heuristic is to prefer the value (from those available) that leads to an easiest to solve CSP. This requires to estimate the difficulty of solving a CSP. One method proposes to convert a CSP into a tree-structured CSP by deleting a minimum number of arcs and then to find all solutions of the resulting CSP (higher the solution count, easier the CSP).

For randomly generated problems, and probably in general, the work involved in assessing each value is not worth the benefit of choosing a value which will on average be more likely to lead to a solution than the default choice. In particular problems, on the other hand, there may be information available which allows the values to be ordered according to the principle of choosing first those most likely to succeed.

4.6 Heuristic Search in CSP

In the last few years, greedy local search strategies became popular, again. These algorithms incrementally alter inconsistent value assignments to all the variables. They use a "repair" or "hill climbing" metaphor to move towards more and more complete solutions. To avoid getting stuck at "local optima" they are equipped with various heuristics for randomizing the search. Their stochastic nature generally voids the guarantee of "completeness" provided by the systematic search methods.

The local search methodology uses the following terms:

- **state (configuration):** one possible assignment of all variables; the number of states is equal to the product of domains' sizes
- **evaluation value:** the number of constraint violations of the state (sometimes weighted)
- **neighbor:** the state which is obtained from the current state by changing one variable value
- **local-minimum:** the state that is not a solution and the evaluation values of all of its neighbors are larger than or equal to the evaluation value of this state
- **strict local-minimum:** the state that is not a solution and the evaluation values of all of its neighbors are larger than the evaluation value of this state
- **non-strict local-minimum:** the state that is a local-minimum but not a strict local-minimum.

4.6.1 Hill-Climbing

Hill-climbing is probably the most known algorithm of local search. The idea of hill-climbing is:

1. start at randomly generated state
2. move to the neighbor with the best evaluation value
3. if a strict local-minimum is reached then restart at other randomly generated state.

This procedure repeats till the solution is found. In the algorithm, that we present here, the parameter `Max_Flips` is used to limit the maximal number of moves between restarts which helps to leave non-strict local-minimum.

Algorithm Hill-Climbing

```
procedure hill-climbing (Max_Flips)
    restart: s <- random valuation of variables;
    for j:=1 to Max_Flips do
        if eval(s)=0 then return s endif;
        if s is a strict local minimum then
            goto restart
        else
            s <- neighborhood with smallest evaluation value
        endif
    endfor
    goto restart
end hill-climbing
```

Note, that the hill-climbing algorithm has to explore all neighbors of the current state before choosing the move. This can take a lot of time.

4.6.2 Min-Conflicts

To avoid exploring all neighbors of the current state some heuristics were proposed to find a next move. *Min-conflicts* heuristics chooses randomly any conflicting variable, i.e., the variable that is involved in any unsatisfied constraint, and then picks a value which minimizes the number of violated constraints (break ties randomly). If no such value exists, it picks randomly one value that does not increase the number of violated constraints (the current value of the variable is picked only if all the other values increase the number of violated constraints).

Algorithm Min-Conflicts

```
procedure MC (Max Moves)
    s <- random valuation of variables;
    nb_moves <- 0;
    while eval(s)>0 & nb_moves<Max Moves do
        choose randomly a variable V in conflict;
        choose a value v' that minimizes the number of conflicts for V;
        if v' # current value of V then
            assign v' to V;
            nb_moves <- nb_moves+1;
        endif
    endwhile
    return s
end MC
```

Note, that the pure min-conflicts algorithm presented above is not able to leave local-minimum. In addition, if the algorithm achieves a strict local-minimum it does not perform any move at all and, consequently, it does not terminate.

4.6.3 GSAT

GSAT is a greedy local search procedure for satisfying logic formulas in a conjunctive normal form (CNF). Such problems are called SAT or k-SAT (k is a number of literals in each clause of the formula) and are known to be NP-c (each NP-hard problem can be transformed to NP-complex problem).

The procedure starts with an arbitrary instantiation of the problem variables and offers to reach the highest satisfaction degree by succession of small transformations called repairs or flips (flipping a variable is a changing its value).

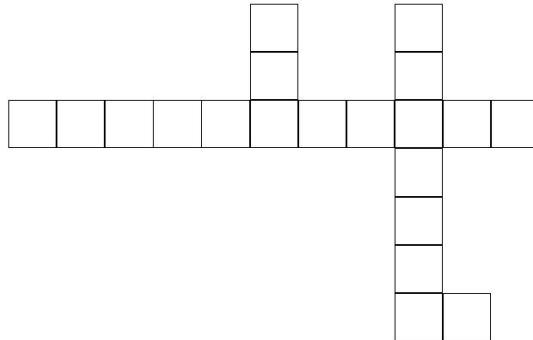
Algorithm GSAT

```
procedure GSAT(A,Max Tries,Max Flips)
A: is a CNF formula
for i:=1 to Max Tries do
    S <- instantiation of variables
    for j:=1 to Max Iter do
        if A satisfiable by S then
            return S
        endif
        V <- the variable whose flip yield the most important raise in the
number of satisfied clauses;
        S <- S with V flipped;
    endfor
endfor
return the best instantiation found
end GSAT
```

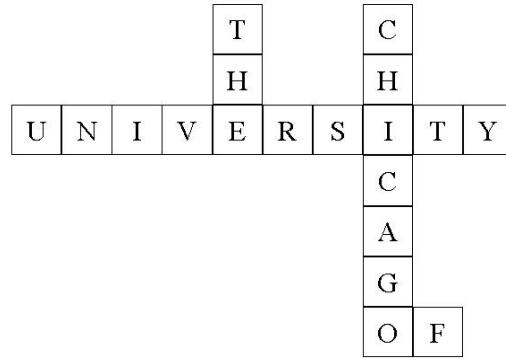
Questions

1. Consider a variant of the crossword puzzle problem. In this variant, we assume that we have a set of words W_1, W_2, \dots, W_n and a crossword puzzle grid. Our goal is to fill the crossword grid with the words such that letters of intersecting words match. An example of an uncompleted puzzle and a completed puzzle appear below.

THE
UNIVERSITY
OF
CHICAGO



THE
UNIVERSITY
OF
CHICAGO



Provide a constraint satisfaction problem formulation for this variant of the crossword puzzle problem.

- Specify the variables to which values are to be assigned.
- Specify the domains from which the variables take their values.
- Define the constraints that must hold between variables. Please provide pseudo-code defining the constraints explicitly.
- Give a simple example of a "fill-in" (crossword) puzzle of the type above that demonstrates the limitations of arc-consistency type constraint propagation for solving constraint satisfaction problems.

- e. Explain why constraint satisfaction procedures based on backtracking search are not subject to this problem.
 - f. Briefly describe the use of the iterative refinement, min-conflict strategy to solve the crossword puzzle problem.
 - g. Demonstrate the application of your procedure on a simple 4 word example puzzle.
2. Consider the following classroom scheduling problem: There are 4 classes, C1, C2, C3, and C4, and 3 class rooms, R1, R2, and R3. The following table shows the class schedule:

Class	Time
C1	8am-10:30am
C2	9am-11:30pm
C3	10am-12:30pm
C4	11am-1:30pm

In addition, there are the following restrictions:

- Each class must use one of the 3 rooms, R1, R2, R3.
- R3 is too small for C3.
- R2 and R3 are too small for C4.

One way of formulating this problem as a constraint satisfaction problem is to let each class, C1, ..., C4, be a variable, and each room, R1, R2, R3, be the possible values for these variables.

- (a) Show the initial possible values for each variable, C1, ..., C4, given the restrictions above.
- (b) Express formally all the constraints in this problem.
- (c) Consider each pair of variables appearing in the same constraint in (b), please point out which pairs are arc-consistent for the initial values provided in (a). For those pairs that are not arc-consistent, please provide the necessary operations so that they become arc-consistent.

Solution

1. A. Variables.

We use rows or columns of boxes as variables. In this case we have four variables H1,H2,V1,V2 (we use H for horizontal and V for vertical)

1. B. Domains

All the variables can take values out of the same domain D. In our case we define D = {THE, UNIVERSITY, OF, CHICAGO}

1. C. Constraints

We define two kinds of constraints.

Length constraints:

```
length(H1) == 10  
length(H2) == 2  
length(V1) == 3  
length(V2) == 7
```

Cross constraints:

```
H1(5) == V1(3)  
H1(8) == V2(3)  
H2(1) == V2(7)
```

1.D. Arc consistency problems

There are two kinds of problems, when there are no legal assignments or there are more than one legal assignment.

Example of more than one legal assignment.

Assume three variables V1,H1,H2 taking values out of the domain D = {bit, its, hit, sit, ion, one}

	V1		
H1			
H2			

after applying the arc consistency algorithm (page 146) the domains for each variable are equal to

$$D(H1) = \{\text{bit, hit, sit}\}$$

$$D(V1) = \{\text{ion}\}$$

$$D(H2) = \{\text{one}\}$$

There is more than one legal assignment for the variable H1

Example of no legal assignment.

In the previous example change the domain from D to E = {bit, its, hit, sit, ion }

1.E. Procedures based on backtracking do not have problems with multiple legal assignments because they pick the first one that satisfies the constraints without looking for more options. When there are no legal assignments, they search the whole space, then return a failure value

1. F.

- (0). Start a counter of steps count := 0
- (1). Assign to each variable a random word taken from its respective domain.
- (2). Count the number of conflicts each assignment produced (number of constraints unsatisfied)
- (3). Pick the variable with the highest number of conflicts and change its value until its number of conflicts is reduced
- (4). Increase count by one

(5). If count is less than a predetermined maximum number of steps, repeat from step (2)

Note: Explain why it is necessary to have a maximum number of steps. Is this an optimal method?

1. G. Now that you have the algorithm, try it at home!
- 2.a. C1: { R1, R2, R3 } C2: { R1, R2, R3 } C3: { R1, R2 } C4: { R1 }
- 2.b C1 != C2, C1 != C3, C2 != C3, C2 != C4, C3 != C4
We may add C3 != R3, C4 != R2, C4 != R3 even though they are contained in (a).
- 2.c. All the five pairs of variables in the five binary constraints in (b) are not arc consistent. To make them consistent, we need to remove R1 from the domain of C3, R1, R2 from the domain of C2 and R2, R3 from the domain of C1.

Module 5

Knowledge Representation and Logic – (Propositional Logic)

5.1 Instructional Objective

- Students should understand the importance of knowledge representation in intelligent agents
- Students should understand the use of formal logic as a knowledge representation language
- The student should be familiar with the following concepts of logic
 - syntax
 - semantics
 - validity
 - satisfiability
 - interpretation and models
 - entailment
- Students should understand each of the above concepts in propositional logic
- Students should learn different inference mechanisms in propositional logic

At the end of this lesson the student should be able to do the following:

- Represent a natural language description as statements in logic
- Deduct new sentences by applying inference rules.

Lesson

11

Propositional Logic

5.2 Knowledge Representation and Reasoning

Intelligent agents should have capacity for:

- **Perceiving**, that is, acquiring information from environment,
- **Knowledge Representation**, that is, representing its understanding of the world,
- **Reasoning**, that is, inferring the implications of what it knows and of the choices it has, and
- **Acting**, that is, choosing what it wants to do and carry it out.

Representation of knowledge and the reasoning process are central to the entire field of artificial intelligence. The primary component of a knowledge-based agent is its knowledge-base. A knowledge-base is a set of sentences. Each sentence is expressed in a language called the knowledge representation language. Sentences represent some assertions about the world. There must be mechanisms to derive new sentences from old ones. This process is known as inferencing or reasoning. Inference must obey the primary requirement that the new sentences should follow logically from the previous ones.

Logic is the primary vehicle for representing and reasoning about knowledge. Specifically, we will be dealing with formal logic. The advantage of using formal logic as a language of AI is that it is precise and definite. This allows programs to be written which are declarative - they describe what is true and not how to solve problems. This also allows for automated reasoning techniques for general purpose inferencing.

This, however, leads to some severe limitations. Clearly, a large portion of the reasoning carried out by humans depends on handling knowledge that is uncertain. Logic cannot represent this uncertainty well. Similarly, natural language reasoning requires inferring hidden state, namely, the intention of the speaker. When we say, "One of the wheels of the car is flat.", we know that it has three wheels left. Humans can cope with virtually infinite variety of utterances using a finite store of commonsense knowledge. Formal logic has difficulty with this kind of ambiguity.

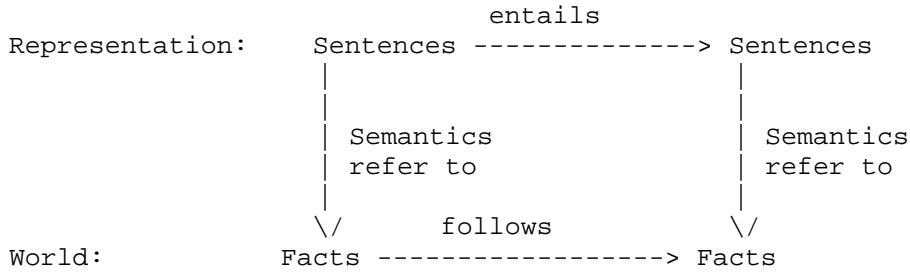
A logic consists of two parts, a language and a method of reasoning. The logical language, in turn, has two aspects, syntax and semantics. Thus, to specify or define a particular logic, one needs to specify three things:

Syntax: The atomic symbols of the logical language, and the rules for constructing well-formed, non-atomic expressions (symbol structures) of the logic. Syntax specifies the symbols in the language and how they can be combined to form sentences. Hence facts about the world are represented as sentences in logic.

Semantics: The meanings of the atomic symbols of the logic, and the rules for determining the meanings of non-atomic expressions of the logic. It specifies what facts in the world a sentence refers to. Hence, also specifies how you assign a truth value to a sentence based on its meaning in the world. A **fact** is a claim about the world, and may be true or false.

Syntactic Inference Method: The rules for determining a subset of logical expressions, called theorems of the logic. It refers to a mechanical method for computing (deriving) new (true) sentences from existing sentences.

Facts are claims about the world that are True or False, whereas a **representation** is an expression (sentence) in some language that can be encoded in a computer program and stands for the objects and relations in the world. We need to ensure that the representation is consistent with reality, so that the following figure holds:



There are a number of logical systems with different syntax and semantics. We list below a few of them.

- Propositional logic

All objects described are fixed or unique

"John is a student" student(john)

Here John refers to one unique person.

- First order predicate logic

Objects described can be unique or variables to stand for a unique object

"All students are poor"

ForAll(S) [student(S) \rightarrow poor(S)]

Here S can be replaced by many different unique students.

This makes programs much more compact:

eg. ForAll(A,B)[brother(A,B) \rightarrow brother(B,A)]

replaces half the possible statements about brothers

- Temporal

Represents truth over time.
 - Modal

Represents doubt
 - Higher order logics

Allows variable to represent many relations between objects
 - Non-monotonic

Represents defaults
- Propositional logic is one of the simplest systems of logic.
- ### 5.3 Propositional Logic
- In propositional logic (PL) an user defines a set of propositional symbols, like P and Q . User defines the semantics of each of these symbols. For example,
- P means "It is hot"
 - Q means "It is humid"
 - R means "It is raining"
 -
 - A **sentence** (also called a formula or well-formed formula or wff) is defined as:
 1. A symbol
 2. If S is a sentence, then $\sim S$ is a sentence, where " \sim " is the "not" logical operator
 3. If S and T are sentences, then $(S \vee T)$, $(S \wedge T)$, $(S \Rightarrow T)$, and $(S \Leftrightarrow T)$ are sentences, where the four logical connectives correspond to "or," "and," "implies," and "if and only if," respectively
 4. A finite number of applications of (1)-(3)
 - Examples of PL sentences:
 - $(P \wedge Q) \Rightarrow R$ (here meaning "If it is hot and humid, then it is raining")
 - $Q \Rightarrow P$ (here meaning "If it is humid, then it is hot")
 - Q (here meaning "It is humid.")

- Given the truth values of all of the constituent symbols in a sentence, that sentence can be "evaluated" to determine its truth value (True or False). This is called an **interpretation** of the sentence.
- A **model** is an interpretation (i.e., an assignment of truth values to symbols) of a set of sentences such that each sentence is True. A model is just a formal mathematical structure that "stands in" for the world.
- A **valid** sentence (also called a **tautology**) is a sentence that is True under *all* interpretations. Hence, no matter what the world is actually like or what the semantics is, the sentence is True. For example "It's raining or it's not raining."
- An **inconsistent** sentence (also called **unsatisfiable** or a **contradiction**) is a sentence that is False under *all* interpretations. Hence the world is never like what it describes. For example, "It's raining and it's not raining."
- Sentence P **entails** sentence Q, written $P \models Q$, means that whenever P is True, so is Q. In other words, all models of P are also models of Q

Entailment (\models): Given 2 sentences p and q we say p entails q , written $p \models q$, if q holds in every model that p holds.

Example: Entailment

$$p \wedge (p \Rightarrow q) \models q$$

Show that:

$$p \wedge (p \Rightarrow q)$$

Proof: For any model M in which $p \wedge (p \Rightarrow q)$ holds then we know that p holds in M

and $p \Rightarrow q$ holds in M. Since p holds in M then since $p \Rightarrow q$ holds in M, q must hold in M. Therefore q holds in every model that $p \wedge (p \Rightarrow q)$ holds and so $p \wedge (p \Rightarrow q) \models q$.

As we have noted models affect equivalence and so we repeat the definition again and give an example of a proof of equivalence.

Equivalence: Two sentences are *equivalent* if they hold in exactly the same models.

Example: Equivalence

$$p \Rightarrow q \equiv \neg p \vee q$$

Show that:

Proof: We need to provide two proofs as above for

$$p \Rightarrow q \models \neg p \vee q$$

- For any model M in which $p \Rightarrow q$ holds then we know that either p holds in M and so q holds in M, or $\neg p$ does not hold in M and so $\neg p$ holds in M. Since either q holds in M or $\neg p$ holds in M, then $\neg p \vee q$ holds in M.

$$\neg p \vee q \models p \Rightarrow q$$

and

- For any model M in which $\neg p \vee q$ holds then we know that either $\neg p$ holds in M or q holds in M. If $\neg p$ holds in M then $p \Rightarrow q$ holds in M. Otherwise, if q holds in M then $p \Rightarrow q$ holds in M. Therefore $p \Rightarrow q$ holds in M.

Knowledge based programming relies on concluding new knowledge from existing knowledge. Entailment is a required justification; i.e. if p_1, \dots, p_n is known then there is justification to conclude q if

$$p_1 \wedge \dots \wedge p_n \models q$$

In some circumstances we insist on this strong form of justification; i.e. we cannot conclude q unless the entailment holds. Reasoning like this is the equivalent for knowledge based programs of running a piece of conventional software.



Note: Entailment (\models) is concerned with *truth* and is determined by considering the truth of the sentences in all models.

5.4 Propositional Logic Inference

Let $KB = \{ S_1, S_2, \dots, S_M \}$ be the set of all sentences in our Knowledge Base, where each S_i is a sentence in Propositional Logic. Let $\{ X_1, X_2, \dots, X_N \}$ be the set of all the symbols (i.e., variables) that are contained in all of the M sentences in KB . Say we want to know if a goal (aka query, conclusion, or theorem) sentence G follows from KB .

5.4.1 Model Checking

Since the computer doesn't know the interpretation of these sentences in the world, we don't know whether the constituent symbols represent facts in the world that are True or False. So, instead, consider *all* possible combinations of truth values for all the symbols, hence enumerating all logically distinct cases:

X1	X2	...	XN	S1	S2	...	SM	$S_1 \wedge S_2 \wedge \dots \wedge S_M$	G	$(S_1 \wedge \dots \wedge S_M) \Rightarrow G$
F	F	...	F							
F	F	...	T							
...										
T	T	...	T							

- There are 2^N rows in the table.
- Each row corresponds to an equivalence class of worlds that, under a given interpretation, have the truth values for the N symbols assigned in that row.
- The **models** of KB are the rows where the third-to-last column is *true*, i.e., where all of the sentences in KB are *true*.
- A sentence R is **valid** if and only if it is true under all possible interpretations, i.e., if the entire column associated with R contains all *true* values.
- Since we don't know the semantics and therefore whether each symbol is True or False, to determine if a sentence G is **entailed** by KB , we must determine if **all** models of KB are also models of G . That is, whenever KB is true, G is true too. In other words, whenever the third-to-last column has a T, the same row in the second-to-last column also has a T. But this is logically equivalent to saying that the sentence $(KB \Rightarrow G)$ is valid (by definition of the "implies" connective). In other words, if the last column of the table above contains only *True*, then **KB entails G**; or conclusion G logically follows from the premises in KB , no matter what the interpretations (i.e., semantics) associated with all of the sentences!
- The truth table method of inference is **complete** for PL (Propositional Logic) because we can always enumerate all 2^n rows for the n propositional symbols that occur. But this is exponential in n . In general, it has been shown that the problem of checking if a set of sentences in PL is satisfiable is NP-complete. (The truth table method of inference is *not* complete for FOL (First-Order Logic).)

Example

Using the "weather" sentences from above, let $KB = (((P \wedge Q) \Rightarrow R) \wedge (Q \Rightarrow P) \wedge Q)$

corresponding to the three facts we know about the weather: (1) "If it is hot and humid, then it is raining," (2) "If it is humid, then it is hot," and (3) "It is humid." Now let's ask the query "Is it raining?" That is, is the query sentence R entailed by KB? Using the truth-table approach to answering this query we have:

P	Q	R	$(P \wedge Q) \Rightarrow R$	$Q \Rightarrow P$	Q	KB	R	$KB \Rightarrow R$
T	T	T	T	T	T	T	T	T
T	T	F	F	T	F	F	T	T
T	F	T	T	F	F	T	T	T
T	F	F	T	F	F	F	T	T
F	T	T	T	F	T	F	T	T
F	T	F	T	F	T	F	T	T
F	F	T	T	F	F	T	T	T
F	F	F	T	F	F	F	T	T

Hence, in this problem there is only one model of KB, when P, Q, and R are all True. And in this case R is also True, so R is entailed by KB. Also, you can see that the last column is all True values, so the sentence $KB \Rightarrow R$ is valid.

Instead of an exponential length proof by truth table construction, is there a faster way to implement the inference process? Yes, using a **proof procedure** or **inference procedure** that uses **sound rules of inference** to deduce (i.e., derive) new sentences that are true in all cases where the premises are true. For example, consider the following:

P	Q	P	$P \Rightarrow Q$	$P \wedge (P \Rightarrow Q)$	Q	$(P \wedge (P \Rightarrow Q)) \Rightarrow Q$
F	F	F	T	F	F	T
F	T	F	T	F	T	T
T	F	T	F	F	F	T
T	T	T	T	T	T	T

Since whenever P and $P \Rightarrow Q$ are both true (last row only), Q is true too, Q is said to be **derived** from these two premise sentences. We write this as $KB \vdash Q$. This local pattern referencing only two of the M sentences in KB is called the **Modus Ponens** inference rule. The truth table shows that this inference rule is **sound**. It specifies how to make one kind of step in deriving a conclusion sentence from a KB.

Therefore, given the sentences in KB, construct a **proof** that a given conclusion sentence can be derived from KB by applying a sequence of sound inferences using either sentences in KB or sentences derived earlier in the proof, until the conclusion sentence is derived. This method is called the **Natural Deduction** procedure. (Note: This step-by-step, local proof process also relies on the **monotonicity** property of PL and FOL. That is, adding a new sentence to KB does not affect what can be entailed from the original KB and does not invalidate old sentences.)

Module 5

Knowledge Representation and Logic – (Propositional Logic)

Lesson 12

Propositional Logic inference rules

5.5 Rules of Inference

Here are some examples of sound rules of inference. Each can be shown to be sound once and for all using a truth table. The left column contains the premise sentence(s), and the right column contains the derived sentence. We write each of these derivations as $A \vdash B$, where A is the premise and B is the derived sentence.

Name	Premise(s)	Derived Sentence
Modus Ponens	$A, A \Rightarrow B$	B
And Introduction	A, B	$A \wedge B$
And Elimination	$A \wedge B$	A
Double Negation	$\sim\sim A$	A
Unit Resolution	$A \vee B, \sim B$	A
Resolution	$A \vee B, \sim B \vee C$	$A \vee C$

In addition to the above rules of inference one also requires a set of equivalences of propositional logic like " $A \wedge B$ " is equivalent to " $B \wedge A$ ". A number of such equivalences were presented in the discussion on propositional logic.

5.6 Using Inference Rules to Prove a Query/Goal/Theorem

A proof is a sequence of sentences, where each sentence is either a premise or a sentence derived from earlier sentences in the proof by one of the rules of inference. The last sentence is the query (also called goal or theorem) that we want to prove.

Example for the "weather problem" given above.

1. Q Premise
2. $Q \Rightarrow P$ Premise
3. P Modus Ponens(1,2)
4. $(P \wedge Q) \Rightarrow R$ Premise
5. $P \wedge Q$ And Introduction(1,3)
6. R Modus Ponens(4,5)

5.6.1 Inference vs Entailment

There is a subtle difference between entailment and inference.

Inference (\vdash): Given 2 sentences p and q we say q is inferred from p , written $p \vdash q$, if there is a sequence of rules of inference that apply to p and allow q to be added.

Notice that inference is not directly related to truth; i.e. we can infer a sentence provided we have rules of inference that produce the sentence from the original sentences.

However, if rules of inference are to be useful we wish them to be related to entailment. Ideally we would like:

$$p \vdash q \quad p \models q \\ \text{iff}$$

but this equivalence may fail in two ways:

- $p \vdash q \quad p \not\models q$
but

We have inferred q by applying rules of inference to p , but there is some model in which p holds but q does not hold. In this case the rules of inference have inferred ``too much".

- $p \models q \quad p \not\vdash q$
but

q is a sentence which holds in all models in which p holds, but we cannot find rules of inference that will infer q from p . In this case the rules of inference are insufficient to infer the things we want to be able to infer.

5.7 Soundness and Completeness

These notions are so important that there are 2 properties of logics associated with them.

Soundness: An inference procedure \vdash is *sound* if whenever $p \vdash q$ then it is also the case that $p \models q$.

``A sound inference procedure infers things that are valid consequences"

Completeness: An inference procedure \vdash is *complete* if whenever $p \models q$ then it is also the case that $p \vdash q$.

``A complete inference procedure is able to infer anything that is a valid consequence''

The ``best'' inference procedures are both sound and complete, but gaining completeness is often computationally expensive. Notice that even if inference is not complete it is desirable that it is sound.

Propositional Logic and Predicate Logic each with Modus Ponens as their inference procedure are sound but not complete. We shall see that we need further (sound) rules of inference to achieve completeness. In fact we shall see that we shall even restrict the language in order to achieve an effective inference procedure that is sound and complete for a subset of First Order Predicate Logic.

The notion of soundness and completeness is more generally applied than in logic. Whenever we create a knowledge based program we use the syntax of the knowledge representation language, we assign a semantics in some way and the reasoning mechanism defines the inference procedures. The semantics will define what entailment means in this representation and we will be interested in how well the reasoning mechanism achieves entailment.

5.7.1 Decidability

Determining whether $p \models q$ is computationally hard. If q is a consequence then if \vdash is complete then we know that $p \vdash q$ and we can apply the rules of inference exhaustively knowing that we will eventually find the sequence of rules of inference. It may take a long time but it is finite.

However if q is not a consequence (remember the task is *whether or not $p \models q$*) then we can happily apply rules of inference generating more and more irrelevant consequences. So the procedure is guaranteed to eventually stop if q is derivable, but may not terminate otherwise.

Decidability: A problem is *decidable* if there is a procedure that is guaranteed to terminate having determined whether the answer is ``yes'' or ``no''.

Semi-Decidability: A problem is only semi-decidable if there is a procedure that is guaranteed to terminate in one of these cases but not both.

Entailment in Propositional Logic is decidable since truth tables can be applied in a finite number of steps to determine whether or not $p \models q$.

Entailment in Predicate Logic is only semi-decidable; it is only guaranteed to terminate when q is a consequence. One result of this semi-decidability is that many problems are not decidable; if they rely on failing to prove some sentence. Planning is typically not decidable. A common reaction to a non-decidable problem is to *assume* the answer after some reasoning time threshold has been reached. Another reaction to the semi-decidability of Predicate Logic is to restrict attention to subsets of the logic; however even if its entailment is decidable the procedure may be computationally expensive.

Questions

1. Consider a knowledge base KB that contains the following propositional logic sentences:

$$\begin{aligned} Q &\Rightarrow P \\ P &\Rightarrow \neg Q \\ Q \vee R \end{aligned}$$

- a) Construct a truth table that shows the truth value of each sentence in KB and indicate the models in which the KB is true.
- b) Does KB entail R ? Use the definition of entailment to justify your answer.
- c) Does KB entail $R \Rightarrow P$? Extend the truth table and use the definition of entailment to justify your answer.
- d) Does KB entail $Q \Rightarrow R$? Extend the truth table and use the definition of entailment to justify your answer.

2. Using propositional logic prove (D) from (A,B,C):

- A. $P \Rightarrow (Q \Leftrightarrow R)$
- B. $\neg(Q \Leftrightarrow R)$
- C. $(S \wedge Q) \Rightarrow P$
- D. $\neg P \wedge (S \Rightarrow \neg Q)$

Solution

1.a. Truth table:

P	Q	R	$Q \Rightarrow P$	$P \Rightarrow \neg Q$	$Q \vee R$	$R \Rightarrow P$	$Q \Rightarrow R$
T	T	T		F			
T	T	F		F			F
T	F	T	Model in which KB is true				T
T	F	F			F		
F	T	T	F			F	
F	T	F	F				F
F	F	T	Model in which KB is true				T
F	F	F			F		

1.b. $\text{KB} \models R$. In every model in which the KB is true (indicated in the table), R is also true; thus the KB entails R.

1.c. The KB does not entail P. When (P,Q,R) is (F,F,T), $R \Rightarrow P$ is false.

1.d $\text{KB} \models (Q \Rightarrow R)$. In every model in which the KB is true, $Q \Rightarrow R$ is also true.

2. The proof steps are described below.

Transforming A,B,C and the negation of D to CNF gives the following clauses:

A.1 $\neg P \vee \neg Q \vee R$.

A.2 $\neg P \vee Q \vee \neg R$.

B.1 $Q \vee R$

B.2 $\neg Q \vee \neg R$

C. $\neg S \vee \neg Q \vee P$.

D.1 $P \vee S$.

D.2 $P \vee Q$.

Resolution proof may then proceed as follows:

E. $Q \vee \neg R$ (D.2 + A.2, factored)

F. $Q.$ (E + B.1)

G. $\neg R$ (F+B.2).

H. $\neg Q \vee P.$ (C+D.1, factored)

I. P (H+D.2, factored)

J. $\neg Q \vee R$ (I+A.1).

K. R (J+F)

L. $\emptyset.$ (K+G).

Module 6

Knowledge Representation and Logic – (First Order Logic)

6.1 Instructional Objective

- Students should understand the advantages of first order logic as a knowledge representation language
- Students should be able to convert natural language statement to FOL statements
- The student should be familiar with the following concepts of first order logic
 - syntax
 - interpretation
 - semantics
 - semantics of quantifiers
 - entailment
 - unification
 - Skolemization
- Students should be familiar with different inference rules in FOL
- Students should understand soundness and completeness properties of inference mechanisms and the notions of decidability
- Students should learn in details first order resolution techniques
- The use of search in first order resolution should be discussed, including some search heuristics

At the end of this lesson the student should be able to do the following:

- Represent a natural language description as statements in first order logic
- Applying inference rules
- Implement automated theorem provers using resolution mechanism

Lesson 13

First Order Logic - I

6.2 First Order Logic

6.2.1 Syntax

Let us first introduce the symbols, or alphabet, being used. Beware that there are all sorts of slightly different ways to define FOL.

6.2.1.1 Alphabet

- Logical Symbols: These are symbols that have a standard meaning, like: AND, OR, NOT, ALL, EXISTS, IMPLIES, IFF, FALSE, =.
- Non-Logical Symbols: divided in:
 - Constants:
 - Predicates: 1-ary, 2-ary, .., n-ary. These are usually just identifiers.
 - Functions: 0-ary, 1-ary, 2-ary, .., n-ary. These are usually just identifiers. 0-ary functions are also called **individual constants**.

Where predicates return true or false, functions can return any value.

- Variables: Usually an identifier.

One needs to be able to distinguish the identifiers used for predicates, functions, and variables by using some appropriate convention, for example, capitals for function and predicate symbols and lower cases for variables.

6.2.1.2 Terms

A Term is either an individual constant (a 0-ary function), or a variable, or an n-ary function applied to n terms: $F(t_1 t_2 .. t_n)$
[We will use both the notation $F(t_1 t_2 .. t_n)$ and the notation $(F t_1 t_2 .. t_n)$]

6.2.1.3 Atomic Formulae

An Atomic Formula is either FALSE or an n-ary predicate applied to n terms: $P(t_1 t_2 .. t_n)$. In the case that "=" is a logical symbol in the language, $(t_1 = t_2)$, where t_1 and t_2 are terms, is an atomic formula.

6.2.1.4 Literals

A Literal is either an atomic formula (a **Positive Literal**), or the negation of an atomic formula (a **Negative Literal**). A **Ground Literal** is a variable-free literal.

6.2.1.5 Clauses

A Clause is a disjunction of literals. A **Ground Clause** is a variable-free clause. A **Horn Clause** is a clause with at most one positive literal. A **Definite Clause** is a Horn Clause with exactly one positive Literal.

Notice that implications are equivalent to Horn or Definite clauses:

(A IMPLIES B) is equivalent to ((NOT A) OR B)

(A AND B IMPLIES FALSE) is equivalent to ((NOT A) OR (NOT B)).

6.2.1.6 Formulae

A Formula is either:

- an atomic formula, or
- a **Negation**, i.e. the NOT of a formula, or
- a **Conjunctive Formula**, i.e. the AND of formulae, or
- a **Disjunctive Formula**, i.e. the OR of formulae, or
- an **Implication**, that is a formula of the form (formula1 IMPLIES formula2), or
- an **Equivalence**, that is a formula of the form (formula1 IFF formula2), or
- a **Universally Quantified Formula**, that is a formula of the form (ALL variable formula). We say that occurrences of variable are **bound** in formula [we should be more precise]. Or
- a **Existentially Quantified Formula**, that is a formula of the form (EXISTS variable formula). We say that occurrences of variable are **bound** in formula [we should be more precise].

An occurrence of a variable in a formula that is not bound, is said to be **free**. A formula where all occurrences of variables are bound is called a **closed formula**, one where all variables are free is called an **open formula**.

A formula that is the disjunction of clauses is said to be in **Clausal Form**. We shall see that there is a sense in which every formula is equivalent to a clausal form.

Often it is convenient to refer to terms and formulae with a single name. **Form** or **Expression** is used to this end.

6.2.2 Substitutions

- Given a term s, the result [**substitution instance**] of **substituting a term t in s for a variable x**, $s[t/x]$, is:
 - t, if s is the variable x
 - y, if s is the variable y different from x
 - $F(s_1[t/x] s_2[t/x] \dots s_n[t/x])$, if s is $F(s_1 s_2 \dots s_n)$.

- Given a formula A, the result (**substitution instance**) of **substituting a term t in A for a variable x**, $A[t/x]$, is:
 - FALSE, if A is FALSE,
 - $P(t_1[t/x] t_2[t/x] \dots t_n[t/x])$, if A is $P(t_1 t_2 \dots t_n)$,
 - $(B[t/x] \text{ AND } C[t/x])$ if A is $(B \text{ AND } C)$, and similarly for the other connectives,
 - $(\text{ALL } x \ B)$ if A is $(\text{ALL } x \ B)$, (similarly for EXISTS),
 - $(\text{ALL } y \ B[t/x])$, if A is $(\text{ALL } y \ B)$ and y is different from x (similarly for EXISTS).

The substitution $[t/x]$ can be seen as a map from terms to terms and from formulae to formulae. We can define similarly $[t_1/x_1 t_2/x_2 \dots t_n/x_n]$, where $t_1 t_2 \dots t_n$ are terms and $x_1 x_2 \dots x_n$ are variables, as a map, the **[simultaneous] substitution of x_1 by t_1 , x_2 by t_2 , ..., of x_n by t_n** . [If all the terms $t_1 \dots t_n$ are variables, the substitution is called an **alphabetic variant**, and if they are ground terms, it is called a **ground substitution**.] Note that a simultaneous substitution is not the same as a sequential substitution.

6.2.3 Unification

- Given two substitutions $S = [t_1/x_1 \dots t_n/x_n]$ and $V = [u_1/y_1 \dots u_m/y_m]$, the **composition** of S and V, $S . V$, is the substitution obtained by:
 - Applying V to $t_1 \dots t_n$ [the operation on substitutions with just this property is called **concatenation**], and
 - adding any pair u_j/y_j such that y_j is not in $\{x_1 \dots x_n\}$.

For example: $[G(x \ y)/z].[A/x \ B/y \ C/w \ D/z]$ is $[G(A \ B)/z \ A/x \ B/y \ C/w]$.

Composition is an operation that is associative and non commutative

- A set of forms $f_1 \dots f_n$ is **unifiable** iff there is a substitution S such that $f_1.S = f_2.S = \dots = f_n.S$. We then say that S is a **unifier** of the set.
For example $\{P(x \ F(y) \ B) \ P(x \ F(B) \ B)\}$ is unified by $[A/x \ B/y]$ and also unified by $[B/y]$.
- A **Most General Unifier** (MGU) of a set of forms $f_1 \dots f_n$ is a substitution S that unifies this set and such that for any other substitution T that unifies the set there is a substitution V such that $S.V = T$. The result of applying the MGU to the forms is called a **Most General Instance** (MGI). Here are some examples:
-

FORMULAE	MGU	MGI
$(P \ x), \ (P \ A)$	$[A/x]$	$(P \ A)$
$(P \ (F \ x) \ y \ (G \ y)), \ (P \ (F \ x) \ z \ (G \ x))$	$[x/y \ x/z]$	$(P \ (F \ x) \ x \ (G \ x))$
$(F \ x \ (G \ y)),$	$[(G \ u)/x \ y/z]$	$(F \ (G \ u) \ (G \ y))$

```

(F (G u) (G z))
-----
(F x (G y)),           Not Unifiable
(F (G u) (H z))
-----
(F x (G x) x),         Not Unifiable
(F (G u) (G (G z)) z)
-----
```

This last example is interesting: we first find that (G u) should replace x, then that (G z) should replace x; finally that x and z are equivalent. So we need x->(G z) and x->z to be both true. This would be possible only if z and (G z) were equivalent. That cannot happen for a finite term. To recognize cases such as this that do not allow unification [we cannot replace z by (G z) since z occurs in (G z)], we need what is called an Occur Test . Most Prolog implementation use Unification extensively but do not do the occur test for efficiency reasons.

The determination of Most General Unifier is done by the **Unification Algorithm**. Here is the pseudo code for it:

```
FUNCTION Unify WITH PARAMETERS form1, form2, and assign RETURNS MGU,
where form1 and form2 are the forms that we want to unify, and assign
is initially nil.
```

1. Use the Find-Difference function described below to determine the first elements where form1 and form2 differ and one of the elements is a variable. Call difference-set the value returned by Find-Difference. This value will be either the atom Fail, if the two forms cannot be unified; or null, if the two forms are identical; or a pair of the form (Variable Expression).
2. If Find-Difference returned the atom Fail, Unify also returns Fail and we cannot unify the two forms.
3. If Find-Difference returned nil, then Unify will return assign as MGU.
4. Otherwise, we replace each occurrence of Variable by Expression in form1 and form2; we compose the given assignment assign with the assignment that maps Variable into Expression, and we repeat the process for the new form1, form2, and assign.

```
FUNCTION Find-Difference WITH PARAMETERS form1 and form2 RETURNS pair,
where form1 and form2 are e-expressions.
```

1. If form1 and form2 are the same variable, return nil.
2. Otherwise, if either form1 or form2 is a variable, and it does not appear anywhere in the other form, then return the pair (Variable Other-Form), otherwise return Fail.
3. Otherwise, if either form1 or form2 is an atom then if they are the same atom then return nil otherwise return Fail.

4. Otherwise both form1 and form2 are lists.

Apply the Find-Difference function to corresponding elements of the two lists until either a call returns a non-null value or the two lists are simultaneously exhausted, or some elements are left over in one list.

In the first case, that non-null value is returned; in the second, nil is returned; in the third, Fail is returned

6.2.4 Semantics

Before we can continue in the "syntactic" domain with concepts like Inference Rules and Proofs, we need to clarify the Semantics, or meaning, of First Order Logic.

An **L-Structure** or **Conceptualization** for a language L is a structure $M = (U, I)$, where:

- U is a non-empty set, called the **Domain**, or **Carrier**, or **Universe of Discourse** of M, and
- I is an **Interpretation** that associates to each n-ary function symbol F of L a map

$$I(F) : \prod_{x_1 \in U} \dots \prod_{x_n \in U} U \rightarrow U$$

and to each n-ary predicate symbol P of L a subset of $\prod_{x_1 \in U} \dots \prod_{x_n \in U} U$.

The set of functions (predicates) so introduced form the **Functional Basis (Relational Basis)** of the conceptualization.

Given a language L and a conceptualization (U, I) , an **Assignment** is a map from the variables of L to U. An **X-Variant** of an assignment s is an assignment that is identical to s everywhere except at x where it differs.

Given a conceptualization $M = (U, I)$ and an assignment s it is easy to extend s to map each term t of L to an individual $s(t)$ in U by using induction on the structure of the term.

Then

- **M satisfies a formula A under s iff**
 - A is atomic, say $P(t_1 \dots t_n)$, and $(s(t_1) \dots s(t_n))$ is in $I(P)$.
 - A is $(\text{NOT } B)$ and M does not satisfy B under s.
 - A is $(B \text{ OR } C)$ and M satisfies B under s, or M satisfies C under s.
[Similarly for all other connectives.]
 - A is $(\text{ALL } x \text{ } B)$ and M satisfies B under all x-variants of s.
 - A is $(\text{EXISTS } x \text{ } B)$ and M satisfies B under some x-variants of s.
- **Formula A is satisfiable in M iff** there is an assignment s such that M satisfies A under s.
- **Formula A is satisfiable iff** there is an L-structure M such that A is satisfiable in M.

- **Formula A is valid or logically true in M** iff M satisfies A under any s. We then say that M is a **model** of A.
 - **Formula A is Valid or Logically True** iff for any L-structure M and any assignment s, M satisfies A under s.

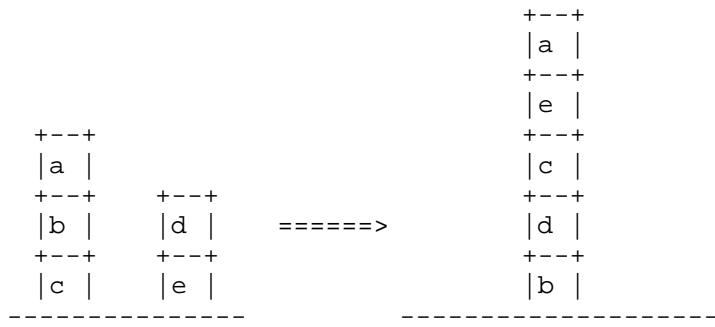
Some of these definitions can be made relative to a set of formulae GAMMA:

- **Formula A is a Logical Consequence of GAMMA in M** iff M satisfies A under any s that also satisfies all the formulae in GAMMA.
 - **Formula A is a Logical Consequence of GAMMA** iff for any L-structure M, A is a logical consequence of GAMMA in M. At times instead of "A is a logical consequence of GAMMA" we say "GAMMA entails A".

We say that formulae A and B are (logically) **equivalent** iff A is a logical consequence of {B} and B is a logical consequence of {A}.

EXAMPLE 1: A Block World

Here we look at a problem and see how to represent it in a language. We consider a simple world of blocks as described by the following figures:



We see two possible states of the world. On the left is the current state, on the right a desired new state. A robot is available to do the transformation. To describe these worlds we can use a structure with domain $U = \{a\ b\ c\ d\ e\}$, and with predicates {ON, ABOVE, CLEAR, TABLE} with the following meaning:

- **ON**: (**ON** x y) iff x is immediately above y.
The interpretation of **ON** in the left world is {(a b) (b c) (d e)}, and in the right world is {(a e) (e c) (c d) (d b)}.
 - **ABOVE**: (**ABOVE** x y) iff x is above y.
The interpretation of **ABOVE** [in the left world] is {(a b) (b c) (a c) (d e)} and in the right world is {(a e) (a c) (a d) (a b) (e c) (e d) (e b) (c d) (c b) (d b)}
 - **CLEAR**: (**CLEAR** x) iff x does not have anything above it.
The interpretation of **CLEAR** [in the left world] is {a d} and in the right world is {a}

- TABLE: (TABLE x) iff x rests directly on the table.
The interpretation of TABLE [in the left world] is {c e} and in the right world id {b}.

Examples of formulae true in the block world [both in the left and in the right state] are [these formulae are known as **Non-Logical Axioms**]:

- (ON x y) IMPLIES (ABOVE x y)
- ((ON x y) AND (ABOVE y z)) IMPLIES (ABOVE x z)
- (ABOVE x y) IMPLIES (NOT (ABOVE y x))
- (CLEAR x) IFF (NOT (EXISTS y (ON y x)))
- (TABLE x) IFF (NOT (EXISTS y (ON x y)))

Note that there are things that we cannot say about the block world with the current functional and predicate basis unless we use equality. Namely, we cannot say as we would like that a block can be ON at most one other block. For that we need to say that if x is ON y and x is ON z then y is z. That is, we need to use a logic with equality.

Not all formulae that are true on the left world are true on the right world and viceversa. For example, a formula true in the left world but not in the right world is (ON a b). Assertions about the left and right world can be in contradiction. For example (ABOVE b c) is true on left, (ABOVE c b) is true on right and together they contradict the non-logical axioms. This means that the theory that we have developed for talking about the block world can talk of only one world at a time. To talk about two worlds simultaneously we would need something like the Situation Calculus that we will study later.

Module 6

Knowledge Representation and Logic – (First Order Logic)

Lesson 14

First Order Logic - II

6.2.5 Herbrand Universe

It is a good exercise to determine for given formulae if they are satisfied/valid on specific L-structures, and to determine, if they exist, models for them. A good starting point in this task, and useful for a number of other reasons, is the **Herbrand Universe** for this set of formulae. Say that $\{F_01 \dots F_{0n}\}$ are the individual constants in the formulae [if there are no such constants, then introduce one, say, F_0]. Say that $\{F_1 \dots F_m\}$ are all the non 0-ary function symbols occurring in the formulae. Then the set of (constant) terms obtained starting from the individual constants using the non 0-ary functions, is called the Herbrand Universe for these formulae.

For example, given the formula $(P x A) \text{ OR } (Q y)$, its Herbrand Universe is just $\{A\}$. Given the formulae $(P x (F y)) \text{ OR } (Q A)$, its Herbrand Universe is $\{A (F A) (F (F A)) (F (F (F A))) \dots\}$.

Reduction to Clausal Form

In the following we give an algorithm for deriving from a formula an equivalent clausal form through a series of truth preserving transformations.

We can state an (unproven by us) theorem:

Theorem: Every formula is equivalent to a clausal form

We can thus, when we want, restrict our attention only to such forms.

6.2.6 Deduction

An **Inference Rule** is a rule for obtaining a new formula [**the consequence**] from a set of given formulae [**the premises**].

A most famous inference rule is **Modus Ponens**:

$$\{A, \text{ NOT } A \text{ OR } B\}$$

B

For example:

$$\{\text{Sam is tall,}$$

$$\quad \text{if Sam is tall then Sam is unhappy}\}$$

Sam is unhappy

When we introduce inference rules we want them to be **Sound**, that is, we want the consequence of the rule to be a logical consequence of the premises of the rule. Modus Ponens is sound. But the following rule, called **Abduction**, is not:

$$\{B, \text{ NOT } A \text{ OR } B\}$$

A

is not. For example:

```
John is wet  
If it is raining then John is wet  
-----  
It is raining
```

gives us a conclusion that is usually, but not always true [John takes a shower even when it is not raining].

A **Logic or Deductive System** is a language, plus a set of inference rules, plus a set of logical axioms [formulae that are valid].

A **Deduction or Proof or Derivation** in a deductive system D, given a set of formulae GAMMA, is a sequence of formulae B1 B2 .. Bn such that:

- for all i from 1 to n, Bi is either a logical axiom of D, or an element of GAMMA, or is obtained from a subset of {B1 B2 .. Bi-1} by using an inference rule of D.

In this case we say that Bn is **Derived** from GAMMA in D, and in the case that GAMMA is empty, we say that Bn is a **Theorem** of D.

6.2.7 Soundness, Completeness, Consistency, Satisfiability

A Logic D is **Sound** iff for all sets of formulae GAMMA and any formula A:

- if A is derived from GAMMA in D, then A is a logical consequence of GAMMA

A Logic D is **Complete** iff for all sets of formulae GAMMA and any formula A:

- If A is a logical consequence of GAMMA, then A can be derived from GAMMA in D.

A Logic D is **Refutation Complete** iff for all sets of formulae GAMMA and any formula A:

- If A is a logical consequence of GAMMA, then the union of GAMMA and (NOT A) is inconsistent

Note that if a Logic is Refutation Complete then we can enumerate all the logical consequences of GAMMA and, for any formula A, we can reduce the question if A is or not a logical consequence of GAMMA to the question: the union of GAMMA and NOT A is or not consistent.

We will work with logics that are both Sound and Complete, or at least Sound and Refutation Complete.

A **Theory** T consists of a logic and of a set of Non-logical axioms. For convenience, we may refer, when not ambiguous, to the logic of T , or the non-logical axioms of T , just as T .

The common situation is that we have in mind a well defined "world" or set of worlds. For example we may know about the natural numbers and the arithmetic operations and relations. Or we may think of the block world. We choose a language to talk about these worlds. We introduce function and predicate symbols as it is appropriate. We then introduce formulae, called **Non-Logical Axioms**, to characterize the things that are true in the worlds of interest to us. We choose a logic, hopefully sound and (refutation) complete, to derive new facts about the worlds from the non-logical axioms.

A **Theorem** in a theory T is a formula A that can be derived in the logic of T from the non-logical axioms of T .

A Theory T is **Consistent** iff there is no formula A such that both A and $\text{NOT } A$ are theorems of T ; it is **Inconsistent** otherwise. If a theory T is inconsistent, then, for essentially any logic, any formula is a theorem of T . [Since T is inconsistent, there is a formula A such that both A and $\text{NOT } A$ are theorems of T . It is hard to imagine a logic where from A and $(\text{NOT } A)$ we cannot infer FALSE, and from FALSE we cannot infer any formula. We will say that a logic that is at least this powerful is **Adequate**.]

A Theory T is **Unsatisfiable** if there is no structure where all the non-logical axioms of T are valid. Otherwise it is **Satisfiable**.

Given a Theory T , a formula A is a **Logical Consequence of T** if it is a logical consequence of the non logical axioms of T .

Theorem: If the logic we are using is sound then:

1. If a theory T is satisfiable then T is consistent
2. If the logic used is also adequate then if T is consistent then T is satisfiable
3. If a theory T is satisfiable and by adding to T the non-logical axiom $(\text{NOT } A)$ we get a theory that is not satisfiable Then A is a logical consequence of T .
4. If a theory T is satisfiable and by adding the formula $(\text{NOT } A)$ to T we get a theory that is inconsistent, then A is a logical consequence of T .

Module 6

Knowledge Representation and Logic – (First Order Logic)

Lesson 15

Inference in FOL - I

6.2.8 Resolution

We have introduced the inference rule Modus Ponens. Now we introduce another inference rule that is particularly significant, Resolution.

Since it is not trivial to understand, we proceed in two steps. First we introduce Resolution in the Propositional Calculus, that is, in a language with only truth valued variables. Then we generalize to First Order Logic.

6.2.8.1 Resolution in the Propositional Calculus

In its simplest form Resolution is the inference rule:

$$\begin{array}{c} \{A \text{ OR } C, B \text{ OR } (\text{NOT } C)\} \\ \hline A \text{ OR } B \end{array}$$

More in general the **Resolution Inference Rule** is:

- Given as premises the clauses C1 and C2, where C1 contains the literal L and C2 contains the literal (NOT L), infer the clause C, called the **Resolvent** of C1 and C2, where C is the union of (C1 - {L}) and (C2 - {(NOT L)})

In symbols:

$$\begin{array}{c} \{C_1, C_2\} \\ \hline (C_1 - \{L\}) \text{ UNION } (C_2 - \{(\text{NOT } L)\}) \end{array}$$

Example:

The following set of clauses is inconsistent:

1. $(P \text{ OR } (\text{NOT } Q))$
2. $((\text{NOT } P) \text{ OR } (\text{NOT } S))$
3. $(S \text{ OR } (\text{NOT } Q))$
4. Q

In fact:

5. $((\text{NOT } Q) \text{ OR } (\text{NOT } S))$ from 1. and 2.
6. $(\text{NOT } Q)$ from 3. and 5.
7. FALSE from 4. and 6.

Notice that 7. is really the empty clause [why?].

Theorem: The Propositional Calculus with the Resolution Inference Rule is sound and Refutation Complete.

NOTE: This theorem requires that clauses be represented as sets, that is, that each element of the clause appear exactly once in the clause. This requires some form of membership test when elements are added to a clause.

$$C_1 = \{P, P\}$$

$$\begin{aligned} C_2 &= \{(NOT P) (NOT P)\} \\ C &= \{P (NOT P)\} \end{aligned}$$

From now on by resolution we just get again C1, or C2, or C.

6.2.8.2 Resolution in First Order Logic

Given clauses C1 and C2, a clause C is a **RESOLVENT** of C1 and C2, if

1. There is a subset $C_1' = \{A_1, \dots, A_m\}$ of C1 of literals of the same sign, say positive, and a subset $C_2' = \{B_1, \dots, B_n\}$ of C2 of literals of the opposite sign, say negative,
2. There are substitutions s_1 and s_2 that replace variables in C_1' and C_2' so as to have new variables,
3. C_2'' is obtained from C2 removing the negative signs from $B_1 \dots B_n$
4. There is an Most General Unifier s for the union of $C_1'.s_1$ and $C_2''.s_2$

and C is

$$((C_1 - C_1').s_1 \text{ UNION } (C_2 - C_2').s_2).s$$

In symbols this **Resolution** inference rule becomes:

$$\frac{\begin{array}{c} \{C_1, C_2\} \\ \hline \end{array}}{C}$$

If C_1' and C_2' are singletons (i.e. contain just one literal), the rule is called **Binary Resolution**.

Example:

$$\begin{aligned} C_1 &= \{(P z (F z)) (P z A)\} \\ C_2 &= \{(NOT (P z A)) (NOT (P z x)) (NOT (P x z))\} \\ C_1' &= \{(P z A)\} \\ C_2' &= \{(NOT (P z A)) (NOT (P z x))\} \\ C_2'' &= \{(P z A) (P z x)\} \\ s_1 &= [z1/z] \\ s_2 &= [z2/z] \\ C_1'.s_1 \text{ UNION } C_2'.s_2 &= \{(P z1 A) (P z2 A) (P z2 x)\} \\ s &= [z1/z2 A/x] \\ C &= \{(NOT (P A z1)) (P z1 (F z1))\} \end{aligned}$$

Notice that this application of Resolution has eliminated more than one literal from C2, i.e. it is not a binary resolution.

Theorem: First Order Logic, with the Resolution Inference Rule, is sound and refutation complete.

We will not develop the proof of this theorem. We will instead look at some of its steps, which will give us a wonderful opportunity to revisit Herbrand. But before that let's observe that in a sense, if we replace in this theorem "Resolution" by "Binary Resolution", then the theorem does not hold and Binary Resolution is not Refutation Complete. This is the case when in the implementation we do not use sets but instead use bags. This can be shown using the same example as in the case of propositional logic.

Given a clause C, a subset D of C, and a substitution s that unifies D, we define C.s to be a **Factor** of C.

The **Factoring Inference Rule** is the rule with premise C and as consequence C.s.

Theorem: For any set of clauses S and clause C, if C is derivable from S using Resolution, then C is derivable from S using Binary Resolution and Factoring.

When doing proofs it is efficient to have as few clauses as possible. The following definition and rule are helpful in eliminating redundant clauses:

A clause C1 **Subsumes** a clause C2 iff there is a substitution s such that C1.s is a subset of C2.

Subsumption Elimination Rule: If C1 subsumes C2 then eliminate C2.

Herbrand Revisited

We have presented the concept of Herbrand Universe H_S for a set of clauses S. Here we meet the concept of **Herbrand Base**, $H(S)$, for a set of clauses S. $H(S)$ is obtained from S by considering the ground instances of the clauses of S under all the substitutions that map all the variables of S into elements of the Herbrand universe of S. Clearly, if in S occurs some variable and the Herbrand universe of S is infinite then $H(S)$ is infinite.

[NOTE: Viceversa, if S has no variables, or S has variables and possibly individual constants, but no other function symbol, then $H(S)$ is finite. If $H(S)$ is finite then we can, as we will see, decide if S is or not satisfiable.]

[NOTE: it is easy to determine if a finite subset of $H(S)$ is satisfiable: since it consists of ground clauses, the truth table method works now as in propositional cases.]

The importance of the concepts of Herbrand Universe and of Herbrand Base is due to the following theorems:

Herbrand Theorem: If a set S of clauses is unsatisfiable then there is a finite subset of $H(S)$ that is also unsatisfiable.

Because of the theorem, when $H(S)$ is finite we will be able to decide if S is or not satisfiable. Herbrand theorem immediately suggests a general refutation complete proof procedure:

given a set of clauses S, enumerate subsets of $H(S)$ until you find one that is unsatisfiable.

But, as we shall soon see, we can come up with a better refutation complete proof procedure.

Refutation Completeness of the Resolution Proof Procedure

Given a set of clauses S , the **Resolution Closure** of S , $R(S)$, is the smallest set of clauses that contains S and is closed under Resolution. In other words, it is the set of clauses obtained from S by applying repeatedly resolution.

Ground Resolution Theorem: If S is an unsatisfiable set of ground clauses, then $R(S)$ contains the clause FALSE.

In other words, there is a resolution deduction of FALSE from S .

Lifting Lemma: Given clauses C_1 and C_2 that have no variables in common, and ground instances C'_1 and C'_2 , respectively, of C_1 and C_2 , if C' is a resolvent of C'_1 and C'_2 , then there is a clause C which is a resolvent of C_1 and C_2 which has C' as a ground instance

With this we have our result, that the Resolution Proof procedure is Refutation Complete. Note the crucial role played by the Herbrand Universe and Basis. Unsatisfiability of S is reduced to unsatisfiability for a finite subset $H_S(S)$ of $H(S)$, which in turn is reduced to the problem of finding a resolution derivation for FALSE in $H_S(S)$, derivation which can be "lifted" to a resolution proof of FALSE from S .

Dealing with Equality

Up to now we have not dealt with equality, that is, the ability to recognize terms as being equivalent (i.e. always denoting the same individual) on the basis of some equational information. For example, given the information that

$$S(x) = x+1$$

then we can unify:

$$F(S(x), y) \text{ and } F(x+1, 3).$$

There are two basic approaches to dealing with this problem.

- The first is to add inference rules to help us replace terms by equal terms. One such rule is the **Demodulation Rule**: Given terms t_1 , t_2 , and t_3 where t_1 and t_2 are unifiable with MGU s , and t_2 occurs in a formula A , then

$$\frac{\{t_1 = t_3, A(\dots t_2 \dots)\}}{A(\dots t_3.s \dots)}$$

Another more complex, and useful, rule is **Paramodulation**.

- The second approach is not to add inference rules and instead to add non-logical axioms that characterize equality and its effect for each non logical symbol. We first establish the reflexive, symmetric and transitive properties of "=":

$x=x$
 $x=y \text{ IMPLIES } y=x$
 $x=y \text{ AND } y=z \text{ IMPLIES } x=z$

Then for each unary function symbol F we add the equality axiom

$x=y \text{ IMPLIES } F(x)=F(y)$

Then for each binary function symbol F we add the equality axiom

$x=z \text{ AND } y=w \text{ IMPLIES } F(x,y)=F(z,w)$

And similarly for all other function symbols.

The treatment of predicate symbols is similar. For example, for the binary predicate symbol P we add

$x=z \text{ AND } y=w \text{ IMPLIES } (P(x,y) \text{ IFF } P(z,w))$

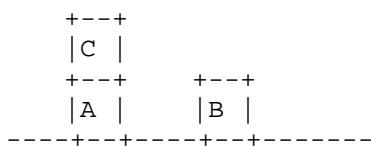
Whatever approach is chosen, equality substantially complicates proofs.

Answering True/False Questions

If we want to show that a clause C is derivable from a set of clauses $S=\{C_1 C_2 \dots C_n\}$, we add to S the clauses obtained by negating C, and apply resolution to the resulting set S' of clauses until we get the clause FALSE.

Example:

We are back in the Block World with the following state



which gives us the following State Clauses:

- ON(C A)
- ONTABLE(A)
- ONTABLE(B)
- CLEAR(C)
- CLEAR(B)

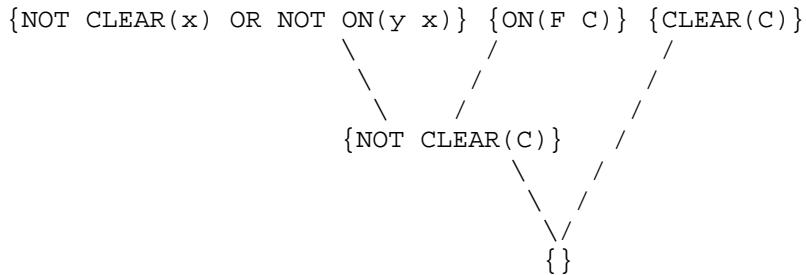
In addition we consider the non-logical axiom:

- $(\text{ALL } x (\text{CLEAR}(x) \text{ IMPLIES } (\text{NOT } (\text{EXISTS } y \text{ ON}(y x)))))$

which in clause form becomes

- $\text{NOT CLEAR}(x) \text{ OR NOT ON}(y x)$

If we now ask whether $(\text{NOT } (\text{EXISTS } y (\text{ON}(y C))))$, we add to the clauses considered above the clause $\text{ON}(F C)$ and apply resolution:



Example:

We are given the following predicates:

- $S(x)$: x is Satisfied
- $H(x)$: x is Healthy
- $R(x)$: x is Rich
- $P(x)$: x is Philosophical

The premises are the non-logical axioms:

- $S(x) \text{ IMPLIES } (H(x) \text{ AND } R(x))$
- $\text{EXISTS } x (S(x) \text{ and } P(x))$

The conclusion is

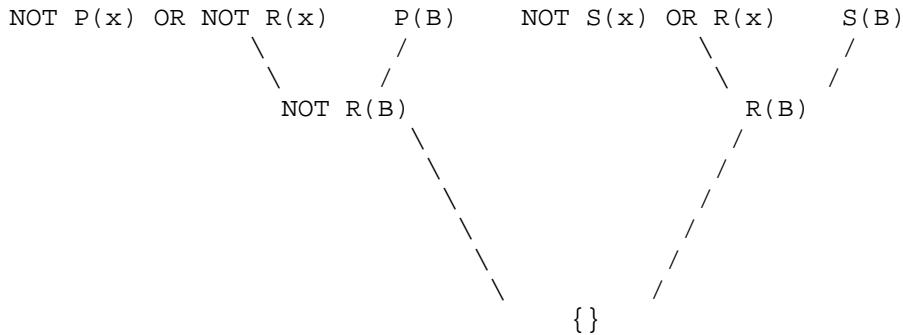
- $\text{EXISTS } x (P(x) \text{ AND } R(x))$

The corresponding clauses are:

1. $\text{NOT } S(x) \text{ OR } H(x)$
2. $\text{NOT } S(x) \text{ OR } R(x)$
3. $S(B)$
4. $P(B)$
5. $\text{NOT } P(x) \text{ OR NOT } R(x)$

where B is a Skolem constant.

The proof is then:



Answering Fill-in-Blanks Questions

We now determine how we can identify individual(s) that satisfy specific formulae.

EXAMPLE:

NON-LOGICAL SYMBOLS:

- $SW(x y)$: x is staying with y
- $A(x y)$: x is at place y
- $R(x y)$: x can be reached at phone number y
- $PH(x)$: the phone number for place x
- Sally, Morton, UnionBldg: Individuals

NON-LOGICAL AXIOMS:

1. $SW(\text{Sally Morton})$
2. $A(\text{Morton UnionBldg})$
3. $SW(x \quad y) \quad \text{AND} \quad A(y \quad z) \quad \text{IMPLIES} \quad A(x \quad z)$,
which is equivalent to the clause
 1. $\text{NOT } SW(x y) \text{ OR NOT } A(y z) \text{ OR } A(x z)$
4. $A(x y) \text{ IMPLIES } R(x PH(y))$, which is equivalent to the clause
 1. $\text{NOT } A(u v) \text{ OR } R(u PH(v))$

GOAL: Determine where to call Sally

- $\text{NOT EXISTS } x R(\text{Sally } x)$, which is equivalent to the clause
 1. $\text{NOT } R(\text{Sally } w)$.

To this clause we add as a disjunct the literal, **Answer Literal**, $\text{Ans}(w)$ to obtain the clause :

5. $\text{Ans}(w) \text{ OR NOT } R(\text{Sally } w)$.

PROOF

6. $\text{Ans}(v) \text{ OR NOT } A(\text{Sally } v)$, from 4. and 5.
7. $\text{Ans}(v) \text{ OR NOT } SW(\text{Sally } y) \text{ OR NOT } A(y \ v)$, from 6. and 3.
8. $\text{Ans}(v) \text{ OR NOT } A(\text{Morton } v)$, from 7. and 1.
9. $\text{Ans}(\text{UnionBldg})$, from 8. and 2.

The proof procedure terminates when we get a clause that is an instance of the Answer Literal. 9. and gives us the place where we can call Sally.

General Method

If A is the Fill-In-Blanks question that we need to answer and $x_1 \dots x_n$ are the free variables occurring in A , then we add to the Non-Logical axioms and Facts GAMMA the clause

$$\text{NOT } A \text{ OR } \text{ANS}(x_1 \dots x_n)$$

We terminate the proof when we get a clause of the form

$$\text{ANS}(t_1 \dots t_n)$$

$t_1 \dots t_n$ are terms that denote individuals that simultaneously satisfy the query for, respectively $x_1 \dots x_n$.

We can obtain all the individuals that satisfy the original query by continuing the proof looking for alternative instantiations for the variables $x_1 \dots x_n$.

If we build the proof tree for $\text{ANS}(t_1 \dots t_n)$ and consider the MGUs used in it, the composition of these substitutions, restricted to $x_1 \dots x_n$, gives us the individuals that answer the original Fill-In-Blanks question.

Module 6

Knowledge Representation and Logic – (First Order Logic)

Lesson 16

Inference in FOL - II

6.2.9 Proof as Search

Up to now we have exhibited proofs as if they were found miraculously. We gave formulae and showed proofs of the intended results. We did not exhibit how the proofs were derived.

We now examine how proofs can actually be found. In so doing we stress the close ties between theorem proving and search.

A General Proof Procedure

We use binary resolution [we represent clauses as sets] and we represent the proof as a tree, the **Proof Tree**. In this tree nodes represent clauses. We start with two disjoint sets of clauses INITIAL and OTHERS.

1. We create a node START and introduce a hyperarc from START to new nodes, each representing a distinct element of INITIAL. We put in a set OPEN all these new nodes. These nodes are called **AND Nodes**.
2. If OPEN is empty, we terminate. Otherwise we remove an element N from OPEN using a selection function SELECT.
3. If N is an AND node, we connect N with a single hyperarc to new nodes N₁ ... N_m, one for each literal in the clause C represented at N. These nodes are also labeled with C and are called **OR Nodes**. All of these nodes are placed into OPEN.

[NOTE 1: In the case that C consists of a single literal, we can just say that N is now an OR node.]

[NOTE 2: One may decide not to create all the nodes N₁ .. N_m in a single action and instead to choose one of the literals of C and to create a node N_i to represent C with this choice of literal. N_i is inserted into OPEN. N also goes back into OPEN if not all of its literals have been considered. The rule used to choose the literal in C is called a **Selection Rule**]

Repeat from 2.

4. If N is an OR node, say, corresponding to the ith literal of a clause C, we consider all the possible ways of applying binary resolution between C and clauses from the set OTHERS, resolving on the ith literal of C.

Let D₁ .. D_p be the resulting clauses. We represent each of these clauses D_j by an AND node N(D_j) as in 1. above. We put an arc from N to N(D_j). We set OPEN to NEXT-OPEN(OPEN, C, {D₁, .., D_p}). We set OTHERS to NEXT-OITHERS(OTHERS, C, {D₁, .., D_p}). If in the proof tree we have, starting at START, a hyperpath (i.e. a path that may include hyperarcs) whose leaves have all label {}, we terminate with success.

[NOTE 3: One may decide, instead of resolving C on its ith literal with all possible element of OTHERS, to select one compatible element of OTHERS and to resolve C with it, putting this resolvent and C back into OPEN. We would call a rule for selecting an element of OTHERS a **Search Rule**.]

Repeat from 2.

In this proof procedure we have left indetermined:

- The sets INITIAL and OTHERS of clauses
- The function SELECT by which we choose which node to expand
- The function NEXT-OPEN for computing the next value of OPEN
- The function NEXT-OTHERS for computing the next value of OTHERS

There is no guaranty that for any choice of INITIAL, OTHERS, SELECT, NEXT-OPEN and NEXT-OTHERS the resulting theorem prover will be "complete", i.e. that everything that is provable will be provable with this theorem prover.

Example

Suppose that we want to prove that

1. NOT P(x) OR NOT R(x)
is inconsistent with the set of clauses:
2. NOT S(x) OR H(x)
3. NOT S(x) OR R(x)
4. S(b)
5. P(b)

The following are possible selections for the indeterminates:

INITIAL: {1.}, that is, it consists of the clauses representing the negation of the goal.

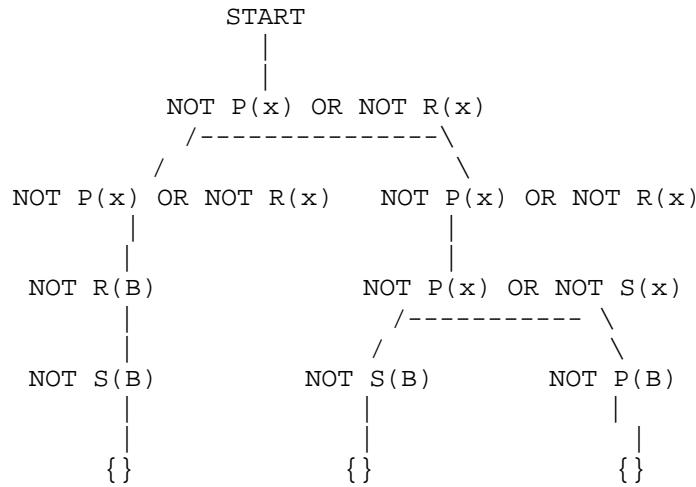
OTHERS: {2. 3. 4. 5.}, that is, it consists of the non-logical axioms.

SELECT: We use OPEN as a FIFO queue, i.e. we do breadth-first search.

NEXT-OPEN: It sets NEXT-OPEN(OPEN, C, {D1, .., Dp}) to the union of OPEN, {C}, and {D1, .., Dp}.

NEXT-OTHERS: It leaves OTHERS unchanged

The Proof Tree is then (we underline AND nodes and all their outgoing arcs are assumed to form a single hyperlink)



At an OR node we apply resolution between the current clause at its selected literal and all the compatible elements of OTHERS.

Example

The following example uses Horn clauses in propositional logic. I use the notation that is common in Prolog: we represent the implication:

A1 AND A2 AND .. AND An IMPLIES A

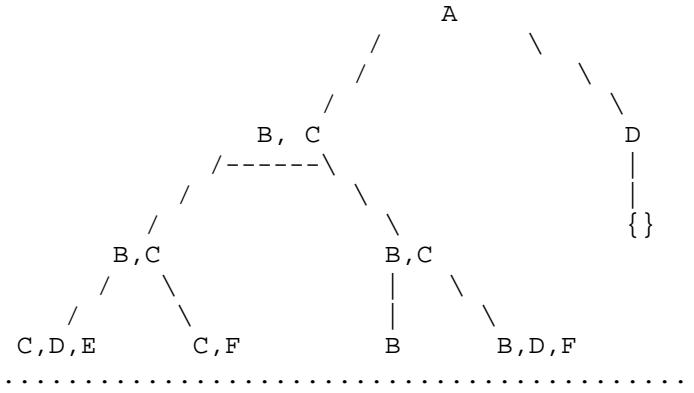
as

$A \leq A_1, A_2, \dots, A_n$

Our problem is:

1. A, This is what we want to prove
2. $A \leq B, C$
3. $A \leq D$
4. $B \leq D, E$
5. $B \leq F$
6. $C \leq$
7. $C \leq D, F$
8. $D \leq$
9. $F \leq$

We now partially represent the proof tree. We do not apply breadth first because we want to see how great is the impact of the choice of SELECT.



You can keep on expanding the tree and see how depth first would generate a large tree while breadth first rapidly derives D from A, and {} from D.

In other circumstances other strategies would be appropriate as we see below.

6.2.10 Some Proof Strategies

From the early sixties people have looked for strategies that would simplify the search problem in theorem proving. Here are some of the strategies suggested.

6.2.10.1 Unit Preference

When we apply resolution if one of the premises is a unit clause (it has a single literal), the resolvent will have one less literal than the largest of the premises, thus getting closer to the desired goal {}. Thus it appears desirable to use resolution between clauses one of which is the unit clause. This unit preference is applied both when selecting from the OPEN set (i.e. at the leaves of the tree) and when we at an OR node we select an element of OTHERS to resolve with it.

6.2.10.2 Set of Support Strategy

When we use the Set of Support Strategy we have:

NEXT-OPEN(OPEN, C, {D₁, ..., D_p}) is the union of OPEN, {C}, and {D₁, ..., D_p}

NEXT-OTHERS(OTHERS, C, {D₁, ..., D_p}) is OTHERS.

In simple words, the set of support strategy uses the OPEN set as its set of support. Each application of resolution has as a premise an element of the set of support and adds that premise and the resolvent to the set of support.

Usually the INITIAL set (i.e. the initial set of support) consists of all the clauses obtained by negating the intended "theorem".

The set of support strategy is complete if

- The OTHERS set of clauses is satisfiable. That is the case when we are given a satisfiable set of non-logical axioms and we use it as OTHERS.

6.2.10.3 Input Resolution

In Input Resolution:

- INITIAL consists of the union of the negation of the "theorem" and of the set of non-logical axioms.
- OTHERS usually is the set of non-logical axioms.
- NEXT-OPEN(OPEN, C, {D₁,...,D_p}) is the union of OPEN and {C}, that is, OPEN does not change in successive iterations
- NEXT-OTHERS(OTHERS, C, {D₁,...,D_p}) is the union of OTHERS, {C}, and {D₁,...,D_p}.

In other words, in each resolution one of the premises is one of the original clauses.

In general Input Resolution is incomplete, as it can be seen with the following unsatisfiable set of clauses (from Nilsson) from which we are unable to derive FALSE using Input Resolution:

- Q(u) OR P(A)
- NOT Q(w) OR P(w)
- NOT Q(x) OR NOT P(x)
- Q(y) OR NOT P(y)

[You can use this set of clauses as both OPEN and OTHERS.]

Input resolution is complete in the case that all the clauses are Horn clauses.

6.2.10.4 Linear Resolution

Linear Resolution, also known as Ancestry-Filtered Form Strategy, is a generalization of Input Resolution. The generalization is that now in each resolution one of the clauses is one of the original clauses or it is an ancestor in the proof tree of the second premise.

Linear Resolution is complete.

6.2.10.5 SLD-Resolution

Selected Linear Resolution for Definite Clauses, or simply SLD-Resolution, is the basis for a proof procedure for theorems that are in the form of conjunctions of positive literals (this conjunction is called a **Goal**) given non-logical axioms that are definite clauses.

More precisely:

- A Goal is the conjunction of positive literals
- A **Selection Rule** is a method for selecting one of the literals of a goal (the first literal, or the last literal, etc.)

- In SLD-Resolution, given a goal $G = A_1 \dots A_n$, a definite clause $C: A \leq B_1 \dots B_m$, and a subgoal A_i selected from G using a Selection Rule S , where A_i and A are unifiable with MGU s , the **Resolvent of G and C under S** is

$(A_1 \dots A_{i-1} B_1 \dots B_m A_{i+1} \dots A_n).s$

- An **SLD-Derivation** of a goal G given a selection rule S and a set P of definite clauses, is a sequence of triples (G_i, C_i, s_i) , for i from 0 to k , where
 - G_0 is G
 - for each $i > 0$, C_i is obtained from a clause in P by replacement of all of its variables with new variables
 - G_{i+1} is the SLD-Resolvent of G_i and C_i by use of S and with MGU s_i .
- A **Refutation** of a goal G given definite clauses P and selection rule S , is a finite SLD-derivation of G given S and P whose last goal is the null clause.
If $s_1 \dots s_k$ are the MGUs used in the refutation, then $s = s_1.s_2\dots.s_k$ is a substitution that, restricted to the variables of G , makes G true whenever the clauses of P are true.
- The goal G **succeeds** for given selection rule S and set of definite clauses P if it has a refutation for P and S ; otherwise it **Fails**.

Theorem: SLD-Resolution is Sound and Complete for conjunctive positive goals and definite clauses.

An important consequence of this theorem is that it remains true no matter the selection rule we use to select literals in goals. Thus we can select literals as we please, for instance left-to right. An other important aspect is that the substitution $s = s_1.s_2\dots.s_n$ gives us a method for finding the individuals that satisfy the goal in the structures that satisfy the clauses in P .

Nothing has been said in SLD-Resolution about what rule should be used to select the clause of P to resolve with the current literal of the current goal (such a rule is called a **Search rule**).

Example

Suppose that we are given the following clauses:

- WOMAN(MOTHER(v)) ; Every mother is a woman
- GOOD(HUSBAND(ANN)) ; The husband of Ann is good
- GOOD(z) \leq LIKES(MOTHER(z), z); if z likes his mother then z is good

and we want to find out a woman that likes's someone's husband.

The goal can be stated as:

- WOMAN(x),LIKES(x,HUSBAND(y))
[NOTE: the variables x and y are implicitly existentially quantified.]

The SLD-Derivation is:

```
((WOMAN(x), LIKES(x, HUSBAND(y))), WOMAN(MOTHER(v)), [MOTHER(v)/x])
((LIKES(MOTHER(v), HUSBAND(y))), GOOD(z) <= LIKES(MOTHER(z), z),
[HUSBAND(y)/v, HUSBAND(y)/z])
((GOOD(HUSBAND(y))), GOOD(HUSBAND(ANN)), [ANN/y])
{})
```

and the substitution is:

[MOTHER(HUSBAND(ANN))/x, ANN/y]

6.2.11 Non-Monotonic Reasoning

First order logic and the inferences we perform on it is an example of monotonic reasoning.

In monotonic reasoning if we enlarge at set of axioms we cannot retract any existing assertions or axioms.

Humans do not adhere to this monotonic structure when reasoning:

- we need to jump to conclusions in order to plan and, more basically, survive.
 - we cannot anticipate all possible outcomes of our plan.
 - we must make assumptions about things we do not specifically know about.

6.2.11.1 Default Reasoning

This is a very common form of non-monotonic reasoning. Here *We want to draw conclusions based on what is most likely to be true.*

We have already seen examples of this and possible ways to represent this knowledge.

We will discuss two approaches to do this:

- Non-Monotonic logic.
- Default logic.

DO NOT get confused about the label *Non-Monotonic* and *Default* being applied to reasoning and a particular logic. Non-Monotonic reasoning is generic descriptions of a class of reasoning. Non-Monotonic logic is a specific theory. The same goes for Default reasoning and Default logic.

Non-Monotonic Logic

This is basically an extension of first-order predicate logic to include a *modal* operator, M . The purpose of this is to allow for consistency.

For example: $\forall x: \text{plays_instrument}(x) \wedge M \text{improvises}(x) \rightarrow \text{jazz_musician}(x)$

states that for all x if x plays an instrument and if the fact that x can improvise is consistent with all other knowledge then we can conclude that x is a jazz musician.

How do we define *consistency*?

One common solution (consistent with PROLOG notation) is

to show that fact P is true attempt to prove $\neg P$. If we fail we may say that P is consistent (since $\neg P$ is false).

However consider the famous set of assertions relating to President Nixon.

$\forall x: \text{Republican}(x) \wedge M \neg \text{Pacifist}(x) \rightarrow \neg \text{Pacifist}(x)$

$\forall x: \text{Quaker}(x) \wedge M \text{Pacifist}(x) \rightarrow \neg \text{Pacifist}(x)$

Now this states that Quakers tend to be pacifists and Republicans tend not to be.

BUT Nixon was both a Quaker and a Republican so we could assert:

$\text{Quaker}(\text{Nixon})$

$\text{Republican}(\text{Nixon})$

This now leads to our total knowledge becoming inconsistent.

Default Logic

Default logic introduces a new inference rule:

$$\frac{A \vdash B}{C}$$

which states if A is deducible and it is consistent to assume B then conclude C .

Now this is similar to Non-monotonic logic but there are some distinctions:

- New inference rules are used for computing the set of plausible extensions. So in the Nixon example above Default logic can support both assertions since it does not say anything about how choose between them -- it will depend on the inference being made.
- In Default logic any nonmonotonic expressions are rules of inference rather than expressions

6.2.11.2 Circumscription

Circumscription is a rule of conjecture that allows you to jump to the conclusion that the objects you can show that posses a certain property, p , are in fact all the objects that posses that property.

Circumscription can also cope with default reasoning.

Suppose we know: $\text{bird}(\text{tweety})$

$\forall x: \text{penguin}(x) \rightarrow \text{bird}(x)$

$\forall x: \text{penguin}(x) \rightarrow \neg \text{flies}(x)$

and we wish to add the fact that *typically, birds fly*.

In circumscription this phrase would be stated as:

A bird will fly if it is not abnormal

and can thus be represented by:

$\forall x: \text{bird}(x) \wedge \neg \text{abnormal}(x) \rightarrow \text{flies}(x).$

However, this is not sufficient

We cannot conclude

$\text{flies}(\text{tweety})$

since we cannot prove

$\neg \text{abnormal}(\text{tweety}).$

This is where we apply circumscription and, in this case,

we will assume that those things that are shown to be abnormal are the only things to be abnormal

Thus we can rewrite our *default rule* as:

$$\forall x: \text{bird}(x) \wedge \neg \text{flies}(x) \rightarrow \text{abnormal}(x)$$

and add the following

$$\forall x: \neg \text{abnormal}(x)$$

since there is nothing that cannot be shown to be abnormal.

If we now add the fact:

$$\text{penguin}(\text{tweety})$$

Clearly we can prove

$$\text{abnormal}(\text{tweety}).$$

If we circumscribe abnormal now we would add the sentence,

a penguin (tweety) is the abnormal thing:

$$\forall x: \text{abnormal}(x) \rightarrow \text{penguin}(x).$$

Note the distinction between Default logic and circumscription:

Defaults are sentences in language itself not additional inference rules.

6.2.11.3 Truth Maintenance Systems

A variety of *Truth Maintenance Systems* (TMS) have been developed as a means of implementing Non-Monotonic Reasoning Systems.

Basically TMSs:

- all do some form of dependency directed backtracking
- assertions are connected via a network of dependencies.

Justification-Based Truth Maintenance Systems (JTMS)

- This is a simple TMS in that it does not know anything about the structure of the assertions themselves.
- Each supported belief (assertion) in has a justification.
- Each justification has two parts:

- An *IN-List* -- which supports beliefs held.
- An *OUT-List* -- which supports beliefs *not* held.
- An assertion is connected to its justification by an arrow.
- One assertion can *feed* another justification thus creating the network.
- Assertions may be labelled with a *belief status*.
- An assertion is *valid* if every assertion in the IN-List is believed and none in the OUT-List are believed.
- An assertion is non-monotonic if the OUT-List is not empty or if any assertion in the IN-List is non-monotonic.

Logic-Based Truth Maintenance Systems (LTMS)

Similar to JTMS except:

- Nodes (assertions) assume no relationships among them except ones explicitly stated in justifications.
- JTMS can represent P and $\neg P$ simultaneously. An LTMS would throw a contradiction here.
- If this happens network has to be reconstructed.

Assumption-Based Truth Maintenance Systems (ATMS)

- JTMS and LTMS pursue a single line of reasoning at a time and backtrack (dependency-directed) when needed -- *depth-first search*.
- ATMS maintain alternative paths in parallel -- *breadth-first search*
- Backtracking is avoided at the expense of maintaining multiple contexts.
- However as reasoning proceeds contradictions arise and the ATMS can be *pruned*
 - Simply find assertion with no valid justification.

Questions

1.a. Consider a first-order logical system which uses two 1-place predicates namely, *Big* and *Small*. The set of object constants is given by $\{A, B\}$. Enumerate all possible models in this case.

1.b. For each of the following sentences, identify the models in which the given sentence is true.

- a. $Big(A) \quad Big(B)$
- b. $Big(A) \quad Big(B)$
- c. $x \ Big(x)$
- d. $x \ \neg \ Big(x)$
- e. $x \ Big(x)$
- f. $x \ \neg \ Big(x)$
- g. $x \ Big(x) \quad Small(x)$
- h. $x \ Big(x) \quad Small(x)$
- i. $x \ Big(x) \quad \neg \ Small(x)$

1.c Determine whether the expressions p and q unify with each other in each of the following cases. If so, give the most general unifier; If not, give a brief explanation (Assume that the upper case letters are (object, predicate, or function) constants and that the lower case letters are variables).

- a. $p = F(G(v), H(u, v)); q = F(w, J(x, y))$
- b. $p = F(x, F(u, x)); q = F(F(y, A), F(z, F(B, z)))$
- c. $p = F(x_1, G(x_2, x_3), x_2, B); q = F(G(H(A, x_5), x_2), x_1, H(A, x_4), x_4)$

1.d. Put the following FOL formulas into conjunctive form (CNF):

- a. $x \ y \ ((P(x) \quad Q(y)) \quad z \ R(x, y, z))$
- b. $x \ y \ z \ (P(x) \quad (Q(y) \quad R(z)))$

2. Convert the following English statements to statements in first order logic

- a. every boy or girl is a child
- b. every child gets a doll or a train or a lump of coal
- c. no boy gets any doll
- d. no child who is good gets any lump of coal
- e. Jack is a boy

3. Using the above five axioms construct a proof by refutation by resolution of the statement

“if Jack doesn't get a train, then Jack is not a good boy”

4. Consider the following set of axioms

- i. Everyone who loves someone who loves them back is happy
- ii. Mary loves everyone.
- iii. There is someone who loves Mary.

From the above statements conclude that:

Mary is happy.

Solution

1.a. Possible models are:

$$\begin{aligned}M_0 &= \{\} \\M_1 &= \{\text{Big(A)}\} \\M_2 &= \{\text{Big(B)}\}, \\M_3 &= \{\text{Small(A)}\} \\M_4 &= \{\text{Small(B)}\} \\M_5 &= \{\text{Big(A), Big(B)}\} \\M_6 &= \{\text{Big(A), Small(A)}\} \\M_7 &= \{\text{Big(A), Small(B)}\} \\M_8 &= \{\text{Small(A), Big(B)}\} \\M_9 &= \{\text{Big(B), Small(B)}\} \\M_{10} &= \{\text{Small(A), Small(B)}\} \\M_{11} &= \{\text{Big(A), Big(B), Small(A)}\} \\M_{12} &= \{\text{Big(A), Big(B), Small(B)}\} \\M_{13} &= \{\text{Small(A), Small(B), Big(A)}\} \\M_{14} &= \{\text{Small(A), Small(B), Big(B)}\} \\M_{15} &= \{\text{Big(A), Big(B), Small(A), Small(B)}\}\end{aligned}$$

1.b Answers:

- (a) $\text{Big}(A) \wedge \text{Big}(B) : M_5 M_{11} M_{12} M_{15}$
(b) $\text{Big}(A) \vee \text{Big}(B) : M_1 M_2 M_5 M_6 M_7 M_8 M_9 M_{11} M_{12} M_{13} M_{14} M_{15}$
(c) $\forall x \text{Big}(x)$ (equivalent to $\text{Big}(A) \wedge \text{Big}(B)$) : $M_5 M_{11} M_{12} M_{15}$
(d) $\forall x \neg \text{Big}(x)$ (equivalent to $\neg \text{Big}(A) \wedge \neg \text{Big}(B)$) : $M_0 M_3 M_4 M_{10}$
(e) $\exists x \text{Big}(x)$ (equivalent to $\text{Big}(A) \vee \text{Big}(B)$) : $M_1 M_2 M_5 M_6 M_7 M_8 M_9 M_{11} M_{12} M_{13} M_{14} M_{15}$
(f) $\exists x \neg \text{Big}(x)$ (equivalent to $\neg \text{Big}(A) \vee \neg \text{Big}(B)$ which is equivalent to $\neg(\text{Big}(A) \wedge \text{Big}(B))$) :
 $M_0 M_1 M_2 M_3 M_4 M_6 M_7 M_8 M_9 M_{10} M_{13} M_{14}$
(g) $\forall x \text{Big}(x) \wedge \text{Small}(x)$ (equivalent to $(\text{Big}(A) \wedge \text{Small}(A)) \wedge (\text{Big}(B) \wedge \text{Small}(B))$) : M_{15}
(h) $\forall x \text{Big}(x) \vee \text{Small}(x)$ (equivalent to $(\text{Big}(A) \vee \text{Small}(A)) \wedge (\text{Big}(B) \vee \text{Small}(B))$) : $M_5 M_7 M_8 M_{10} M_{11} M_{12} M_{13} M_{14} M_{15}$
(i) $\forall x \text{Big}(x) \Rightarrow \text{Small}(x)$ (equivalent to $(\neg \text{Big}(A) \vee \neg \text{Small}(A)) \wedge (\neg \text{Big}(B) \vee \neg \text{Small}(B))$) :
 $M_0 M_1 M_2 M_3 M_4 M_5 M_7 M_8 M_{10}$

1.c. Answers:

(a) $p = F(G(v), H(u, v)); q = F(w, J(x, y)).$

Substitute $w/G(v)$: $p = F(G(v), H(u, v)); q = F(G(v), J(x, y))$ but cannot find a unifier for $H(u, v)$ and $J(x, y)$ because they are different objects (predicates, functions, constants).

(b) $p = F(x, F(u, x)); q = F(F(y, A), F(z, F(B, z))).$

Substitute $x/F(y, A)$: $p = F(F(y, A), F(u, F(y, A)); q = F(F(y, A), F(z, F(B, z))).$

Substitute u/z : $p = F(F(B, A), F(z, F(y, A)); q = F(F(y, A), F(z, F(B, z))).$

Substitute y/B and z/A : $p = F(F(B, A), F(A, F(B, A)); q = F(F(B, A), F(A, F(B, A)).$

So $\text{Unify}(p, q) = \{x/F(y, A), u/z, y/B, z/A\}.$

(c) $p = F(x_1, G(x_2, x_3), x_2, B); q = F(G(H(A, x_5), x_2), x_1, H(A, x_4), x_4).$

Substitute x_4/B : $p = F(x_1, G(x_2, x_3), x_2, B); q = F(G(H(A, x_5), x_2), x_1, H(A, B), B).$

Substitute $x_1/G(x_2, x_3)$: $p = F(G(x_2, x_3), G(x_2, x_3), x_2, B); q = F(G(H(A, x_5), x_2), G(x_2, x_3), H(A, B), B).$

Substitute $x_2/H(A, x_5)$: $p = F(G(H(A, x_5), x_3), G(H(A, x_5), x_3), H(A, x_5), B);$

$q = F(G(H(A, x_5), H(A, x_5)), G(H(A, x_5), x_3), H(A, B), B).$

Substitute $x_3/H(A, x_5)$: $p = F(G(H(A, x_5), H(A, x_5)), G(H(A, x_5), H(A, x_5)), H(A, x_5), B);$

$q = F(G(H(A, x_5), H(A, x_5)), G(H(A, x_5), H(A, x_5)), H(A, B), B).$

Substitute x_5/B so $\text{Unify}(p, q) = \{x_4/B, x_1/G(x_2, x_3), x_2/H(A, x_5), x_3/H(A, x_5), x_5/B\}.$

1.d Answers

(a) $\forall x \forall y ((P(x) \wedge Q(y) \Rightarrow \exists z R(x, y, z)) =$

$\forall x \forall y (\neg(P(x) \wedge Q(y)) \vee \exists z R(x, y, z)) =$

$\forall x \forall y (\neg P(x) \vee \neg Q(y) \vee R(x, y, G(x, y))) =$

$\forall x \forall y ((P(x) \wedge Q(y) \Rightarrow R(x, y, G(x, y)))$

(b) $\exists x \forall y \exists z (P(x) \Rightarrow$

$(Q(y) \Rightarrow R(z))) =$

$\exists x \forall y \exists z (\neg P(x) \vee (Q(y) \Rightarrow R(z))) =$

$\exists x \forall y \exists z (\neg P(x) \vee (\neg Q(y) \vee R(z))) =$

$\exists x \forall y \exists z (\neg P(x) \vee \neg Q(y) \vee R(z))$ (where B is any constant not used elsewhere) =

$\forall y (\neg P(B) \vee \neg Q(y) \vee R(G(y))) =$

$\forall y (P(B) \wedge Q(y) \Rightarrow R(G(y)))$

2. The FOL statements are

- a. $\forall x ((\text{boy}(x) \text{ or } \text{girl}(x)) \rightarrow \text{child}(x))$
- b. $\forall y (\text{child}(y) \rightarrow (\text{gets}(y,\text{doll}) \text{ or } \text{gets}(y,\text{train}) \text{ or } \text{gets}(y,\text{coal})))$
- c. $\forall w (\text{boy}(w) \rightarrow \neg \text{gets}(w,\text{doll}))$
- d. $\forall z ((\text{child}(z) \text{ and } \text{good}(z)) \rightarrow \neg \text{gets}(z,\text{coal}))$
- e. $\text{boy}(\text{Jack})$

3. The statement to be proved can be written in FOL as:

$\neg \text{gets}(\text{Jack},\text{train}) \rightarrow \neg \text{good}(\text{Jack})$

The proof consists of three steps:

- Negate the conclusion.
- Translate all statements to clausal form.
- Apply resolution to the clauses until you obtain the empty clause

The steps are shown below:

- Transform axioms into clause form:
 1. $\forall x ((\text{boy}(x) \text{ or } \text{girl}(x)) \rightarrow \text{child}(x))$
 $\neg(\text{boy}(x) \text{ or } \text{girl}(x)) \text{ or } \text{child}(x)$
 $(\neg \text{boy}(x) \text{ and } \neg \text{girl}(x)) \text{ or } \text{child}(x)$
 $(\neg \text{boy}(x) \text{ or } \text{child}(x)) \text{ and } (\neg \text{girl}(x) \text{ or } \text{child}(x))$
 2. $\forall y (\text{child}(y) \rightarrow (\text{gets}(y,\text{doll}) \text{ or } \text{gets}(y,\text{train}) \text{ or } \text{gets}(y,\text{coal})))$
 $\neg \text{child}(y) \text{ or } \text{gets}(y,\text{doll}) \text{ or } \text{gets}(y,\text{train}) \text{ or } \text{gets}(y,\text{coal})$
 3. $\forall w (\text{boy}(w) \rightarrow \neg \text{gets}(w,\text{doll}))$
 $\neg \text{boy}(w) \text{ or } \neg \text{gets}(w,\text{doll})$
 4. $\forall z ((\text{child}(z) \text{ and } \text{good}(z)) \rightarrow \neg \text{gets}(z,\text{coal}))$
 $\neg(\text{child}(z) \text{ and } \text{good}(z)) \text{ or } \neg \text{gets}(z,\text{coal})$
 $\neg \text{child}(z) \text{ or } \neg \text{good}(z) \text{ or } \neg \text{gets}(z,\text{coal})$
 5. $\text{boy}(\text{Jack})$
 6. $\neg(\neg \text{gets}(\text{Jack},\text{train}) \rightarrow \neg \text{good}(\text{Jack}))$ negated conclusion
 $\text{gets}(\text{Jack},\text{train}) \text{ or } \text{good}(\text{Jack})$
 $\neg \text{gets}(\text{Jack},\text{train}) \text{ and } \text{good}(\text{Jack})$
- The set of CNF clauses:
 1. (a) $\neg \text{boy}(x) \text{ or } \text{child}(x)$
(b) $\neg \text{girl}(x) \text{ or } \text{child}(x)$
 2. $\neg \text{child}(y) \text{ or } \text{gets}(y,\text{doll}) \text{ or } \text{gets}(y,\text{train}) \text{ or } \text{gets}(y,\text{coal})$
 3. $\neg \text{boy}(w) \text{ or } \neg \text{gets}(w,\text{doll})$
 4. $\neg \text{child}(z) \text{ or } \neg \text{good}(z) \text{ or } \neg \text{gets}(z,\text{coal})$
 5. $\text{boy}(\text{Jack})$

- 6. (a) !gets(Jack,train)
 (b) good(Jack)
- Resolution:
 - o 4. !child(z) or !good(z) or !gets(z,coal)
 6.(b). good(Jack)

 7. !child(Jack) or !gets(Jack,coal) (substituting z by Jack)
 - o 1.(a). !boy(x) or child(x)
 5. boy(Jack)

 8. child(Jack) (substituting x by Jack)
 - o 7. !child(Jack) or !gets(Jack,coal)
 8. child(Jack)

 9. !gets(Jack,coal)
 - o 2. !child(y) or gets(y,doll) or gets(y,train) or gets(y,coal)
 8. child(Jack)

 10. gets(Jack,doll) or gets(Jack,train) or gets(Jack,coal) (substituting y by Jack)
 - o 9. !gets(Jack,coal)
 10. gets(Jack,doll) or gets(Jack,train) or gets(Jack,coal)

 11. gets(Jack,doll) or gets(Jack,train)
 - o 3. !boy(w) or !gets(w,doll)
 5. boy(Jack)

 12. !gets(Jack,doll) (substituting w by Jack)
 - o 11. gets(Jack,doll) or gets(Jack,train)
 12. !gets(Jack,doll)

 13. gets(Jack,train)
 - o 6.(a). !gets(Jack,train)
 13. gets(Jack,train)

 14. empty clause

4. The axioms in FOL are:

A1: for-all x, for-all y, [loves(x,y) & loves(y,x) => happy(x)]

A2: for-all z, [loves(mary,z)]

A3: there-is w, [loves(w,mary)]

The conclusion can be written as

C: happy(mary)

The proof is as follows:

- Translate the axioms and the negation of the conclusion into clausal form.
Show each step of the translation (remember that the order in which you apply the steps is crucial).
 1. **A1:** !loves(x,y) || !loves(y,x) || happy(x)
note that p => q is equivalent to !p || q and that !(r & s) is equivalent to !r || !s
 2. **A2:** loves(mary,z)
 3. **A3:** loves(a,mary), where **a** is a new constant symbol
 4. **!C:** ! happy(mary)
- Apply resolution (state explicitly what clauses are resolved and which substitutions are made) until the empty clause is found.

```
!C: ! happy(mary)
A1: !loves(x,y) || !loves(y,x) || happy(x) _____ with x=mary
      A4: +loves(mary,y) || !loves(y,mary)
      A2: loves(mary,z) _____ with z=y
      A5: +loves(y,mary)
      A3: loves(a,mary) _____ with y=a
      _____ empty clause
```

Since assuming that "mary is not happy" yields a contradiction in the presence of A1, A2, A3, then the statement "mary is happy" is a logical consequence of A1, A2, and A3.

Module 7

Knowledge Representation and Logic – (Rule based Systems)

7.1 Instructional Objective

- The students should understand the use of rules as a restricted class of first order logic statements
- The student should be familiar with the concept of Horn clause
- Students should be able to understand and implement the following reasoning algorithm
 - Forward chaining
 - Backward chaining
- Students should understand the nature of the PROLOG programming language and the reasoning method used in it.
- Students should be able to write elementary PROLOG programs
- Students should understand the architecture and organization of expert systems and issues involved in designing an expert system

At the end of this lesson the student should be able to do the following:

- Represent a knowledge base as a set of rules if possible
- Apply forward/backward chaining algorithm as suitable
- Write elementary programs in PROLOG
- Design expert systems

Lesson 17

Rule based Systems - I

7.2 Rule Based Systems

Instead of representing knowledge in a relatively declarative, static way (as a bunch of things that are true), rule-based system represent knowledge in terms of a bunch of rules that tell you what you should do or what you could conclude in different situations. A rule-based system consists of a bunch of IF-THEN rules, a bunch of facts, and some interpreter controlling the application of the rules, given the facts. Hence, this are also sometimes referred to as production systems. Such rules can be represented using Horn clause logic.

There are two broad kinds of rule system: forward chaining systems, and backward chaining systems. In a forward chaining system you start with the initial facts, and keep using the rules to draw new conclusions (or take certain actions) given those facts. In a backward chaining system you start with some hypothesis (or goal) you are trying to prove, and keep looking for rules that would allow you to conclude that hypothesis, perhaps setting new subgoals to prove as you go. Forward chaining systems are primarily data-driven, while backward chaining systems are goal-driven. We'll look at both, and when each might be useful.

7.2.1 Horn Clause Logic

There is an important special case where inference can be made substantially more focused than in the case of general resolution. This is the case where all the clauses are *Horn clauses*.

Definition: A *Horn clause* is a clause with at most one positive literal.

Any Horn clause therefore belongs to one of four categories:

- A *rule*: 1 positive literal, at least 1 negative literal. A rule has the form " $\sim P_1 \vee \sim P_2 \vee \dots \vee \sim P_k \vee Q$ ". This is logically equivalent to " $[P_1 \wedge P_2 \wedge \dots \wedge P_k] \Rightarrow Q$ "; thus, an if-then implication with any number of conditions but one conclusion. Examples: " $\sim \text{man}(X) \vee \text{mortal}(X)$ " (All men are mortal); " $\sim \text{parent}(X,Y) \vee \sim \text{ancestor}(Y,Z) \vee \text{ancestor}(X,Z)$ " (If X is a parent of Y and Y is an ancestor of Z then X is an ancestor of Z.)
- A *fact or unit*: 1 positive literal, 0 negative literals. Examples: " $\text{man}(\text{socrates})$ ", " $\text{parent}(\text{elizabeth}, \text{charles})$ ", " $\text{ancestor}(\text{X}, \text{X})$ " (Everyone is an ancestor of themselves (in the trivial sense).)
- A *negated goal* : 0 positive literals, at least 1 negative literal. In virtually all implementations of Horn clause logic, the negated goal is the negation of the statement to be proved; the knowledge base consists entirely of facts and goals. The statement to be proven, therefore, called the goal, is therefore a single unit or the conjunction of units; an *existentially* quantified variable in the goal turns into a *free* variable in the negated goal. E.g. If the goal to be proven is " $\exists (X) \text{ male}(X) \wedge \text{ancestor}(\text{elizabeth}, X)$ " (show that there exists a male descendent of Elizabeth) the negated goal will be " $\sim \text{male}(X) \vee \sim \text{ancestor}(\text{elizabeth}, X)$ ".

- The null clause: 0 positive and 0 negative literals. Appears only as the end of a resolution proof.

Now, if resolution is restricted to Horn clauses, some interesting properties appear. Some of these are evident; others I will just state and you can take on faith.

I. If you resolve Horn clauses A and B to get clause C, then the positive literal of A will resolve against a negative literal in B, so the only positive literal left in C is the one from B (if any). Thus, the resolvent of two Horn clauses is a Horn clause.

II. If you resolve a negated goal G against a fact or rule A to get clause C, the positive literal in A resolves against a negative literal in G. Thus C has no positive literal, and thus is either a negated goal or the null clause.

III. Therefore: Suppose you are trying to prove Phi from Gamma, where $\sim\text{Phi}$ is a negated goal, and Gamma is a knowledge base of facts and rules. Suppose you use the set of support strategy, in which no resolution ever involves resolving two clauses from Gamma together. Then, inductively, every resolution combines a negated goal with a fact or rule from Gamma and generates a new negated goal. Moreover, if you take a resolution proof, and trace your way back from the null clause at the end to $\sim\text{Phi}$ at the beginning, since every resolution involves combining one negated goal with one clause from Gamma, it is clear that the sequence of negated goals involved can be linearly ordered. That is, the final proof, ignoring dead ends has the form

```
 $\sim\text{Phi}$  resolves with C1 from Gamma, generating negated goal P2  

P2 resolves with C2 from Gamma, generating negated goal P3  

...
Pk resolves with C2 from Gamma, generating the null clause.
```

IV. Therefore, the process of generating the null clause can be viewed as a state space search where:

- A state is a negated goal.
- A operator on negated goal P is to resolve it with a clause C from Gamma.
- The start state is $\sim\text{Phi}$
- The goal state is the null clause.

V. Moreover, it turns out that it doesn't really matter which literal in P you choose to resolve. All the literals in P will have to be resolved away eventually, and the order doesn't really matter. (This takes a little work to prove or even to state precisely, but if you work through a few examples, it becomes reasonably evident.)

7.2.2 Backward Chaining

Putting all the above together, we formulate the following non-deterministic algorithm for resolution in Horn theories. This is known as backward chaining.

```

bc(in P0 : negated goal;
    GAMMA : set of facts and rules;)
{ if P0 = null then succeed;
  pick a literal L in P0;
  choose a clause C in GAMMA whose head resolves with L;
  P := resolve(P0,GAMMA);
  bc(P,GAMMA)
}

```

If $\text{bc}(\neg\Phi, \Gamma)$ succeeds, then Φ is a consequence of Γ ; if it fails, then Φ is not a consequence of Γ .

Moreover: Suppose that Φ contains existentially quantified variables. As remarked above, when $\neg\Phi$ is Skolemized, these become free variables. If you keep track of the successive bindings through the successful path of resolution, then the final bindings of these variables gives you a *value* for these variables; all proofs in Horn theories are constructive (assuming that function symbols in Γ are constructive.) Thus the attempt to prove a statement like " $\exists(X,Y) p(X,Y) \wedge q(X,Y)$ " can be interpreted as "*Find X and Y such that p(X,Y) and q(X,Y).*"

The successive negated goals P_i can be viewed as negations of *subgoals* of Φ . Thus, the operation of resolving $\neg P$ against C to get $\neg Q$ can be interpreted, "One way to prove P would be to prove Q and then use C to infer P ". For instance, suppose P is "mortal(socrates)," C is "man(X) \Rightarrow mortal(X)" and Q is "man(socrates)." Then the step of resolving $\neg P$ against C to get $\neg Q$ can be viewed as, "One way to prove mortal(socrates) would be to prove man(socrates) and then combine that with C ."

Propositional Horn theories can be decided in polynomial time. First-order Horn theories are only semi-decidable, but in practice, resolution over Horn theories runs much more efficiently than resolution over general first-order theories, because of the much restricted search space used in the above algorithm.

Backward chaining is complete for Horn clauses. If Φ is a consequence of Γ , then there is a backward-chaining proof of Φ from Γ .

7.2.3 Pure Prolog

We are now ready to deal with (pure) Prolog, the major Logic Programming Language. It is obtained from a variation of the backward chaining algorithm that allows Horn clauses with the following rules and conventions:

- The Selection Rule is to select the leftmost literals in the goal.
- The Search Rule is to consider the clauses in the order they appear in the current list of clauses, from top to bottom.
- **Negation as Failure**, that is, Prolog assumes that a literal L is proven if it is unable to prove $(\neg L)$
- Terms can be set equal to variables but not in general to other terms. For example, we can say that $x=A$ and $x=F(B)$ but we cannot say that $A=F(B)$.

- Resolvents are added to the bottom of the list of available clauses.

These rules make for very rapid processing. Unfortunately:

The Pure Prolog Inference Procedure is Sound but not Complete

This can be seen by example. Given

- $P(A,B)$
- $P(C,B)$
- $P(y,x) \leq P(x,y)$
- $P(x,z) \leq P(x,y), P(y,z)$

we are unable to derive in Prolog that $P(A,C)$ because we get caught in an ever deepening depth-first search.

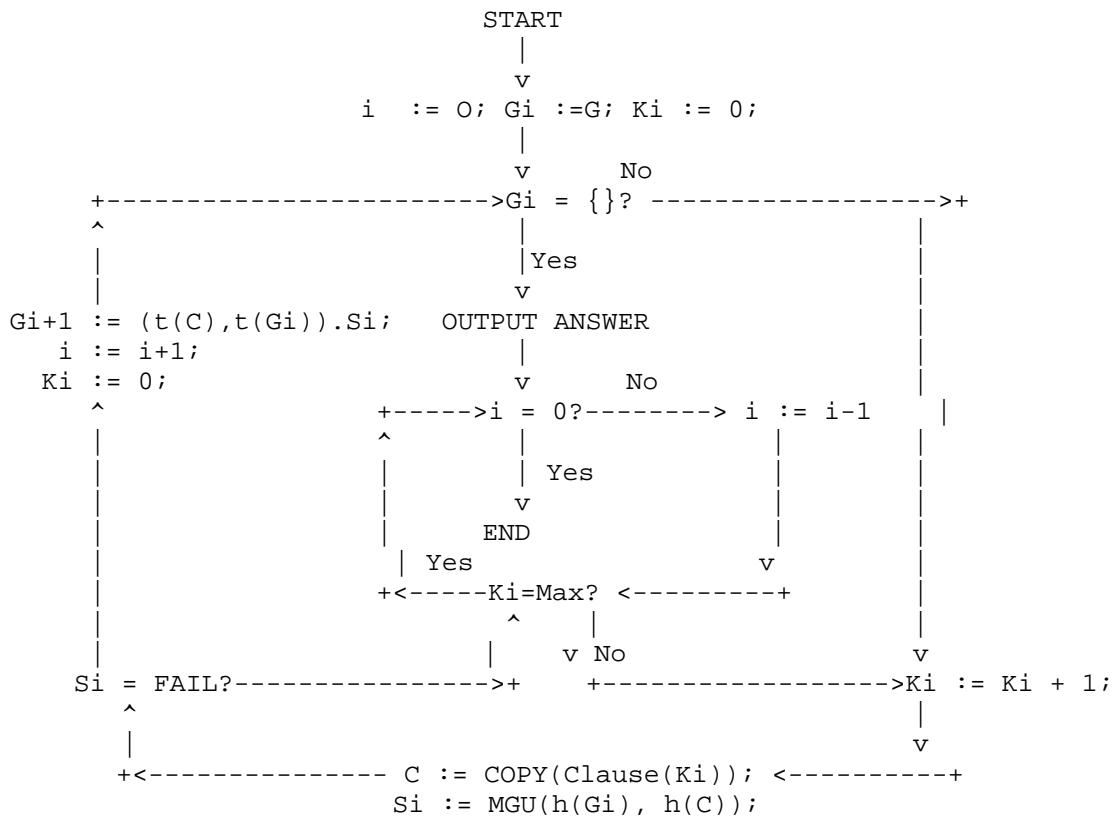
A Prolog "program" is a knowledge base Gamma. The program is invoked by posing a query Phi. The value returned is the bindings of the variables in Phi, if the query succeeds, or failure. The interpreter returns one answer at a time; the user has the option to request it to continue and to return further answers.

The derivation mechanism of Pure Prolog has a very simple form that can be described by the following flow chart.

Interpreter for Pure Prolog

Notational conventions:

- i: used to index literals in a goal
- K_i : indexes the clauses in the given program (i.e. set of clauses) P
- Max: the number of clauses in P
- $h(G)$: the first literal of the goal G
- $t(G)$: the rest of goal G, i.e. G without its first literal
- $\text{clause}(K_i)$: the kith clause of the program



7.2.3.1 Real Prolog

Real Prolog systems differ from pure Prolog for a number of reasons. Many of which have to do with the ability in Prolog to modify the control (search) strategy so as to achieve efficient programs. In fact there is a dictum due to Kowalski:

$$\text{Logic} + \text{Control} = \text{Algorithm}$$

But the reason that is important to us now is that Prolog uses a Unification procedure which does not enforce the **Occur Test**. This has an unfortunate consequence that, while Prolog may give origin to efficient programs, but

$$\text{Prolog is not Sound}$$

Actual Prolog differs from pure Prolog in three major respects:

- There are additional functionalities besides theorem proving, such as functions to assert statements, functions to do arithmetic, functions to do I/O.
- The "cut" operator allows the user to prune branches of the search tree.
- The unification routine is not quite correct, in that it does not check for circular bindings e.g. $X \rightarrow Y, Y \rightarrow f(X).$

Notation: The clause " $\sim p \vee \sim q \vee r$ " is written in Prolog in the form "r :- p,q."

Example: Let Gamma be the following knowledge base:

```
1. ancestor(X,X).
2. ancestor(X,Z) :- parent(X,Y), ancestor(Y,Z).
3. parent(george,sam).
4. parent(george,andy).
5. parent(andy,mary).
6. male(george).
7. male(sam).
8. male(andy).
9. female(mary).
```

Let Phi be the query "exists(Q) ancestor(george,Q) \wedge female(Q)." (i.e. find a female descendant of george.) Then the Skolemization of Phi is " $\sim \text{ancestor}(\text{george},Q) \vee \sim \text{female}(Q)$." A Prolog search proceeds as follows: (The indentation indicates the subgoal structure. Note that the variables in clauses in Gamma are renamed each time.)

```
G0: ~ancestor(george,Q) V ~female(Q).      Resolve with 1: Q=X1=george.
    G1: ~female(george)                      Fail. Return to G0.
G0: ~ancestor(george,Q) V ~female(Q).      Resolve with 2: x2=george.
Z2=Q.
    G2: ~parent(george,Y2) V ~ancestor(Y2,Q) V ~female(Q).
                                                Resolve with 3: Y2=sam.
        G3: ~ancestor(sam,Q) V ~female(Q).   Resolve with 1: Q=X3=sam.
            G4: ~female(sam).                Fail. Return to G3.
        G3: ~ancestor(sam,Q) V ~female(Q).   Resolve with 2: X4=sam, Z4=Q
            G5: ~parent(sam,Y2) V ~ancestor(Y2,Q) V ~female(Q).
                                                Fail. Return to G3.
        G3: ~ancestor(sam,Q) V ~female(Q).   Fail. Return to G2.
    G2: ~parent(george,Y2) V ~ancestor(Y2,Q) V ~female(Q).
                                                Resolve with 4: Y2=andy.
        G6: ~ancestor(andy,Q) V ~female(Q).  Resolve with 1: X5=Q=andy
            G7: ~female(andy).                Fail. Return to G6.
        G6: ~parent(andy,Y6) V ~ancestor(Y6,Q) V ~female(Q).
                                                Resolve with 5: Y6=mary.
            G8: ~ancestor(mary,Q) V ~female(mary). Resolve with 1:
X7=Q=mary.
            G9: ~female(mary)                  Resolve with 9.
                                                Null.
```

Return the binding Q=mary.

7.2.4 Forward chaining

An alternative mode of inference in Horn clauses is *forward chaining*. In forward chaining, one of the resolvents in every resolution is a fact. (Forward chaining is also known as "unit resolution.")

Forward chaining is generally thought of as taking place in a dynamic knowledge base, where facts are gradually added to the knowledge base Gamma. In that case, forward chaining can be implemented in the following routines.

```

procedure add_fact(in F; in out GAMMA)
  /* adds fact F to knowledge base GAMMA and forward chains */
if F is not in GAMMA then {
  GAMMA := GAMMA union {F};
  for each rule R in GAMMA do {
    let ~L be the first negative literal in R;
    if L unifies with F then
      then { resolve R with F to get C;
              if C is a fact then add_fact(C,GAMMA)
              else /* C is a rule */ add_rule(C,GAMMA)
            }
    }
  }
end add_fact.

procedure add_rule(in R; in out GAMMA)
  /* adds rule R to knowledge base GAMMA and forward chains */
if R is not in GAMMA then {
  GAMMA := GAMMA union {R};
  let ~L be the first negative literal in R;
  for each fact F in GAMMA do
    if L unifies with F
      then { resolve R with F to get C;
              if C is a fact then add_fact(C,GAMMA)
              else /* C is a rule */ add_rule(C,GAMMA)
            }
  }
end add_fact.

procedure answer_query(in Q, GAMMA) return boolean /* Success or
failure */
{ QQ := {Q} /* A queue of queries
  while QQ is non-empty do {
    Q1 := pop(QQ);
    L1 := the first literal in Q1;
    for each fact F in GAMMA do
      if F unifies with L
        then { resolve F with Q1 to get Q2;
                if Q2=null then return(true)
                else add Q2 to QQ;
              }
  }
  return(false)
}

```

The forward chaining algorithm may not terminate if GAMMA contains recursive rules.

Forward chaining is complete for Horn clauses; if Phi is a consequence of Gamma, then there is a forward chaining proof of Phi from Gamma. To be sure of finding it if Gamma contains recursive rules, you have to modify the above routines to use an exhaustive search technique, such as a breadth-first search.

In a forward chaining system the facts in the system are represented in a *working memory* which is continually updated. Rules in the system represent possible actions to take when specified conditions hold on items in the working memory - they are sometimes called condition-action rules. The conditions are usually *patterns* that must *match* items in the working memory, while the actions usually involve *adding* or *deleting* items from the working memory.

The interpreter controls the application of the rules, given the working memory, thus controlling the system's activity. It is based on a cycle of activity sometimes known as a *recognise-act* cycle. The system first checks to find all the rules whose conditions hold, given the current state of working memory. It then selects one and performs the actions in the action part of the rule. (The selection of a rule to fire is based on fixed strategies, known as *conflict resolution* strategies.) The actions will result in a new working memory, and the cycle begins again. This cycle will be repeated until either no rules fire, or some specified goal state is satisfied.

Conflict Resolution Strategies

A number of conflict resolution strategies are typically used to decide which rule to fire. These include:

- Don't fire a rule twice on the same data.
- Fire rules on more recent working memory elements before older ones. This allows the system to follow through a single chain of reasoning, rather than keeping on drawing new conclusions from old data.
- Fire rules with more specific preconditions before ones with more general preconditions. This allows us to deal with non-standard cases. If, for example, we have a rule ``IF (bird X) THEN ADD (flies X)" and another rule ``IF (bird X) AND (penguin X) THEN ADD (swims X)" and a penguin called tweety, then we would fire the second rule first and start to draw conclusions from the fact that tweety swims.

These strategies may help in getting reasonable behavior from a forward chaining system, but the most important thing is how we write the rules. They should be carefully constructed, with the preconditions specifying as precisely as possible when different rules should fire. Otherwise we will have little idea or control of what will happen. Sometimes special working memory elements are used to help to control the behavior of the system. For example, we might decide that there are certain basic stages of processing in doing some task, and certain rules should only be fired at a given stage - we could have a special working memory element (stage 1) and add (stage 1) to the preconditions of all the relevant rules, removing the working memory element when that stage was complete.

Choice between forward and backward chaining

Forward chaining is often preferable in cases where there are many rules with the same conclusions. A well-known category of such rule systems are taxonomic hierarchies. E.g. the taxonomy of the animal kingdom includes such rules as:

```
animal(X) :- sponge(X).  
animal(X) :- arthropod(X).  
animal(X) :- vertebrate(X).  
...  
vertebrate(X) :- fish(X).  
vertebrate(X) :- mammal(X)  
...  
mammal(X) :- carnivore(X)  
...  
carnivore(X) :- dog(X).  
carnivore(X) :- cat(X).  
...
```

(I have skipped family and genus in the hierarchy.)

Now, suppose we have such a knowledge base of rules, we add the fact "dog(fido)" and we query whether "animal(fido)". In forward chaining, we will successively add "carnivore(fido)", "mammal(fido)", "vertebrate(fido)", and "animal(fido)". The query will then succeed immediately. The total work is proportional to the height of the hierarchy. By contrast, if you use backward chaining, the query " \sim animal(fido)" will unify with the first rule above, and generate the subquery " \sim sponge(fido)", which will initiate a search for Fido through all the subdivisions of sponges, and so on. Ultimately, it searches the entire taxonomy of animals looking for Fido.

In some cases, it is desirable to *combine* forward and backward chaining. For example, suppose we augment the above animal with features of these various categories:

```
breathes(X) :- animal(X).  
...  
backbone(X) :- vertebrate(X).  
has(X,brain) :- vertebrate(X).  
...  
furry(X) :- mammal(X).  
warm_blooded(X) :- mammal(X).  
...
```

If all these rules are implemented as forward chaining, then as soon as we state that Fido is a dog, we have to add all his known properties to Gamma; that he breathes, is warm-blooded, has a liver and kidney, and so on. The solution is to mark these property rules as backward chaining and mark the hierarchy rules as forward chaining. You then implement the knowledge base with both the forward chaining algorithm, restricted to rules marked as forward chaining, and backward chaining rules, restricted to rules marked as backward chaining. However, it is hard to guarantee that such a mixed inference system will be complete.

AND/OR Trees

We will next show the use of AND/OR trees for inferencing in Horn clause systems. The problem is, given a set of axioms in Horn clause form and a goal, show that the goal can be proven from the axioms.

An AND/OR tree is a tree whose internal nodes are labeled either "AND" or "OR". A *valuation* of an AND/OR tree is an assignment of "TRUE" or "FALSE" to each of the leaves. Given a tree T and a valuation over the leaves of T, the values of the internal nodes and of T are defined recursively in the obvious way:

- An OR node is TRUE if at least one of its children is TRUE.
- An AND node is TRUE if all of its children are TRUE.

The above is an *unconstrained* AND/OR tree. Also common are *constrained* AND/OR trees, in which the leaves labeled "TRUE" must satisfy some kind of constraint. A *solution* to a constrained AND/OR tree is a valuation that satisfies the constraint and gives the tree the value "TRUE".

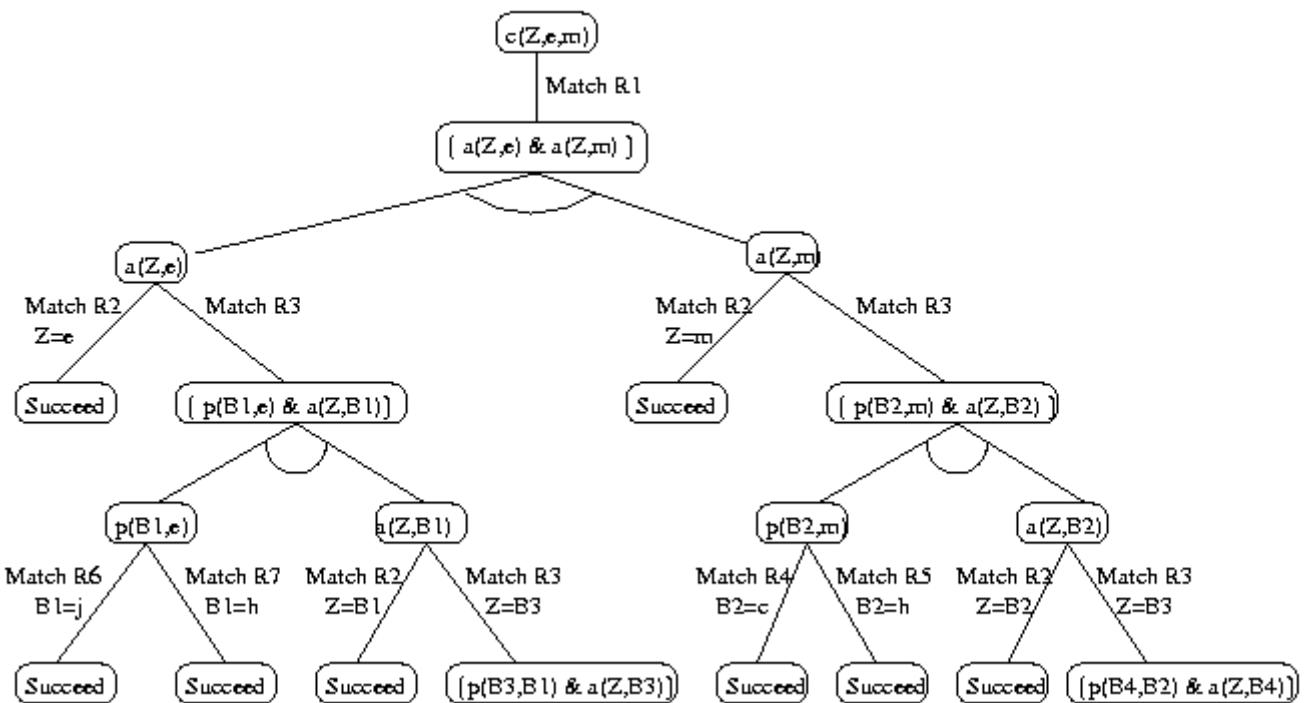
An OR node is a goal to be proven. A goal G has one downward arc for each rule R whose head resolves with G. This leads to an AND node. The children in the AND node are the literals in the tail of R. Thus, a rule is satisfied if all its subgoals are satisfied (the AND node); a goal is satisfied if it is established by one of its rules (the OR node). The leaves are unit clauses, with no tail, labeled TRUE, and subgoals with no matching rules, labeled FALSE . The constraint is that the variable bindings must be consistent. The figure below show the AND/OR tree corresponding to the following Prolog rule set with the goal "common_ancestor(Z,edward,mary)"

Axioms:

```
/* Z is a common ancestor of X and Y */
R1: common_ancestor(Z,X,Y) :- ancestor(Z,X), ancestor(Z,Y).

/* Usual recursive definition of ancestor, going upward. */
R2: ancestor(A,A).
R3: ancestor(A,C) :- parent(B,C), ancestor(A,B).

/* Mini family tree */
R4: parent(catherine,mary).
R5: parent(henry,mary).
R6: parent(jane,edward).
R7: parent(henry,edward).
```



Module 7

Knowledge Representation and Logic – (Rule based Systems)

Rule based Systems - II

Version 2 CSE IIT, Kharagpur

7.2.5 Programs in PROLOG

These minimal notes on Prolog show only some of its flavor.

Here are facts

```
plays(ann,fido).  
friend(john,sam).
```

where ann, fido, john, and sam are individuals, and plays and friend are **functors**. And here is a rule

```
likes(X,Y) :- plays(X,Y),friend(X,Y).
```

It says that if X plays with Y and X is a friend of Y then X likes Y. Variables start with capital letters (If we are not interested in the value of a variable, we can just use _ (underscore)).

In a rule the left-hand side and the right-hand side are called respectively the **head** and the **tail** of the rule.

The prompt in prolog is

```
| ?-
```

You exit prolog with the statement
`halt.`

You can add rules and facts to the current session in a number of ways:

1. You can enter clauses from the terminal as follows:

```
2. | ?- consult(user).  
3. | like(b,a).  
4. | like(d,c).  
5. ^D
```

which adds the two clauses to the working space.

6. Or you can read in a file:

```
7. | ?- consult('filename').
```

which is added at the end of the working space.

8. Or you can assert a clause

```
9. | ?- assert(< clause >).  
10.
```

Here are some confusing "equalities" in prolog:

predicate	relation	variable substitution	arithmetic computation
<hr/>			
==	identical	no	no
=	unifiable	yes	no
=:=	same value	no	yes
is	is the value of	yes	yes

and some examples of their use:

```
?- X == Y.  
no  
  
?- X + 1 == 2 + Y.  
no  
  
?- X + 1 = 2 + Y.  
X = 2  
Y = 1  
  
?- X + 1 = 1 + 2.  
no
```

Example: Factorial1

```
factorial(0,1).  
factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1), F is N*F1.  
then  
?- factorial(5,F).  
F = 120  
  
?- factorial(X,120).  
instantiation error
```

Example: Factorial2: dropping N>0 from factorial1

```
factorial(0,1).  
factorial(N,F) :- N1 is N-1, factorial(N1,F1), F is N*F1.  
then  
?- factorial(5,F).  
F = 120; Here ";" asks for next value of F  
keeps on going until stack overflow
```

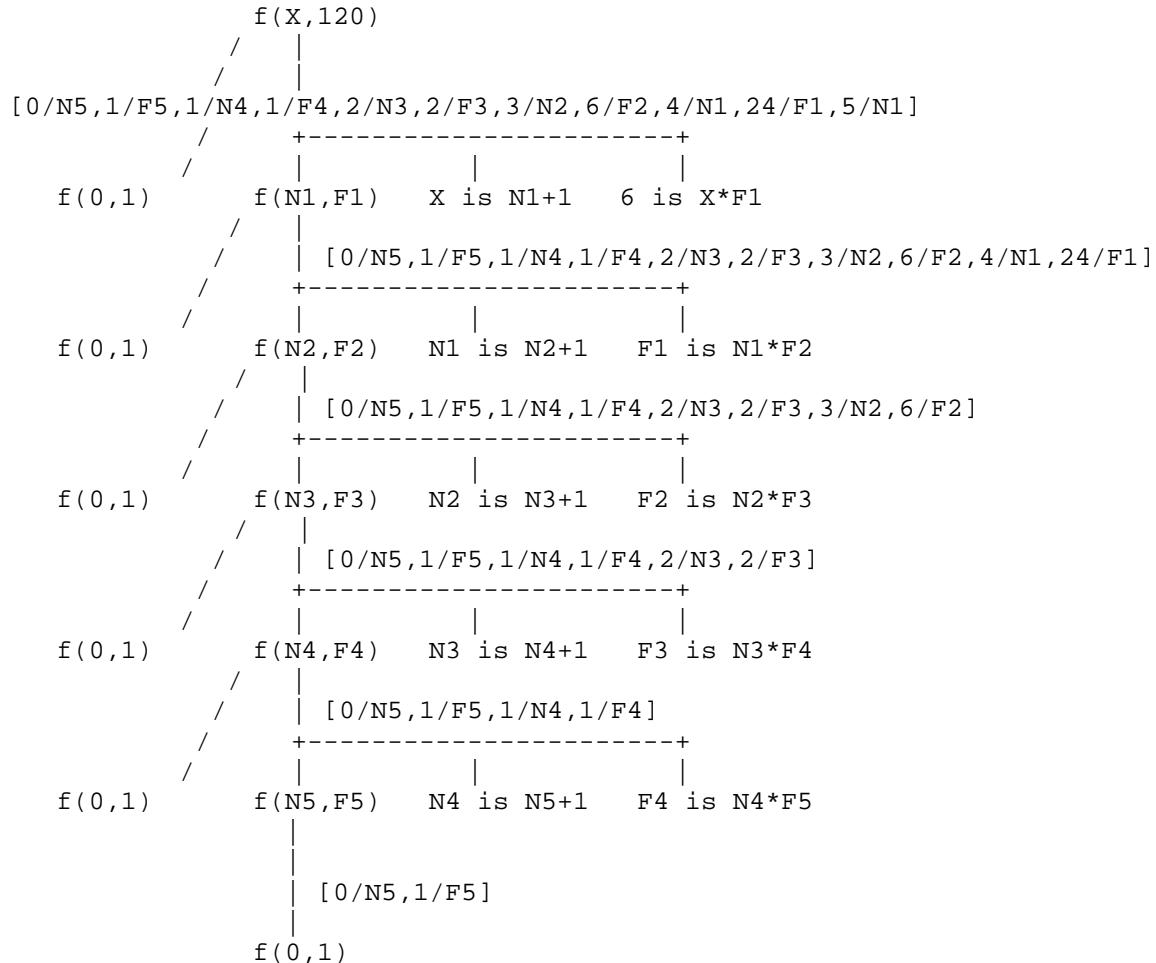
Example: Factorial3: Changing order of terms

```
(1) factorial(0,1).  
(2) factorial(N,F) :- factorial(N1,F1), N is N1+1, F is N*F1.
```

then

```
?- factorial(5,F).  
F = 120  
  
?- factorial(X,120).  
X = 5;  
integer overflow
```

Here is why factorial(X,120) returns 5. For brevity, instead of "factorial" we will write "f".



In this diagram we see the substitutions computed. Much is not said in the diagram, for example why we abandon the unifications with the various f(0,1)s. [Let's say it for the second f(0,1) from the top: because it forces the substitution [0/N1,1/F1,1/X] and this cause 6 is X*F1 to fail.]

Lists

Lists are very much as in lisp. In place of Lisp's cons, in Prolog we use the "." or dot:

Dot Notation	List Notation	Lisp Notation
.(X,Y)	[X Y]	(X . Y)
.(X, .(Y,Z))	[X,Y Z]	(X (Y . Z))
.(X, .(Y, .(Z, [])))	[X,Y,Z]	(X Y Z)

Example: len

```

len([],0).
len([_|T], N) :- len(T,M), N is M+1.

?- len([a,b,c],X).
X = 3

?- len([a,b,c], 3).
yes

?- len([a,b,c], 5).
no

```

Example: member

member(X,Y) is intended to mean X is one of the top level elements of the list Y.

```

member(X,[X|_]).
member(X,[_|T]) :- member(X,T).

?- member(X, [1,2,3,4,5]).
X=1;
X=2;
X=3;
X=4;
X=5;
no

```

Example: select

select(X,A,R) is intended to mean that X is a top level element of the list A and that R is what is left of A after taking X out of it.

```

select(H,[H|T],T).
select(X,[H|T],[H|T1]) :- select(X,T,T1).

?- select(X,[a,b,c],R).
X=a
R=[b,c];
X=b
R=[a,c];
X=c
R=[a,b];
No

```

The Cryptography Problem

Here is a problem:

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

to be solved by mapping the letters into distinct digits and then doing regular arithmetic.
We add variables to represent the various carries:

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

We observe that carries can only be 0 or 1 and thus that M has to be 1. Then here is a solution:

```
solve([S,E,N,D], [M,O,R,E], [M,O,N,E,Y]) :-  
    M=1, L=[2,3,4,5,6,7,8,9],  
    select(S,L,L1), S>0, (C3=0; C3=1),           ";" means OR  
    O is S+M+C3-10*M, select(O, L1, L2),  
    select(E,L2,L3), (C2=0;C2=1),  
    N is E+O+C2-10*C3, select(N,L3,L4), (C1=0;C1=1),  
    R is E+10*C2-(N+C1), select(R,L4,L5),  
    select(D,L5,L6),  
    Y is D+E-10*C1, select(Y,L6,_).  
  
?- solve([S,E,N,D], [M,O,R,E], [M,O,N,E,Y]).  
S=9  
E=5  
N=6  
D=7  
M=1  
O=0  
R=8  
Y=2;  
No
```

7.2.6 Expert Systems

An **expert system** is a computer program that contains some of the subject-specific knowledge of one or more human experts. An expert systems are meant to solve real problems which normally would require a specialized human expert (such as a doctor or a mineralogist). Building an expert system therefore first involves extracting the relevant knowledge from the human expert. Such knowledge is often heuristic in nature, based on useful ``rules of thumb" rather than absolute certainties. Extracting it from the expert in a way that can be used by a computer is generally a difficult task, requiring its own expertise. A *knowledge engineer* has the job of extracting this knowledge and building the expert system *knowledge base*.

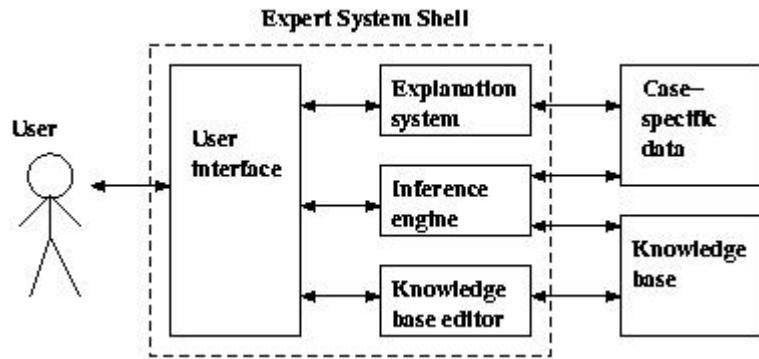
A first attempt at building an expert system is unlikely to be very successful. This is partly because the expert generally finds it very difficult to express exactly what knowledge and rules they use to solve a problem. Much of it is almost subconscious, or appears so obvious they don't even bother mentioning it. *Knowledge acquisition* for expert systems is a big area of research, with a wide variety of techniques developed. However, generally it is important to develop an initial prototype based on information extracted by interviewing the expert, then iteratively refine it based on feedback both from the expert and from potential users of the expert system.

In order to do such iterative development from a prototype it is important that the expert system is written in a way that it can easily be inspected and modified. The system should be able to explain its reasoning (to expert, user and knowledge engineer) and answer questions about the solution process. Updating the system shouldn't involve rewriting a whole lot of code - just adding or deleting localized chunks of knowledge.

The most widely used knowledge representation scheme for expert systems is rules. Typically, the rules won't have certain conclusions - there will just be some degree of certainty that the conclusion will hold if the conditions hold. Statistical techniques are used to determine these certainties. Rule-based systems, with or without certainties, are generally easily modifiable and make it easy to provide reasonably helpful traces of the system's reasoning. These traces can be used in providing explanations of what it is doing.

Expert systems have been used to solve a wide range of problems in domains such as medicine, mathematics, engineering, geology, computer science, business, law, defence and education. Within each domain, they have been used to solve problems of different types. Types of problem involve *diagnosis* (e.g., of a system fault, disease or student error); *design* (of a computer systems, hotel etc); and *interpretation* (of, for example, geological data). The appropriate problem solving technique tends to depend more on the problem type than on the domain. Whole books have been written on how to choose your knowledge representation and reasoning methods given characteristics of your problem.

The following figure shows the most important modules that make up a rule-based expert system. The user interacts with the system through a *user interface* which may use menus, natural language or any other style of interaction). Then an *inference engine* is used to reason with both the *expert knowledge* (extracted from our friendly expert) and data specific to the particular problem being solved. The expert knowledge will typically be in the form of a set of IF-THEN rules. The *case specific data* includes both data provided by the user and partial conclusions (along with certainty measures) based on this data. In a simple forward chaining rule-based system the case specific data will be the elements in *working memory*.



Almost all expert systems also have an *explanation subsystem*, which allows the program to explain its reasoning to the user. Some systems also have a *knowledge base editor* which help the expert or knowledge engineer to easily update and check the knowledge base.

One important feature of expert systems is the way they (usually) separate domain specific knowledge from more general purpose reasoning and representation techniques. The general purpose bit (in the dotted box in the figure) is referred to as an *expert system shell*. As we see in the figure, the shell will provide the inference engine (and knowledge representation scheme), a user interface, an explanation system and sometimes a knowledge base editor. Given a new kind of problem to solve (say, car design), we can usually find a shell that provides the right sort of support for that problem, so all we need to do is provide the expert knowledge. There are numerous commercial expert system shells, each one appropriate for a slightly different range of problems. (Expert systems work in industry includes both writing expert system shells and writing expert systems using shells.) Using shells to write expert systems generally greatly reduces the cost and time of development.

Questions

1. Consider the first-order logic sentences defined below.

$$\begin{aligned} & \forall x, y P(x, y) \wedge Q(y, x) \Rightarrow R(x, y) \\ & \forall x, y S(x, Bob) \wedge S(y, x) \Rightarrow P(x, y) \\ & \forall x, y S(x, y) \Rightarrow Q(y, x) \\ & \forall x, y T(x, y, x) \Rightarrow Q(x, y) \\ & \forall x, y T(x, x, y) \Rightarrow Q(x, y) \\ & T(Alice, Dawn, Alice) \\ & T(Eve, Carl, Eve) \\ & T(Alice, Bob, Dawn) \\ & T(Carl, Carl, Alice) \\ & S(Bob, Alice) \\ & S(Carl, Bob) \\ & S(Dawn, Carl) \\ & S(Carl, Dawn) \\ & S(Alice, Dawn) \\ & S(Eve, Carl) \end{aligned}$$

Use backward chaining to find **ALL** answers for the following queries. When matching rules, proceed from top to bottom, and evaluate subgoals from left to right.

Query 1: $\exists x Q(Alice, x)$

Query 2: $\exists x, y R(x, y)$.

2. Translate the following first-order logic sentences into Prolog. Note, some sentences may require more than one Prolog statement.

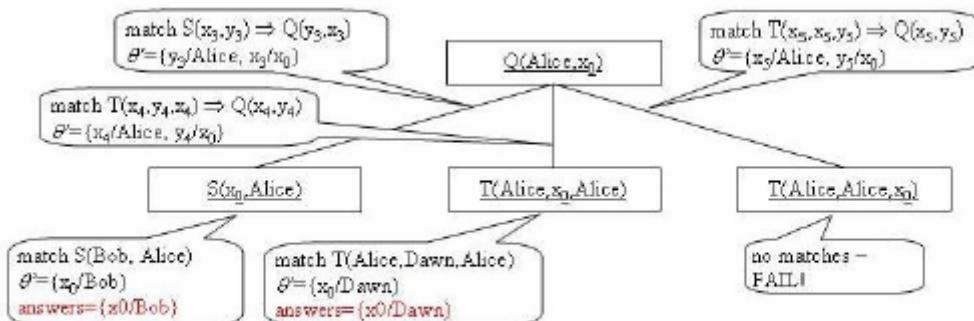
- Knows(Sylvester, Tweetie)
- $\forall x, y \text{ Friend}(x, y) \Rightarrow \text{Knows}(x, y)$
- $\forall x, y (\text{Cat}(x) \wedge \text{Bird}(y)) \Rightarrow \text{LikesToEat}(x, y)$
- $\forall x (\text{Parakeet}(x) \vee \text{Penguin}(x)) \Rightarrow \text{Bird}(x)$
- $\forall x \text{ Parakeet}(x) \Rightarrow (\text{Flies}(x) \wedge \text{Chirps}(x))$

3. Write a PROLOG programs to append two lists.

Solution

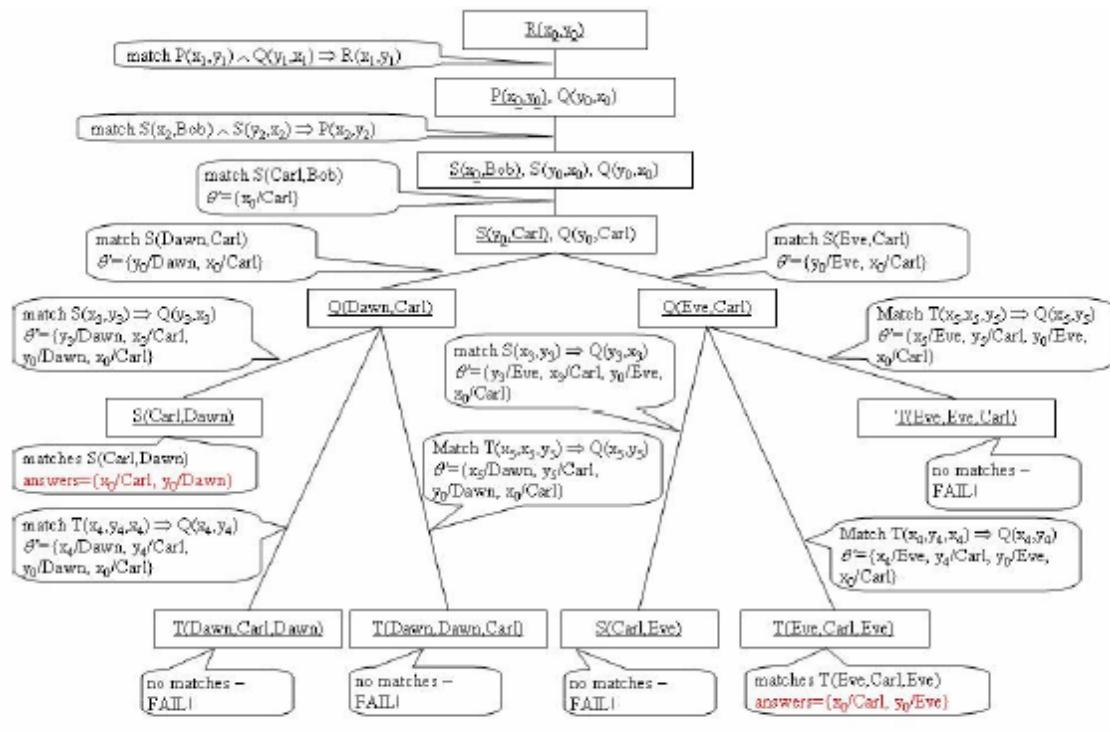
1. The proof trees are shown below:

Query 1:



Answers = { {x₀/Bob}, {x₁/Dawn} }

Query 2:



Answers = { $\{x_0/\text{Carl}, y_0/\text{Dawn}\}$, $\{x_0/\text{Carl}, y_0/\text{Eve}\}$ }

2. Prolog statements are

- a. *knows(sylvester,tweetie).*
- b. *knows(X,Y):- friend(X,Y).*
- c. *likesToEat(X,Y) :- cat(X), bird(Y).*
- d. *bird(X) :- parakeet(X).*
- e. *bird(X) :- penguin(X).*
- f. *flies(X) :- parakeet(X).*
- g. *chirps(X) :- parakeet(X).*

3. Prolog program

```
append(nil,L,L).  
append(c(X,L),M,c(X,N)) :- append(L,M,N).
```

Module 8

Other representation formalisms

8.1 Instructional Objective

- The students should understand the syntax and semantic of semantic networks
- Students should learn about different constructs and relations supported by semantic networks
- Students should be able to design semantic nets to represent real life problems
- The student should be familiar with reasoning techniques in semantic nets
- Students should be familiar with syntax and semantic of frames
- Students should be able to obtain frame representation of a real life problem
- Inferencing mechanism in frames should be understood.

At the end of this lesson the student should be able to do the following:

- Represent a real life problem in terms of semantic networks and frames
- Apply inferencing mechanism on this knowledge-base.

Lesson 19

Semantic nets

8. 2 Knowledge Representation Formalisms

Some of the abstract knowledge representation mechanisms are the following:

Simple relational knowledge

The simplest way of storing facts is to use a relational method where each fact about a set of objects is set out systematically in columns. This representation gives little opportunity for inference, but it can be used as the knowledge basis for inference engines.

- Simple way to store facts.
- Each fact about a set of objects is set out systematically in columns.
- Little opportunity for inference.
- Knowledge basis for inference engines.

Musician	Style	Instrument	Age
Miles Davis	Jazz	Trumpet	deceased
John Zorn	Avant Garde	Saxophone	35
Frank Zappa	Rock	Guitar	deceased
John McLaughlin	Jazz	Guitar	47

We can ask things like:

- Who is dead?
- Who plays Jazz/Trumpet etc.?

This sort of representation is popular in database systems.

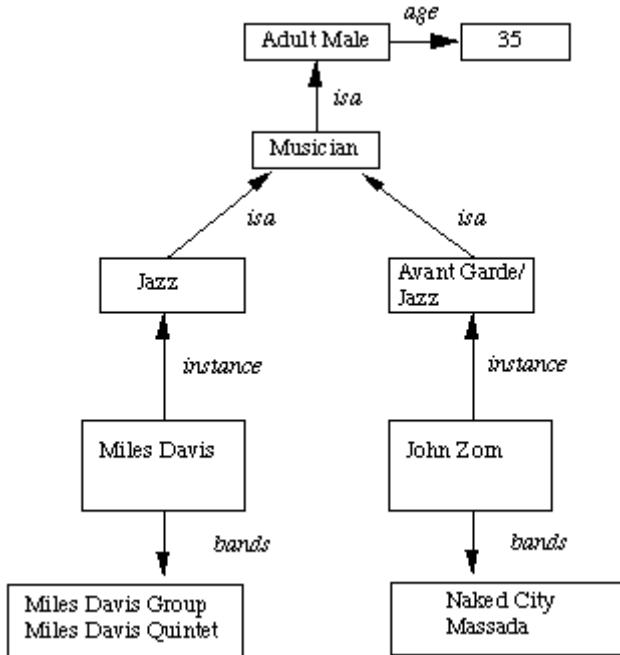
Inheritable knowledge

Relational knowledge is made up of objects consisting of

- attributes
- corresponding associated values.

We extend the base more by allowing inference mechanisms:

- Property inheritance
 - elements inherit values from being members of a class.
 - data must be organised into a hierarchy of classes.



- Boxed nodes -- objects and values of attributes of objects.
- Values can be objects with attributes and so on.
- Arrows -- point from object to its value.
- This structure is known as a slot and filler structure, semantic network or a collection of frames.

The algorithm to retrieve a value for an attribute of an instance object:

1. Find the object in the knowledge base
2. If there is a value for the attribute report it
3. Otherwise look for a value of instance if none fail
4. Otherwise go to that node and find a value for the attribute and then report it
5. Otherwise search through using *isa* until a value is found for the attribute.

Inferential Knowledge

Represent knowledge as *formal logic*:

All dogs have tails $\forall x: \text{dog}(x) \rightarrow \text{hasatail}(x)$ Advantages:

- A set of strict rules.
 - Can be used to derive more facts.
 - Truths of new statements can be verified.
 - Guaranteed correctness.
- Many inference procedures available to implement standard rules of logic.
- Popular in AI systems. e.g Automated theorem proving.

Procedural knowledge

Basic idea:

- Knowledge encoded in some procedures
 - small programs that know how to do specific things, how to proceed.
 - *e.g* a parser in a natural language understander has the knowledge that a *noun phrase* may contain articles, adjectives and nouns. It is represented by calls to routines that know how to process articles, adjectives and nouns.

Advantages:

- *Heuristic* or domain specific knowledge can be represented.
- *Extended logical inferences*, such as default reasoning facilitated.
- *Side effects* of actions may be modelled. Some rules may become false in time. Keeping track of this in large systems may be tricky.

Disadvantages:

- Completeness -- not all cases may be represented.
- Consistency -- not all deductions may be correct.

e.g If we know that *Fred is a bird* we might deduce that *Fred can fly*. Later we might discover that *Fred is an emu*.

- Modularity is sacrificed. Changes in knowledge base might have far-reaching effects.
- Cumbersome control information.

The following properties should be possessed by a knowledge representation system.

Representational Adequacy

-- the ability to represent the required knowledge;

Inferential Adequacy

- the ability to manipulate the knowledge represented to produce new knowledge corresponding to that inferred from the original;

Inferential Efficiency

- the ability to direct the inferential mechanisms into the most productive directions by storing appropriate guides;

Acquisitional Efficiency

- the ability to acquire new knowledge using automatic methods wherever possible rather than reliance on human intervention.

To date no single system optimises all of the above. We will discuss in this module two formalisms, namely, semantic networks and frames, which trades off representational adequacy for inferential and acquisitional efficiency.

5.3 Semantic Networks

A semantic network is often used as a form of knowledge representation. It is a directed graph consisting of vertices which represent concepts and edges which represent semantic relations between the concepts.

The following semantic relations are commonly represented in a semantic net.

Meronymy (A is part of B)

Holonymy (B has A as a part of itself)

Hyponymy (or *troponymy*) (A is subordinate of B; A is kind of B)

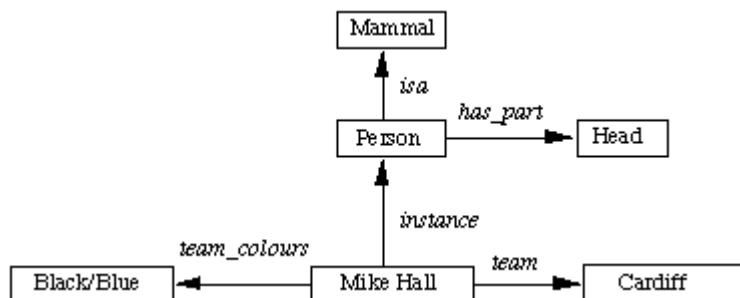
Hypernymy (A is superordinate of B)

Synonymy (A denotes the same as B)

Antonymy (A denotes the opposite of B)

Example

The physical attributes of a person can be represented as in the following figure using a semantic net.

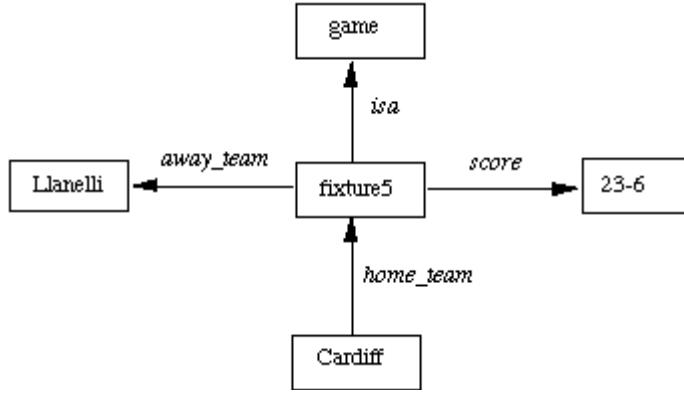


These values can also be represented in logic as: *isa(person, mammal)*, *instance(Mike-Hall, person)* *team(Mike-Hall, Cardiff)*

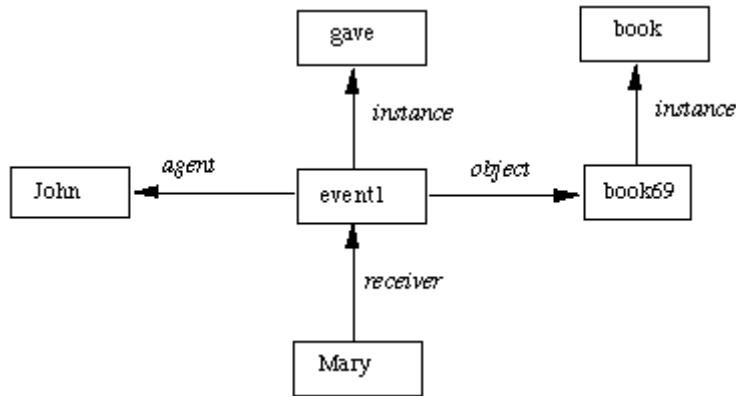
We know that conventional predicates such as *lecturer(dave)* can be written as *instance(dave, lecturer)*. Recall that *isa* and *instance* represent inheritance and are popular in many knowledge representation schemes. But we have a problem: *How we can have more than 2 place predicates in semantic nets?* E.g. *score(Cardiff, Llanelli, 23-6)*

Solution:

- Create new nodes to represent new objects either contained or alluded to in the knowledge, *game* and *fixture* in the current example.
- Relate information to nodes and fill up slots.



As a more complex example consider the sentence: *John gave Mary the book*. Here we have several aspects of an event.



5.3.1 Inference in a Semantic Net

Basic inference mechanism: *follow links between nodes*.

Two methods to do this:

Intersection search

-- the notion that *spreading activation* out of two nodes and finding their intersection finds relationships among objects. This is achieved by assigning a special tag to each visited node.

Many advantages including entity-based organisation and fast parallel implementation. However very structured questions need highly structured networks.

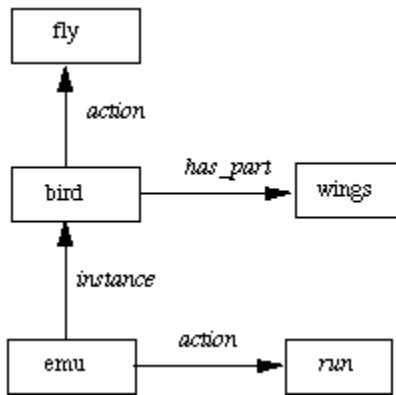
Inheritance

-- the *isa* and *instance* representation provide a mechanism to implement this.

Inheritance also provides a means of dealing with *default reasoning*. E.g. we could represent:

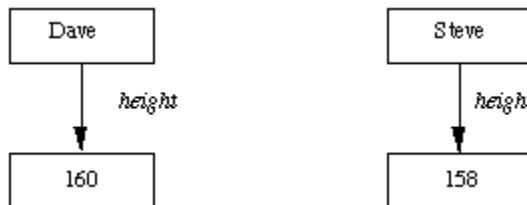
- Emus are birds.
- Typically birds fly and have wings.
- Emus run.

in the following Semantic net:

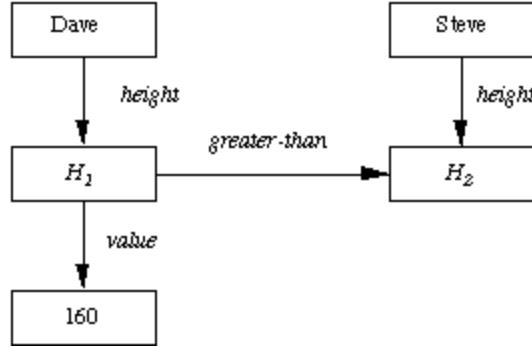


In making certain inferences we will also need to *distinguish between the link that defines a new entity and holds its value and the other kind of link that relates two existing entities*. Consider the example shown where the height of two people is depicted and we also wish to compare them.

We need extra nodes for the concept as well as its value.



Special procedures are needed to process these nodes, but without this distinction the analysis would be very limited.



Extending Semantic Nets

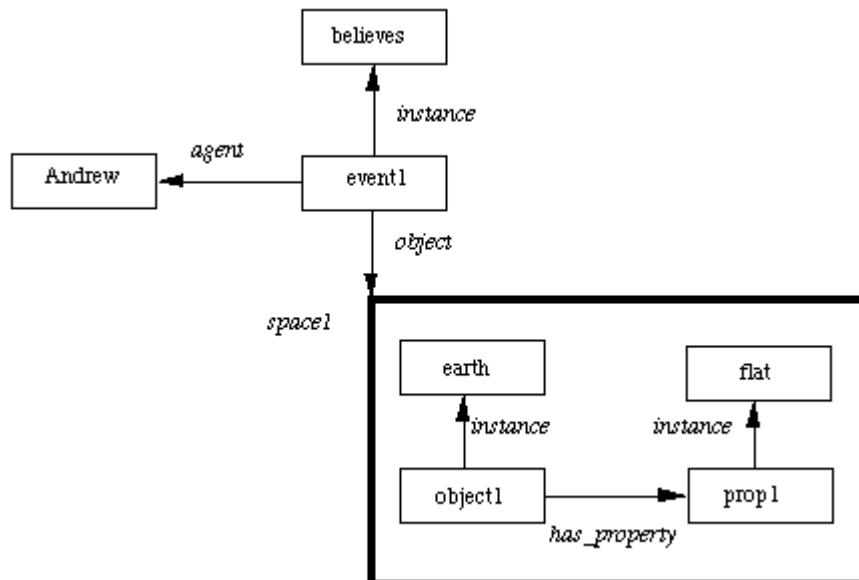
Here we will consider some extensions to Semantic nets that overcome a few problems or extend their expression of knowledge.

Partitioned Networks *Partitioned* Semantic Networks allow for:

- propositions to be made without commitment to truth.
- expressions to be quantified.

Basic idea: *Break network into spaces* which consist of groups of nodes and arcs and regard each *space* as a node.

Consider the following: *Andrew believes that the earth is flat*. We can encode the proposition *the earth is flat* in a *space* and within it have nodes and arcs that represent the fact (next figure). We can then have nodes and arcs to link this *space* to the rest of the network to represent Andrew's belief.

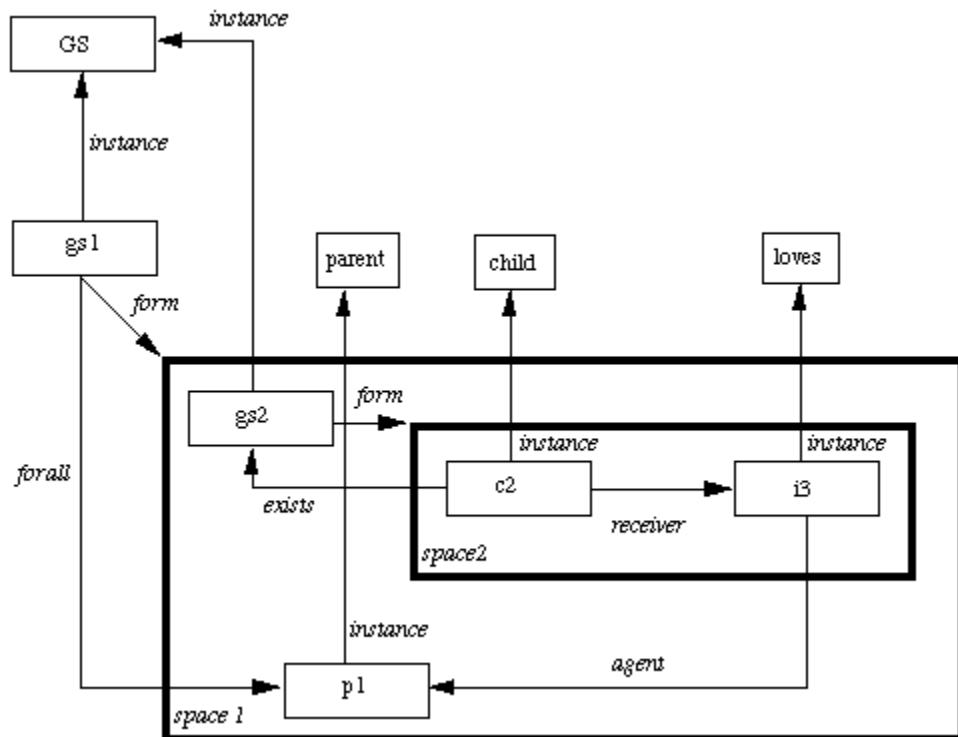


Now consider the quantified expression: *Every parent loves their child* To represent this we:

- Create a *general statement*, GS, special class.
- Make node g an instance of GS.
- Every element will have at least 2 attributes:
 - a *form* that states which relation is being asserted.
 - one or more *forall* (\forall) or *exists* (\exists) connections -- these represent universally quantifiable variables in such statements e.g. x, y in $\forall x \exists y : child(y) \wedge loves(x,y)$

Here we have to construct two *spaces one* for each x, y . **NOTE:** We can express \exists variables as *existentially qualified* variables and express the event of *love* having an agent p and receiver b for every parent p which could simplify the network.

Also If we change the sentence to *Every parent loves child* then the node of the object being acted on (*the child*) lies outside the form of the general statement. Thus it is not viewed as an existentially qualified variable whose value may depend on the agent. So we could construct a partitioned network as in the next figure.



Module 8

Other representation formalisms

Lesson 20

Frames - I

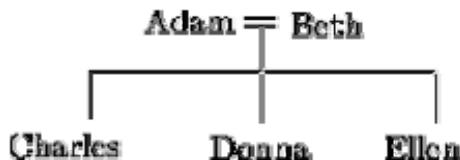
8.4 Frames

Frames are descriptions of conceptual individuals. Frames can exist for ``real'' objects such as ``The Watergate Hotel'', sets of objects such as ``Hotels'', or more ``abstract'' objects such as ``Cola-Wars'' or ``Watergate''.

A Frame system is a collection of objects. Each object contains a number of *slots*. A slot represents an attribute. Each slot has a value. The value of an attribute can be another object. Each object is like a C struct. The struct has a name and contains a bunch of named values (which can be pointers)

Frames are essentially defined by their relationships with other frames. Relationships between frames are represented using *slots*. If a frame f is in a relationship r to a frame g , then we put the *value* g in the r *slot* of f .

For example, suppose we are describing the following genealogical tree:



The frame describing Adam might look something like:

Adam:

```
sex: Male
spouse: Beth
child: (Charles Donna Ellen)
```

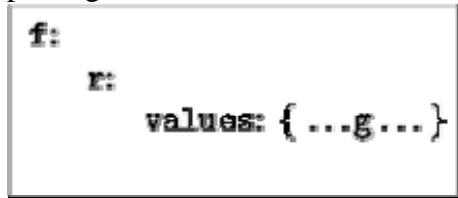
where *sex*, *spouse*, and *child* are slots. Note that a single slot may hold several values (*e.g.* the children of Adam).

The genealogical tree would then be described by (at least) seven frames, describing the following individuals: Adam, Beth, Charles, Donna, Ellen, Male, and Female.

A frame can be considered just a convenient way to represent a set of predicates applied to constant symbols (*e.g.* *ground instances* of predicates.). For example, the frame above could be written:

```
sex(Adam, Male)
spouse(Adam, Beth)
child(Adam, Charles)
child(Adam, Donna)
child(Adam, Ellen)
```

More generally, the ground predicate $r(f,g)$ is represented, in a frame based system, by placing the value g in the r slot of the frame $f : r(f,g) \equiv$



Frames can also be regarded as an extension to Semantic nets. Indeed it is not clear where the distinction between a semantic net and a frame ends. Semantic nets initially were used to represent labelled connections between objects. As tasks became more complex the representation needs to be more structured. The more structured the system it becomes more beneficial to use frames. A *frame* is a collection of attributes or slots and associated values that describe some real world entity. Frames on their own are not particularly helpful but frame systems are a powerful way of encoding information to support reasoning. Set theory provides a good basis for understanding frame systems. Each frame represents:

- a class (set), or
 - an instance (an element of a class).

Consider the example below.

Person

isa: *Mammal*

Cardinality:

Adult-Male

isa: *Person*

Cardinality:

Rugby-Player

isa: *Adult-Male*

Cardinality:

Height:

Weight:

Position:

Team:

Team-Colours:

Back

isa: *Rugby-Player*

Cardinality: ...

Tries:

Mike-Hall

instance: *Back*

Height: *6-0*

Position: *Centre*

Team: *Cardiff-RFC*

Team-Colours: *Black/Blue*

Rugby-Team

isa: *Team*

Cardinality: ...

Team-size: *15*

Coach:

Cardiff-RFC

instance: *Rugby-Team*

Team-size: *15*

Coach: *Terry Holmes*

Players: *{Robert-Howley, Gwyn-Jones, ... }*

Here the frames *Person*, *Adult-Male*, *Rugby-Player* and *Rugby-Team* are all **classes** and the frames *Robert-Howley* and *Cardiff-RFC* are instances.

Note

- The *isa* relation is in fact the subset relation.
- The *instance* relation is in fact *element of*.
- The *isa* attribute possesses a transitivity property. This implies: *Robert-Howley* is a *Back* and a *Back* is a *Rugby-Player* who in turn is an *Adult-Male* and also a *Person*.
- Both *isa* and *instance* have inverses which are called subclasses or all instances.
- There are attributes that are associated with the class or set such as cardinality and on the other hand there are attributes that are possessed by each member of the class or set.

DISTINCTION BETWEEN SETS AND INSTANCES

It is important that this distinction is clearly understood.

Cardiff-RFC can be thought of as a set of players or as an instance of a *Rugby-Team*.

If *Cardiff-RFC* were a *class* then

- its instances would be players
- it could not be a subclass of *Rugby-Team* otherwise its elements would be members of *Rugby-Team* which we do not want.

Instead we make it a subclass of *Rugby-Player* and this allows the players to inherit the correct properties enabling us to let the *Cardiff-RFC* to inherit information about teams.

This means that *Cardiff-RFC* is an instance of *Rugby-Team*.

BUT There is a problem here:

- A class is a set and its elements have properties.
- We wish to use inheritance to bestow values on its members.
- But there are properties that the set or class itself has such as the manager of a team.

This is why we need to view *Cardiff-RFC* as a subset of one class players and an instance of teams. We seem to have a CATCH 22. *Solution: Metaclasses*

A metaclass is a special class whose elements are themselves classes.

Now consider our rugby teams as:

<i>Class</i>	
<i>instance:</i>	<i>Class</i>
<i>isa:</i>	<i>Class</i>
<i>Cardinality:</i>	...
 <i>Team</i>	
<i>instance:</i>	<i>Class</i>
<i>isa:</i>	<i>Class</i>
<i>Cardinality:</i>	{ The number of teams }
<i>Team-Size:</i>	15
 <i>Rugby-Team</i>	
<i>isa:</i>	<i>Team</i>
<i>Cardinality:</i>	{ The number of teams }
<i>Team-size:</i>	15
<i>Coach:</i>	
 <i>Cardiff-RFC</i>	
<i>instance:</i>	<i>Rugby-Team</i>
<i>Team-size:</i>	15
<i>Coach:</i>	Terry Holmes
 <i>Robert-Howley</i>	
<i>instance:</i>	<i>Back</i>
<i>Height:</i>	6-0
<i>Position:</i>	Serum Half
<i>Team:</i>	<i>Cardiff-RFC</i>
<i>Team-Colours:</i>	Black/Blue

The basic metaclass is *Class*, and this allows us to

- define classes which are instances of other classes, and (thus)
- inherit properties from this class.

Inheritance of default values occurs when one element or class is an instance of a class.

Module 8

Other representation formalisms

Lesson 21

Frames – II

Slots as Objects

How can we represent the following properties in frames?

- Attributes such as *weight*, *age* be attached and make sense.
- Constraints on values such as *age* being less than a hundred
- Default values
- Rules for inheritance of values such as children inheriting parent's names
- Rules for computing values
- Many values for a slot.

A slot is a relation that maps from its domain of classes to its range of values.

A relation is a set of ordered pairs so one relation is a subset of another.

Since slot is a set the set of all slots can be represented by a metaclass called *Slot*, say.

Consider the following:

SLOT

isa: Class
instance: Class
domain:
range:
range-constraint:
definition:
default:
to-compute:
single-valued:

Coach

instance: SLOT
domain: Rugby-Team
range: Person

range-constraint: $\lambda x \ (experience \ x.manager)$
default:
single-valued: *TRUE*
Colour

instance: *SLOT*
domain: *Physical-Object*
range: *Colour-Set*
single-valued: *FALSE*
Team-Colours

instance: *SLOT*
isa: *Colour*
domain: *team-player*
range: *Colour-Set*
range-constraint: *not Pink*
single-valued: *FALSE*
Position

instance: *SLOT*
domain: *Rugby-Player*
range: { *Back, Forward, Reserve* }
to-compute: $\lambda x \ x.position$
single-valued: *TRUE*

NOTE the following:

- Instances of *SLOT* are slots
- Associated with *SLOT* are attributes that each instance will inherit.

- Each slot has a domain and range.
- Range is split into two parts one the class of the elements and the other is a constraint which is a logical expression if absent it is taken to be true.
- If there is a value for default then it must be passed on unless an instance has its own value.
- The *to-compute* attribute involves a procedure to compute its value. *E.g.* in *Position* where we use the dot notation to assign values to the slot of a frame.
- Transfers through lists other slots from which values can be derived

Interpreting frames

A frame system interpreter must be capable of the following in order to exploit the frame slot representation:

- Consistency checking -- when a slot value is added to the frame relying on the domain attribute and that the value is legal using range and range constraints.
- Propagation of *definition* values along *isa* and *instance* links.
- Inheritance of default. values along *isa* and *instance* links.
- Computation of value of slot as needed.
- Checking that only correct number of values computed.

Access Paths

One advantage of a frame based representation is that the (conceptual) objects related to a frame can be easily accessed by looking in a slot of the frame (there is no need, for example, to search the entire knowledge-base). We define an *access path*, in a network of frames, as a sequence of frames each directly accessible from (*i.e.* appearing in a slot of) its predecessor. A sequence of predicates defines an access path iff any variable appearing as the first argument to a predicate has appeared previously in the sequence. For example, ``John's parent's sister" can be expressed in Algernon as the path:

((parent John ?x) (sister ?x ?y))

The access path **((parent John ?x) (sister ?x ?y))** is equivalent to the syntactically similar predicate calculus statement:

parent(John, ?x) \wedge sister(?x, ?y).

In predicate calculus this statement is equivalent to

sister(?x, ?y) \wedge parent(John, ?x).

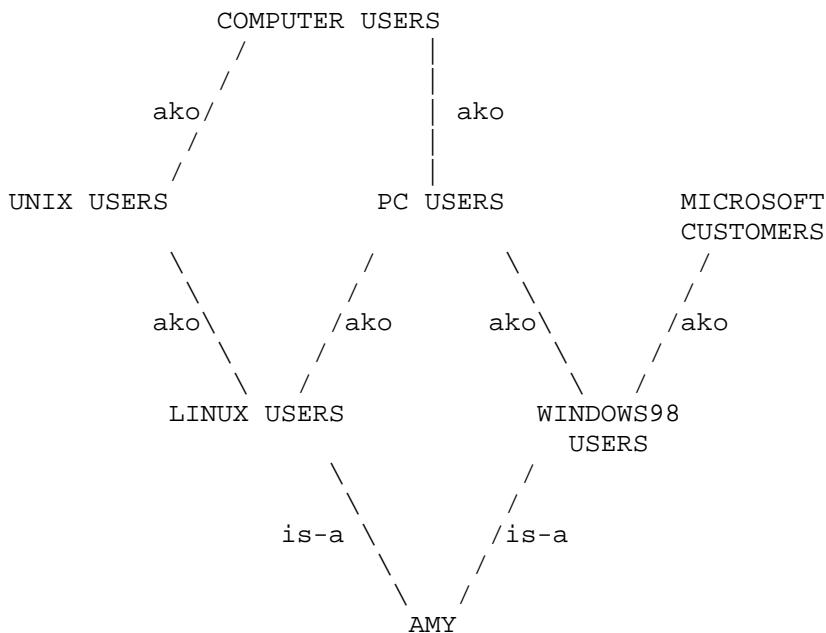
However, the corresponding sequence of predicates:

((sister ?x ?y) (parent John ?x))

is *not* an access path because a query of (`sister ?x ?y`) requires a search of every frame in the entire knowledge-base.

Questions

1. Construct semantic network representations for the information below.
 - a. Richard Nixon is a Quaker and a Republican. Quakers and Republicans are Persons. Every Quaker every quaker follows the doctrine of pacifism.
 - b. Mary gave the green flowered vase to her cousin.
2. Consider the following hierarchy of frames.

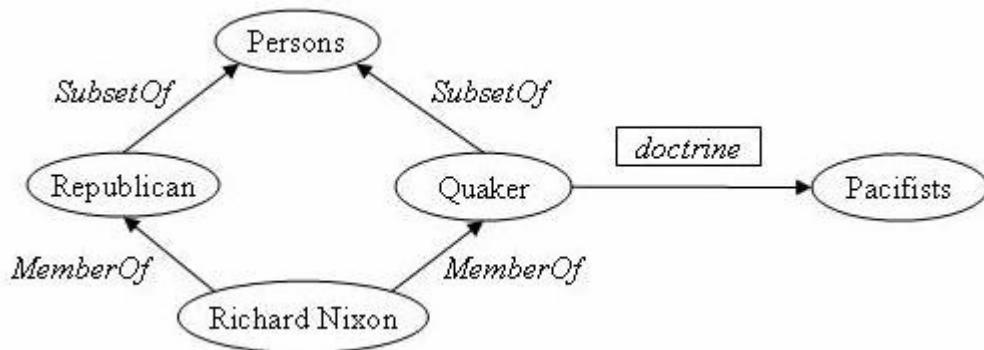


- a. Give the class-precedence list for Amy that would be obtained by applying the topological-sorting algorithm to the above graph.
- b. Suppose that each of the classes *Unix users*, *PC users* and *Computer Users* contains a *favorite programming language* slot. The default value for this slot is:
 - o Fortran, for the *Computer Users* class.
 - o C, for the *Unix Users* class.
 - o C++, for the *PC Users* class.

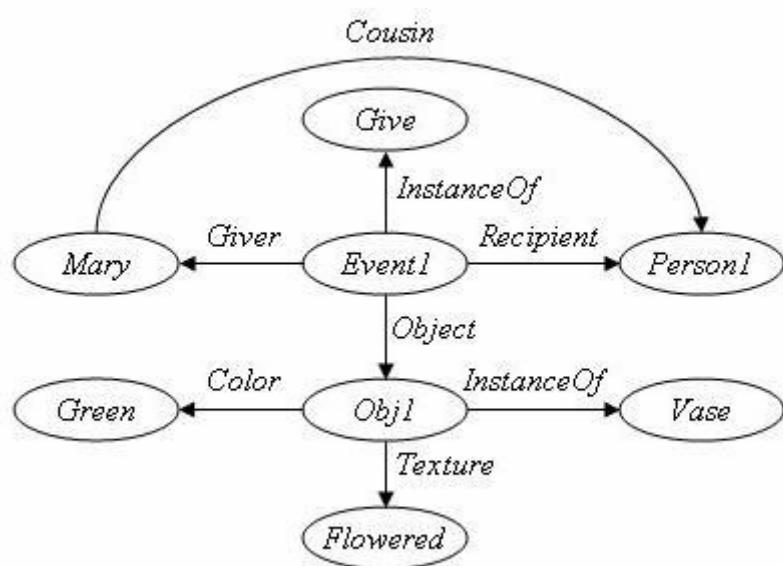
What is the value obtained for Amy's favorite programming language according to the class-precedence list you constructed above?

Solutions

1.a



1.b



2.a.

Node	Fish-hook pairs
Amy	Amy-Linux Users, Linux Users-Windows98 Users
Linux Users	Linux Users-Unix Users, Unix Users-PC Users
Windows98 Users	Windows98 Users-PC Users, PC Users-Microsoft Customers
Unix Users	Unix Users-Computer Users
PC Users	PC Users-Computer Users

Class Precedence list :

Amy	
Linux Users	- Use C
Unix Users	- Use C++
Windows98 Users	- Use Fortran
PC Users	
Computer Users	
Microsoft Customers	

1. Amy's favorite programming language is C

Module 9

Planning

9.1 Instructional Objective

- The students should understand the formulation of planning problems
- The student should understand the difference between problem solving and planning and the need for knowledge representation in large scale problem solving
- Students should understand the STRIPS planning language
- **Students should be able to represent a real life planning problem using STRIPS operators**
- Students should understand planning using situation calculus and the related frame problems
- Students should understand the formulation of planning as a search problem
- Students should learn the following planning algorithms
 - Situation space planning
 - Plan space planning
 - Progression planning
 - Regression planning
- The student should understand the difficulties of full commitment planning
- Students should understand the necessity of least commitment
- Students should learn partial order planning algorithms

At the end of this lesson the student should be able to do the following:

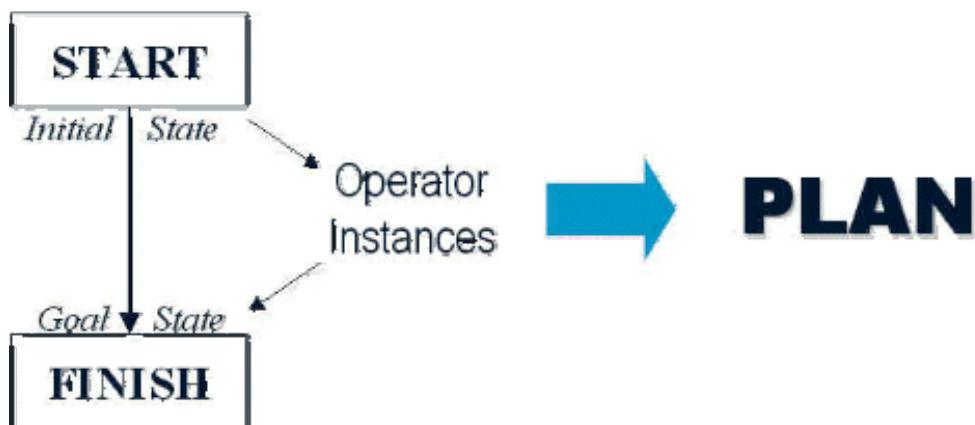
- **Represent a planning problem in STRIPS language**
- **Use a suitable planning algorithm to solve the problem.**

Lesson 22

Logic based planning

9. 1 Introduction to Planning

The purpose of planning is to find a sequence of actions that achieves a given goal when performed starting in a given state. In other words, given a set of operator instances (defining the possible primitive actions by the agent), an initial state description, and a goal state description or predicate, the planning agent computes a plan.



What is a plan? A sequence of operator instances, such that "executing" them in the initial state will change the world to a state satisfying the goal state description. Goals are usually specified as a conjunction of goals to be achieved.

9.1.1 A Simple Planning Agent:

Earlier we saw that **problem-solving agents** are able to plan ahead - to consider the consequences of *sequences* of actions - before acting. We also saw that a **knowledge-based agents** can select actions based on explicit, logical representations of the current state and the effects of actions. This allows the agent to succeed in complex, inaccessible environments that are too difficult for a problem-solving agent

Problem Solving Agents + Knowledge-based Agents = Planning Agents

In this module, we put these two ideas together to build **planning agents**. At the most abstract level, the task of planning is the same as problem solving. Planning can be viewed as a type of problem solving in which the agent uses beliefs about actions and their consequences to search for a solution over the more abstract space of plans, rather than over the space of situations

6.1.2 Algorithm of a simple planning agent:

1. Generate a goal to achieve
2. Construct a plan to achieve goal from current state
3. Execute plan until finished
4. Begin again with new goal

The agent first generates a goal to achieve, and then constructs a plan to achieve it from the current state. Once it has a plan, it keeps executing it until the plan is finished, then begins again with a new goal.

This is illustrated in the following pseudocode:

```

function SIMPLE-PLANNING-AGENT (percept) returns an action
  static: KB, a knowledge base(includes action descriptions)
    p , a plan, initially NoPlan
    t, a counter, initially 0, indicating time
  local variables: G, a goal
    current, a current state description

  TELL(KB, MAKE-PERCEPT-SENTENCE (percept,t))
  current <- STATE-DESCRIPTION(KB, t)
  if p = NoPlan then
    G <- ASK (KB, MAKE-GOAL-QUERY(t))
    p <- IDEAL-PLANNER(current, G, KB)
  if p = NoPlan or p is empty then action <- NoOp
  else
    action <- FIRST(p)
    p <- REST(p)
  TELL(KB, MAKE-ACTION-SENTENCE(action,t))
  t <- t+1
  return action

```

Functions:

STATE-DESCRIPTION: uses a percept as input and returns the description of the initial state in a format required for the planner.

IDEAL-PLANNER: is the planning algorithm and can be any planner described in these notes or chapter 11.

MAKE-GOAL-QUERY: asks the knowledge base what the next goal will be.

The agent in the above algorithm has to check if the goal is feasible or if the complete plan is empty. If the goal is not feasible, it ignores it and tries another goal. If the complete plan was empty then the initial state was also the goal state.

Assumptions:

A simple planning agent create and use plans based on the following assumptions:

- **Atomic time:** each action is indivisible
- **No concurrent actions** allowed
- **Deterministic actions:** result of each actions is completely determined by the definition of the action, and there is no uncertainty in performing it in the world.
- **Agent is the sole cause of change** in the world.
- **Agent is omniscient:** has complete knowledge of the state of the world

- **Closed world assumption:** everything known to be true in the world is included in a state description. Anything not listed is false

9.1.3 Problem Solving vs. Planning

A simple planning agent is very similar to problem-solving agents in that it constructs plans that achieve its goals, and then executes them. The limitations of the problem-solving approach motivates the design of planning systems.

To solve a planning problem using a state-space search approach we would let the:

- initial state = initial situation
- goal-test predicate = goal state description
- successor function computed from the set of operators
- once a goal is found, solution plan is the sequence of operators in the path from the start node to the goal node

In searches, operators are used simply to generate successor states and we can not look "inside" an operator to see how it's defined. The goal-test predicate also is used as a "black box" to test if a state is a goal or not. The search cannot use properties of how a goal is defined in order to reason about finding path to that goal. *Hence this approach is all algorithm and representation weak.*

Planning is considered different from problem solving because of the difference in the way they represent states, goals, actions, and the differences in the way they construct action sequences.

Remember the search-based problem solver had four basic elements:

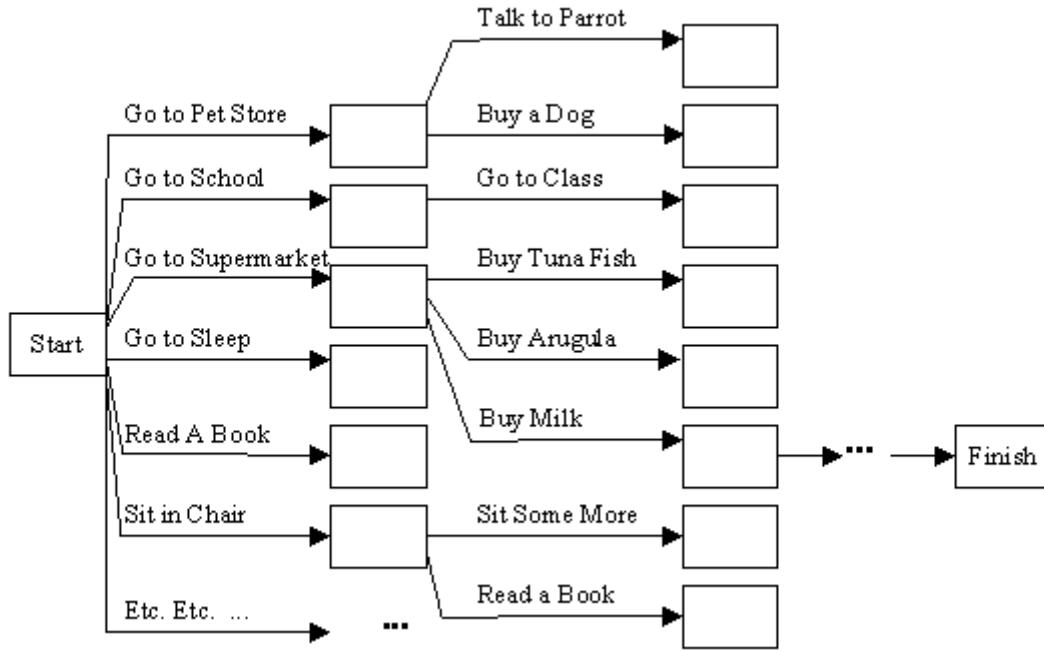
- Representations of actions: programs that develop successor state descriptions which represent actions.
- Representation of state: every state description is complete. This is because a complete description of the initial state is given, and actions are represented by a program that creates complete state descriptions.
- Representation of goals: a problem solving agent has only information about its goal, which is in terms of a goal test and the heuristic function.
- Representation of plans: in problem solving, the solution is a sequence of actions.

In a simple problem: "*Get a quart of milk and a bunch of bananas and a variable speed cordless drill*" for a problem solving exercise we need to specify:

Initial State: the agent is at home without any objects that he is wanting.

Operator Set: everything the agent can do.

Heuristic function: the # of things that have not yet been acquired.



Problems with Problem solving agent:

- It is evident from the above figure that the actual branching factor would be in the thousands or millions. The heuristic evaluation function can only choose states to determine which one is closer to the goal. It cannot eliminate actions from consideration. The agent makes guesses by considering actions and the evaluation function ranks those guesses. The agent picks the best guess, but then has no idea what to try next and therefore starts guessing again.
- It considers sequences of actions beginning from the initial state. The agent is forced to decide what to do in the initial state first, where possible choices are to go to any of the next places. Until the agent decides how to acquire the objects, it can't decide where to go.

Planning emphasizes what is in operator and goal representations. There are three key ideas behind planning:

- *to "open up" the representations* of state, goals, and operators so that a reasoner can more intelligently select actions when they are needed
- *the planner is free to add actions to the plan* wherever they are needed, rather than in an incremental sequence starting at the initial state
- *most parts of the world are independent of most other parts* which makes it feasible to take a conjunctive goal and solve it with a divide-and-conquer strategy

9.2 Logic Based Planning

9.2.1 Situation Calculus

Situation calculus is a version of first-order-logic (FOL) that is augmented so that it can reason about actions in time.

- Add *situation variables* to specify time. A **situation** is a snapshot of the world at an interval of time when nothing changes
- Add a special predicate *holds(f,s)* that means "f is true in situation s"
- Add a function *result(a,s)* that maps the current situation s into a new situation as a result of performing action a.

Example:

The action "*agent-walks-to-location-y*" could be represented by:

$$(Ax)(Ay)(As)(at(Agent,x,s) \rightarrow at(Agent,y,result(walk(y),s)))$$

9.2.1.1 Frame Problem

Actions in Situation Calculus describe what they *change*, not what they *don't change*. So, how do we know what's still true in the new situation? To fix this problem we add a set of frame axioms that explicitly state what doesn't change.

Example:

When the agent walks to a location, the locations of most other objects in the world do not change. So for each object (i.e. a bunch of bananas), add an axiom like:

$$(Ax)(Ay)(As)(at(Bananas,x,s) \rightarrow at(Bananas,x,result(walk(y),s)))$$

9.2.2 Solving planning problems using situation calculus

Using situation calculus a planning algorithm is represented by logical sentences that describe the three main parts of a problem. This is illustrated using the problem described in AI: A Modern Approach by Russell & Norvig.

1. Initial State: a logical sentence of a situation So.

For the shopping problem it could be:

$$At(Home,So) \wedge \neg Have(Milk,So) \wedge \neg Have(Bananas,So) \wedge \neg Have(Drill,So)$$

2. Goal State: a logical query asking for a suitable situation.

For the shopping problem, a query is:

$$\$ s At(Home,s) \wedge Have(Milk,s) \wedge Have(Bananas,s) \wedge Have(Drill,s)$$

3. Operators: a set of descriptions of actions.

Ex. A successor-state action with the Buy(Milk) action

$\exists a, s \text{ Have(Milk, Result}(a, s)) \Leftrightarrow [(a = \text{Buy(milk)} \wedge \text{At(supermarket}, s) \\ \vee (\text{Have(Milk}, s) \wedge a \neq \text{Drop(Milk)})]$

Result(a,s) names the situation resulting from being in s while executing action a.
 It will be useful to handle action sequences than just a single action. We can use Result'(l,s) to mean the situation resulting from executing l (the sequence of actions) beginning in s. Result' is described by saying that an empty sequence of actions will not effect the situation, and a non-empty sequence of actions is the same as doing the first action and then finishing the rest of the actions from the resulting situation.

Empty sequence of actions is represented as:

$\exists s \text{ Result}'([], s) = s$

Non-empty sequence of actions is represented as:

$\exists a, p, s \text{ Result}'([a|p], s) = \text{Result}'(p, \text{Result}(a, s))$

p is a plan and when applied to start state So, develops a situation which satisfies the goal query.

p could be:

$\text{At(home, Result}'(p, So)) \wedge \text{Have(Milk, Result}'(p, So)) \wedge \text{Have(Bananas, Result}'(p, So)) \wedge \text{Have(Drill, Result}'(p, So))$

The goal query could be:

$G = [\text{Go(Supermarket)}, \text{Buy(Milk)}, \text{Buy(Banana)}, \text{Go(Hardware Store)}, \text{Buy(Drill)}, \text{Go(Home)}]$

In theory, there isn't much more to say. Doing planning this way doesn't guarantee a practical solution. Plan p guarantees to achieve the goal but not necessarily efficiently. This approach is representation strong, but reasoning weak because of the poor efficiency of resolution.

Practical Planning:

To make planning practical we need to do 2 things:

1. Restrict the language with which we define problems. With a restrictive language, there are fewer possible solutions to search through.

Use a special-purpose algorithm called a **planner** rather than a general-purpose theorem prover to search for a solution.

Module 9

Planning

Lesson 23

Planning systems

9.3 Planning Systems

Classical Planners use the STRIPS (Stanford Research Institute Problem Solver) language to describe states and operators. It is an efficient way to represent planning algorithms.

9.3.1 Representation of States and Goals

States are represented by conjunctions of function-free ground literals, that is, predicates applied to constant symbols, possibly negated.

An example of an initial state is:

$$At(Home) \wedge \neg Have(Milk) \wedge \neg Have(Bananas) \wedge \neg Have(Drill) \wedge \dots$$

A state description does not have to be complete. We just want to obtain a successful plan to a set of possible complete states. But if it does not mention a given positive literal, then the literal can be assumed to be false.

Goals are a conjunction of literals. Therefore the goal is

$$At(Home) \wedge Have(Milk) \wedge Have(Bananas) \wedge Have(Drill)$$

Goals can also contain variables. Being at a store that sells milk is equivalent to

$$At(x) \wedge Sells(x, Milk)$$

We have to differentiate between a goal given to a planner which is producing a sequence of actions that makes the goal true if executed, and a query given to a theorem prover that produces true or false if there is truth in the sentences, given a knowledge base.

We also have to keep track of the changes rather than of the states themselves because most actions change only a small part of the state representation.

9.3.2 Representation of Actions

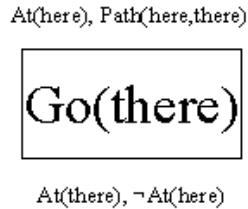
Strips operators consist of three components

- action description: what an agent actually returns to the environment in order to do something.
- precondition: conjunction of atoms (positive literals), that says what must be true before an operator can be applied.
- effect of an operator: conjunction of literals (positive or negative) that describe how the situation changes when the operator is applied.

An example action of going from one place to another:

*Op(ACTION:Go(there), PRECOND:At(here) \wedge Path(here, there)
EFFECT:At(there) \wedge -At(here))*

The following figure shows a diagram of the operator Go(there). The preconditions appear above the action, and the effects below.



Operator Schema: an operator with variables.

- it is a family of actions, one for each of the different values of the variables.
- every variable must have a value

Preconditions and Effects are restrictive. Operator o is applicable in a state s if every one of the preconditions in o are true in s. An example is if the initial situation includes the literals

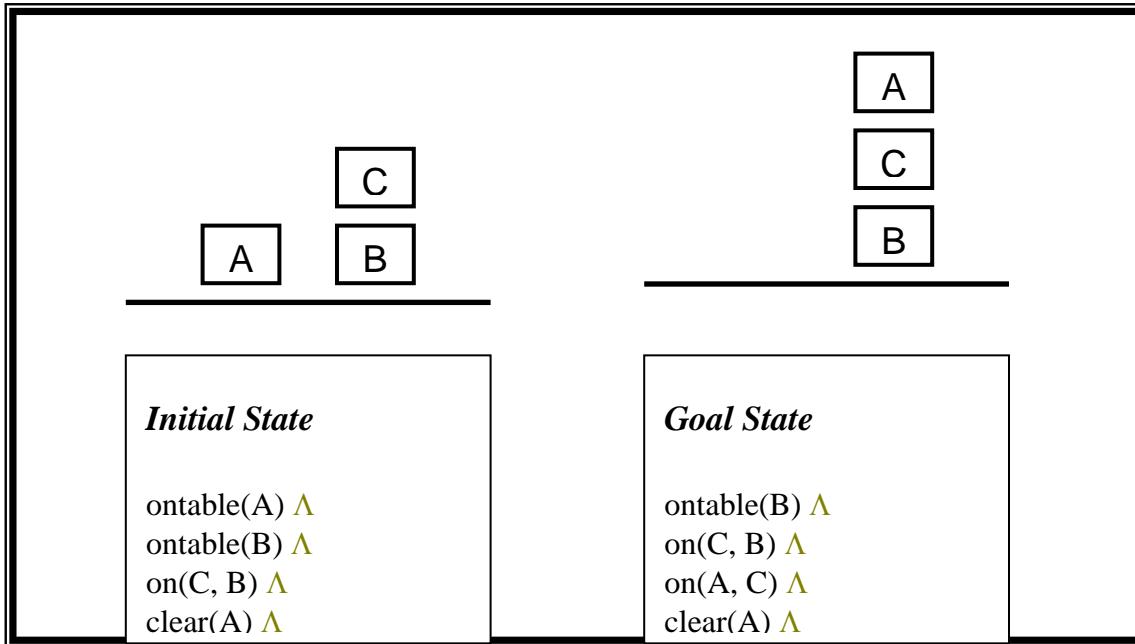
At(Home, Path(Home, Supermarket))...

then the action Go(Supermarket) is applicable, and the resulting situation contains the literals

-At(Home), At(Supermarket), Path(Home, Supermarket)...

The result is all positive literals in Effect(o) hold, all literals in s hold and negative literals in Effect(o) are ignored.

The set of operators for the “Box World” example problem is shown below:



Definitions of Descriptors:

ontable(x): block x is on top of the table

on(x,y): block x is on top of block y

clear(x): there is nothing on top of block x ; therefore it can be picked up

handempty: you are not holding any block

Definitions of Operators:

Op{ACTION: **pickup(x)**

PRECOND: ontable(x), clear(x), handempty

EFFECT: holding(x), ~ontable(x), ~clear(x), ~handempty }

Op{ACTION: **putdown(x)**

PRECOND: holding(x)

EFFECT: ontable(x), clear(x), handempty, ~holding(x) }

Op{ACTION: **stack(x,y)**

PRECOND: holding(x), clear(y)

EFFECT: on(x,y), clear(x), handempty, ~holding(x), ~clear(y) }

Op{ACTION: **unstack(x,y)**

PRECOND: clear(x), on(x,y), handempty

EFFECT: holding(x), clear(y), ~clear(x), ~on(x,y), ~handempty) }

Module 9

Planning

Lesson 24

Planning algorithm - I

9.4 Planning as Search

Planning as Search:

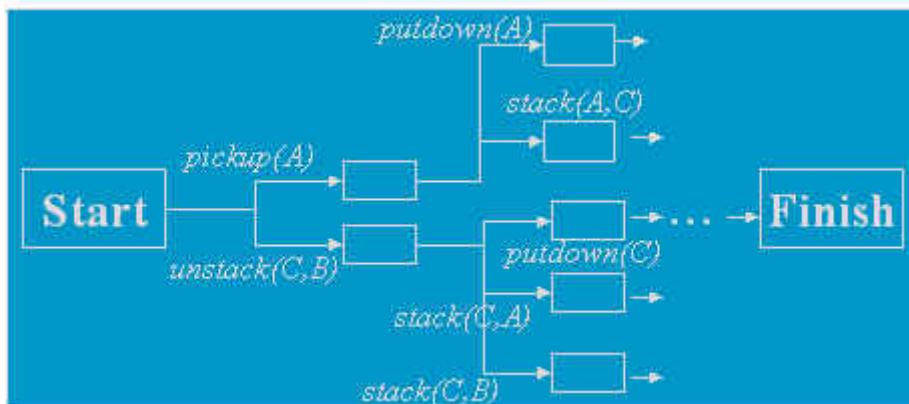
There are two main approaches to solving planning problems, depending on the kind of search space that is explored:

1. Situation-space search
2. Planning-space search

9.4.1 Situation-Space Search

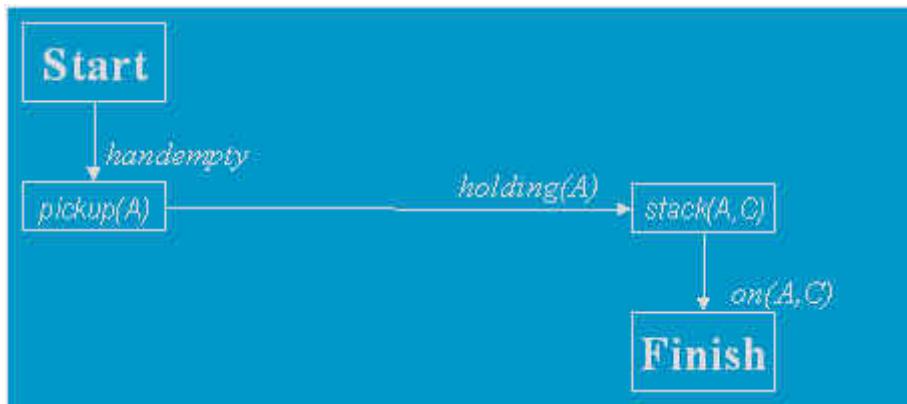
In situation space search

- the search space is the space of all possible states or situations of the world
- initial state defines one node
- a goal node is a state where all goals in the goal state are satisfied
- a solution plan is the sequence of actions (e.g. operator instances) in the path from the start node to a goal node



9.4.2 Plan-Space Search

- the search space is the space of all possible plans
- a node corresponds to a partial plan
- initially we will specify an "initial plan" which is one node in this space
- a goal node is a node containing a plan which is complete, satisfying all of the goals in the goal state
- the node itself contains all of the information for determining a solution plan (e.g. sequence of actions)



9.4.3 Situation-Space Planning Algorithms

There are 2 approaches to situation-space planning:

1. Progression situation-space planning
2. Regression situation-space planning

Progression Planning:

- *Forward-chaining* from initial state to goal state
- Looks just like a state-space search except STRIPS operators are specified instead of a set of next-move functions
- You can use any search method you like (i.e. BFS, DFS, A*)
- **Disadvantage:** huge search space to explore, so usually very inefficient

Algorithm:

1. Start from initial state
2. Find all operators whose preconditions are true in the initial state
3. Compute effects of operators to generate successor states
4. Repeat steps #2-#3 until a new state satisfies the goal conditions

The work through of the progression algorithm for the Blocks World example is shown below:

Step	State	Applicable Operators	Operator Applied
#1	ontable(A) \wedge ontable(B) \wedge $\text{on}(C, B)$ \wedge $\text{clear}(A)$ \wedge $\text{clear}(C)$ \wedge handempty	pickup(A) $\text{unstack}(C, B)$	pickup(A)
#2	$\sim\text{ontable}(A)$ \wedge	$\text{putdown}(A)$ $\text{stack}(A, C)$	$\text{stack}(A, C)$

	ontable(B) \wedge on(C, B) \wedge \sim clear(A) \wedge clear(C) \wedge \sim handempty \wedge holding(A)		
#3	ontable(B) \wedge on(C, B) \wedge on(A, C) \wedge clear(A) \wedge \sim clear(C) \wedge handempty \wedge \sim holding(A)	Matches goal state so STOP!	

Regression Planning

- *Backward-chaining* from goal state to initial state
- Regression situation-space planning is usually more efficient than progression because many operators are applicable at each state, yet only a small number of operators are applicable for achieving a given goal
- Hence, regression is more goal-directed than progression situation-space planning
- **Disadvantage:** cannot always find a plan even if one exists!

Algorithm:

1. Start with goal node corresponding to goal to be achieved
2. Choose an operator that will *add* one of the goals
3. Replace that goal with the operator's preconditions
4. Repeat steps #2-#3 until you have reached the initial state
- 5.

While backward-chaining is performed by STRIPS in terms of the generation of goals, sub-goals, sub-sub-goals, etc., operators are used in the forward direction to generate successor states, starting from the initial state, until a goal is found.

The work through of the regression algorithm for the Blocks World example is shown below.

Step	State	Stack	Plan	Note
#1	ontable(A) \wedge ontable(B)	achieve(on(A,C))		Stack contains original goal. State contains the initial state

	Λ on(C, B) Λ clear(A) Λ clear(C) Λ handempty			description.
#2	Same.	achieve(clear(C), holding(A), apply(Stack(A,C))) achieve(on(A,C))		Choose operator <i>Stack</i> to solve goal popped from top of goal stack.
#3	Same.	achieve(holding(A)) achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C))) achieve(on(A,C))		Order sub-goals arbitrarily.
#4	Same.	achieve(ontable(A), clear(A), handempty), apply(pickup(A)) achieve(holding(A)) achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C))) achieve(on(A,C))		Choose operator <i>pickup</i> to solve goal popped from top of goal stack.
#5	ontable(B) Λ on(C, B) Λ clear(C) Λ holding(A)	achieve(clear(C)) achieve(clear(C), holding(A), apply(Stack(A,C))) achieve(on(A,C))	Pickup(A)	Top goal is true in current state, so pop it and apply operator <i>pickup(A)</i> .
#6	ontable(B) Λ on(C, B) Λ on(A,C) Λ clear(A) Λ handempty	achieve(on(A,C))	pickup(A) stack(A,C)	Top goal <i>achieve(C)</i> true so pop it. Re-verify that goals that are the preconditions of the <i>stack(A,C)</i> operator still true, then pop that and the operator is applied.
#7	Same.	<empty>		Re-verify that original goal is true in current state, then pop and halt with empty goal stack and state description satisfying original goal.

Goal Interaction

Most planning algorithms assume that the goals to be achieved are independent or nearly independent in the sense that each can be solved separately and then the solutions concatenated together. If the order of solving a set of goals (either the original goals or a set of sub-goals which are the preconditions of an operator) fails because solving a latter goal undoes an earlier goal, then this version of the STRIPS algorithm **fails**. Hence, situation-space planners do not allow for interleaving of steps in any solution it finds.

Principle of Least Commitment

The principle of least commitment is the idea of never making a choice unless required to do so. The advantage of using this principle is you won't have to backtrack later!

In planning, one application of this principle is to never order plan steps unless it's necessary for some reason. So, partial-order planners exhibit this property because constraint ordering steps will only be inserted when necessary. On the other hand, situation-space progression planners make commitments about the order of steps as they try to find a solution and therefore may make mistakes from poor guesses about the right order of steps.

Module 9

Planning

Lesson 25

Planning algorithm - II

9.4.5 Partial-Order Planning

Total-Order vs. Partial-Order Planners

Any planner that maintains a partial solution as a totally ordered list of steps found so far is called a **total-order planner**, or a **linear planner**. Alternatively, if we only represent partial-order constraints on steps, then we have a **partial-order planner**, which is also called a **non-linear planner**. In this case, we specify a set of temporal constraints between pairs of steps of the form $s_1 < s_2$ meaning that step S_1 comes before, but not necessarily immediately before, step S_2 . We also show this temporal constraint in graph form as

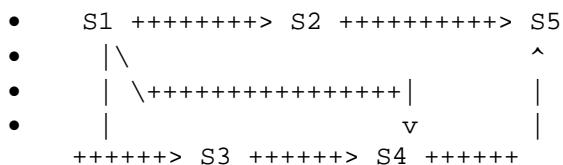
$S_1 ++++++> S_2$

STRIPS is a total-order planner, as are situation-space progression and regression planners

Partial-order planners exhibit the property of least commitment because constraints ordering steps will only be inserted when necessary. On the other hand, situation-space progression planners make commitments about the order of steps as they try to find a solution and therefore may make mistakes from poor guesses about the right order of steps.

Representing a Partial-Order Plan

A partial-order plan will be represented as a graph that describes the temporal constraints between plan steps selected so far. That is, each node will represent a single step in the plan (i.e., an instance of one of the operators), and an arc will designate a temporal constraint between the two steps connected by the arc. For example,



graphically represents the temporal constraints $S_1 < S_2$, $S_1 < S_3$, $S_1 < S_4$, $S_2 < S_5$, $S_3 < S_4$, and $S_4 < S_5$. This partial-order plan implicitly represents the following three total-order plans, each of which is consistent with all of the given constraints:

$[S_1, S_2, S_3, S_4, S_5]$, $[S_1, S_3, S_2, S_4, S_5]$, and $[S_1, S_3, S_4, S_2, S_5]$.

9.5 Plan-Space Planning Algorithms

An alternative is to search through the space of *plans* rather than a space of *situations*. That is, we start with a simple, incomplete plan, which we call a **partial plan**. Then we consider ways of expanding the partial plan until we come up with a complete plan that

solves the problem. We use this approach when the ordering of sub-goals affects the solution.

Here one starts with a simple, incomplete plan, a partial plan, and we look at ways of expanding the partial plan until we come up with a complete plan that solves the problem. The operators for this search are operators on plans: adding a step, imposing an ordering that puts one step before another, instantiating a previously unbound variable, and so on. Therefore the solution is the final plan.

Two types of operators are used:

- Refinement operators take a partial plan and add constraints to it. They eliminate some plans from the set and they never add new plans to it.
- A modification operator debugs incorrect plans that the planner may make, therefore we can worry about bugs later.

9.5.1 Representation of Plans

A **plan** is formally defined as a data structure consisting of the following 4 components:

1. A set of plan steps
2. A set of step ordering constraints
3. A set of variable binding constraints
4. A set of causal links

Example:

Plan(

 STEPS:{S₁:*Op(ACTION: Start,*
 S₂:*Op(ACTION: Finish,*
 PRECOND: *OnTable(c), On(b,c), On(a,b) },*
 ORDERINGS: {S₁ < S₂},
 BINDINGS: {},
 LINKS: {})

Key Difference Between Plan-Space Planning and Situation-Space Planning
In Situation-Space planners all operations, all variables, and all orderings must be fixed when each operator is applied. Plan-Space planners make commitments (i.e., what steps in what order) only as necessary. Hence, Plan-Space planners do least-commitment planning.

Start Node in Plan Space

The initial plan is created from the initial state description and the goal description by creating two "pseudo-steps:"

Start

```
P: none  
E: all positive literals defining the initial state
```

Finish

```
P: literals defining the conjunctive goal to be achieved  
E: none
```

and then creating the initial plan as: Start -----> Finish

Searching Through Plan Space

There are two main reasons why a given plan may not be a solution:

Unsatisfied goal. That is, there is a goal or sub-goal that is not satisfied by the current plan steps.

Possible threat caused by a plan step that could cause the undoing of a needed goal if that step is done at the wrong time

So, define a set of **plan modification operators** that detect and fix these problems.

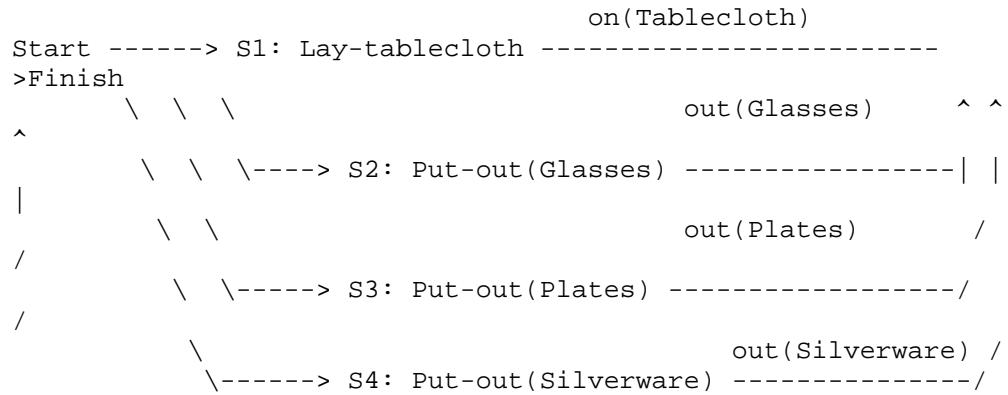
Example

- Goal: Set the table, i.e., `on(Tablecloth) ^ out(Glasses) ^ out(Plates) ^ out(Silverware)`
- Initial State: `clear(Table)`
- Operators:
 1. Lay-tablecloth
 2. `P: clear(Table)
E: on(Tablecloth), ~clear(Table)`
 3. Put-out(*x*)
 4. `P: none
E: out(x), ~clear(Table)`
- Searching for a Solution in Plan Space

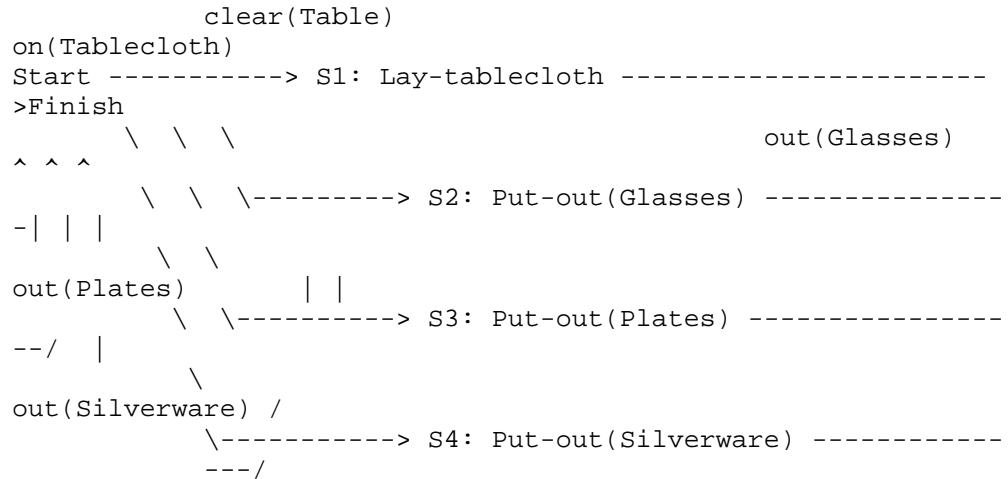
1. Initial Plan

Start -----> Finish

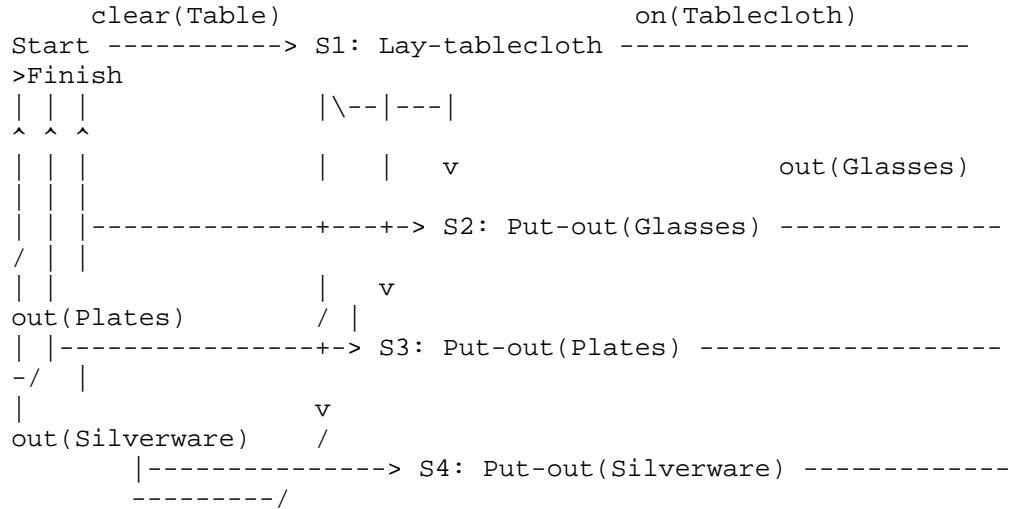
2. Solve 4 unsolved goals in Finish by adding 4 new steps with the minimal temporal constraints possible:



3. Solve unsolved subgoal `clear(Table)` which is a precondition of step `S1`:



4. Fix threats caused by steps `S2`, `S3`, and `S4` on the link from `Start` to `S1`. That is, `clear(Table)` is a necessary precondition of `S1` that is created by step `Start`. But `S2` causes `clear(Table)` to be deleted (negated), so if `S2` came *before* `S1`, `clear(Table)` wouldn't be true and step `S1` couldn't be performed. Therefore, add a temporal constraint that forces `S2` to come *anytime after* `S1`. That is, add constraint `S1 < S2`. Similarly, add `S1 < S3`, and `S1 < S4`, resulting in the new plan:



5. No threats and no unsolved goals in this plan, so it is a complete plan (i.e., a solution to the planning problem). Any total ordering of the steps implied by this partial-order plan is a solution plan. Here, there are six possible plans, where the first step is S1, and the steps S2, S3, and S4 follow in any order. (Don't include the pseudo-steps Start and Finish.)

Interleaving vs. Non-Interleaving of Sub-Plan Steps

Given a conjunctive goal, $G_1 \wedge G_2$, if the steps that solve G_1 must either all come before or all come after the steps that solve G_2 , then the planner is called a **non-interleaving planner**. Otherwise, the planner allows interleaving of sub-plan steps. This constraint is different from the issue of partial-order vs. total-order planners. STRIPS is a non-interleaving planner because of its use of a stack to order goals to be achieved.

Partial-Order Planner (POP) Algorithm

```

function pop(initial-state, conjunctive-goal, operators)
  // non-deterministic algorithm
  plan = make-initial-plan(initial-state, conjunctive-goal);
loop:
  begin
    if solution?(plan) then return plan;
    ( $S\text{-need}$ ,  $c$ ) = select-subgoal(plan); // choose an unsolved goal
    choose-operator(plan, operators,  $S\text{-need}$ ,  $c$ );
      // select an operator to solve that goal and revise plan
    resolve-threats(plan); // fix any threats created
  end
end

function solution?(plan)

```

```

if causal-links-establishing-all-preconditions-of-all-steps(plan)
    and all-threats-resolved(plan)
    and all-temporal-ordering-constraints-consistent(plan)
    and all-variable-bindings-consistent(plan)
then return true;
else return false;
end

function select-subgoal(plan)
    pick a plan step  $S\text{-need}$  from steps(plan) with a precondition  $c$ 
        that has not been achieved;
    return ( $S\text{-need}$ ,  $c$ );
end

procedure choose-operator(plan, operators,  $S\text{-need}$ ,  $c$ )
    // solve "open precondition" of some step
    choose a step  $S\text{-add}$  by either
        Step Addition: adding a new step from operators that
            has  $c$  in its Add-list
        or Simple Establishment: picking an existing step in Steps(plan)
            that has  $c$  in its Add-list;
    if no such step then return fail;
    add causal link " $S\text{-add} \rightarrow c S\text{-need}$ " to Links(plan);
    add temporal ordering constraint " $S\text{-add} < S\text{-need}$ " to Orderings(plan);
    if  $S\text{-add}$  is a newly added step then
        begin
            add  $S\text{-add}$  to Steps(plan);
            add " $Start < S\text{-add}$ " and " $S\text{-add} < Finish$ " to Orderings(plan);
        end
    end

procedure resolve-threats(plan)
    foreach  $S\text{-threat}$  that threatens link " $Si \rightarrow c Sj$ " in Links(plan)
        begin // "declobber" threat
            choose either
                Demotion: add " $S\text{-threat} < Si$ " to Orderings(plan)
                or Promotion: add " $Sj < S\text{-threat}$ " to Orderings(plan);
            if not(consistent(plan)) then return fail;
        end
    end

```

Plan Modification Operations

The above algorithm uses four basic plan modification operations to revise a plan, two for solving a goal and two for fixing a threat:

- **Establishment -- "Solve an Open Precondition" (i.e., unsolved goal)**
If a precondition p of a step s does not have a causal link to it, then it is not yet solved. This is called an **open precondition**. Two ways to solve:
 - **Simple Establishment**
Find an existing step t prior to s in which p is necessarily true (i.e., it's in the Effects list of t). Then add causal link from t to s .

- **Step Addition**
Add a new plan step τ that contains in its Effects list p . Then add causal link from τ to s .
- **Declobbering -- Threat Removal**
A threat is a relationship between a step s_3 and a causal link $s_1 \dashrightarrow_p s_2$, where p is a precondition in step s_2 , that has the following form:
 - $\dashrightarrow \quad s_1 \dashrightarrow_p s_2$
 - $\quad |$
 - $\quad |$
 - $\dashrightarrow \quad s_3 \sim p$

That is, step s_3 has effect $\sim p$ and from the temporal links could possibly occur in-between steps s_1 and s_2 , which have a causal link between them. If this occurred, then s_3 would "clobber" the goal p "produced" by s_1 before it can be "consumed" by s_2 . Fix by ensuring that s_3 cannot occur in the "protection interval" in between s_1 and s_2 by doing either of the following:

- **Promotion**
Force threatening step to come *after* the causal link. I.e., add temporal link $S_2 < S_3$.
- Demotion**
Force threatening step to come *before* the causal link. I.e., add temporal link $S_3 < S_1$.

9.5.2 Simple Sock/Shoe Example

In the following example, we will show how the planning algorithm derives a solution to a problem that involves putting on a pair of shoes. In this problem scenario, Pat is walking around his house in his bare feet. He wants to put some shoes on to go outside.
Note: There are no threats in this example and therefore is no mention of checking for threats though it a necessary step

To correctly represent this problem, we must break down the problem into simpler, more atomic states that the planner can recognize and work with. We first define the Start operator and the Finish operator to create the minimal partial order plan. As mentioned before, we must simplify and break down the situation into smaller, more appropriate states. The Start operator is represented by the effects: $\sim \text{LeftSockOn}$, $\sim \text{LeftShoeOn}$, $\sim \text{RightSockOn}$, and $\sim \text{RightShoeOn}$. The Finish operator has the preconditions that we wish to meet: LeftShoeOn and RightShoeOn . Before we derive a plan that will allow Pat to reach his goal (i.e. satisfying the condition of having both his left and right shoe on) from the initial state of having nothing on, we need to define some operators to get us

there. Here are the operators that we will use and the possible state (already mentioned above).

Operators

Op(ACTION: PutLeftSockOn() PRECOND: \sim LeftSockOn EFFECT: LeftSockOn)
Op(ACTION: PutRightSockOn() PRECOND: \sim RightSockOn EFFECT: RightSockOn)
Op(ACTION: PutLeftShoeOn() PRECOND:LeftSockOn EFFECT: LeftShoeOn)
Op(ACTION: PutRightShoeOn() PRECOND:RightShoeOn EFFECT: RightShoeOn)

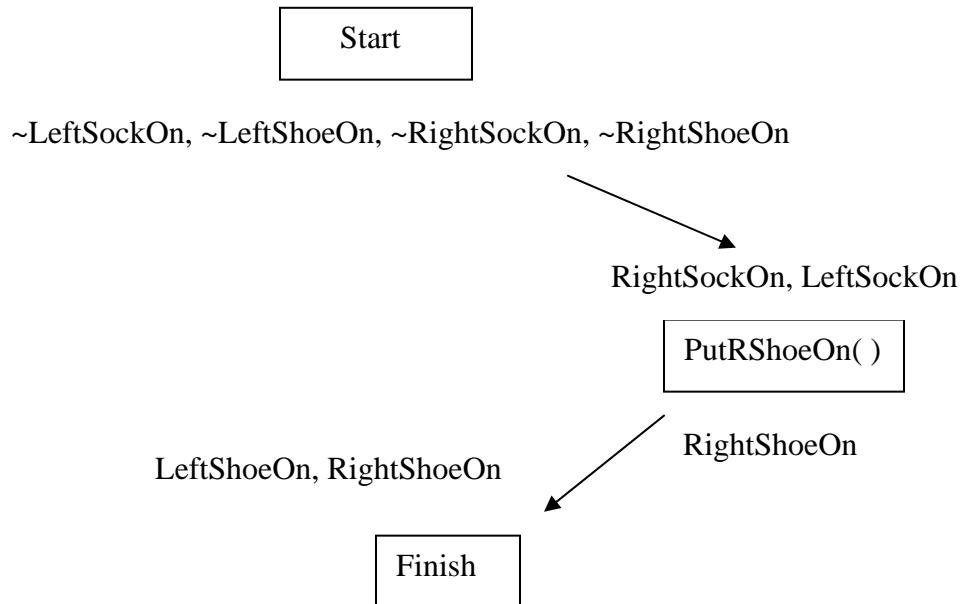
States

LeftSockOn, LeftShoeOn, RightSockOn, RightShoeOn

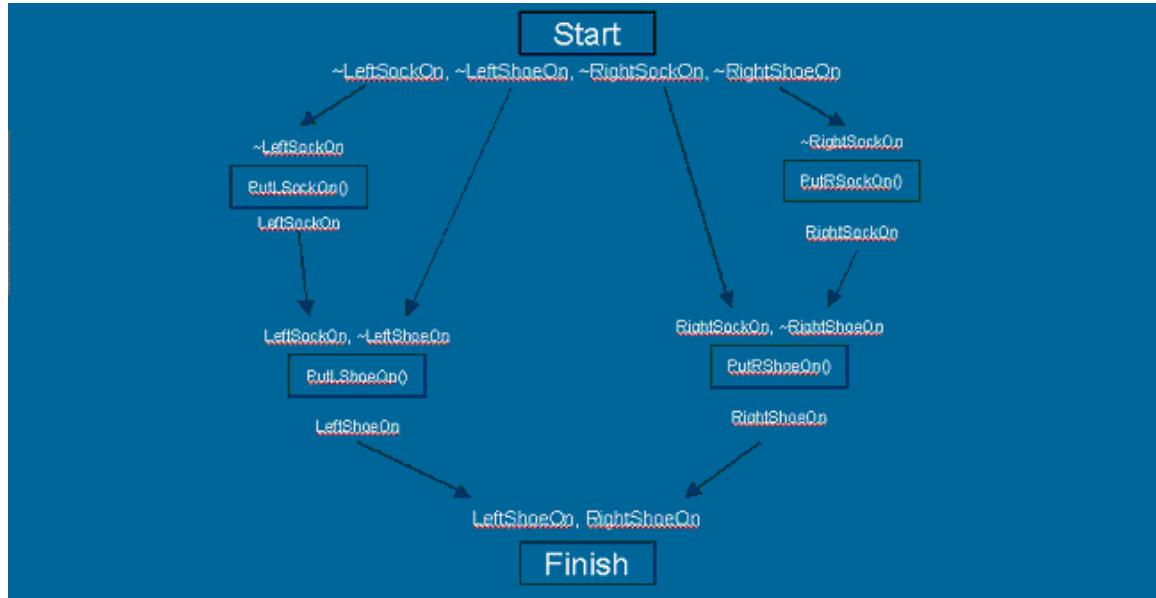
Creating A Plan

From the states listed above, we first create a minimal partial order plan. We can represent bare feet (Start operator) by saying that Pat is not wearing any socks or shoes and shoes on (Finish operator) with the two shoe on states. Here is the minimal partial order plan.

Initially we have two preconditions to achieve; RightShoeOn and LeftShoeOn. Let's start with the condition of having our right shoe on. We must choose an operator that will result in this condition. To meet this condition we need to the operator 'PutRightShoeOn()'. We add the operator and create a causal link between it and the Finish operator. However, adding this operator results a new condition (i.e. precondition of PutRightShoeOn()) of having the right sock on.



At this point we still have two conditions to meet: having our left shoe on and having our right sock on. We continue by selecting one of these two preconditions and trying to achieve it. Let's pick the precondition of having our right sock on. To satisfy this condition, we must add another step, operator 'PutRightSockOn()'. The effects of this operator will satisfy the precondition of having our right sock on. At this point, we have achieved the 'RightSockOn' state. Since the precondition of the 'PutRightSockOn()' operator is one of the effects of the Start operator, we can simply draw a causal link between the two operators. These two steps can be repeated for Pat's left shoe. The plan is complete when all preconditions are resolved.

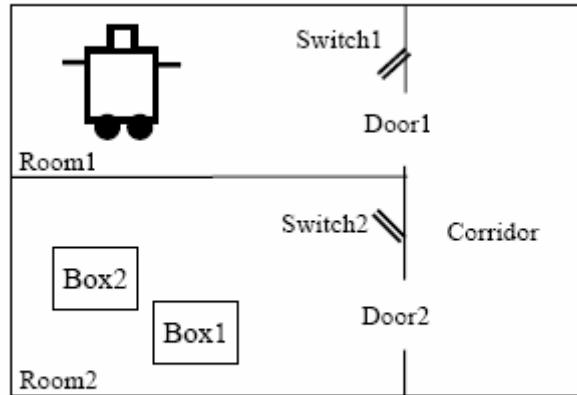


The Partial Order Planning algorithm can be described as a form of regression planning that uses the principle of least commitment. It starts with a minimal partial order plan that consists of a Start operator (initial state) and a Finish operator (goal state). It then chooses a precondition that has not been resolved and chooses an operator whose effect matches the precondition. It then checks if any threats were created by the addition of the operator and if one is detected, resolves it either by demoting the operator, promoting the operator, or backtracking (removing the operator). It continues to choose operators until a solution is found (i.e. all preconditions are resolved).

Solutions created by the Partial Order Planning algorithm are very flexible. They may be executed in many ways. They can represent many different total order plans (partial order plans can be converted to total order plans using a process called linearization). Lastly they can more efficiently if steps are executed simultaneously.

Questions

1. Consider the world of Shakey the robot, as shown below.



NOTE: The intended interpretation of the switches drawn is that Switch1 is in the off position and Switch2 is in the on position.

Shakey has the following six actions available:

- $Go(x,y)$, which moves Shakey from x to y . It requires Shakey to be at x and that x and y are locations in the same room. By convention a door between two rooms is in both of them, and the corridor counts as a room.
- $Push(b,x,y)$, which allows Shakey to push a box b from location x to location y . Both Shakey and the box must be at the same location before this action can be used.
- $ClimbUp(b,x)$, which allows Shakey to climb onto box b at location x . Both Shakey and the box must be at the same location before this action can be used. Also Shakey must be on the *Floor*.
- $ClimbDown(b,x)$, which allows Shakey to climb down from a box b at location x . Shakey must be on the box and the box must be in location x before this action can be used.
- $TurnOn(s,x)$, which allows Shakey to turn on switch s which is located at location x . Shakey must be on top of a box at the switch's location before this action can be used.
- $TurnOff(s,x)$, which allows Shakey to turn off switch s which is located at location x . Shakey must be on top of a box at the switch's location before this action can be used.

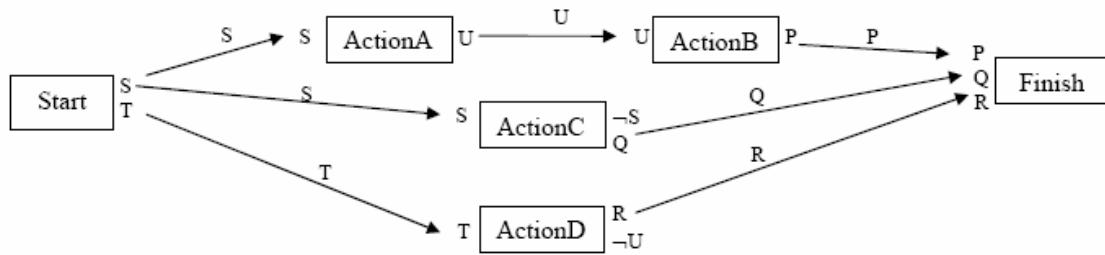
Using STRIPS syntax, define the six actions from above. In your action definitions, use only the following predicates: $Box(b)$ to mean that b is a box, $In(x,r)$ to mean that location x is in room r , $At(x,y)$ to mean that the object x is at location y , $ShakeyOn(x)$ to mean that Shakey is on the object x , $Switch(s)$ to mean that s is a switch, and $SwitchOn(s)$ to mean

that the switch s is on. You may also use the constants *Shakey* and *Floor* in the action definitions.

2. In the above problem, using STRIPS, define the initial state depicted on the previous page. Use only the predicates from part (a) and the constants *Box1*, *Box2*, *Switch1*, *Switch2*, *Floor*, *Shakey*, *Room1*, *Room2*, *Corridor*, *LDoor1*, *LDoor2*, *LShakeyStart*, *LSwitch1*, *LBox1Start*, *LBox2Start*, *LSwitch2*. The L_x constants are intended to represent the locations of x ,

3. Provide a totally ordered plan for Shakey to turn off *Switch2* using the actions and the initial state defined in (2) and (3).

4. Consider the inconsistent partially ordered plan below. Identify the conflicts in this plan and show all ways of resolving them that follow the principle of least commitment. For each solution, draw the new partially ordered plan, and list all of its linearizations.



Solutions

1. STRIPS description of the operators are:

```

Action( Go(x, y),
  Pre: At(Shakey, x) ∧ In(x, r) ∧ In(y, r) ∧ ShakeyOn(Floor)
  Eff: At(Shakey, y) ∧ ¬At(Shakey, x) )
Action( Push(b, x, y),
  Pre: At(Shakey, x) ∧ At(b, x) ∧ Box(b) ∧ ShakeyOn(Floor) ∧ In(x, r) ∧ In(y, r)
  Eff: At(Shakey, y) ∧ At(b, y) ∧ ¬At(Shakey, x) ∧ ¬At(b, x) )
Action( ClimbUp(b, x),
  Pre: At(Shakey, x) ∧ At(b, x) ∧ Box(b) ∧ ShakeyOn(Floor)
  Eff: ¬ShakeyOn(Floor) ∧ ShakeyOn(b) )
Action( ClimbDown(b, x),
  Pre: ShakeyOn(b) ∧ Box(b) ∧ At(b, x) ∧ At(Shakey, x)
  Eff: ¬ShakeyOn(b) ∧ ShakeyOn(Floor) )
Action( TurnOn(s, x),
  Pre: At(Shakey, x) ∧ At(s, x) ∧ Switch(s) ∧ Box(b) ∧ At(b,x) ∧ ShakeyOn(b)
  Eff: SwitchOn(s) )
Action( TurnOff(s, x),
  Pre: At(Shakey, x) ∧ At(s, x) ∧ Switch(s) ∧ Box(b) ∧ At(b,x) ∧ ShakeyOn(b) ∧ SwitchOn(s)
  Eff: ¬SwitchOn(s) )
  
```

2. The initial state is:

$\text{Box}(\text{Box1}) \wedge \text{Box}(\text{Box2}) \wedge \text{Switch}(\text{Switch1}) \wedge \text{Switch}(\text{Switch2}) \wedge$
 $\text{SwitchOn}(\text{Switch2}) \wedge \text{ShakeyOn}(\text{Floor}) \wedge \text{In}(\text{L}_{\text{ShakeyStart}}, \text{Room1}) \wedge \text{In}(\text{L}_{\text{Door1}}, \text{Room1}) \wedge$
 $\text{In}(\text{L}_{\text{Door1}}, \text{Corridor}) \wedge \text{In}(\text{L}_{\text{Door2}}, \text{Room2}) \wedge \text{In}(\text{L}_{\text{Door2}}, \text{Corridor}) \wedge \text{In}(\text{L}_{\text{Switch1}}, \text{Room1}) \wedge$
 $\text{In}(\text{L}_{\text{Box1Start}}, \text{Room2}) \wedge \text{In}(\text{L}_{\text{Box2Start}}, \text{Room2}) \wedge \text{In}(\text{L}_{\text{Switch2}}, \text{Room2}) \wedge \text{At}(\text{Shakey}, \text{L}_{\text{ShakeyStart}}) \wedge$
 $\text{At}(\text{Box1}, \text{L}_{\text{Box1Start}}) \wedge \text{At}(\text{Box2}, \text{L}_{\text{Box2Start}}) \wedge \text{At}(\text{Switch1}, \text{L}_{\text{Switch1}}) \wedge \text{At}(\text{Switch2}, \text{L}_{\text{Switch2}})$

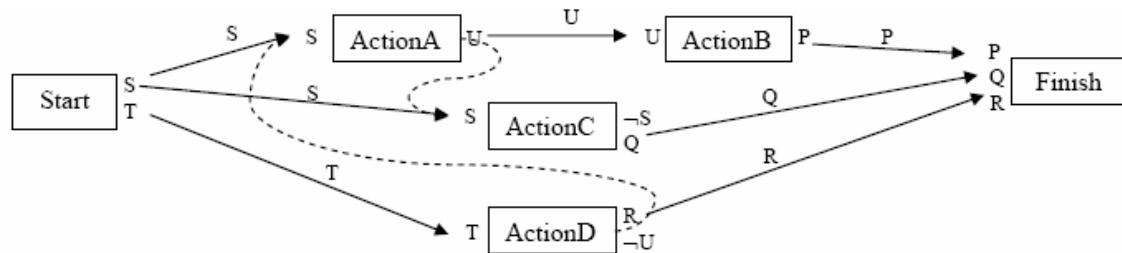
3. The plan is

$\text{Go}(\text{L}_{\text{ShakeyStart}}, \text{L}_{\text{Door1}})$
 $\text{Go}(\text{L}_{\text{Door1}}, \text{L}_{\text{Door2}})$
 $\text{Go}(\text{L}_{\text{Door2}}, \text{L}_{\text{Box1Start}})$
 $\text{Push}(\text{Box1}, \text{L}_{\text{Box1Start}}, \text{L}_{\text{Switch2}})$
 $\text{ClimbUp}(\text{Box1}, \text{L}_{\text{Switch2}})$
 $\text{TurnOff}(\text{Switch2}, \text{L}_{\text{Switch2}})$

4. Conflicts:

- ActionC cannot come between Start and ActionA.
- ActionD cannot come between ActionA and ActionB.

Resolution 1:

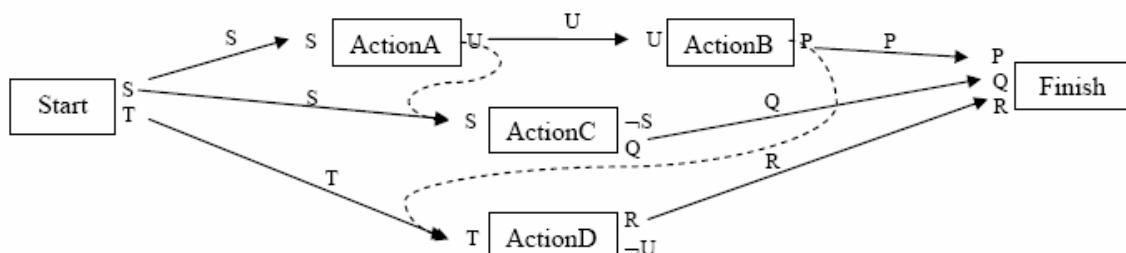


Linearizations of Resolution 1: (ActionA abbreviated as A, and so on...)

Start, D, A, B, C, Finish

Start, D, A, C, B, Finish

Resolution 2:



Linearizations of Resolution 2: (ActionA abbreviated as A, and so on...)

Start, A, B, D, C, Finish

Start, A, B, C, D, Finish

Start, A, C, B, D, Finish

Module 10

Reasoning with Uncertainty - Probabilistic reasoning

10.1 Instructional Objective

- The students should understand the role of uncertainty in knowledge representation
- Students should learn the use of probability theory to represent uncertainty
- Students should understand the basic of probability theory, including
 - Probability distributions
 - Joint probability
 - Marginal probability
 - Conditional probability
 - Independence
 - Conditional independence
- Should learn inference mechanisms in probability theory including
 - Bayes rule
 - Product rule
- Should be able to convert natural language statements into probabilistic statements and apply inference rules
- Students should understand Bayesian networks as a data structure to represent conditional independence
- Should understand the syntax and semantics of Bayes net
- Should understand inferencing mechanisms in Bayes net
- Should understand efficient inferencing techniques like variable ordering
- Should understand the concept of d-separation
- Should understand inference mechanism for the special case of polytrees
- Students should have idea about approximate inference techniques in Bayesian networks

At the end of this lesson the student should be able to do the following:

- **Represent a problem in terms of probabilistic statement**
- Apply Bayes rule and product rule for inferencing
- **Represent a problem using Bayes net**
- **Perform probabilistic inferencing using Bayes net.**

Lesson 26

Reasoning with Uncertain information

10.2 Probabilistic Reasoning

Using logic to represent and reason we can represent knowledge about the world with facts and rules, like the following ones:

bird(tweety).
fly(X) :- bird(X).

We can also use a theorem-prover to reason about the world and deduct new facts about the world, for e.g.,

?- fly(tweety).

Yes

However, this often does not work outside of toy domains - non-tautologous certain rules are hard to find.

A way to handle knowledge representation in real problems is to extend logic by using certainty factors.

In other words, replace
IF condition THEN fact
with
IF condition with certainty x THEN fact with certainty $f(x)$

Unfortunately cannot really adapt logical inference to probabilistic inference, since the latter is not context-free.

Replacing rules with conditional probabilities makes inferencing simpler.

Replace
smoking \rightarrow lung cancer
or
lotsofconditions, smoking \rightarrow lung cancer
with
 $P(\text{lung cancer} | \text{smoking}) = 0.6$

Uncertainty is represented explicitly and quantitatively within probability theory, a formalism that has been developed over centuries.

A probabilistic model describes the world in terms of a set S of possible states - the sample space. We don't know the true state of the world, so we (somehow) come up with a probability distribution over S which gives the probability of any state being the true one. The world usually described by a set of variables or attributes.

Consider the probabilistic model of a fictitious medical expert system. The 'world' is described by 8 binary valued variables:

Visit to Asia? A
 Tuberculosis? T
 Either tub. or lung cancer? E
 Lung cancer? L
 Smoking? S
 Bronchitis? B
 Dyspnoea? D
 Positive X-ray? X

We have $2^8 = 256$ possible states or configurations and so 256 probabilities to find.

10.3 Review of Probability Theory

The primitives in probabilistic reasoning are *random variables*. Just like primitives in Propositional Logic are propositions. A random variable is not in fact a variable, but a function from a sample space S to another space, often the real numbers.

For example, let the random variable Sum (representing outcome of two die throws) be defined thus:

$$Sum(\text{die1}, \text{die2}) = \text{die1} + \text{die2}$$

Each random variable has an associated probability distribution determined by the underlying distribution on the sample space

Continuing our example : $P(\text{Sum} = 2) = 1/36$,
 $P(\text{Sum} = 3) = 2/36, \dots, P(\text{Sum} = 12) = 1/36$

Consider the probabilistic model of the fictitious medical expert system mentioned before. The sample space is described by 8 binary valued variables.

Visit to Asia? A
 Tuberculosis? T
 Either tub. or lung cancer? E
 Lung cancer? L
 Smoking? S
 Bronchitis? B
 Dyspnoea? D
 Positive X-ray? X

There are $2^8 = 256$ events in the sample space. Each event is determined by a joint instantiation of all of the variables.

$$\begin{aligned} S = & \{(A = f, T = f, E = f, L = f, S = f, B = f, D = f, X = f), \\ & (A = f, T = f, E = f, L = f, S = f, B = f, D = f, X = t), \dots \\ & (A = t, T = t, E = t, L = t, S = t, B = t, D = t, X = t)\} \end{aligned}$$

Since S is defined in terms of joint instantiations, any distribution defined on it is called a joint distribution. All underlying distributions will be joint distributions in this module. The variables {A,T,E,L,S,B,D,X} are in fact random variables, which ‘project’ values.

$$L(A = f, T = f, E = f, L = f, S = f, B = f, D = f, X = f) = f$$

$$L(A = f, T = f, E = f, L = f, S = f, B = f, D = f, X = t) = f$$

$$L(A = t, T = t, E = t, L = t, S = t, B = t, D = t, X = t) = t$$

Each of the random variables {A,T,E,L,S,B,D,X} has its own distribution, determined by the underlying joint distribution. This is known as the margin distribution. For example, the distribution for L is denoted P(L), and this distribution is defined by the two probabilities P(L = f) and P(L = t). For example,

$$P(L = f)$$

$$= P(A = f, T = f, E = f, L = f, S = f, B = f, D = f, X = f)$$

$$+ P(A = f, T = f, E = f, L = f, S = f, B = f, D = f, X = t)$$

$$+ P(A = f, T = f, E = f, L = f, S = f, B = f, D = t, X = f)$$

...

$$P(A = t, T = t, E = t, L = f, S = t, B = t, D = t, X = t)$$

P(L) is an example of a marginal distribution.

Here's a joint distribution over two binary value variables A and B

	A=0	A=1	
B=0	0.2	0.3	
B=1	0.4	0.1	

We get the marginal distribution over B by simply adding up the different possible values of A for any value of B (and put the result in the “margin”).

	A=0	A=1	
B=0	0.2	0.3	0.5 (= 0.2 + 0.3)
B=1	0.4	0.1	0.5 (= 0.4 + 0.1)

In general, given a joint distribution over a set of variables, we can get the marginal distribution over a subset by simply summing out those variables not in the subset.

In the medical expert system case, we can get the marginal distribution over, say, A,D by simply summing out the other variables:

$$P(A, D) = \sum_{T,E,L,S,B,X} P(A, T, E, L, S, B, D, X)$$

However, computing marginals is not an easy task always. For example,

$$\begin{aligned} P(A = t, D = f) &= P(A = t, T = f, E = f, L = f, S = f, B = f, D = f, X = f) \\ &+ P(A = t, T = f, E = f, L = f, S = f, B = f, D = f, X = t) \\ &+ P(A = t, T = f, E = f, L = f, S = f, B = t, D = f, X = f) \\ &+ P(A = t, T = f, E = f, L = f, S = f, B = t, D = f, X = t) \\ &\dots \\ P(A = t, T = t, E = t, L = t, S = t, B = t, D = f, X = t) \end{aligned}$$

This has 64 summands! Each of whose value needs to be estimated from empirical data. For the estimates to be of good quality, each of the instances that appear in the summands should appear sufficiently large number of times in the empirical data. Often such a large amount of data is not available.

However, computation can be simplified for certain special but common conditions. This is the condition of *independence* of variables.

Two random variables A and B are independent iff

$$P(A, B) = P(A)P(B)$$

i.e. can get the joint from the marginals

This is quite a strong statement: It means for **any** value x of A and **any** value y of B

$$P(A = x, B = y) = P(A = x)P(B = y)$$

Note that the independence of two random variables is a property of a the underlying

$$P(A, B) = P(A)P(B) \quad P'(A, B) \neq P'(A)P'(B)$$

probability distribution. We can have

Conditional probability is defined as:

$$P(A|B) \stackrel{\text{def}}{=} \frac{P(A, B)}{P(B)}$$

It means for any value x of A and any value y of B

$$P(A = x | B = y) = \frac{P(A = x, B = y)}{P(B = y)}$$

If A and B are independent then

$$P(A|B) = P(A)$$

Conditional probabilities can represent causal relationships in both directions.

From cause to (probable) effects

$$\begin{aligned} \text{Car_start} = f &\leftarrow \text{Cold_battery} = t \\ P(\text{Car_start} = f \mid \text{Cold_battery} = t) &= 0.8 \end{aligned}$$

From effect to (probable) cause

$$\begin{aligned} \text{Cold_battery} = t &\leftarrow \text{Car_start} = f \\ P(\text{Cold_battery} = t \mid \text{Car_start} = f) &= 0.7 \end{aligned}$$

Module 10

Reasoning with Uncertainty - Probabilistic reasoning

Lesson 27

Probabilistic Inference

10.4 Probabilistic Inference Rules

Two rules in probability theory are important for inferencing, namely, the product rule and the Bayes' rule.

Product rule:

$$\begin{aligned} P(A, B|C) &= P(A|B, C)P(B|C) \\ &= P(B|A, C)P(A|C) \end{aligned}$$

Bayes' rule:

$$P(A|B, C) = \frac{P(B|A, C)P(A|C)}{P(B|C)}$$

Used in Bayesian statistics :

$$P(Model|Data) = \frac{P(Model)P(Data|Model)}{P(Data)}$$

Here is a simple example, of application of Bayes' rule.

Suppose you have been tested positive for a disease; what is the probability that you actually have the disease?

It depends on the accuracy and sensitivity of the test, and on the background (*prior*) probability of the disease.

Let $P(\text{Test}=+\text{ve} | \text{Disease}=\text{true}) = 0.95$, so the false negative rate, $P(\text{Test}=-\text{ve} | \text{Disease}=\text{true})$, is 5%.

Let $P(\text{Test}=+\text{ve} | \text{Disease}=\text{false}) = 0.05$, so the false positive rate is also 5%. Suppose the disease is rare: $P(\text{Disease}=\text{true}) = 0.01$ (1%).

Let D denote Disease and "T=+ve" denote the positive Test.

Then,

$$P(D=\text{true}|T=+\text{ve}) = \frac{P(T=+\text{ve}|D=\text{true}) * P(D=\text{true})}{P(T=+\text{ve}|D=\text{true}) * P(D=\text{true}) + P(T=+\text{ve}|D=\text{false}) * P(D=\text{false})}$$

$$= \frac{0.95 * 0.01}{0.95*0.01 + 0.05*0.99} = 0.161$$

So the probability of having the disease given that you tested positive is just 16%. This seems too low, but here is an intuitive argument to support it. Of 100 people, we expect only 1 to have the disease, but we expect about 5% of those (5 people) to test positive. So of the 6 people who test positive, we only expect 1 of them to actually have the disease; and indeed 1/6 is approximately 0.16.

In other words, the reason the number is so small is that you believed that this is a rare disease; the test has made it 16 times more likely you have the disease, but it is still unlikely in absolute terms. If you want to be "objective", you can set the prior to uniform (i.e. effectively ignore the prior), and then get

$$\begin{aligned} P(D=\text{true}|T=\text{+ve}) &= \frac{P(T=\text{+ve}|D=\text{true}) * P(D=\text{true})}{P(T=\text{+ve})} \\ &= \frac{0.95 * 0.5}{0.95*0.5 + 0.05*0.5} = \frac{0.475}{0.5} = 0.95 \end{aligned}$$

This, of course, is just the true positive rate of the test. However, this conclusion relies on your belief that, if you did not conduct the test, half the people in the world have the disease, which does not seem reasonable.

A better approach is to use a plausible prior (eg $P(D=\text{true})=0.01$), but then conduct multiple independent tests; if they all show up positive, then the posterior will increase. For example, if we conduct two (conditionally independent) tests T_1, T_2 with the same reliability, and they are both positive, we get

$$\begin{aligned} P(D=\text{true}|T_1=\text{+ve}, T_2=\text{+ve}) &= \frac{P(T_1=\text{+ve}|D=\text{true}) * P(T_2=\text{+ve}|D=\text{true}) * P(D=\text{true})}{P(T_1=\text{+ve}, T_2=\text{+ve})} \\ &= \frac{0.95 * 0.95 * 0.01}{0.95*0.95*0.01 + 0.05*0.05*0.99} = \frac{0.009}{0.0115} = 0.7826 \end{aligned}$$

The assumption that the pieces of evidence are conditionally independent is called the **naive Bayes** assumption. This model has been successfully used for mainly application including classifying email as spam ($D=\text{true}$) or not ($D=\text{false}$) given the presence of various key words ($T_i=\text{+ve}$ if word i is in the text, else $T_i=\text{-ve}$). It is clear that the words are not independent, even conditioned on spam/not-spam, but the model works surprisingly well nonetheless.

In many problems, complete independence of variables do not exist. Though many of them are conditionally independent.

X and Y are conditionally independent given Z iff

$$P(X, Y|Z) = P(X|Z)P(Y|Z)$$

In full: X and Y are conditionally independent given Z iff for any instantiation x, y, z of X, Y, Z we have

$$\begin{aligned} & P(X = x, Y = y|Z = z) \\ &= P(X = x|Z = z)P(Y = y|Z = z) \end{aligned}$$

An example of conditional independence:

Consider the following three Boolean random variables:

LeaveBy8, GetTrain, OnTime

Suppose we can assume that:

$$P(OnTime | GetTrain, LeaveBy8) = P(OnTime | GetTrain)$$

$$\text{but NOT } P(OnTime | LeaveBy8) = P(OnTime)$$

Then, *OnTime* is dependent on *LeaveBy8*, but *independent* of *LeaveBy8* given *GetTrain*.

$$\text{We can represent } P(OnTime | GetTrain, LeaveBy8) = P(OnTime | GetTrain)$$

graphically by: *LeaveBy8* -> *GetTrain* -> *OnTime*

Module 10

Reasoning with Uncertainty - Probabilistic reasoning

Lesson 28

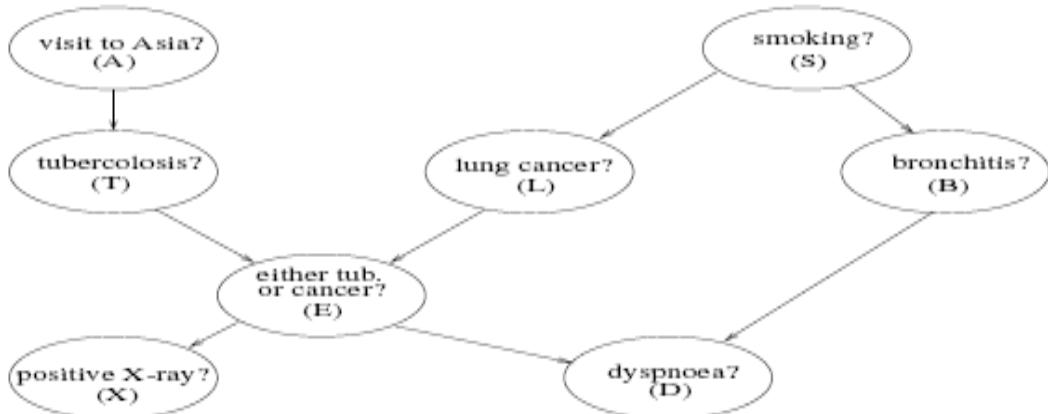
Bayes Networks

10.5 Bayesian Networks

10.5.1 Representation and Syntax

Bayes nets (BN) (also referred to as Probabilistic Graphical Models and Bayesian Belief Networks) are directed acyclic graphs (DAGs) where each node represents a random variable. The intuitive meaning of an arrow from a parent to a child is that the parent directly influences the child. These influences are quantified by conditional probabilities.

BNs are graphical representations of joint distributions. The BN for the medical expert system mentioned previously represents a joint distribution over 8 binary random variables {A,T,E,L,S,B,D,X}.



Conditional Probability Tables

Each node in a Bayesian net has an associated conditional probability table or CPT. (Assume all random variables have only a finite number of possible values). This gives the probability values for the random variable at the node conditional on values for its parents. Here is a part of one of the CPTs from the medical expert system network.

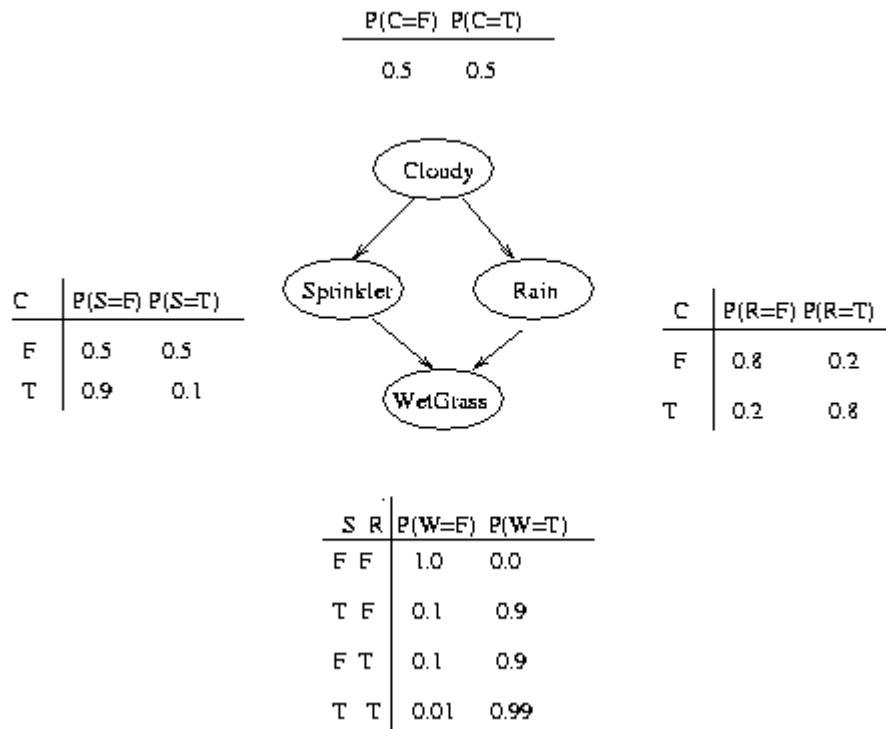
$$\begin{array}{ll} P(D = t | E = t, B = t) = 0.9 & P(D = t | E = t, B = f) = 0.7 \\ P(D = t | E = f, B = t) = 0.8 & P(D = t | E = f, B = f) = 0.1 \end{array}$$

If a node has no parents, then the CPT reduces to a table giving the marginal distribution on that random variable.

$$P(A = t) = 0.1$$

$$P(A = f) = 0.9$$

Consider another example, in which all nodes are binary, i.e., have two possible values, which we will denote by T (true) and F (false).



We see that the event "grass is wet" ($W=true$) has two possible causes: either the water sprinkler is on ($S=true$) or it is raining ($R=true$). The strength of this relationship is shown in the table. For example, we see that $\Pr(W=true | S=true, R=false) = 0.9$ (second row), and hence, $\Pr(W=false | S=true, R=false) = 1 - 0.9 = 0.1$, since each row must sum to one. Since the C node has no parents, its CPT specifies the prior probability that it is cloudy (in this case, 0.5). (Think of C as representing the season: if it is a cloudy season, it is less likely that the sprinkler is on and more likely that the rain is on.)

10.5.2 Semantics of Bayesian Networks

The simplest conditional independence relationship encoded in a Bayesian network can be stated as follows: a node is independent of its ancestors given its parents, where the ancestor/parent relationship is with respect to some fixed topological ordering of the nodes.

In the sprinkler example above, by the chain rule of probability, the joint probability of all the nodes in the graph above is

$$P(C, S, R, W) = P(C) * P(S|C) * P(R|C,S) * P(W|C,S,R)$$

By using conditional independence relationships, we can rewrite this as

$$P(C, S, R, W) = P(C) * P(S|C) * P(R|C) * P(W|S,R)$$

where we were allowed to simplify the third term because R is independent of S given its parent C, and the last term because W is independent of C given its parents S and R. We can see that the conditional independence relationships allow us to represent the joint more compactly. Here the savings are minimal, but in general, if we had n binary nodes, the full joint would require $O(2^n)$ space to represent, but the factored form would require $O(n 2^k)$ space to represent, where k is the maximum fan-in of a node. And fewer parameters makes learning easier.

The intuitive meaning of an arrow from a parent to a child is that the parent directly influences the child. The direction of this influence is often taken to represent causal influence. The conditional probabilities give the strength of causal influence. A 0 or 1 in a CPT represents a deterministic influence.

$$\begin{array}{ll} P(E = t | T = t, C = t) &= 1 & P(E = t | T = t, L = f) = 1 \\ P(E = t | T = f, L = t) &= 1 & P(E = t | T = f, L = f) = 0 \end{array}$$

10.5.2.1 Decomposing Joint Distributions

A joint distribution can always be broken down into a product of conditional probabilities using repeated applications of the product rule.

$$\begin{aligned} P(A, T, E, L, S, B, D, X) &= P(X|A, T, E, L, S, B, D)P(D|A, T, E, L, S, B) \\ P(B|A, T, E, L, S)P(S|A, T, E, L)P(L|A, T, E)P(E|A, T)P(T|A)P(A) \end{aligned}$$

We can order the variables however we like:

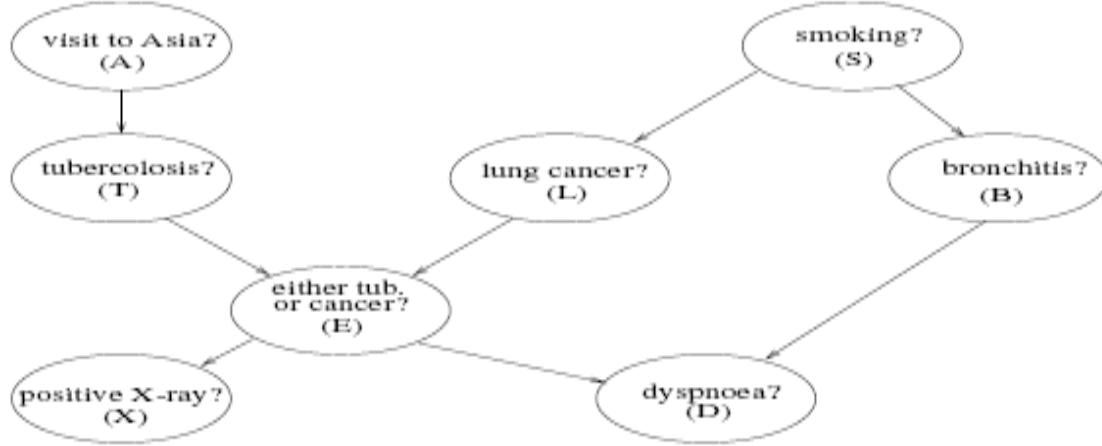
$$\begin{aligned} P(A, T, E, L, S, B, D, X) &= P(X|A, T, E, L, S, B, D)P(D|A, T, E, L, S, B) \\ P(E|A, T, L, S, B)P(B|A, T, L, S)P(L|A, T, S)P(S|A, T)P(T|A)P(A) \end{aligned}$$

10.5.2.2 Conditional Independence in Bayes Net

A Bayes net represents the assumption that each node is conditionally independent of all its non-descendants given its parents.

So for example,

$$P(E|A, T, L, S, B) = P(E|T, L)$$



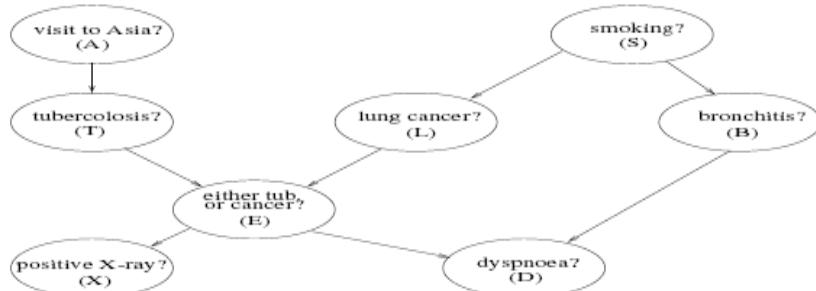
Note that, a node is NOT independent of its descendants given its parents. Generally,

$$P(E|A, T, L, S, B, X) \neq P(E|T, L)$$

10.5.2.3 Variable ordering in Bayes Net

The conditional independence assumptions expressed by a Bayes net allow a compact representation of the joint distribution. First note that the Bayes net imposes a partial order on nodes: $X \leq Y$ iff X is a descendant of Y . We can always break down the joint so that the conditional probability factor for a node only has non-descendants in the condition.

$$\begin{aligned} P(A, T, E, L, S, B, D, X) &= P(X|A, T, E, L, S, B, D)P(D|A, T, E, L, S, B) \\ P(E|A, T, L, S, B)P(B|A, T, L, S)P(L|A, T, S)P(S|A, T)P(T|A)P(A) \end{aligned}$$



10.5.2.4 The Joint Distribution as a Product of CPTs

Because each node is conditionally independent of all its nondescendants given its parents, and because we can write the joint appropriately we have:

$$\begin{aligned} P(A, T, E, L, S, B, D, X) &= \\ &P(X|A, T, E, L, S, B, D)P(D|A, T, E, L, S, B) \\ &\quad P(E|A, T, L, S, B)P(B|A, T, L, S) \\ &\quad P(L|A, T, S)P(S|A, T)P(T|A)P(A) \\ &= \\ &P(X|E)P(D|E, B)P(E|T, L)P(B|S) \\ &\quad P(L|S)P(S|T)P(T|A)P(A) \end{aligned}$$

So the CPTs determine the full joint distribution.

In short,

$$P(X_1, X_2, \dots, X_n) = \prod_{i=1}^n P(X_i | Parents(X_i))$$

Bayesian Networks allow a compact representation of the probability distributions. An unstructured table representation of the “medical expert system” joint would require $2^8 - 1 = 255$ numbers. With the structure imposed by the conditional independence assumptions this reduces to 18 numbers. Structure also allows efficient inference — of which more later.

10.5.2.5 Conditional Independence and d-separation in a Bayesian Network

We can have conditional independence relations between sets of random variables. In the Medical Expert System Bayesian net, $\{X, D\}$ is independent of $\{A, T, L, S\}$ given $\{E, B\}$ which means:

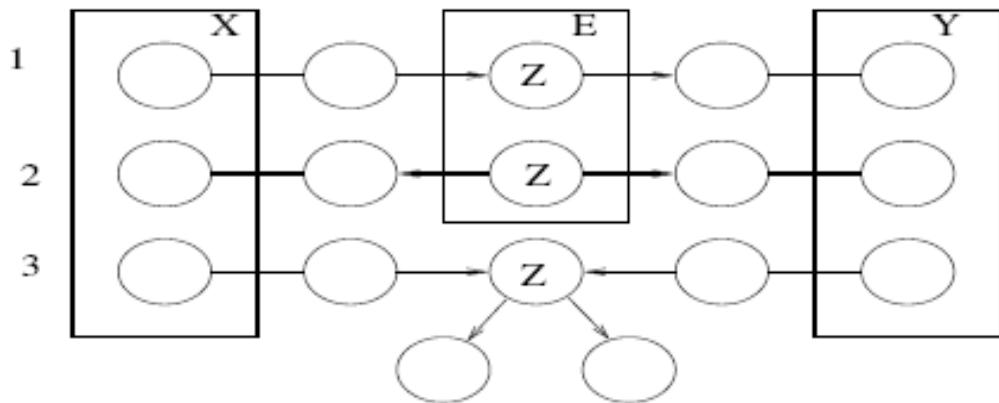
$$P(X, D | E, B) = P(X, D | E, B, A, T, L, S)$$

equivalently . . .

$$P(X, D, A, T, L, S | E, B) = P(A, T, L, S | E, B)P(X, D | E, B)$$

We need a way of checking for these conditional independence relations

Conditional independence can be checked using the **d-separation** property of the Bayes net directed acyclic graph. d-separation is short for direction-dependent separation.



If E d-separates X and Y then X and Y are conditionally independent given E .

E d-separates X and Y if every *undirected path* from a node in X to a node in Y is blocked given E .

Defining d-separation:

A path is blocked given a set of nodes E if there is a node Z on the path for which one of these three conditions holds:

1. Z is in E and Z has one arrow on the path coming in and one arrow going out.
2. Z is in E and Z has both path arrows leading out.
3. Neither Z nor any descendant of Z is in E , and both path arrows lead in to Z .

10.5.3 Building a Bayes Net: The Family Out? Example

We start with a natural language description of the situation to be modeled:

I want to know if my family is at home as I approach the house. Often my wife leaves on a light when she goes out, but also sometimes if she is expecting a guest. When nobody is home the dog is put in the back yard, but he is also put there when he has bowel trouble. If the dog is in the back yard, I will hear her barking, but I may be confused by other dogs barking.

Building the Bayes net involves the following steps.

We build Bayes nets to get probabilities concerning what we don't know given what we do know. What we don't know is not observable. These are called hypothesis events – we need to know what are the hypothesis events in a problem?

Recall that a Bayesian network is composed of related (random) variables, and that a variable incorporates an exhaustive set of mutually exclusive events - one of its events is true. How shall we represent the two hypothesis events in a problem?

Variables whose values are observable and which are relevant to the hypothesis events are called information variables. What are the information variables in a problem?

In this problem we have three variables, what is the causal structure between them? Actually, the whole notion of ‘cause’ let alone ‘determining causal structure’ is very controversial. Often (but not always) your intuitive notion of causality will help you.

Sometimes we need mediating variables which are neither information variables or hypothesis variables to represent causal structures.

10.5.4 Learning of Bayesian Network Parameters

One needs to specify two things to describe a BN: the graph topology (structure) and the parameters of each CPT. It is possible to learn both of these from data. However, learning structure is much harder than learning parameters. Also, learning when some of the nodes are hidden, or we have missing data, is much harder than when everything is observed. This gives rise to 4 cases:

Structure	Observability	Method
Known	Full	Maximum Likelihood Estimation
Known	Partial	EM (or gradient ascent)
Unknown	Full	Search through model space
Unknown	Partial	EM + search through model space

We discuss below the first case only.

Known structure, full observability

We assume that the goal of learning in this case is to find the values of the parameters of each CPT which maximizes the likelihood of the training data, which contains N cases (assumed to be independent). The normalized log-likelihood of the training set D is a sum of terms, one for each node:

$$L = \frac{1}{N} \sum_{i=1}^m \sum_{t=1}^s \log P(X_i | \text{Pa}(X_i), D_t).$$

We see that the log-likelihood scoring function decomposes according to the structure of the graph, and hence we can maximize the contribution to the log-likelihood of each node independently (assuming the parameters in each node are independent of the other nodes). In cases where N is small compared to the number of parameters that require fitting, we can use a numerical prior to regularize the problem. In this case, we call the estimates Maximum A Posteriori (MAP) estimates, as opposed to Maximum Likelihood (ML) estimates.

Consider estimating the Conditional Probability Table for the W node. If we have a set of training data, we can just count the number of times the grass is wet when it is raining and the sprinkler is on, $N(W=1,S=1,R=1)$, the number of times the grass is wet when it is raining and the sprinkler is off, $N(W=1,S=0,R=1)$, etc. Given these counts (which are the sufficient statistics), we can find the Maximum Likelihood Estimate of the CPT as follows:

$$\Pr(W = w | S = s, R = r) \approx N(W = w, S = s, R = r) / N(S = s, R = r)$$

where the denominator is $N(S=s, R=r) = N(W=0, S=s, R=r) + N(W=1, S=s, R=r)$. Thus "learning" just amounts to counting (in the case of multinomial distributions). For Gaussian nodes, we can compute the sample mean and variance, and use linear regression to estimate the weight matrix. For other kinds of distributions, more complex procedures are necessary.

As is well known from the HMM literature, ML estimates of CPTs are prone to sparse data problems, which can be solved by using (mixtures of) Dirichlet priors (pseudo counts). This results in a Maximum A Posteriori (MAP) estimate. For Gaussians, we can use a Wishart prior, etc.

Module 10

Reasoning with Uncertainty - Probabilistic reasoning

Lesson 29

A Basic Idea of Inferencing with Bayes Networks

10.5.5 Inferencing in Bayesian Networks

10.5.5.1 Exact Inference

The basic inference problem in BNs is described as follows:

Given

1. A Bayesian network BN
2. Evidence e - an instantiation of some of the variables in BN (e can be empty)
3. A query variable Q

Compute $P(Q|e)$ - the (marginal) conditional distribution over Q

Given what we do know, compute distribution over what we do not. Four categories of inferencing tasks are usually encountered.

1. *Diagnostic Inferences* (from effects to causes)

Given that John calls, what is the probability of burglary? i.e. Find $P(B|J)$

2. *Causal Inferences* (from causes to effects)

Given Burglary, what is the probability that

John calls, i.e. $P(J|B)$

Mary calls, i.e. $P(M|B)$

3. *Intercausal Inferences* (between causes of a common event)

Given alarm, what is the probability of burglary? i.e. $P(B|A)$

Now given Earthquake, what is the probability of burglary? i.e. $P(B|A \square E)$

4. *Mixed Inferences* (some causes and some effects known)

Given John calls and no Earth quake, what is the probability of Alarm, i.e.

$P(A|J, \neg E)$

We will demonstrate below the inferencing procedure for BNs. As an example consider the following linear BN without any apriori evidence.

$$A \rightarrow B \rightarrow C \rightarrow D$$

Consider computing all the marginals (with no evidence). $P(A)$ is given, and

$$P(B) = \sum_A P(B|A)P(A)$$

We don't need any conditional independence assumption for this.

For example, suppose A, B are binary then we have

$$\begin{aligned} P(B = t) &= \\ P(B = t|A = t)P(A = t) + P(B = t|A = f)P(A = f) \end{aligned}$$

Now,

$$P(C) = \sum_B P(C|B)P(B)$$

$P(B)$ (the marginal distribution over B) was not given originally... but we just computed it in the last step, so we're OK (assuming we remembered to store $P(B)$ somewhere).

If C were not independent of A given B, we would have a CPT for $P(C|A,B)$ not $P(C|B)$. Note that we had to wait for $P(B)$ before $P(C)$ was calculable.

If each node has k values, and the chain has n nodes this algorithm has complexity $O(nk^2)$. Summing over the joint has complexity $O(k^n)$.

Complexity can be reduced by more efficient summation by “pushing sums into products”.

$$\begin{aligned} P(D) &= \sum_{A,B,C} P(A, B, C, D) \\ &= \sum_{A,B,C} P(A)P(B|A)P(C|B)P(D|C) \quad \text{Conditional independence} \\ &= \sum_C \sum_B \sum_A P(A)P(B|A)P(C|B)P(D|C) \quad \text{Commutativity of addition} \\ &= \sum_C P(D|C) \sum_B P(C|B) \sum_A P(A)P(B|A) \quad xy + xz = x(y + z) \end{aligned}$$

Dynamic programming may also be used for the problem of exact inferencing in the above Bayes Net. The steps are as follows:

1. We first compute

$$f_1(B) = \sum_A P(A)P(B|A)$$

2. $f_l(B)$ is a function representable by a table of numbers, one for each possible value of B.

3. Here,

$$f_1(B) = P(B)$$

4. We then use $f_l(B)$ to calculate $f_2(C)$ by summation over B

This method of solving a problem (ie finding $P(D)$) by solving subproblems and storing the results is characteristic of dynamic programming.

The above methodology may be generalized. We eliminated variables starting from the root, but we don't have to. We might have also done the following computation.

$$\begin{aligned} P(A, E) &= \sum_B \sum_C \sum_D P(A, B, C, D, E) \\ &= \sum_B \sum_C \sum_D P(A)P(B|A)P(C|B)P(D|C)P(E|D) \\ &= P(A) \sum_B P(B|A) \sum_C P(C|B) \sum_D P(D|C)P(E|D) \\ &= P(A) \sum_B P(B|A) \sum_C P(C|B) f_1(C, E) \\ &= P(A) \sum_B P(B|A) f_2(B, E) \\ &= P(A) f_3(A, E) \end{aligned}$$

The following points are to be noted about the above algorithm. The algorithm computes intermediate results which are not individual probabilities, but entire tables such as $f_l(C, E)$. It so happens that $f_l(C, E) = P(E|C)$ but we will see examples where the intermediate tables do not represent probability distributions.

Dealing with Evidence

Dealing with evidence is easy. Suppose $\{A, B, C, D, E\}$ are all binary and we want $P(C|A = t, E = t)$. Computing $P(C, A = t, E = t)$ is enough—it's a table of numbers, one for each value of C. We need to just renormalise it so that they add up to 1.

$$\begin{aligned}
P(C|A = t, E = t) &= \frac{P(C, A = t, E = t)}{P(A = t, E = t)} \\
&= \frac{P(C, A = t, E = t)}{\sum_C P(C, A = t, E = t)}
\end{aligned}$$

It was noticed from the above computation that conditional distributions are basically just normalised marginal distributions. Hence, the algorithms we study are only concerned with computing marginals. Getting the actual conditional probability values is a trivial “tidying-up” last step.

Now let us concentrate on computing

$$P(C, A = t, E = t)$$

It can be done by plugging in the observed values for A and E and summing out B and D.

$$\begin{aligned}
&P(C, A = t, E = t) \\
&= \sum_{B,D} P(A = t, B, C, D, E = t) \\
&= \sum_{B,D} P(A = t)P(B|A = t)P(C|B)P(D|C)P(E = t|D) \\
&= P(A = t) \sum_B P(B|A = t)P(C|B) \sum_D P(D|C)P(E = t|D) \\
&= P(A = t) \sum_B P(B|A = t)P(C|B)f_1(C) \\
&= P(A = t)f_1(C) \sum_B P(B|A = t)P(C|B) \\
&= P(A = t)f_1(C)f_2(C)
\end{aligned}$$

We don't really care about $P(A = t)$, since it will cancel out.

Now let us see how evidence-induce independence can be exploited. Consider the following computation.

$$P(A, C = t) \quad (1)$$

$$= \sum_{B,D,E} P(A, B, C = t, D, E) \quad (2)$$

$$= \sum_{B,D,E} P(A)P(B|A)P(C = t|B)P(D|C)P(E|D) \quad (3)$$

$$= P(A) \sum_B P(B|A)P(C = t|B) \sum_D P(D|C = t) \sum_E P(E|D) \quad (4)$$

$$= P(A) \sum_B P(B|A)P(C = t|B) \quad (5)$$

$$= P(A)f(A) \quad (6)$$

Since,

$$\sum_E P(E|D) = 1(D) \text{ and } \sum_D P(D|C = t) = 1$$

Clever variable elimination would jump straight to (5). Choosing an optimal order of variable elimination leads to a large amount of computational swing. However, finding the optimal order is a hard problem.

10.5.5.1.1 Variable Elimination

For a Bayes net, we can sometimes use the factored representation of the joint probability distribution to do marginalization efficiently. The key idea is to "push sums in" as far as possible when summing (marginalizing) out irrelevant terms, e.g., for the water sprinkler network

$$\begin{aligned} \Pr(W = w) &= \sum_c \sum_s \sum_r \Pr(C = c, S = s, R = r, W = w) \\ &= \sum_c \sum_s \sum_r \Pr(C = c) \times \Pr(S = s|C = c) \times \Pr(R = r|C = c) \times \Pr(W = w|S = s, R = r) \\ &= \sum_c \Pr(C = c) \sum_s \Pr(S = s|C = c) \sum_r \Pr(R = r|C = c) \times \Pr(W = w|S = s, R = r) \end{aligned}$$

Notice that, as we perform the innermost sums, we create new terms, which need to be summed over in turn e.g.,

$$\Pr(W = w) = \sum_c \Pr(C = c) \sum_s \Pr(S = s|C = c) \times T1(c, w, s)$$

where,

$$T1(c, w, s) = \sum_r \Pr(R = r|C = c) \times \Pr(W = w|S = s, R = r)$$

Continuing this way,

$$\Pr(W = w) = \sum_c \Pr(C = c) \times T2(c, w)$$

where,

$$T2(c, w) = \sum_s \Pr(S = s | C = c) \times T1(c, w, s)$$

In a nutshell, the variable elimination procedure repeats the following steps.

1. Pick a variable X_i
2. Multiply all expressions involving that variable, resulting in an expression f over a number of variables (including X_i)
3. Sum out X_i , i.e. compute and store

$$f' = \sum_{X_i} f$$

For the multiplication, we must compute a number for each joint instantiation of all variables in f , so complexity is exponential in the largest number of variables participating in one of these multiplicative subexpressions.

If we wish to compute several marginals at the same time, we can use Dynamic Programming to avoid the redundant computation that would be involved if we used variable elimination repeatedly.

Exact inferencing in a general Bayes net is a hard problem. However, for networks with some special topologies efficient solutions inferencing techniques. We discuss one such technique for a class of networks called **Poly-trees**.

10.5.5.2 Inferencing in Poly-Trees

A poly-tree is a graph where there is at most one undirected path between any two pair of nodes. The inferencing problem in poly-trees may be stated as follows.

U: $U_1 \dots U_m$, parents of node X

Y: $Y_1 \dots Y_n$, children of node X

X: Query variable

E: Evidence variables (whose truth values are known)

Objective: compute $P(X | E)$

E^+_X is the set of causal support for X comprising of the variables above X connected through its parents, which are known.

E^-_X is the set of evidential support for X comprising of variables below X connected through its children.

In order to compute $P(X | E)$ we have

$$P(X|E) = P(X|E_X^+, E_X^-)$$

$$= \frac{P(E_X^-|X, E_X^+)P(X|E_X^+)}{P(E_X^-|E_X^+)}$$

Since X d-separates E_X^+ from E_X^- we can simplify the numerator as

$$P(X|E) = \alpha P(E_X^-|X)P(X|E_X^+)$$

where $1/\alpha$ is the constant representing denominator.

Both the terms – $P(X|E^-_X)$ and $P(E^+_X|X)$ can be computed recursively using the conditional independence relations. If the parents are known, X is conditionally independent from all other nodes in the Causal support set. Similarly, given the children, X is independent from all other variables in the evidential support set.

10.5.6 Approximate Inferencing in Bayesian Networks

Many real models of interest, have large number of nodes, which makes exact inference very slow. Exact inference is NP-hard in the worst case.) We must therefore resort to approximation techniques. Unfortunately, approximate inference is #P-hard, but we can nonetheless come up with approximations which often work well in practice. Below is a list of the major techniques.

Variational methods. The simplest example is the mean-field approximation, which exploits the law of large numbers to approximate large sums of random variables by their means. In particular, we essentially decouple all the nodes, and introduce a new parameter, called a variational parameter, for each node, and iteratively update these parameters so as to minimize the cross-entropy (KL distance) between the approximate and true probability distributions. Updating the variational parameters becomes a proxy for inference. The mean-field approximation produces a lower bound on the likelihood. More sophisticated methods are possible, which give tighter lower (and upper) bounds.

Sampling (Monte Carlo) methods. The simplest kind is importance sampling, where we draw random samples x from $P(X)$, the (unconditional) distribution on the hidden variables, and then weight the samples by their likelihood, $P(y|x)$, where y is the evidence. A more efficient approach in high dimensions is called Monte Carlo Markov

Chain (MCMC), and includes as special cases Gibbs sampling and the Metropolis-Hastings algorithm.

Bounded cutset conditioning. By instantiating subsets of the variables, we can break loops in the graph. Unfortunately, when the cutset is large, this is very slow. By instantiating only a subset of values of the cutset, we can compute lower bounds on the probabilities of interest. Alternatively, we can sample the cutsets jointly, a technique known as block Gibbs sampling.

Parametric approximation methods. These express the intermediate summands in a simpler form, e.g., by approximating them as a product of smaller factors. "Minibuckets" and the Boyen-Koller algorithm fall into this category.

Questions

1. 1% of women over age forty who are screened, have breast cancer. 80% of women who really do have breast cancer will have a positive mammography (meaning the test indicates she has cancer). 9.6% of women who do *not* actually have breast cancer will have a positive mammography (meaning that they are incorrectly diagnosed with cancer). Define two Boolean random variables, M meaning a positive mammography test and $\sim M$ meaning a negative test, and C meaning the woman has breast cancer and $\sim C$ means she does not.

- (a) If a woman in this age group gets a positive mammography, what is the probability that she actually has breast cancer?
- (b) True or False: The "Prior" probability, indicating the percentage of women with breast cancer, is not needed to compute the "Posterior" probability of a woman having breast cancer given a positive mammography.
- (c) Say a woman who gets a positive mammography test, M_1 , goes back and gets a second mammography, M_2 , which also is positive. Use the Naive Bayes assumption to compute the probability that she has breast cancer given the results from these two tests.
- (d) True or False: $P(C | M_1, M_2)$ can be calculated in general given only $P(C)$ and $P(M_1, M_2 | C)$.

2. Let A, B, C, D be Boolean random variables. Given that:

A and B are (absolutely) independent.

C is independent of B given A .

D is independent of C given A and B .

$\text{Prob}(A=T) = 0.3$

$\text{Prob}(B=T) = 0.6$

$\text{Prob}(C=T|A=T) = 0.8$

$$\begin{aligned}
 \text{Prob}(C=T|A=F) &= 0.4 \\
 \text{Prob}(D=T|A=T, B=T) &= 0.7 \\
 \text{Prob}(D=T|A=T, B=F) &= 0.8 \\
 \text{Prob}(D=T|A=F, B=T) &= 0.1 \\
 \text{Prob}(D=T|A=F, B=F) &= 0.2
 \end{aligned}$$

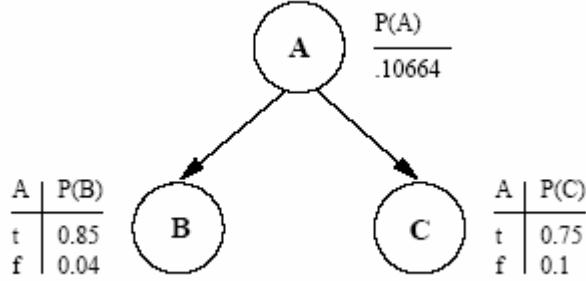
Compute the following quantities:

- 1) $\text{Prob}(D=T)$
- 2) $\text{Prob}(D=F, C=T)$
- 3) $\text{Prob}(A=T|C=T)$
- 4) $\text{Prob}(A=T|D=F)$
- 5) $\text{Prob}(A=T, D=T|B=F)$.

3. Consider a situation in which we want to reason about the relationship between smoking and lung cancer. We'll use 5 Boolean random variables representing "has lung cancer" (C), "smokes" (S), "has a reduced life expectancy" (RLE), "exposed to second-hand smoke" (SHS), and "at least one parent smokes" (PS). Intuitively, we know that whether or not a person has cancer is directly influenced by whether she is exposed to second-hand smoke and whether she smokes. Both of these things are affected by whether her parents smoke. Cancer reduces a person's life expectancy.

- i. Draw the network (nodes and arcs only)
- ii. How many independent values are required to specify all the conditional probability tables (CPTs) for your network?
- iii. How many independent values are in the full joint probability distribution for this problem domain?

4. Consider the following Bayesian Network containing 3 Boolean random variables:



(a) Compute the following quantities:

- (i) $P(\sim B, C | A)$

$$(ii) P(A | \sim B, C)$$

4.b. Now add on to the network above a fourth node containing Boolean random variable D, with arcs to it from both B and C.

(i) Yes or No: Is A conditionally independent of D given B?

(ii) Yes or No: Is B conditionally independent of C given A?

5. Consider the following probability distribution over 6 variables A,B,C,D,E, and F for which the factorization as stated below holds. Find and draw a Bayesian network that for which this factorization is true, but for which no additional factorizations nor any fewer factorizations are true.

$$p(a, b, c, d, e, f) = p(a)p(b)p(c|a, b)p(d|b)p(e|c, d)p(f|e)$$

Solution

1.a. Given:

$$P(C) = 0.01, P(M|C) = 0.8, P(M|\sim C) = 0.096.$$

$$\begin{aligned} P(C|M) &= [P(M|C)P(C)]/P(M) \\ &= [P(M|C)P(C)]/[P(M|C)P(C) + P(M|\sim C)P(\sim C)] \\ &= (0.8)(0.01)/[(0.8)(0.01) + (0.096)(0.99)] \\ &= (0.008)/(0.008 + 0.09504) \\ &= 0.0776 \end{aligned}$$

So, there is a 7.8% chance.

1.b. False, as seen in the use of Bayes's Rule in (a).

$$\begin{aligned} 1.c. P(C|M_1, M_2) &= [P(M_1, M_2|C)P(C)]/P(M_1, M_2) \\ &= [P(M_1|C)P(M_2|C)P(C)]/P(M_1, M_2) \\ &= (.8)(.8)(.01)/P(M_1, M_2) = 0.0064/P(M_1, M_2) \end{aligned}$$

Now, if we further assume that M1 and M2 are independent, then

$$\begin{aligned} P(M_1, M_2) &= P(M_1)P(M_2) \text{ and } P(M) = (P(M|C)P(C) + P(M|\sim C)P(\sim C)) \\ &= (.8)(.01) + (.096)(1-.01) = 0.103 \end{aligned}$$

$$\text{Then, } P(C|M_1, M_2) = .0064 / .103 = 0.603 \text{ (i.e., 60.3%)}$$

More correctly, we don't assume that M1 and M2 are independent, but only use the original Naïve Bayes assumption that M1 and M2 are conditionally independent given C. In this case we need to compute $P(M1, M2)$

$$\begin{aligned} P(M1, M2) &= P(M1, M2|C)P(C) + P(M1, M2|\sim C)P(\sim C) \\ &= P(M1|C)P(M2|C)P(C) + P(M1|\sim C)P(M2|\sim C)P(\sim C) \\ &= (.8)(.8)(.01) + (.096)(.096)(.99) = 0.0155 \end{aligned}$$

$$\text{So, } P(C|M1, M2) = 0.603 / 0.0155 = 0.4129 \text{ (i.e., 41.3%)}$$

1.d. False. Need either $P(M1, M2)$ or $P(M1, M2 | \sim C)$.

2. The values of the quantities are given below:

$$\begin{aligned} P(D=T) &= \\ P(D=T, A=T, B=T) &+ P(D=T, A=T, B=F) + P(D=T, A=F, B=T) + P(D=T, A=F, B=F) = \\ P(D=T|A=T, B=T) P(A=T, B=T) &+ P(D=T|A=T, B=F) P(A=T, B=F) + \\ P(D=T|A=F, B=T) P(A=F, B=T) &+ P(D=T|A=F, B=F) P(A=F, B=F) = \\ (\text{since A and B are independent absolutely}) & \\ P(D=T|A=T, B=T) P(A=T) P(B=T) &+ P(D=T|A=T, B=F) P(A=T) P(B=F) + \\ P(D=T|A=F, B=T) P(A=F) P(B=T) &+ P(D=T|A=F, B=F) P(A=F) P(B=F) = \\ 0.7*0.3*0.6 + 0.8*0.3*0.4 &+ 0.1*0.7*0.6 + 0.2*0.7*0.4 = 0.32 \end{aligned}$$

$$\begin{aligned} P(D=F, C=T) &= \\ P(D=F, C=T, A=T, B=T) &+ P(D=F, C=T, A=T, B=F) + P(D=F, C=T, A=F, B=T) + \\ P(D=F, C=T, A=F, B=F) &= \\ P(D=F, C=T|A=T, B=T) P(A=T, B=T) &+ P(D=F, C=T|A=T, B=F) P(A=T, B=F) + \\ P(D=F, C=T|A=F, B=T) P(A=F, B=T) &+ P(D=F, C=T|A=F, B=F) P(A=F, B=F) = \\ (\text{since C and D are independent given A and B}) & \\ P(D=F|A=T, B=T) P(C=T|A=T, B=T) P(A=T, B=T) &+ P(D=F|A=T, B=F) P(C=T|A=T, B=F) \\ P(A=T, B=F) &+ \\ P(D=F|A=F, B=T) P(C=T|A=F, B=T) P(A=F, B=T) &+ \\ P(D=F|A=F, B=F) P(C=T|A=F, B=F) P(A=F, B=F) = \\ (\text{since C is independent of B given A and A and B are independent absolutely}) & \\ P(D=F|A=T, B=T) P(C=T|A=T) P(A=T) P(B=T) &+ P(D=F|A=T, B=F) P(C=T|A=T) \\ P(A=T) P(B=F) &+ \\ P(D=F|A=F, B=T) P(C=T|A=F) P(A=F) P(B=T) &+ P(D=F|A=F, B=F) P(C=T|A=F) \\ P(A=F) P(B=F) = 0.3*0.8*0.3*0.6 &+ 0.2*0.8*0.3*0.4 + 0.9*0.4*0.7*0.6 + \\ 0.8*0.4*0.7*0.4 &= 0.3032 \end{aligned}$$

$$P(A=T|C=T) = P(C=T|A=T)P(A=T) / P(C=T).$$

$$\text{Now } P(C=T) = P(C=T, A=T) + P(C=T, A=F) =$$

$$P(C=T|A=T)P(A=T) + P(C=T|A=F)P(A=F) = 0.8*0.3 + 0.4*0.7 = 0.52$$

So $P(C=T|A=T)P(A=T) / P(C=T) = 0.8*0.3/0.52= 0.46.$

$P(A=T|D=F) = P(D=F|A=T) P(A=T)/P(D=F).$

Now $P(D=F) = 1-P(D=T) = 0.68$ from the first question above.

$P(D=F|A=T) = P(D=T,B=T|A=T) + P(D=F,B=F|A=T) =$

$P(D=F|B=T,A=T) P(B=T|A=T) + P(D=F|B=F,A=T) P(B=F|A=T) =$
(since B is independent of A)

$P(D=F|B=T,A=T) P(B=T) + P(D=F|B=F,A=T) P(B=F) = 0.3*0.6 + 0.2*0.4 = 0.26.$

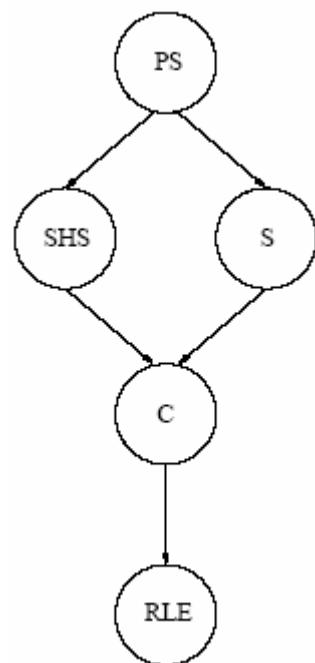
So $P(A=T|D=F) = P(D=F|A=T) P(A=T)/P(D=F) =$

$0.26 * 0.3 / 0.68 = 0.115$

$P(A=T,D=T|B=F) = P(D=T|A=T,B=F) P(A=T|B=F) =$ (since A and B are independent)

$P(D=T|A=T,B=F) P(A=T) = 0.8*0.3 = 0.24.$

3.i. The network is shown below



ii. $1 + 2 + 2 + 4 + 2 = 11$

iii. $2^5 - 1 = 31$

4.a.i. $P(\sim B, C | A) = P(\sim B | A) P(C | A) = (0.15)(0.75) = 0.1125$

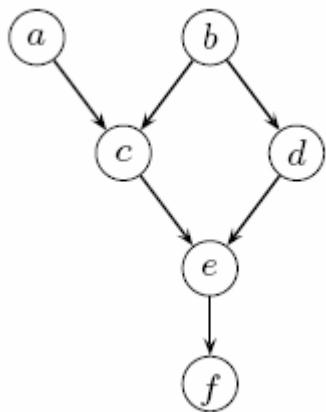
4.a.ii. The steps are shown below

$$\begin{aligned}
P(A \mid \neg B, C) &= \frac{P(\neg B, C \mid A) P(A)}{P(\neg B, C)} \\
&= \frac{P(\neg B, C \mid A) P(A)}{P(\neg B, C \mid A) P(A) + P(\neg B, C \mid \neg A) P(\neg A)} \\
&= \frac{(0.1125)(0.10664)}{(0.1125)(0.10664) + (0.096)(0.89336)} \\
&= 0.12272
\end{aligned}$$

4.b.i. No

4.b.ii. Yes

5. The Bayesian network can be obtained by applying chain rule of probability in the order of factorization mentioned in the question.



Module 11

Reasoning with uncertainty-Fuzzy Reasoning

11.1 Instructional Objective

- The students should understand the use of fuzzy logic as a method of handling uncertainty
- **The student should learn the definition of fuzzy sets and fuzzy operations like union and intersection**
- Students should understand the use of hedges in linguistic description using fuzzy sets
- Students should be able to convert linguistic description of a uncertain problem in terms of statements in fuzzy logic
- Students should understand the fuzzy inferencing mechanism
- Students should understand the steps of
 - Fuzzification
 - Fuzzy rule evaluation
 - Defuzzification
- Students should understand the design of fuzzy expert systems

At the end of this lesson the student should be able to do the following:

- **Represent a given problem using fuzzy logic**
- **Design a fuzzy expert system for a given problem.**

Lesson 30

Other Paradigms of Uncertain Reasoning

11.2 Reasoning with Uncertainty

Fuzzy systems is an alternative to traditional notions of set membership and logic that has its origins in ancient Greek philosophy, and applications at the leading edge of Artificial Intelligence.

11.2.1 THE PROBLEM: REAL-WORLD VAGUENESS

Natural language abounds with vague and imprecise concepts, such as "Sally is tall," or "It is very hot today." Such statements are difficult to translate into more precise language without losing some of their semantic value: for example, the statement "Sally's height is 152 cm." does not explicitly state that she is tall, and the statement "Sally's height is 1.2 standard deviations about the mean height for women of her age in her culture" is fraught with difficulties: would a woman 1.1999999 standard deviations above the mean be tall? Which culture does Sally belong to, and how is membership in it defined?

While it might be argued that such vagueness is an obstacle to clarity of meaning, only the most staunch traditionalists would hold that there is no loss of richness of meaning when statements such as "Sally is tall" are discarded from a language. Yet this is just what happens when one tries to translate human language into classic logic. Such a loss is not noticed in the development of a payroll program, perhaps, but when one wants to allow for natural language queries, or "knowledge representation" in expert systems, the meanings lost are often those being searched for.

For example, when one is designing an expert system to mimic the diagnostic powers of a physician, one of the major tasks is to codify the physician's decision-making process. The designer soon learns that the physician's view of the world, despite her dependence upon precise, scientific tests and measurements, incorporates evaluations of symptoms, and relationships between them, in a "fuzzy," intuitive manner: deciding how much of a particular medication to administer will have as much to do with the physician's sense of the relative "strength" of the patient's symptoms as it will their height/weight ratio. While some of the decisions and calculations could be done using traditional logic, we will see how fuzzy systems affords a broader, richer field of data and the manipulation of that data than do more traditional methods.

11.2.2 HISTORIC FUZZINESS

The precision of mathematics owes its success in large part to the efforts of Aristotle and the philosophers who preceded him. In their efforts to devise a concise theory of logic, and later mathematics, the so-called "Laws of Thought" were posited. One of these, the "Law of the Excluded Middle," states that every proposition must either be True or False. Even when Parmenides proposed the first version of this law (around 400 B.C.) there were strong and immediate objections: for example, Heraclitus proposed that things could be simultaneously True and not True.

It was Plato who laid the foundation for what would become fuzzy logic, indicating that there was a third region (beyond True and False) where these opposites "tumbled about." Other, more modern philosophers echoed his sentiments, notably Hegel, Marx, and Engels. But it was Lukasiewicz who first proposed a systematic alternative to the bi-valued logic of Aristotle.

In the early 1900's, Lukasiewicz described a three-valued logic, along with the mathematics to accompany it. The third value he proposed can best be translated as the term "possible," and he assigned it a numeric value between True and False. Eventually, he proposed an entire notation and axiomatic system from which he hoped to derive modern mathematics.

Later, he explored four-valued logics, five-valued logics, and then declared that in principle there was nothing to prevent the derivation of an infinite-valued logic. Lukasiewicz felt that three- and infinite-valued logics were the most intriguing, but he ultimately settled on a four-valued logic because it seemed to be the most easily adaptable to Aristotelian logic.

Knuth proposed a three-valued logic similar to Lukasiewicz's, from which he speculated that mathematics would become even more elegant than in traditional bi-valued logic. His insight, apparently missed by Lukasiewicz, was to use the integral range [-1, 0 +1] rather than [0, 1, 2]. Nonetheless, this alternative failed to gain acceptance, and has passed into relative obscurity.

It was not until relatively recently that the notion of an infinite-valued logic took hold. In 1965 Lotfi A. Zadeh published his seminal work "Fuzzy Sets" which described the mathematics of fuzzy set theory, and by extension fuzzy logic. This theory proposed making the membership function (or the values False and True) operate over the range of real numbers [0.0, 1.0]. New operations for the calculus of logic were proposed, and showed to be in principle at least a generalization of classic logic.

Module 11

Reasoning with uncertainty-Fuzzy Reasoning

Lesson 31

Fuzzy Set Representation

11.3 Fuzzy Sets: BASIC CONCEPTS

The notion central to fuzzy systems is that truth values (in fuzzy logic) or membership values (in fuzzy sets) are indicated by a value on the range [0.0, 1.0], with 0.0 representing absolute Falseness and 1.0 representing absolute Truth. For example, let us take the statement:

"Jane is old."

If Jane's age was 75, we might assign the statement the truth value of 0.80. The statement could be translated into set terminology as follows:

"Jane is a member of the set of old people."

This statement would be rendered symbolically with fuzzy sets as:

$$m_{OLD}(Jane) = 0.80$$

where m is the membership function, operating in this case on the fuzzy set of old people, which returns a value between 0.0 and 1.0.

At this juncture it is important to point out the distinction between fuzzy systems and probability. Both operate over the same numeric range, and at first glance both have similar values: 0.0 representing False (or non-membership), and 1.0 representing True (or membership). However, there is a distinction to be made between the two statements: The probabilistic approach yields the natural-language statement, "There is an 80% chance that Jane is old," while the fuzzy terminology corresponds to "Jane's degree of membership within the set of old people is 0.80." The semantic difference is significant: the first view supposes that Jane is or is not old (still caught in the Law of the Excluded Middle); it is just that we only have an 80% chance of knowing which set she is in. By contrast, fuzzy terminology supposes that Jane is "more or less" old, or some other term corresponding to the value of 0.80. Further distinctions arising out of the operations will be noted below.

The next step in establishing a complete system of fuzzy logic is to define the operations of EMPTY, EQUAL, COMPLEMENT (NOT), CONTAINMENT, UNION (OR), and INTERSECTION (AND). Before we can do this rigorously, we must state some formal definitions:

Definition 1: Let X be some set of objects, with elements noted as x . Thus,
 $X = \{x\}$.

Definition 2: A fuzzy set A in X is characterized by a membership function $m_A(x)$ which maps each point in X onto the real interval [0.0, 1.0]. As $m_A(x)$ approaches 1.0, the "grade of membership" of x in A increases.

Definition 3: A is EMPTY iff for all x, $m_A(x) = 0.0$.

Definition 4: $A = B$ iff for all x: $m_A(x) = m_B(x)$ [or, $m_A = m_B$].

Definition 5: $m_A' = 1 - m_A$.

Definition 6: A is CONTAINED in B iff $m_A \leq m_B$.

Definition 7: $C = A \text{ UNION } B$, where: $m_C(x) = \text{MAX}(m_A(x), m_B(x))$.

Definition 8: $C = A \text{ INTERSECTION } B$ where: $m_C(x) = \text{MIN}(m_A(x), m_B(x))$.

It is important to note the last two operations, UNION (OR) and INTERSECTION (AND), which represent the clearest point of departure from a probabilistic theory for sets to fuzzy sets. Operationally, the differences are as follows:

For independent events, the probabilistic operation for AND is multiplication, which (it can be argued) is counterintuitive for fuzzy systems. For example, let us presume that x = Bob, S is the fuzzy set of smart people, and T is the fuzzy set of tall people. Then, if $m_S(x) = 0.90$ and $m_T(x) = 0.90$, the probabilistic result would be:

$$m_S(x) * m_T(x) = 0.81$$

whereas the fuzzy result would be:

$$\text{MIN}(m_S(x), m_T(x)) = 0.90$$

The probabilistic calculation yields a result that is lower than either of the two initial values, which when viewed as "the chance of knowing" makes good sense.

However, in fuzzy terms the two membership functions would read something like "Bob is very smart" and "Bob is very tall." If we presume for the sake of argument that "very" is a stronger term than "quite," and that we would correlate "quite" with the value 0.81, then the semantic difference becomes obvious. The probabilistic calculation would yield the statement

If Bob is very smart, and Bob is very tall, then Bob is a quite tall, smart person.

The fuzzy calculation, however, would yield

If Bob is very smart, and Bob is very tall, then Bob is a very tall, smart person.

Another problem arises as we incorporate more factors into our equations (such as the fuzzy set of heavy people, etc.). We find that the ultimate result of a series of AND's approaches 0.0, even if all factors are initially high. Fuzzy theorists argue that this is wrong: that five factors of the value 0.90 (let us say, "very") AND'ed together, should yield a value of 0.90 (again, "very"), not 0.59 (perhaps equivalent to "somewhat").

Similarly, the probabilistic version of A OR B is $(A+B - A*B)$, which approaches 1.0 as additional factors are considered. Fuzzy theorists argue that a string of low membership grades should not produce a high membership grade instead, the limit of the resulting membership grade should be the strongest membership value in the collection.

The skeptical observer will note that the assignment of values to linguistic meanings (such as 0.90 to "very") and vice versa, is a most imprecise operation. Fuzzy systems, it should be noted, lay no claim to establishing a formal procedure for assignments at this level; in fact, the only argument for a particular assignment is its intuitive strength. What fuzzy logic does propose is to establish a formal method of operating on these values, once the primitives have been established.

11.3.1 HEDGES

Another important feature of fuzzy systems is the ability to define "hedges," or modifier of fuzzy values. These operations are provided in an effort to maintain close ties to natural language, and to allow for the generation of fuzzy statements through mathematical calculations. As such, the initial definition of hedges and operations upon them will be quite a subjective process and may vary from one project to another. Nonetheless, the system ultimately derived operates with the same formality as classic logic.

The simplest example is in which one transforms the statement "Jane is old" to "Jane is very old." The hedge "very" is usually defined as follows:

$$m\text{"very"}A(x) = mA(x)^2$$

Thus, if $mOLD(Jane) = 0.8$, then $mVERYOLD(Jane) = 0.64$.

Other common hedges are "more or less" [typically $SQRT(mA(x))$], "somewhat," "rather," "sort of," and so on. Again, their definition is entirely subjective, but their operation is consistent: they serve to transform membership/truth values in a systematic manner according to standard mathematical functions.

A more involved approach to hedges is best shown through the work of Wenstop in his attempt to model organizational behavior. For his study, he constructed arrays of values for various terms, either as vectors or matrices. Each term and

hedge was represented as a 7-element vector or 7x7 matrix. He then intuitively assigned each element of every vector and matrix a value between 0.0 and 1.0, inclusive, in what he hoped was intuitively a consistent manner. For example, the term "high" was assigned the vector

0.0 0.0 0.1 0.3 0.7 1.0 1.0

and "low" was set equal to the reverse of "high," or

1.0 1.0 0.7 0.3 0.1 0.0 0.0

Wenstop was then able to combine groupings of fuzzy statements to create new fuzzy statements, using the APL function of Max-Min matrix multiplication.

These values were then translated back into natural language statements, so as to allow fuzzy statements as both input to and output from his simulator. For example, when the program was asked to generate a label "lower than sortof low," it returned "very low;" "(slightly higher) than low" yielded "rather low," etc.

The point of this example is to note that algorithmic procedures can be devised which translate "fuzzy" terminology into numeric values, perform reliable operations upon those values, and then return natural language statements in a reliable manner.

Module 11

Reasoning with uncertainty-Fuzzy Reasoning

Version 2 CSE IIT, Kharagpur

Lesson 32

Fuzzy Reasoning - Continued

11.4 Fuzzy Inferencing

The process of fuzzy reasoning is incorporated into what is called a Fuzzy Inferencing System. It is comprised of three steps that process the system inputs to the appropriate system outputs. These steps are 1) Fuzzification, 2) Rule Evaluation, and 3) Defuzzification. The system is illustrated in the following figure.

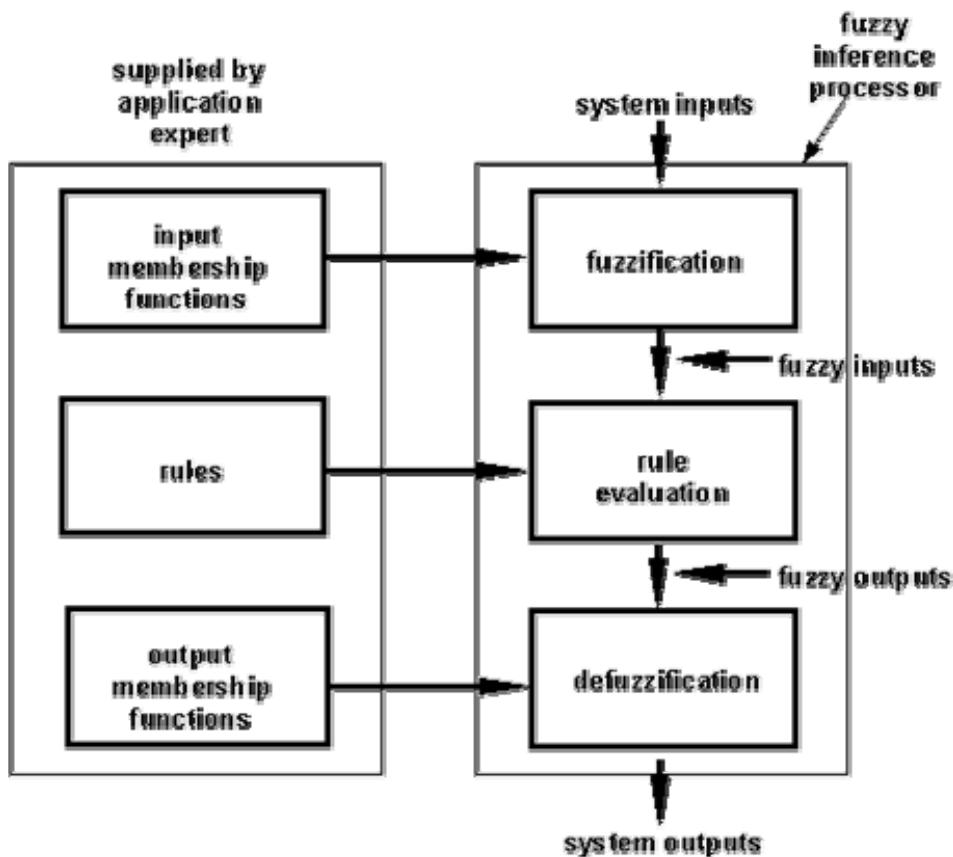


Figure 3: Fuzzy Inferencing Unit

Each step of fuzzy inferencing is described in the following sections.

11.4.1 Fuzzification

Fuzzification is the first step in the fuzzy inferencing process. This involves a domain transformation where crisp inputs are transformed into fuzzy inputs. Crisp inputs are exact inputs measured by sensors and passed into the control system for processing, such as temperature, pressure, rpm's, etc.. Each crisp input that is to be processed by the FIU has its own group of membership functions or sets to which they are transformed. This group of membership functions exists within a universe of discourse that holds all relevant values that the crisp input can possess. The following shows the structure of membership functions within a universe of discourse for a crisp input.

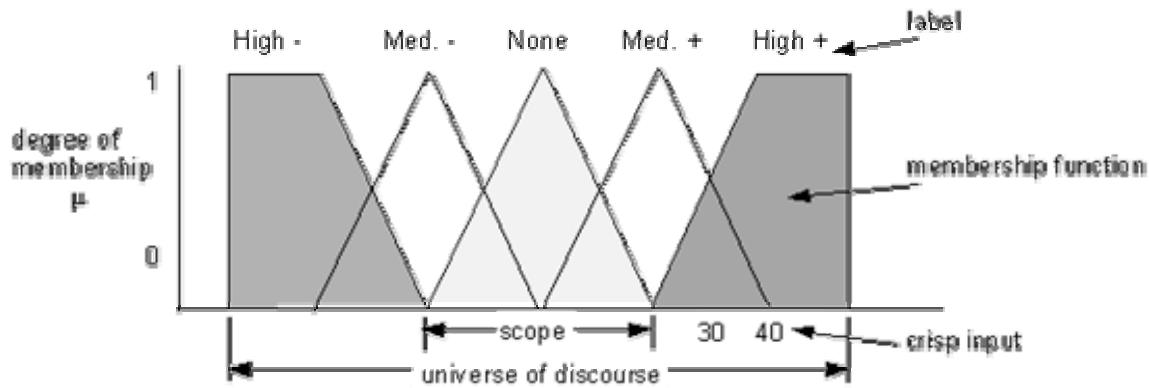


Figure 4: Membership Function Structure

where:

degree of membership: degree to which a crisp value is compatible to a membership function, value from 0 to 1, also known as truth value or fuzzy input.

membership function, MF: defines a fuzzy set by mapping crisp values from its domain to the sets associated degree of membership.

crisp inputs: distinct or exact inputs to a certain system variable, usually measured parameters external from the control system, e.g. 6 Volts.

label: descriptive name used to identify a membership function.

scope: or domain, the width of the membership function, the range of concepts, usually numbers, over which a membership function is mapped.

universe of discourse: range of all possible values, or concepts, applicable to a system variable.

When designing the number of membership functions for an input variable, labels must initially be determined for the membership functions. The number of labels correspond to the number of regions that the universe should be divided, such that each label describes a region of behavior. A scope must be assigned to each membership function that numerically identifies the range of input values that correspond to a label.

The shape of the membership function should be representative of the variable. However this shape is also restricted by the computing resources available. Complicated shapes require more complex descriptive equations or large lookup tables. The next figure shows examples of possible shapes for membership functions.

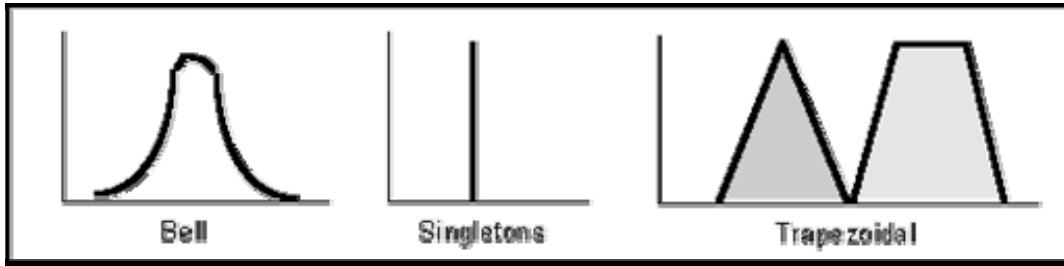


Figure 5: Membership Function Shapes

When considering the number of membership functions to exist within the universe of discourse, one must consider that:

- i) too few membership functions for a given application will cause the response of the system to be too slow and fail to provide sufficient output control in time to recover from a small input change. This may also cause oscillation in the system.
- ii) too many membership functions may cause rapid firing of different rule consequents for small changes in input, resulting in large output changes, which may cause instability in the system.

These membership functions should also be overlapped. No overlap reduces a system based on Boolean logic. Every input point on the universe of discourse should belong to the scope of at least one but no more than two membership functions. No two membership functions should have the same point of maximum truth, (1). When two membership functions overlap, the sum of truths or grades for any point within the overlap should be less than or equal to 1. Overlap should not cross the point of maximal truth of either membership function. Marsh has proposed two indices to describe the overlap of membership functions quantitatively. These are overlap ratio and overlap robustness. The next figure illustrates their meaning.

$$\text{Overlap Ratio} = \frac{\text{overlap scope}}{\text{adjacent MF scope}} \quad (1)$$

$$\text{Overlap Robustness} = \frac{\text{area of summed overlap}}{\max. \text{ area of summed overlap}} = \frac{\int_L^U (\mu_1 + \mu_2) dx}{2(U-L)} \quad (2)$$

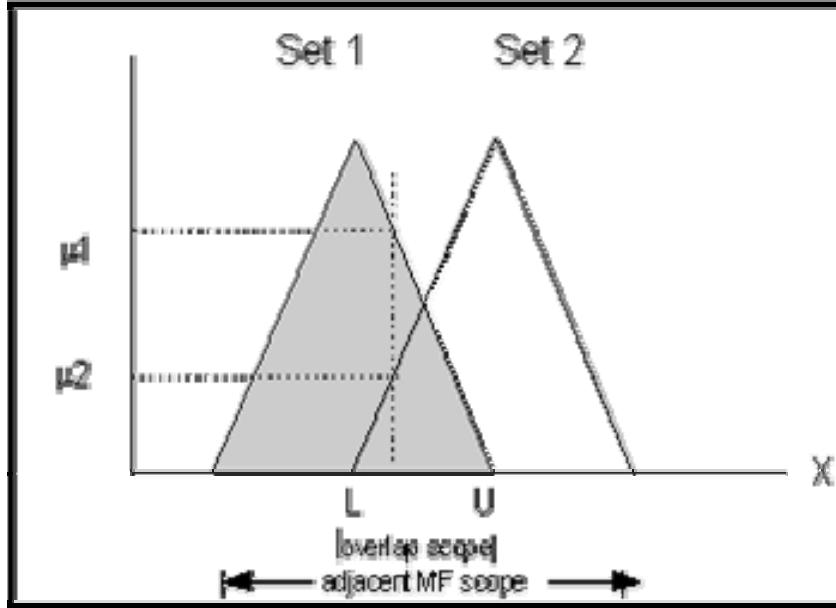


Figure 7: Overlap Indices

The fuzzification process maps each crisp input on the universe of discourse, and its intersection with each membership function is transposed onto the μ axis as illustrated in the previous figure. These μ values are the degrees of truth for each crisp input and are associated with each label as fuzzy inputs. These fuzzy inputs are then passed on to the next step, Rule Evaluation.

Fuzzy Rules

We briefly comment on so-called *fuzzy IF-THEN rules* introduced by Zadeh. They may be understood as partial imprecise knowledge on some crisp function and have (in the simplest case) the form IF x is A_i THEN y is B_i . They should **not** be immediately understood as implications; think of a *table* relating values of a (dependent) variable y to values of an (independent variable) x :

x	A_1	...	A_n
y	B_1	...	B_n

A_i, B_i may be crisp (concrete numbers) or fuzzy (small, medium, ...) It may be understood in two, in general non-equivalent ways: (1) as a listing of n possibilities, called Mamdani's formula:

$$MAMD(x,y) \equiv \bigvee_{i=1}^n (A_i(x) \& B_i(y)).$$

(where x is A_1 and y is B_1 or x is A_2 and y is B_2 or ...). (2) as a conjunction of implications:

$$RULES(x,y) \equiv \bigwedge_{i=1}^n (A_i(x) \rightarrow B_i(y)).$$

(if x is A_1 then y is B_1 and ...).

Both $MAMD$ and $RULES$ define a binary fuzzy relation (given the interpretation of A_i 's, B_i 's and truth functions of connectives). Now given a *fuzzy input* $A^*(x)$ one can consider the image B^* of $A^*(x)$ under this relation, i.e.,

$$B^*(y) \equiv \exists x(A(x) \& R(x,y)),$$

where $R(x,y)$ is $MAMD(x,y)$ (most frequent case) or $RULES(x,y)$. Thus one gets an operator assigning to each fuzzy input set A^* a corresponding fuzzy output B^* . Usually this is combined with some *fuzzifications* converting a crisp input x_0 to some fuzzy $A^*(x)$ (saying something as "x is similar to x_0 ") and a *defuzzification* converting the fuzzy image B^* to a crisp output y_0 . Thus one gets a crisp function; its relation to the set of rules may be analyzed.

11.4.2 Rule Evaluation

Rule evaluation consists of a series of IF-Zadeh Operator-THEN rules. A decision structure to determine the rules require familiarity with the system and its desired operation. This knowledge often requires the assistance of interviewing operators and experts. For this thesis this involved getting information on tremor from medical practitioners in the field of rehabilitation medicine.

There is a strict syntax to these rules. This syntax is structured as:

IF antecedent 1 ZADEH OPERATOR antecedent 2 THEN consequent 1 ZADEH OPERATOR consequent 2.....

The antecedent consists of: input variable IS label, and is equal to its associated fuzzy input or truth value $\mu(x)$.

The consequent consists of: output variable IS label, its value depends on the Zadeh Operator which determines the type of inferencing used. There are three Zadeh Operators, AND, OR, and NOT. The label of the consequent is associated with its output membership function. The Zadeh Operator is limited to operating on two membership functions, as discussed in the fuzzification process. Zadeh Operators are similar to Boolean Operators such that:

AND represents the intersection or *minimum* between the two sets, expressed as:

$$\mu_{A \cap B} = \min[\mu_A(x), \mu_B(x)]$$

OR represents the union or *maximum* between the two sets, expressed as:

$$\mu_{A \cup B} = \max[\mu_A(x), \mu_B(x)]$$

NOT represents the opposite of the set, expressed as:

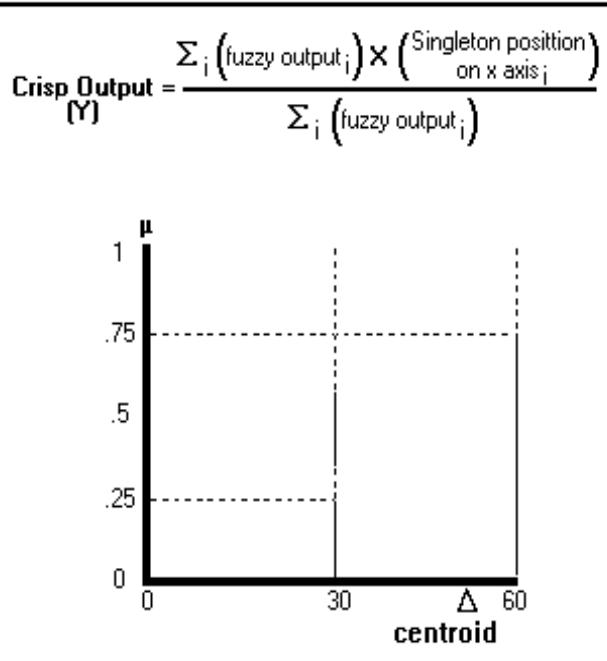
$$\overline{\mu_A} = [1 - \mu_A(x)]$$

The process for determining the result or rule strength of the rule may be done by taking the minimum fuzzy input of antecedent 1 AND antecedent 2, min. inferencing. This minimum result is equal to the consequent rule strength. If there are any consequents that are the same then the maximum rule strength between similar consequents is taken, referred to as maximum or max. inferencing, hence min./max. inferencing. This infers that the rule that is most true is taken. These rule strength values are referred to as fuzzy outputs.

11.4.3 Defuzzification

Defuzzification involves the process of transposing the fuzzy outputs to crisp outputs. There are a variety of methods to achieve this, however this discussion is limited to the process used in this thesis design.

A method of averaging is utilized here, and is known as the Center of Gravity method or COG, it is a method of calculating centroids of sets. The output membership functions to which the fuzzy outputs are transposed are restricted to being singletons. This is so to limit the degree of calculation intensity in the microcontroller. The fuzzy outputs are transposed to their membership functions similarly as in fuzzification. With COG the singleton values of outputs are calculated using a weighted average, illustrated in the next figure. The crisp output is the result and is passed out of the fuzzy inferencing system for processing elsewhere.



11.5 APPLICATIONS

Areas in which fuzzy logic has been successfully applied are often quite concrete. The first major commercial application was in the area of cement kiln control, an operation which requires that an operator monitor four internal states of the kiln, control four sets of operations, and dynamically manage 40 or 50 "rules of thumb" about their interrelationships, all with the goal of controlling a highly complex set of chemical interactions. One such rule is "If the oxygen percentage is rather high and the free-lime and kiln- drive torque rate is normal, decrease the flow of gas and slightly reduce the fuel rate".

Other applications which have benefited through the use of fuzzy systems theory have been information retrieval systems, a navigation system for automatic cars, a predicative fuzzy-logic controller for automatic operation of trains, laboratory water level controllers, controllers for robot arc-welders, feature-definition controllers for robot vision, graphics controllers for automated police sketchers, and more.

Expert systems have been the most obvious recipients of the benefits of fuzzy logic, since their domain is often inherently fuzzy. Examples of expert systems with fuzzy logic central to their control are decision-support systems, financial planners, diagnostic systems for determining soybean pathology, and a meteorological expert system in China for determining areas in which to establish rubber tree orchards.

Questions

1. In a class of 10 students (the universal set), 3 students speaks German to some degree, namely Alice to degree 0.7, Bob to degree 1.0, Cathrine to degree 0.4. What is the size of the subset A of German speaking students in the class?
2. In the above class, argue that the fuzzy subset B of students speaking a *very good* German is a fuzzy subset of A.
3. Let A and B be fuzzy subsets of a universal set X. Show that
$$|A \cup B| = |A| + |B| - |A \cap B|$$
4. For arbitrary fuzzy subsets A and B, show that
$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$
5. Let X = {0, 1, 2, ..., 6}, and let two fuzzy subsets, A and B, of X be defined by:

x	0	1	2	3	4	5	6
$\mu_A(x)$	1	0,7	0	1	0,5	0	0,4
$\mu_B(x)$	0,9	0,7	1	0,2	0,8	0,3	0

Notice that the description in a table as above is just a convenient notation for a fuzzy set description on the form: $A = \sum_i \left(\mu_A(x_i) / x_i \right)$. In its notation, the fuzzy set A in the example in this exercise is described by: $A = 1/0 + 0.7/1 + 1/3 + 0.5/4 + 0.4/6$.

Find:

$A \cap B$, $A \cup B$, \overline{A} and \overline{B}

Solutions

1. $|A| = 0.7 + 1.0 + 0.4 = 2.1$
2. The addition of “very” strengthens the requirement, which consequently will be less satisfied. Thus for all $x \in X$, $B(x) \leq A(x)$, which is precisely what characterized the fuzzy subset relation $B \subseteq A$

3. Answer

$$\begin{aligned}
 |A \cup B| &= |A| + |B| - |A \cap B| \\
 \Leftrightarrow |A \cup B| + |A \cap B| &= |A| + |B| \quad (\text{we added } |A \cap B| \text{ on both sides of } =) \\
 \Leftrightarrow \sum_{x \in X} \max(A(x), B(x)) + \sum_{x \in X} \min(A(x), B(x)) &= \sum_{x \in X} A(x) + \sum_{x \in X} B(x) \quad (\text{per definition of } |\cdot|)
 \end{aligned}$$

For an arbitrary element x we have:

$$\text{if } A(x) < B(x), \text{ then } \max(A(x), B(x)) + \min(A(x), B(x)) = B(x) + A(x)$$

$$\text{if } A(x) \geq B(x), \text{ then } \max(A(x), B(x)) + \min(A(x), B(x)) = A(x) + B(x)$$

and therefore, for all x ,

$$\max(A(x), B(x)) + \min(A(x), B(x)) = A(x) + B(x)$$

and

$$\begin{aligned}
 \sum_{x \in X} (\max(A(x), B(x)) + \min(A(x), B(x))) &= \sum_{x \in X} (A(x) + B(x)) \\
 \Leftrightarrow \sum_{x \in X} \max(A(x), B(x)) + \sum_{x \in X} \min(A(x), B(x)) &= \sum_{x \in X} A(x) + \sum_{x \in X} B(x).
 \end{aligned}$$

4. Answer

$$\overline{A \cup B} : \mu_{\overline{A \cup B}}(x) = 1 - \max(\mu_A(x), \mu_B(x)) \quad (\text{for alle } x)$$

$$\overline{A \cap B} : \mu_{\overline{A \cap B}}(x) = \min(\mu_A(x), \mu_B(x)) = \min(1 - \mu_A(x), 1 - \mu_B(x))$$

$$= \begin{cases} 1 - \mu_A(x) & \text{if } \mu_A(x) \geq \mu_B(x) \\ 1 - \mu_B(x) & \text{if } \mu_B(x) \geq \mu_A(x) \end{cases} = 1 - \max(\mu_A(x), \mu_B(x))$$

5. Answer

x	0	1	2	3	4	5	6
$\mu_{A \cap B}(x)$	0,9	0,7	0	0,2	0,5	0	0
$\mu_{A \cup B}(x)$	1	0,7	1	1	0,8	0,3	0,4
$\mu_{\overline{A}}(x)$	0	0,3	1	0	0,5	1	0,6
$\mu_{\overline{B}}(x)$	0,1	0,3	0	0,8	0,2	0,7	1

Module 12

Machine Learning

Version 2 CSE IIT, Kharagpur

12.1 Instructional Objective

- The students should understand the concept of learning systems
- Students should learn about different aspects of a learning system
- Students should learn about taxonomy of learning systems
- Students should learn about different aspects of a learning systems like inductive bias and generalization
- The student should be familiar with the following learning algorithms, and should be able to code the algorithms
 - Concept learning
 - Decision trees
 - Neural networks
- Students understand the merits and demerits of these algorithms and the problem domain where they should be applied.

At the end of this lesson the student should be able to do the following:

- Represent a problem as a learning problem
- Apply a suitable learning algorithm to solve the problem.

Lesson 33

Learning : Introduction

12.1 Introduction to Learning

Machine Learning is the study of how to build computer systems that adapt and improve with experience. It is a subfield of Artificial Intelligence and intersects with cognitive science, information theory, and probability theory, among others.

Classical AI deals mainly with *deductive* reasoning, learning represents *inductive* reasoning. Deductive reasoning arrives at answers to queries relating to a particular situation starting from a set of general axioms, whereas inductive reasoning arrives at general axioms from a set of particular instances.

Classical AI often suffers from the knowledge acquisition problem in real life applications where obtaining and updating the knowledge base is costly and prone to errors. Machine learning serves to solve the knowledge acquisition bottleneck by obtaining the result from data by induction.

Machine learning is particularly attractive in several real life problem because of the following reasons:

- Some tasks cannot be defined well except by example
- Working environment of machines may not be known at design time
- Explicit knowledge encoding may be difficult and not available
- Environments change over time
- Biological systems learn

Recently, learning is widely used in a number of application areas including,

- Data mining and knowledge discovery
- Speech/image/video (pattern) recognition
- Adaptive control
- Autonomous vehicles/robots
- Decision support systems
- Bioinformatics
- WWW

Formally, a computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E .

Thus a learning system is characterized by:

- task T
- experience E, and
- performance measure P

Examples:

Learning to play chess

T: Play chess

P: Percentage of games won in world tournament

E: Opportunity to play against self or other players

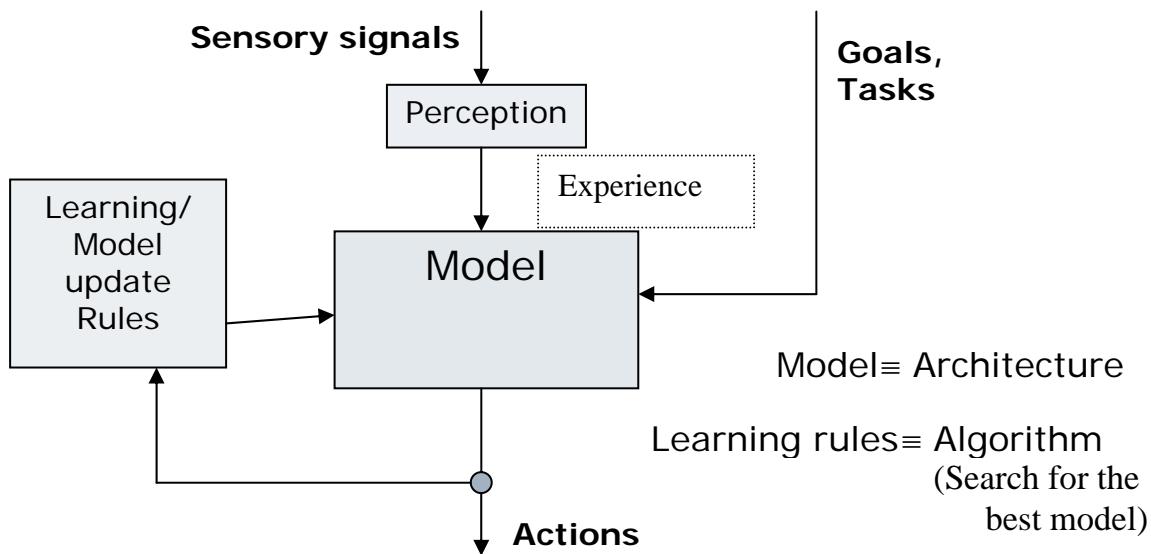
Learning to drive a van

T: Drive on a public highway using vision sensors

P: Average distance traveled before an error (according to human observer)

E: Sequence of images and steering actions recorded during human driving.

The block diagram of a generic learning system which can realize the above definition is shown below:



As can be seen from the above diagram the system consists of the following components:

- *Goal*: Defined with respect to the task to be performed by the system
- *Model*: A mathematical function which maps perception to actions
- *Learning rules*: Which update the model parameters with new experience such that the performance measures with respect to the goals is optimized
- *Experience*: A set of perception (and possibly the corresponding actions)

12.1.1 Taxonomy of Learning Systems

Several classification of learning systems are possible based on the above components as follows:

Goal/Task/Target Function:

Prediction: To predict the desired output for a given input based on previous input/output pairs. E.g., to predict the value of a stock given other inputs like market index, interest rates etc.

Categorization: To classify an object into one of several categories based on features of the object. E.g., a robotic vision system to categorize a machine part into one of the categories, spanner, hammer etc based on the parts' dimension and shape.

Clustering: To organize a group of objects into homogeneous segments. E.g., a satellite image analysis system which groups land areas into forest, urban and water body, for better utilization of natural resources.

Planning: To generate an optimal sequence of actions to solve a particular problem. E.g., an Unmanned Air Vehicle which plans its path to obtain a set of pictures and avoid enemy anti-aircraft guns.

Models:

- Propositional and FOL rules
- Decision trees
- Linear separators
- Neural networks
- Graphical models
- Temporal models like hidden Markov models

Learning Rules:

Learning rules are often tied up with the model of learning used. Some common rules are gradient descent, least square error, expectation maximization and margin maximization.

Experiences:

Learning algorithms use experiences in the form of perceptions or perception action pairs to improve their performance. The nature of experiences available varies with applications. Some common situations are described below.

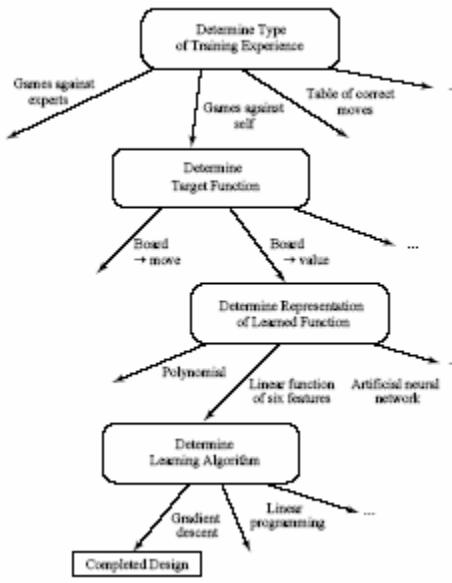
Supervised learning: In supervised learning a teacher or oracle is available which provides the desired action corresponding to a perception. A set of perception action pair provides what is called a training set. Examples include an automated vehicle where a set of vision inputs and the corresponding steering actions are available to the learner.

Unsupervised learning: In unsupervised learning no teacher is available. The learner only discovers persistent patterns in the data consisting of a collection of perceptions. This is also called exploratory learning. Finding out malicious network attacks from a sequence of anomalous data packets is an example of unsupervised learning.

Active learning: Here not only a teacher is available, the learner has the freedom to ask the teacher for suitable perception-action example pairs which will help the learner to improve its performance. Consider a news recommender system which tries to learn an users preferences and categorize news articles as interesting or uninteresting to the user. The system may present a particular article (of which it is not sure) to the user and ask whether it is interesting or not.

Reinforcement learning: In reinforcement learning a teacher is available, but the teacher instead of directly providing the desired action corresponding to a perception, return reward and punishment to the learner for its action corresponding to a perception. Examples include a robot in a unknown terrain where its get a punishment when its hits an obstacle and reward when it moves smoothly.

In order to design a learning system the designer has to make the following choices based on the application.



12.1.2 Mathematical formulation of the inductive learning problem

- Extrapolate from a given set of examples so that we can make accurate predictions about future examples.

Supervised versus Unsupervised learning

Want to learn an unknown function $f(\mathbf{x}) = \mathbf{y}$, where \mathbf{x} is an input example and \mathbf{y} is the desired output. Supervised learning implies we are given a set of (\mathbf{x}, \mathbf{y}) pairs by a "teacher." Unsupervised learning means we are only given the \mathbf{x} s. In either case, the goal is to estimate f .

Inductive Bias

- Inductive learning is an inherently conjectural process because any knowledge created by generalization from specific facts cannot be proven true; it can only be proven false. Hence, inductive inference is **falsity preserving**, not truth preserving.
- To generalize beyond the specific training examples, we need constraints or **biases** on what f is best. That is, learning can be viewed as searching the **Hypothesis Space H** of possible f functions.
- A bias allows us to choose one f over another one
- A completely unbiased inductive algorithm could only memorize the training examples and could not say anything more about other unseen examples.
- Two types of biases are commonly used in machine learning:
 - **Restricted Hypothesis Space Bias**
Allow only certain types of f functions, not arbitrary ones

- o **Preference Bias**

Define a metric for comparing f_s so as to determine whether one is better than another

Inductive Learning Framework

- Raw input data from sensors are preprocessed to obtain a **feature vector**, \mathbf{x} , that adequately describes all of the relevant features for classifying examples.
- Each \mathbf{x} is a list of (attribute, value) pairs. For example,

```
 $\mathbf{x} = (\text{Person} = \text{Sue}, \text{Eye-Color} = \text{Brown}, \text{Age} = \text{Young}, \text{Sex} = \text{Female})$ 
```

The number of attributes (also called features) is fixed (positive, finite). Each attribute has a fixed, finite number of possible values.

Each example can be interpreted as a *point* in an n -dimensional **feature space**, where n is the number of attributes.

Module 12

Machine Learning

Version 2 CSE IIT, Kharagpur

Lesson 34

Learning From Observations

12.2 Concept Learning

Definition:

The problem is to learn a function mapping examples into two classes: positive and negative. We are given a database of examples already classified as positive or negative. Concept learning: the process of inducing a function mapping input examples into a Boolean output.

Examples:

- Classifying objects in astronomical images as stars or galaxies
- Classifying animals as vertebrates or invertebrates

Example: Classifying Mushrooms

Class of Tasks: Predicting poisonous mushrooms

Performance: Accuracy of classification

Experience: Database describing mushrooms with their class

Knowledge to learn: Function mapping mushrooms to {0,1} where 0:not-poisonous and 1:poisonous

Representation of target knowledge: conjunction of attribute values.

Learning mechanism: candidate-elimination

Representation of instances:

Features:

- color {red, brown, gray}
- size {small, large}
- shape {round, elongated}
- land {humid, dry}
- air humidity {low, high}
- texture {smooth, rough}

Input and Output Spaces:

X : The space of all possible examples (input space).

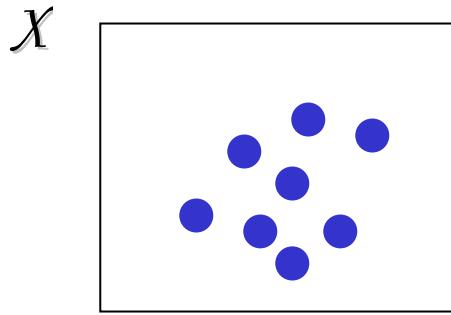
Y: The space of classes (output space).

An example in X is a feature vector X.

For instance: $X = (\text{red}, \text{small}, \text{elongated}, \text{humid}, \text{low}, \text{rough})$

X is the cross product of all feature values.

Only a small subset of instances is available in the database of examples.



$$Y = \{0, 1\}$$

Training Examples:

D : The set of training examples.

D is a set of pairs { (x,c(x)) }, where c is the target concept. c is a subset of the universe of discourse or the set of all possible instances.

Example of D:

((red,small,round,humid,low,smooth),	poisonous)
((red,small,elongated,humid,low,smooth),	poisonous)
((gray,large,elongated,humid,low,rough),	not-poisonous)
((red,small,elongated,humid,high,rough),	poisonous)

Hypothesis Representation

Any hypothesis h is a function from X to Y

$$h: X \rightarrow Y$$

We will explore the space of conjunctions.

Special symbols:

- ? Any value is acceptable
- 0 no value is acceptable

Consider the following hypotheses:

(?, ?, ?, ?, ?, ?): all mushrooms are poisonous

(0,0,0,0,0,0): no mushroom is poisonous

Hypotheses Space:

The space of all hypotheses is represented by H

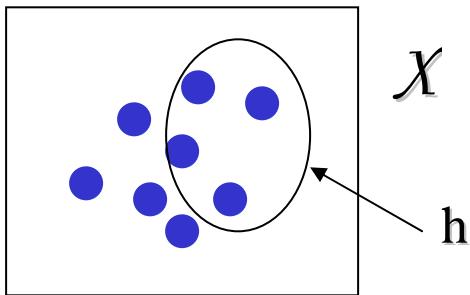
Let h be a hypothesis in H .

Let X be an example of a mushroom.

if $h(X) = 1$ then X is poisonous, otherwise X is not-poisonous

Our goal is to find the hypothesis, h^* , that is very “close” to target concept c .

A hypothesis is said to “cover” those examples it classifies as positive.



Assumptions:

We will explore the space of all conjunctions.

We assume the target concept falls within this space.

A hypothesis close to target concept c obtained after seeing many training examples will result in high accuracy on the set of unobserved examples. (Inductive Learning Hypothesis)

12.2.1 Concept Learning as Search

We will see how the problem of concept learning can be posed as a search problem.

We will illustrate that there is a general to specific ordering inherent to any hypothesis space.

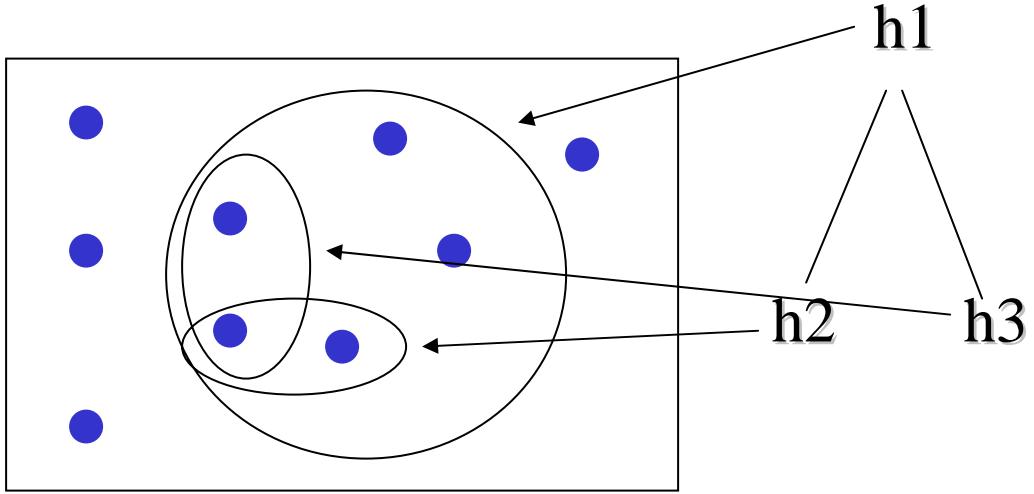
Consider these two hypotheses:

$$h_1 = (\text{red}, ?, ?, \text{humid}, ?, ?, ?)$$

$$h_2 = (\text{red}, ?, ?, ?, ?, ?, ?)$$

We say h_2 is more general than h_1 because h_2 classifies more instances than h_1 and h_1 is covered by h_2 .

For example, consider the following hypotheses



h_1 is more general than h_2 and h_3 .

h_2 and h_3 are neither more specific nor more general than each other.

Definitions:

Let h_j and h_k be two hypotheses mapping examples into $\{0,1\}$.

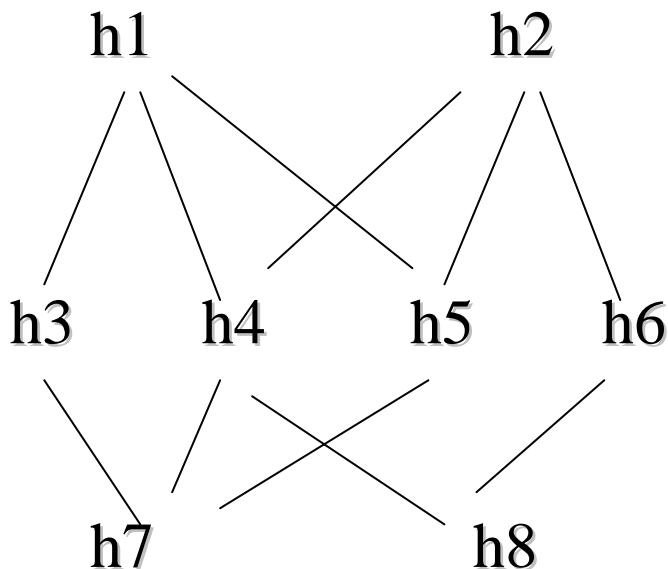
We say h_j is more general than h_k iff

For all examples X , $h_k(X) = 1 \Rightarrow h_j(X) = 1$

We represent this fact as $h_j \geq h_k$

The \geq relation imposes a partial ordering over the hypothesis space H (reflexive, antisymmetric, and transitive).

Any input space X defines then a lattice of hypotheses ordered according to the general-specific relation:



12.2.1.1 Algorithm to Find a Maximally-Specific Hypothesis

Algorithm to search the space of conjunctions:

- Start with the most specific hypothesis
- Generalize the hypothesis when it fails to cover a positive example

Algorithm:

1. Initialize \mathbf{h} to the most specific hypothesis
2. For each positive training example \mathbf{X}
 - For each value \mathbf{a} in \mathbf{h}
 - If example \mathbf{X} and \mathbf{h} agree on \mathbf{a} , do nothing
 - Else generalize \mathbf{a} by the next more general constraint
3. Output hypothesis \mathbf{h}

Example:

Let's run the learning algorithm above with the following examples:

((red,small,round,humid,low,smooth),	poisonous)
((red,small,elongated,humid,low,smooth),	poisonous)
((gray,large,elongated,humid,low,rough),	not-poisonous)
((red,small,elongated,humid,high,rough),	poisonous)

We start with the most specific hypothesis: $\mathbf{h} = (0,0,0,0,0,0)$

The first example comes and since the example is positive and **h** fails to cover it, we simply generalize **h** to cover exactly this example:

$$h = (\text{red}, \text{small}, \text{round}, \text{humid}, \text{low}, \text{smooth})$$

Hypothesis **h** basically says that the first example is the only positive example, all other examples are negative.

Then comes examples 2: ((red,small,elongated,humid,low,smooth), poisonous)

This example is positive. All attributes match hypothesis **h** except for attribute shape: it has the value *elongated*, not *round*. We generalize this attribute using symbol ? yielding:

$$h: (\text{red}, \text{small}, ?, \text{humid}, \text{low}, \text{smooth})$$

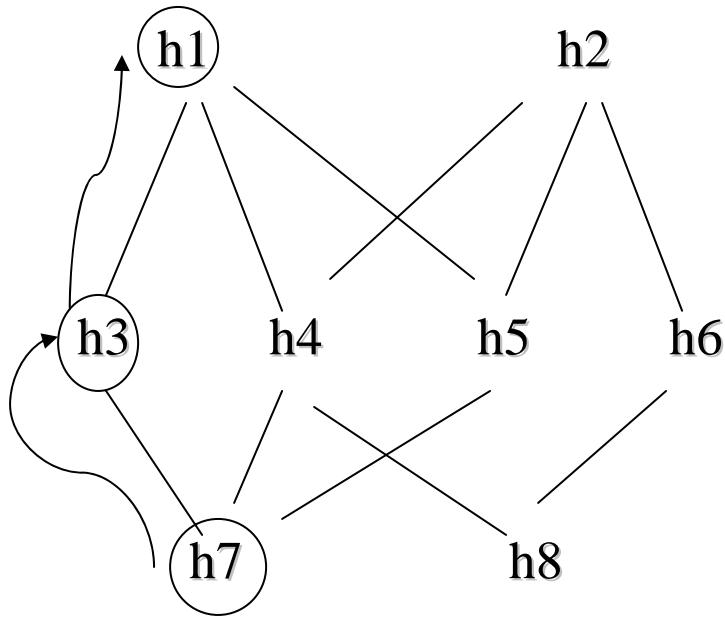
The third example is negative and so we just ignore it.

Why is it we don't need to be concerned with negative examples?

Upon observing the 4th example, hypothesis **h** is generalized to the following:

$$h = (\text{red}, \text{small}, ?, \text{humid}, ?, ?)$$

h is interpreted as any mushroom that is red, small and found on humid land should be classified as poisonous.



The algorithm is guaranteed to find the hypothesis that is most specific and consistent with the set of training examples. It takes advantage of the general-specific ordering to move on the corresponding lattice searching for the next most specific hypothesis.

Note that:

There are many hypotheses consistent with the training data D. Why should we prefer the most specific hypothesis?

What would happen if the examples are not consistent? What would happen if they have errors, noise?

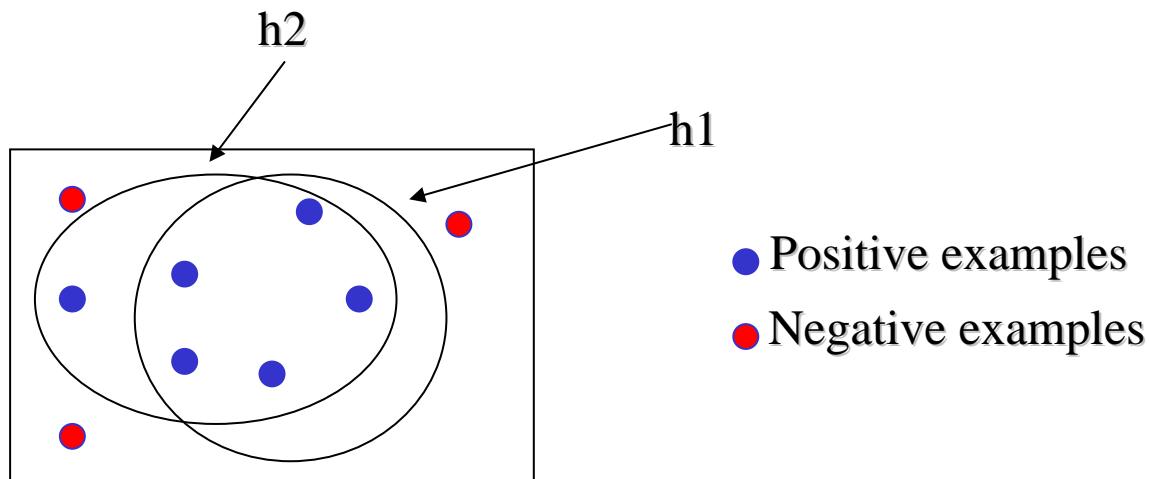
What if there is a hypothesis space H where one can find more than one maximally specific hypothesis h? The search over the lattice must then be different to allow for this possibility.

- The algorithm that finds the maximally specific hypothesis is limited in that it only finds one of many hypotheses consistent with the training data.
- The Candidate Elimination Algorithm (CEA) finds ALL hypotheses consistent with the training data.
- CEA does that without explicitly enumerating all consistent hypotheses.
- Applications:
 - Chemical Mass Spectroscopy
 - Control Rules for Heuristic Search

12.2.2 Candidate Elimination Algorithm

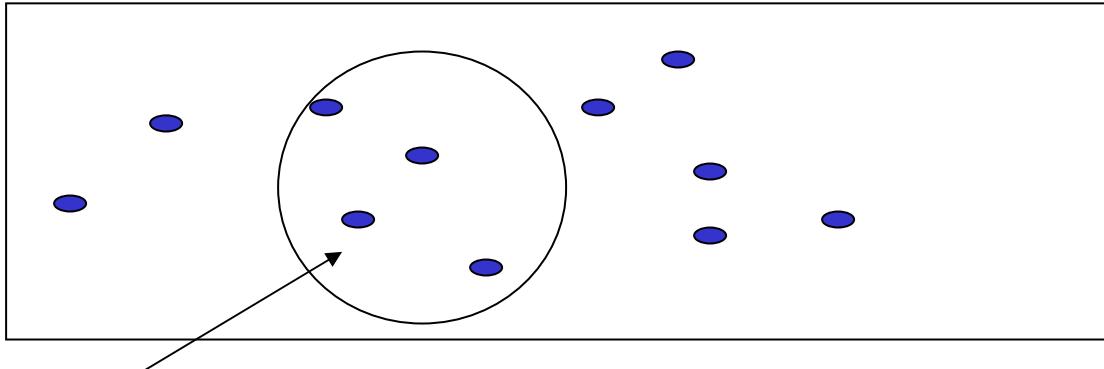
Consistency vs Coverage

In the following example, h1 covers a different set of examples than h2, h2 is consistent with training set D, h1 is not consistent with training set D



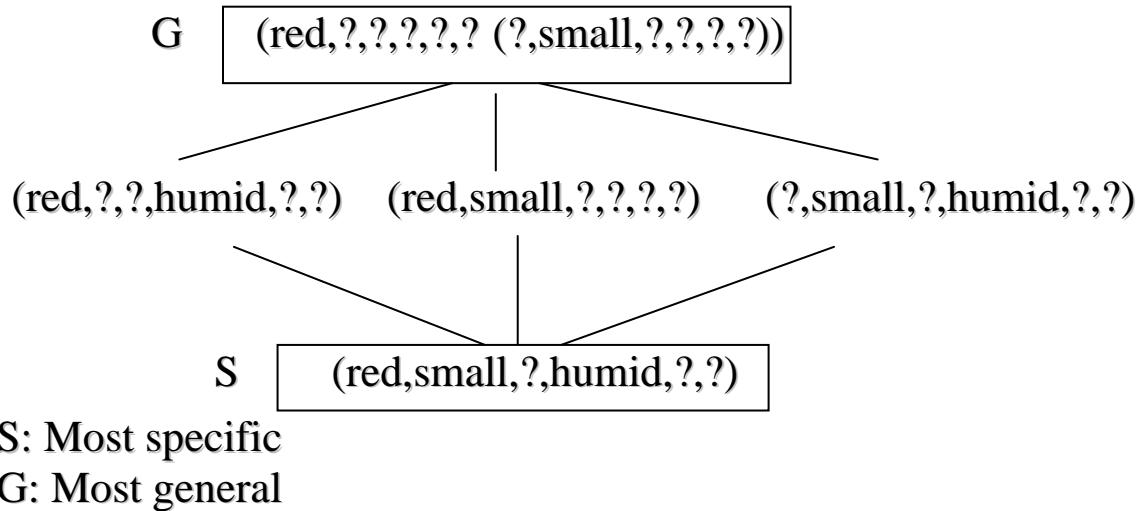
12.2.2.1 Version Space

Hypothesis space \mathcal{H}



Version space: Subset of hypothesis from \mathcal{H} consistent with training set \mathcal{D} .

The version space for the mushroom example is as follows:



The candidate elimination algorithm generates the entire version space.

12.2.2.2 The Candidate-Elimination Algorithm

The candidate elimination algorithm keeps two lists of hypotheses consistent with the training data: (i) The list of most specific hypotheses \mathbf{S} and, (ii) The list of most general hypotheses \mathbf{G} . This is enough to derive the whole version space \mathbf{VS} .

Steps:

1. Initialize \mathbf{G} to the set of maximally general hypotheses in \mathbf{H}
2. Initialize \mathbf{S} to the set of maximally specific hypotheses in \mathbf{H}
3. For each training example X do
 - a) If X is positive: generalize \mathbf{S} if necessary
 - b) If X is negative: specialize \mathbf{G} if necessary
4. Output $\{\mathbf{G}, \mathbf{S}\}$

Step (a) Positive examples

If X is positive:

- Remove from \mathbf{G} any hypothesis inconsistent with X
- For each hypothesis h in \mathbf{S} not consistent with X
 - Remove h from \mathbf{S}
 - Add all minimal generalizations of h consistent with X such that some member of \mathbf{G} is more general than h
 - Remove from \mathbf{S} any hypothesis more general than any other hypothesis in \mathbf{S}

Step (b) Negative examples

If X is negative:

- Remove from \mathbf{S} any hypothesis inconsistent with X
- For each hypothesis h in \mathbf{G} not consistent with X
 - Remove g from \mathbf{G}
 - Add all minimal generalizations of h consistent with X such that some member of \mathbf{S} is more specific than h
 - Remove from \mathbf{G} any hypothesis less general than any other hypothesis in \mathbf{G}

The candidate elimination algorithm is guaranteed to converge to the right hypothesis provided the following:

- a) No errors exist in the examples
- b) The target concept is included in the hypothesis space \mathbf{H}

If there exists errors in the examples:

- a) The right hypothesis would be inconsistent and thus eliminated.
- b) If the \mathbf{S} and \mathbf{G} sets converge to an empty space we have evidence that the true concept lies outside space \mathbf{H} .

Module 12

Machine Learning

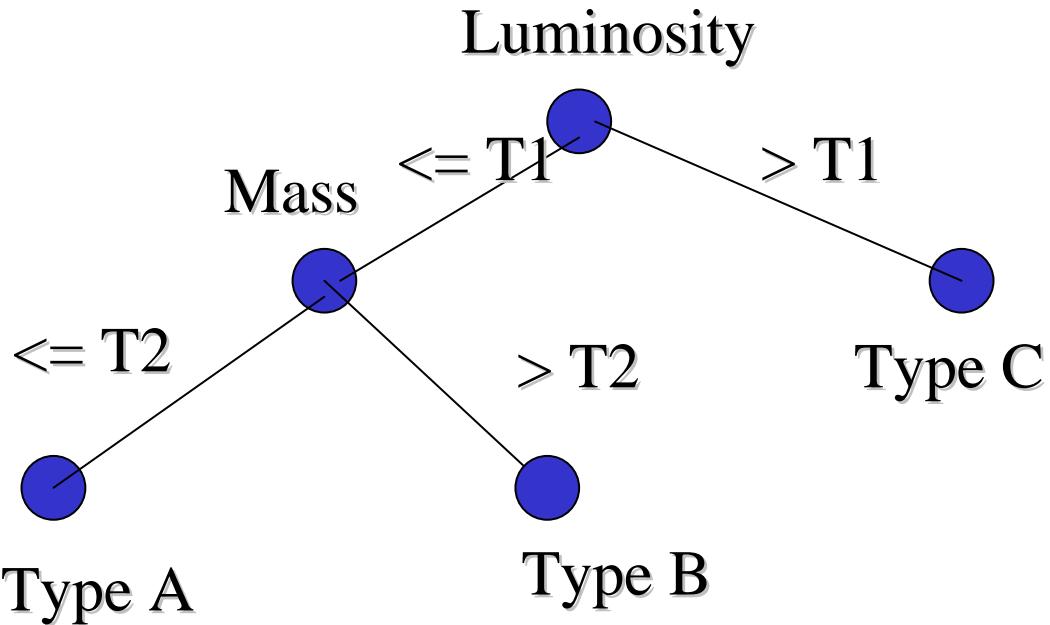
Version 2 CSE IIT, Kharagpur

Lesson 35

Rule Induction and Decision Tree - I

12.3 Decision Trees

Decision trees are a class of learning models that are more robust to noise as well as more powerful as compared to concept learning. Consider the problem of classifying a star based on some astronomical measurements. It can naturally be represented by the following set of decisions on each measurement arranged in a tree like fashion.



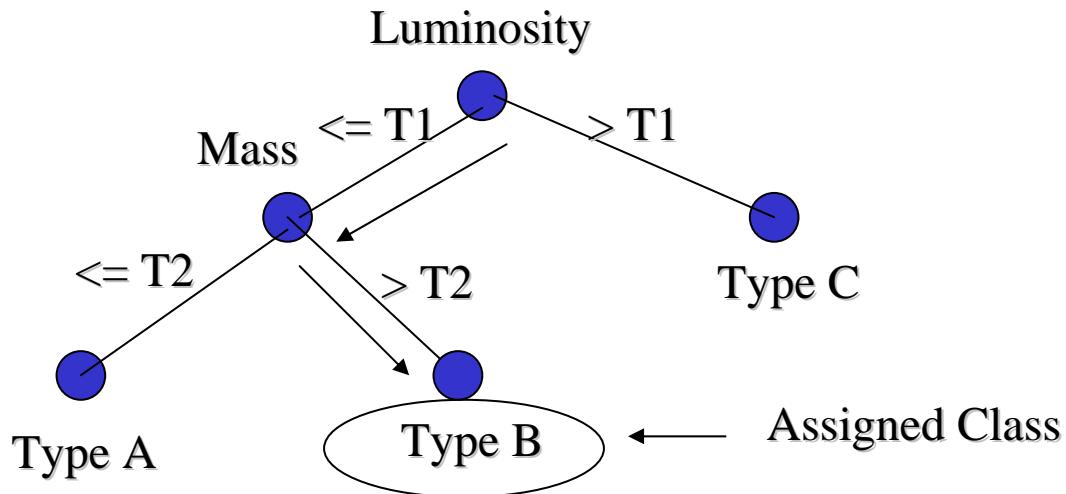
12.3.1 Decision Tree: Definition

- A decision-tree learning algorithm approximates a target concept using a tree representation, where each internal node corresponds to an attribute, and every terminal node corresponds to a class.
- There are two types of nodes:
 - Internal node.- Splits into different branches according to the different values the corresponding attribute can take. Example: luminosity $\leq T_1$ or luminosity $> T_1$.
 - Terminal Node.- Decides the class assigned to the example.

12.3.2 Classifying Examples Using Decision Tree

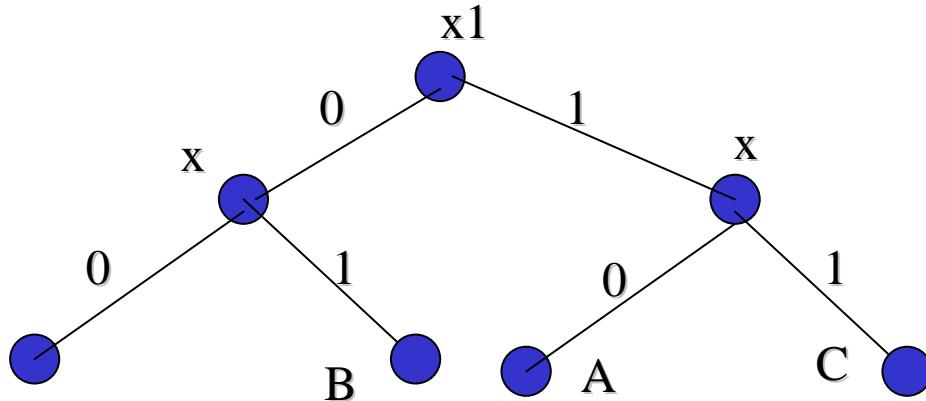
To classify an example X we start at the root of the tree, and check the value of that attribute on X . We follow the branch corresponding to that value and jump to the next node. We continue until we reach a terminal node and take that class as our best prediction.

$$X = (\text{Luminosity} \leq T1, \text{Mass} > T2)$$



Decision trees adopt a DNF (Disjunctive Normal Form) representation. For a fixed class, every branch from the root of the tree to a terminal node with that class is a conjunction of attribute values; different branches ending in that class form a disjunction.

In the following example, the rules for class A are: $(\sim X1 \& \sim x2) \text{ OR } (X1 \& \sim x3)$



12.3.3 Decision Tree Construction

There are different ways to construct trees from data. We will concentrate on the top-down, greedy search approach:

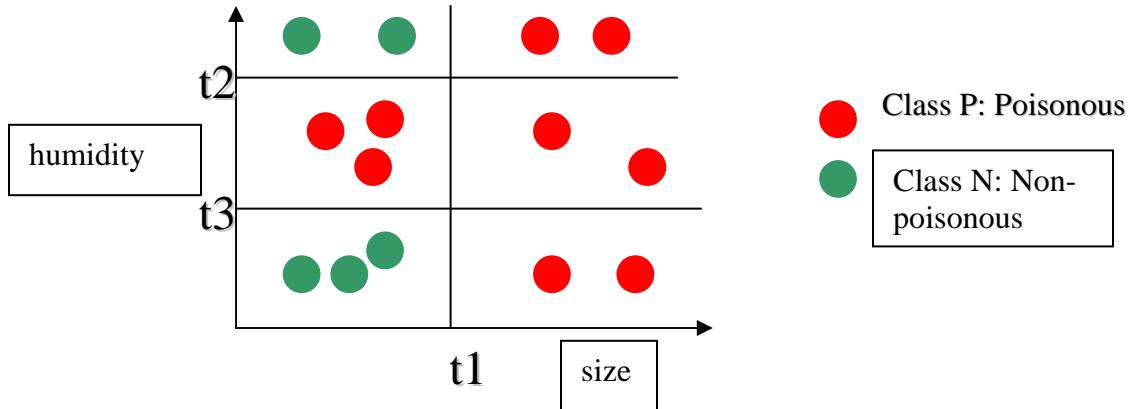
Basic idea:

1. Choose the best attribute a^* to place at the root of the tree.

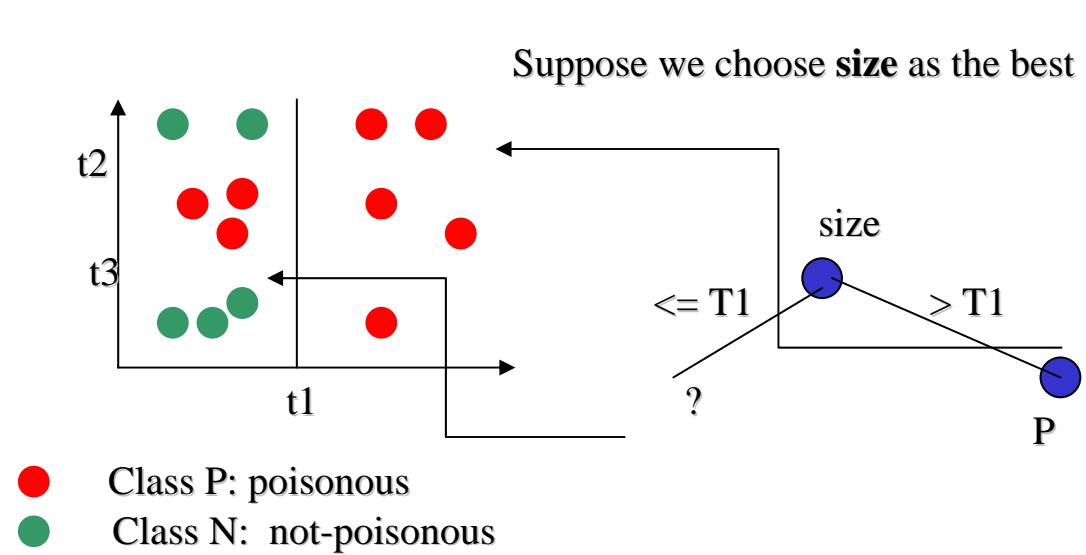
2. Separate training set D into subsets $\{D_1, D_2, \dots, D_k\}$ where each subset D_i contains examples having the same value for a^*

3. Recursively apply the algorithm on each new subset until examples have the same class or there are few of them.

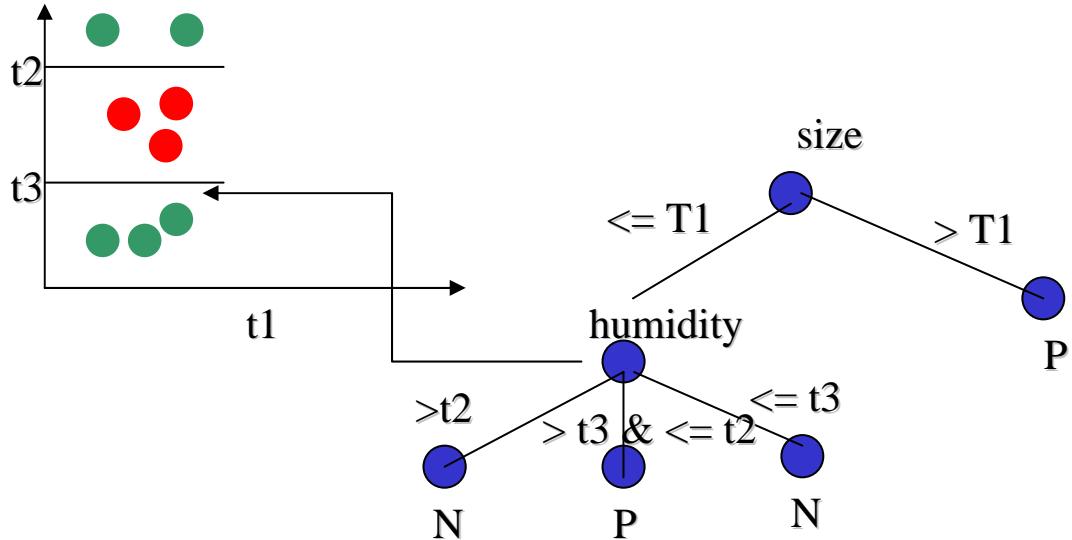
Illustration:



Suppose we choose **size** as the best



Suppose we choose **humidity** as the next best



Steps:

- Create a root for the tree
- If all examples are of the same class or the number of examples is below a threshold return that class
- If no attributes available return majority class
- Let a^* be the best attribute
- For each possible value v of a^*
 - Add a branch below a^* labeled “ $a = v$ ”
 - Let S_v be the subsets of example where attribute $a^*=v$
 - Recursively apply the algorithm to S_v

Module 12

Machine Learning

Version 2 CSE IIT, Kharagpur

Lesson 36

Rule Induction and Decision Tree - II

Splitting Functions

What attribute is the best to split the data? Let us remember some definitions from information theory.

A measure of uncertainty or entropy that is associated to a random variable X is defined as

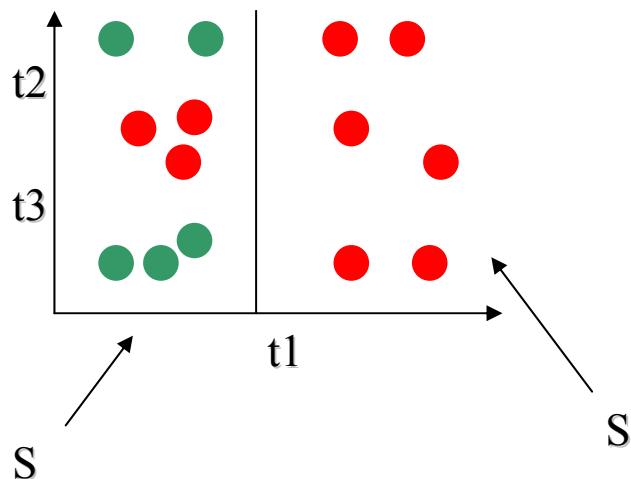
$$H(X) = - \sum p_i \log p_i$$

where the logarithm is in base 2.

This is the “average amount of information or entropy of a finite complete probability scheme”.

We will use a entropy based splitting function.

Consider the previous example:



Size divides the sample in two.

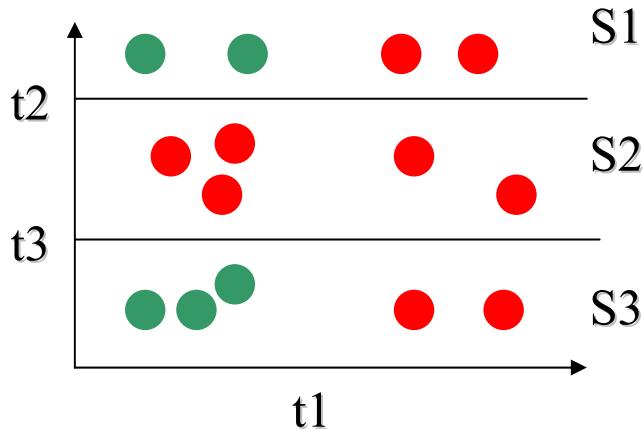
$$S_1 = \{ 6P, 0NP \}$$

$$S_2 = \{ 3P, 5NP \}$$

$$H(S_1) = 0$$

$$H(S_2) = -(3/8)\log_2(3/8)$$

$$-(5/8)\log_2(5/8)$$



humidity divides the sample in three.

$$S_1 = \{2P, 2NP\}$$

$$S_2 = \{5P, 0NP\}$$

$$S_3 = \{2P, 3NP\}$$

$$H(S_1) = 1$$

$$H(S_2) = 0$$

$$H(S_3) = -(2/5)\log_2(2/5)$$

$$-(3/5)\log_2(3/5)$$

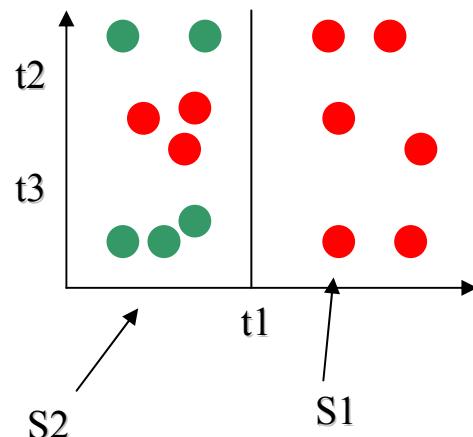
Let us define *information gain* as follows:

Information gain IG over attribute A: $IG(A)$

$$IG(A) = H(S) - \sum_v (S_v/S) H(S_v)$$

$H(S)$ is the entropy of all examples. $H(S_v)$ is the entropy of one subsample after partitioning S based on all possible values of attribute A.

Consider the previous example:



We have,

$$H(S1) = 0$$

$$H(S2) = -(3/8)\log_2(3/8)$$

$$-(5/8)\log_2(5/8)$$

$$H(S) = -(9/14)\log_2(9/14)$$

$$-(5/14)\log_2(5/14)$$

$$|S1|/|S| = 6/14$$

$$|S2|/|S| = 8/14$$

The principle for decision tree construction may be stated as follows:

Order the splits (attribute and value of the attribute) in decreasing order of information gain.

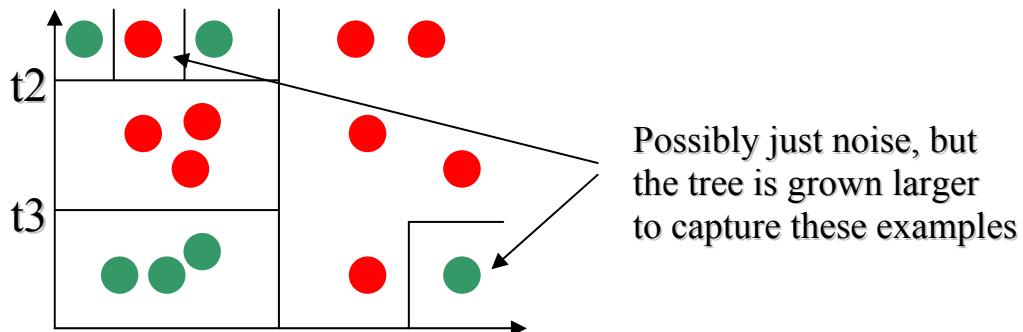
12.3.4 Decision Tree Pruning

Practical issues while building a decision tree can be enumerated as follows:

- 1) How deep should the tree be?
- 2) How do we handle continuous attributes?
- 3) What is a good splitting function?
- 4) What happens when attribute values are missing?
- 5) How do we improve the computational efficiency

The depth of the tree is related to the generalization capability of the tree. If not carefully chosen it may lead to overfitting.

A tree *overfits* the data if we let it grow deep enough so that it begins to capture “aberrations” in the data that harm the predictive power on unseen examples:



There are two main solutions to overfitting in a decision tree:

- 1) Stop the tree early before it begins to overfit the data.

+ In practice this solution is hard to implement because it is not clear what is a good stopping point.

2) Grow the tree until the algorithm stops even if the overfitting problem shows up. Then prune the tree as a post-processing step.

+ This method has found great popularity in the machine learning community.

A common *decision tree pruning algorithm* is described below.

1. Consider all internal nodes in the tree.
2. For each node check if removing it (along with the subtree below it) and assigning the most common class to it does not harm accuracy on the validation set.
3. Pick the node n^* that yields the best performance and prune its subtree.
4. Go back to (2) until no more improvements are possible.

Decision trees are appropriate for problems where:

- Attributes are both numeric and nominal.
- Target function takes on a discrete number of values.
- A DNF representation is effective in representing the target concept.
- Data may have errors.

Module 12

Machine Learning

Version 2 CSE IIT, Kharagpur

Lesson 37

Learning and Neural
Networks - I

12.4 Neural Networks

Artificial neural networks are among the most powerful learning models. They have the versatility to approximate a wide range of complex functions representing multi-dimensional input-output maps. Neural networks also have inherent adaptability, and can perform robustly even in noisy environments.

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired by the way biological nervous systems, such as the brain, process information. The key element of this paradigm is the novel structure of the information processing system. It is composed of a large number of highly interconnected simple processing elements (neurons) working in unison to solve specific problems. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning in biological systems involves adjustments to the synaptic connections that exist between the neurons. This is true of ANNs as well. ANNs can process information at a great speed owing to their highly massive parallelism.

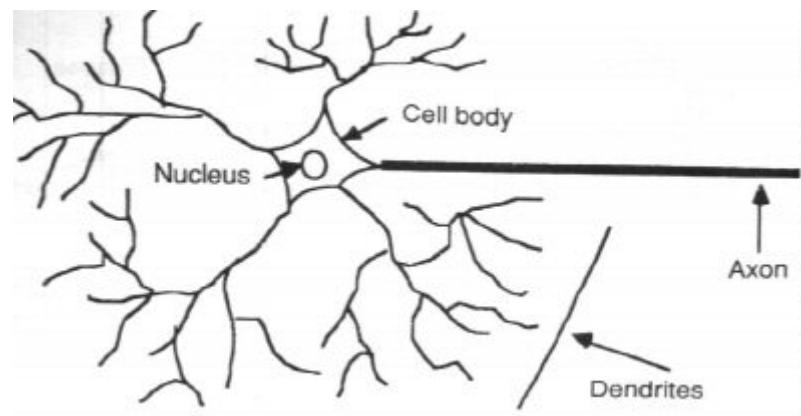
Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyse. This expert can then be used to provide projections given new situations of interest and answer "what if" questions. Other advantages include:

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

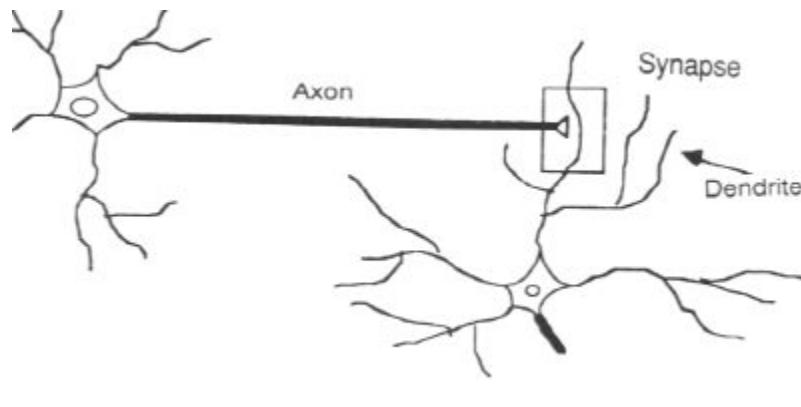
12.4.1 Biological Neural Networks

Much is still unknown about how the brain trains itself to process information, so theories abound. In the human brain, a typical neuron collects signals from others through a host of fine structures called *dendrites*. The neuron sends out spikes of electrical activity through a long, thin stand known as an *axon*, which splits into thousands of branches. At the end of each branch, a structure called a *synapse* converts the activity from the axon into electrical effects that inhibit or excite activity from the axon into electrical effects that inhibit or excite activity in the connected neurones. When a neuron receives

excitatory input that is sufficiently large compared with its inhibitory input, it sends a spike of electrical activity down its axon. Learning occurs by changing the effectiveness of the synapses so that the influence of one neuron on another changes.



Components of a Biological Neuron



The Synapse

12.4.2 Artificial Neural Networks

Artificial neural networks are represented by a set of nodes, often arranged in layers, and a set of weighted directed links connecting them. The nodes are equivalent to neurons, while the links denote synapses. The nodes are the information processing units and the links acts as communicating media.

There are a wide variety of networks depending on the nature of information processing carried out at individual nodes, the topology of the links, and the algorithm for adaptation of link weights. Some of the popular among them include:

Perceptron: This consists of a single neuron with multiple inputs and a single output. It has restricted information processing capability. The information processing is done through a transfer function which is either linear or non-linear.

Multi-layered Perceptron (MLP): It has a layered architecture consisting of input, hidden and output layers. Each layer consists of a number of perceptrons. The output of each layer is transmitted to the input of nodes in other layers through weighted links. Usually, this transmission is done only to nodes of the next layer, leading to what are known as feed forward networks. MLPs were proposed to extend the limited information processing capabilities of simple perceptrons, and are highly versatile in terms of their approximation ability. Training or weight adaptation is done in MLPs using supervised backpropagation learning.

Recurrent Neural Networks: RNN topology involves backward links from output to the input and hidden layers. The notion of time is encoded in the RNN information processing scheme. They are thus used in applications like speech processing where inputs are time sequences data.

Self-Organizing Maps: SOMs or Kohonen networks have a grid topology, with unequal grid weights. The topology of the grid provides a low dimensional visualization of the data distribution. These are thus used in applications which typically involve organization and human browsing of a large volume of data. Learning is performed using a winner take all strategy in an unsupervised mode.

In this module we will discuss perceptrons and multi layered perceptrons.

Module 12

Machine Learning

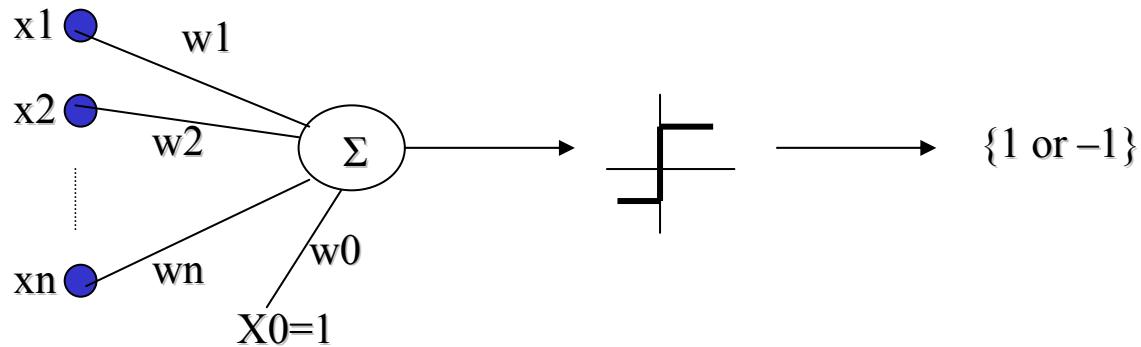
Version 2 CSE IIT, Kharagpur

Lesson 38

Neural Networks - II

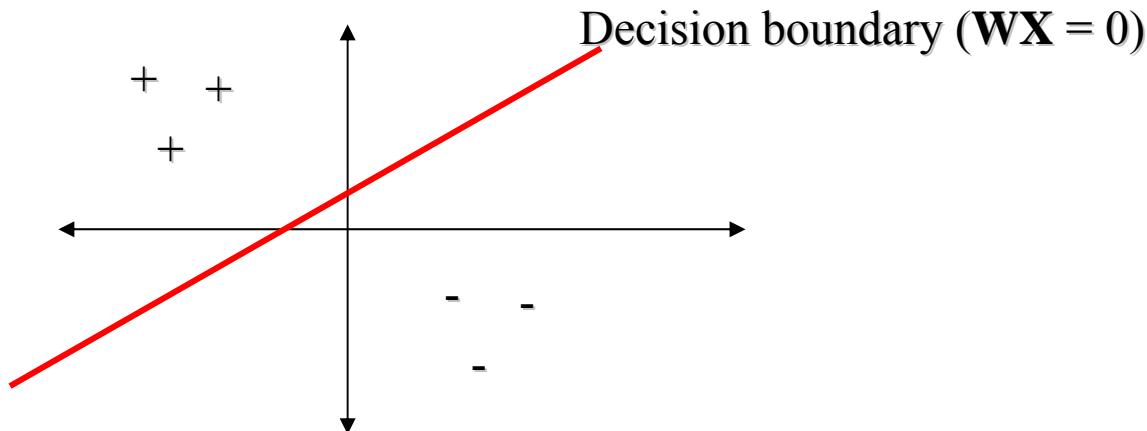
12.4.3 Perceptron

Definition: It's a step function based on a linear combination of real-valued inputs. If the combination is above a threshold it outputs a 1, otherwise it outputs a -1.

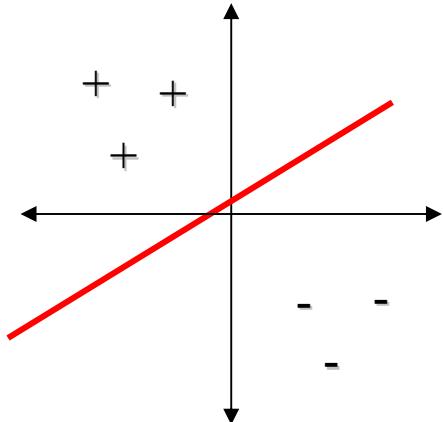


$$O(x_1, x_2, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1x_1 + w_2x_2 + \dots + w_nx_n > 0 \\ -1 & \text{otherwise} \end{cases}$$

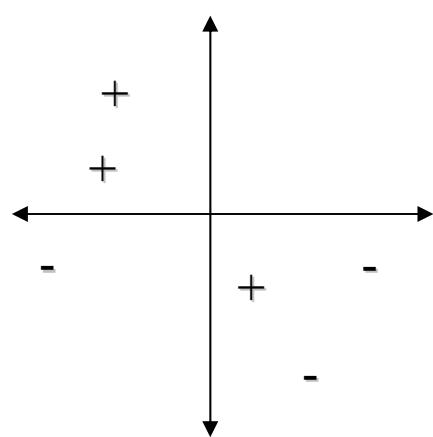
A perceptron draws a hyperplane as the decision boundary over the (n-dimensional) input space.



A perceptron can learn only examples that are called “linearly separable”. These are examples that can be perfectly separated by a hyperplane.



Linearly separable

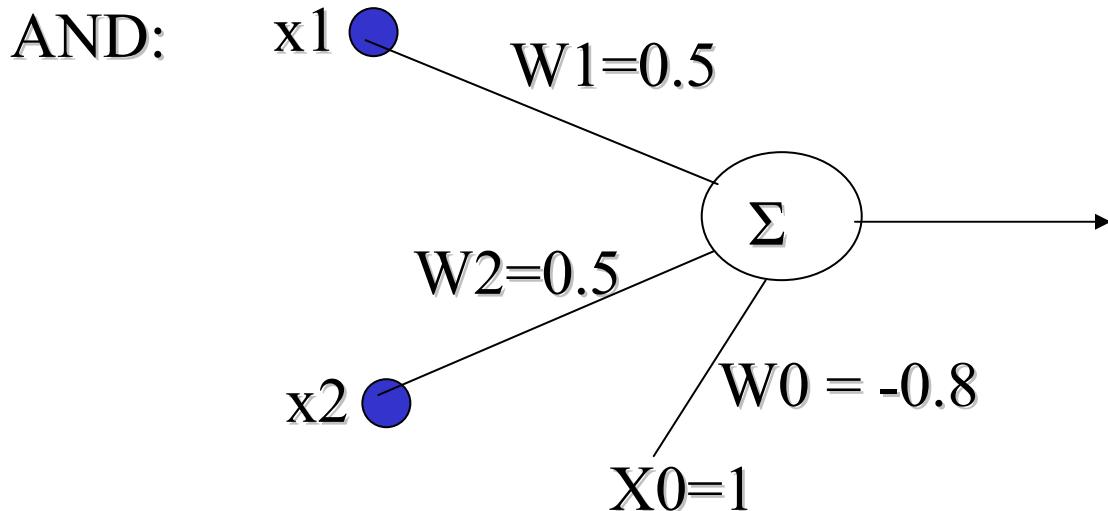


Non-linearly separable

Perceptrons can learn many boolean functions: AND, OR, NAND, NOR, but not XOR

However, every boolean function can be represented with a perceptron network that has two levels of depth or more.

The weights of a perceptron implementing the AND function is shown below.



12.4.3.1 Perceptron Learning

Learning a perceptron means finding the right values for W . The hypothesis space of a perceptron is the space of all weight vectors. The perceptron learning algorithm can be stated as below.

1. Assign random values to the weight vector
2. Apply the *weight update rule* to every training example
3. Are all training examples correctly classified?
 - a. Yes. Quit
 - b. No. Go back to Step 2.

There are two popular weight update rules.

- i) The perceptron rule, and
- ii) Delta rule

The Perceptron Rule

For a new training example $X = (x_1, x_2, \dots, x_n)$, update each weight according to this rule:

$$w_i = w_i + \Delta w_i$$

Where $\Delta w_i = \eta (t - o) x_i$

t: target output

o: output generated by the perceptron

η : constant called the learning rate (e.g., 0.1)

Comments about the perceptron training rule:

- If the example is correctly classified the term $(t - o)$ equals zero, and no update on the weight is necessary.
- If the perceptron outputs -1 and the real answer is 1 , the weight is increased.
- If the perceptron outputs a 1 and the real answer is -1 , the weight is decreased.
- Provided the examples are linearly separable and a small value for η is used, the rule is proved to classify all training examples correctly (i.e, is consistent with the training data).

The Delta Rule

What happens if the examples are not linearly separable?

To address this situation we try to approximate the real concept using the delta rule.

The key idea is to use a *gradient descent search*. We will try to minimize the following error:

$$E = \frac{1}{2} \sum_i (t_i - o_i)^2$$

where the sum goes over all training examples. Here o_i is the inner product WX and not $\text{sgn}(WX)$ as with the perceptron rule. The idea is to find a minimum in the space of weights and the error function E.

The delta rule is as follows:

For a new training example $X = (x_1, x_2, \dots, x_n)$, update each weight according to this rule:

$$w_i = w_i + \Delta w_i$$

Where $\Delta w_i = -\eta E'(W)/w_i$

η : learning rate (e.g., 0.1)

It is easy to see that

$$E'(W)/w_i = \sum_j (t_j - o_j) (-x_j)$$

So that gives us the following equation:

$$w_i = \eta \sum_j (t_j - o_j) x_j$$

There are two differences between the perceptron and the delta rule. The perceptron is based on an output from a step function, whereas the delta rule uses the linear combination of inputs directly. The perceptron is guaranteed to converge to a consistent hypothesis assuming the data is linearly separable. The delta rule converges in the limit but it does not need the condition of linearly separable data.

There are two main difficulties with the gradient descent method:

1. Convergence to a minimum may take a long time.
2. There is no guarantee we will find the global minimum.

These are handled by using momentum terms and random perturbations to the weight vectors.

Module 12

Machine Learning

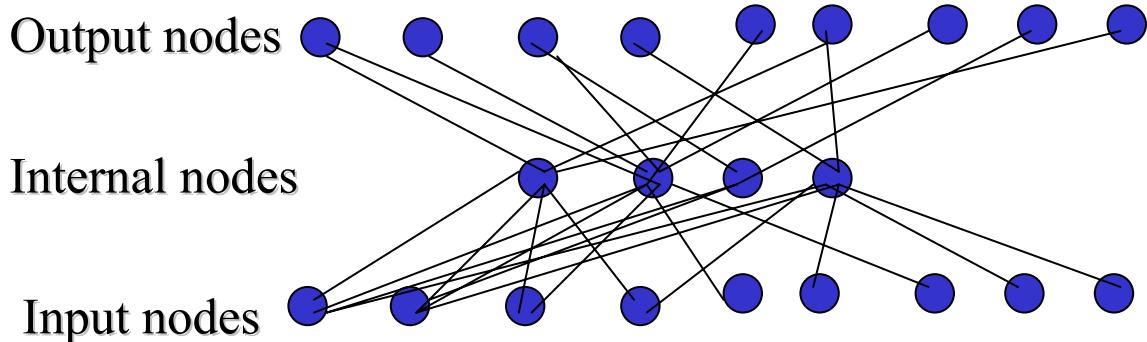
Version 2 CSE IIT, Kharagpur

Lesson 39

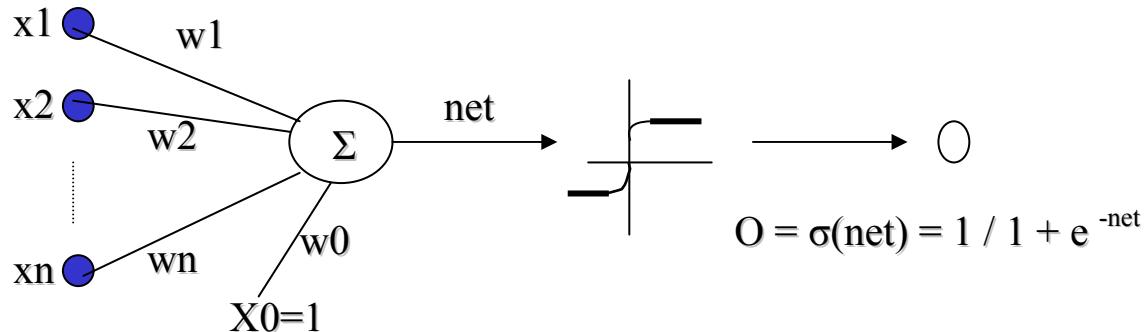
Neural Networks - III

12.4.4 Multi-Layer Perceptrons

In contrast to perceptrons, multilayer networks can learn not only multiple decision boundaries, but the boundaries may be nonlinear. The typical architecture of a multi-layer perceptron (MLP) is shown below.



To make nonlinear partitions on the space we need to define each unit as a nonlinear function (unlike the perceptron). One solution is to use the sigmoid unit. Another reason for using sigmoids are that they are continuous unlike linear thresholds and are thus differentiable at all points.



$$O(x_1, x_2, \dots, x_n) = \sigma(WX)$$

$$\text{where: } \sigma(WX) = 1 / (1 + e^{-WX})$$

Function σ is called the sigmoid or logistic function. It has the following property:

$$d\sigma(y) / dy = \sigma(y)(1 - \sigma(y))$$

12.4.4.1 Back-Propagation Algorithm

Multi-layered perceptrons can be trained using the back-propagation algorithm described next.

Goal: To learn the weights for all links in an interconnected multilayer network.

We begin by defining our measure of error:

$$E(W) = \frac{1}{2} \sum_d \sum_k (t_{kd} - o_{kd})^2$$

k varies along the output nodes and d over the training examples.

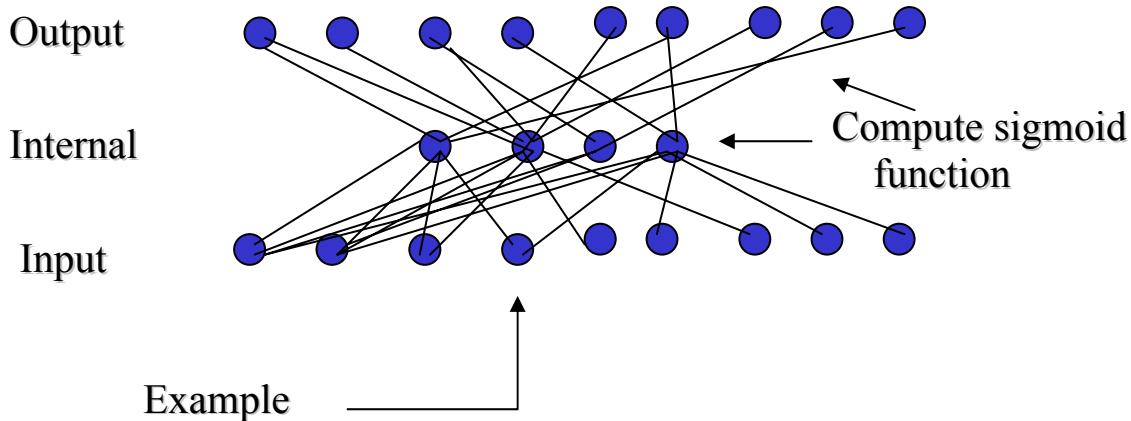
The idea is to use again a gradient descent over the space of weights to find a global minimum (no guarantee).

Algorithm:

1. Create a network with n_{in} input nodes, n_{hidden} internal nodes, and n_{out} output nodes.
2. Initialize all weights to small random numbers.
3. Until error is small do:
 - For each example X do
 - Propagate example X forward through the network
 - Propagate errors backward through the network

Forward Propagation

Given example X, compute the output of every node until we reach the output nodes:



Backward Propagation

- A. For each output node k compute the error:

$$\delta_k = O_k (1-O_k)(t_k - O_k)$$

- B. For each hidden unit h , calculate the error:

$$\delta_h = O_h (1-O_h) \sum_k W_{kh} \delta_k$$

- C. Update each network weight:

$W_{ji} = W_{ji} + \Delta W_{ji}$

where $\Delta W_{ji} = \eta \delta_j X_{ji}$ (W_{ji} and X_{ji} are the input and weight of node i to node j)

A momentum term, depending on the weight value at last iteration, may also be added to the update rule as follows. At iteration n we have the following:

$$\Delta W_{ji}(n) = \eta \delta_j X_{ji} + \alpha \Delta W_{ji}(n)$$

Where α ($0 <= \alpha <= 1$) is a constant called the momentum.

1. It increases the speed along a local minimum.
2. It increases the speed along flat regions.

Remarks on Back-propagation

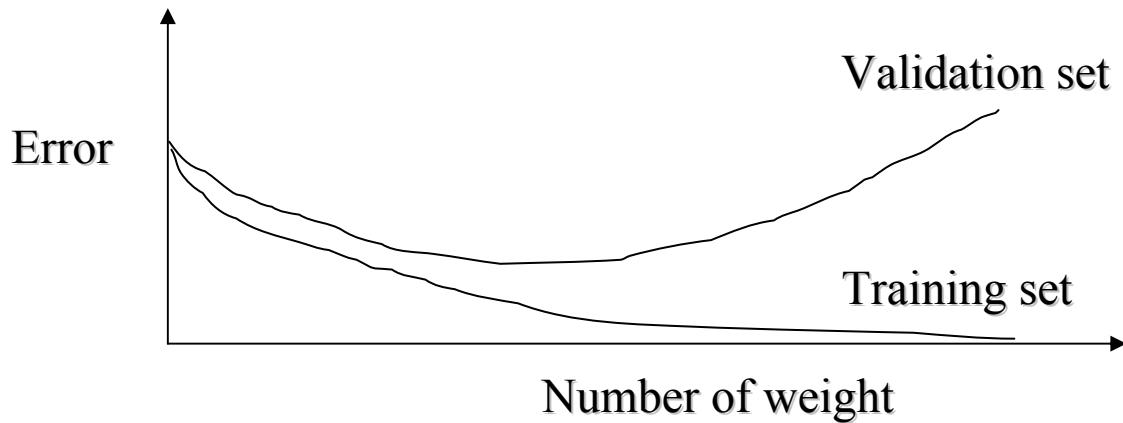
1. It implements a gradient descent search over the weight space.
2. It may become trapped in local minima.
3. In practice, it is very effective.
4. How to avoid local minima?
 - a) Add momentum.
 - b) Use stochastic gradient descent.
 - c) Use different networks with different initial values for the weights.

Multi-layered perceptrons have high representational power. They can represent the following:

1. Boolean functions. Every boolean function can be represented with a network having two layers of units.
2. Continuous functions. All bounded continuous functions can also be approximated with a network having two layers of units.
3. Arbitrary functions. Any arbitrary function can be approximated with a network with three layers of units.

Generalization and overfitting:

One obvious stopping point for backpropagation is to continue iterating until the error is below some threshold; this can lead to overfitting.



Overfitting can be avoided using the following strategies.

- Use a validation set and stop until the error is small in this set.
- Use 10 fold cross validation.
- Use weight decay; the weights are decreased slowly on each iteration.

Applications of Neural Networks

Neural networks have broad applicability to real world business problems. They have already been successfully applied in many industries.

Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs including:

- sales forecasting
- industrial process control
- customer research
- data validation
- risk management
- target marketing

Because of their adaptive and non-linear nature they have also been used in a number of control system application domains like,

- process control in chemical plants
- unmanned vehicles
- robotics
- consumer electronics

Neural networks are also used in a number of other applications which are too hard to model using classical techniques. These include, computer vision, path planning and user modeling.

Questions

1. Assume a learning problem where each example is represented by four attributes. Each attribute can take two values in the set {a,b}. Run the candidate elimination algorithm on the following examples and indicate the resulting version space. What is the size of the space?

((a, b, b, a), +)
((b, b, b, b), +)
((a, b, b, b), +)
((a, b, a, a), -)

2. Decision Trees

The following table contains training examples that help a robot janitor predict whether or not an office contains a recycling bin.

STATUS DEPT. OFFICE SIZE RECYCLING BIN?

1. faculty	ee	large	no
2. staff	ee	small	no
3. faculty	cs	medium	yes
4. student	ee	large	yes
5. staff	cs	medium	no
6. faculty	cs	large	yes
7. student	ee	small	yes
8. staff	cs	medium	no

Use entropy splitting function to construct a minimal decision tree that predicts whether or not an office contains a recycling bin. Show each step of the computation.

3. Translate the above decision tree into a collection of decision rules.

4. Neural networks: Consider the following function of two variables:

A	B	Desired Output
0	0	1
1	0	0
0	1	0
1	1	1

Prove that this function cannot be learned by a single perceptron that uses the step function as its activation function.

4. Construct by hand a perceptron that can calculate the logic function implies (\Rightarrow). Assume that 1 = true and 0 = false for all inputs and outputs.

Solution

1. The general and specific boundaries of the version space are as follows:

$$\begin{aligned} S: & \{(_, b, b, _)\} \\ G: & \{(_, _, b, _)\} \end{aligned}$$

The size of the version space is 2.

2. The solution steps are as follows:

1. **Selecting the attribute for the root node:**

YES	NO
-----	----

status:

$$\begin{aligned} \text{Faculty: } & (3/8) * [-(2/3) * \log_2(2/3) - (1/3) * \log_2(1/3)] \\ \text{Staff: } & + (3/8) * [-(0/3) * \log_2(0/3) - (3/3) * \log_2(3/3)] \\ \text{Student: } & + (2/8) * [-(2/2) * \log_2(2/2) - (0/2) * \log_2(0/2)] \\ \text{TOTAL: } & = 0.35 + 0 + 0 = 0.35 \end{aligned}$$

size:

$$\begin{aligned} \text{Large: } & (3/8) * [-(2/3) * \log_2(2/3) - (1/3) * \log_2(1/3)] \\ \text{Medium: } & + (3/8) * [-(1/3) * \log_2(1/3) - (2/3) * \log_2(2/3)] \\ \text{Small: } & + (2/8) * [-(1/2) * \log_2(1/2) - (1/2) * \log_2(1/2)] \\ \text{TOTAL: } & = 0.35 + 0.35 + 2/8 = 0.95 \end{aligned}$$

Dept:

```

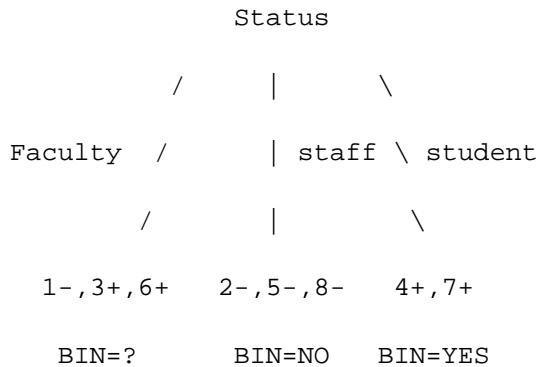
ee:          (4/8)*[ -(2/4)*log_2(2/4) - (2/4)*log_2(2/4) ]

cs:          + (4/8)*[ -(2/4)*log_2(2/4) - (2/4)*log_2(2/4) ]

TOTAL:      = 0.5 + 0.5 = 1

```

Since **status** is the attribute with the lowest entropy, it is selected as the root node:



Only the branch corresponding to Status=Faculty needs further processing.

Selecting the attribute to split the branch Status=Faculty:

YES _____ **NO** _____

Dept :

```

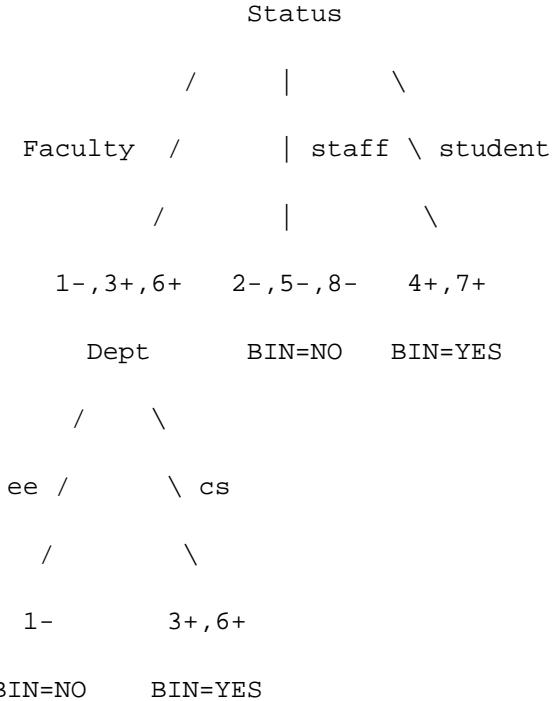
ee:           (1/3)*[ -(0/1)*log_2(0/1) - (1/1)*log_2(1/1) ]

cs:           + (2/3)*[ -(2/2)*log_2(2/2) - (0/2)*log_2(0/2) ]

TOTAL:      = 0 + 0 = 0

```

Since the minimum possible entropy is 0 and **dept** has that minimum possible entropy, it is selected as the best attribute to split the branch Status=Faculty. Note that we don't even have to calculate the entropy of the attribute **size** since it cannot possibly be better than 0.

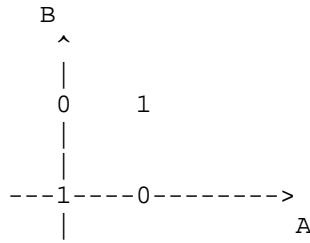


Each branch ends in a homogeneous set of examples so the construction of the decision tree ends here.

3. The rules are:

```
IF status=faculty AND dept=ee THEN recycling-bin=no
IF status=faculty AND dept=cs THEN recycling-bin=yes
IF status=staff THEN recycling-bin=no
IF status=student THEN recycling-bin=yes
```

4. This function cannot be computed using a single perceptron since the function is not linearly separable:



5. Let

$$a_i = g(in_i) = g\left(\sum_{j=0}^n W_{j,i}a_j\right), \quad a_0 = -1.$$

Choose g to be the threshold function (output 1 on input > 0 , output 0 otherwise). There are two inputs to the implies function, $X_1 \Rightarrow X_2$. Therefore,

$$\sum_{j=0}^n W_{j,i}a_j = -W_0 + W_1X_1 + W_2X_2$$

Arbitrarily **choose** $W_2 = 1$, and substitute into above for all values of X_1 and X_2 .

X_1	X_2	$X_1 \Rightarrow X_2$	$-W_0 + W_1X_1 + (1)X_2$
0	0	1	$-W_0 > 0 \quad \therefore W_0 < 0$
0	1	1	$-W_0 + 1 > 0 \quad \therefore W_0 < 1$
1	0	0	$-W_0 + W_1 < 0 \quad \therefore W_0 > W_1$
1	1	1	$-W_0 + W_1 + 1 > 0 \quad \therefore W_0 - W_1 < 1$

Note that greater/less than signs are decided by the threshold fn. and the result of $X_1 \Rightarrow X_2$. For example, when $X_1=0$ and $X_2=0$, the resulting equation must be greater than 0 since g is threshold fn. and $X_1 \Rightarrow X_2$ is True. Similarly, when $X_1=1$ and $X_2=0$, the resulting equation must be less than 0 since g is the threshold fn. and $X_1 \Rightarrow X_2$ is False.

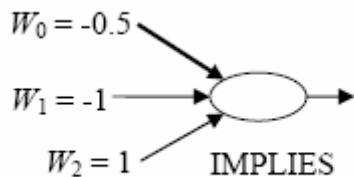
Since $W_0 < 0$, and $W_0 > W_1$, and $W_0 - W_1 < 1$, **choose** $W_1 = -1$. Substitute where possible.

X_1	X_2	$X_1 \Rightarrow X_2$	$-W_0 + (-1)X_1 + (1)X_2$
0	0	1	$W_0 < 0$
0	1	1	$W_0 < 1$
1	0	0	$W_0 > -1$
1	1	1	$W_0 - (-1) < 1 \therefore W_0 < 0$

Since $W_0 < 0$, and $W_0 > -1$, choose $W_0 = -0.5$.

Thus... $W_0 = -0.5$, $W_1 = -1$, $W_2 = 1$.

Finally,



Module 13

Natural Language Processing

Version 2 CSE IIT, Kharagpur

13.1 Instructional Objective

- The students should understand the necessity of natural language processing in building an intelligent system
- Students should understand the difference between natural and formal language and the difficulty in processing the former
- Students should understand the ambiguities that arise in natural language processing
- Students should understand the language information required like like
 - Phonology
 - Morphology
 - Syntax
 - Semantic
 - Discourse
 - World knowledge
- Students should understand the steps involved in natural language understanding and generation
- **The student should be familiar with basic language processing operations like**
 - Morphological analysis
 - Parts-of-Speech tagging
 - Lexical processing
 - Semantic processing
 - Knowledge representation

At the end of this lesson the student should be able to do the following:

- **Design the processing steps required for a NLP task**
- **Implement the processing techniques.**

Lesson 40

Issues in NLP

Version 2 CSE IIT, Kharagpur

13.1 Natural Language Processing

Natural Language Processing (NLP) is the process of computer analysis of input provided in a human language (natural language), and conversion of this input into a useful form of representation.

The field of NLP is primarily concerned with getting computers to perform useful and interesting tasks with human languages. The field of NLP is secondarily concerned with helping us come to a better understanding of human language.

- The input/output of a NLP system can be:
 - **written text**
 - **speech**
- We will mostly concerned with written text (not speech).
- To process written text, we need:
 - **lexical, syntactic, semantic knowledge about the language**
 - **discourse information, real world knowledge**
- To process spoken language, we need everything required to process written text, plus the challenges of speech recognition and speech synthesis.

There are two components of NLP.

- **Natural Language Understanding**
 - Mapping the given input in the natural language into a useful representation.
 - Different level of analysis required:
morphological analysis,
syntactic analysis,
semantic analysis,
discourse analysis, ...
- **Natural Language Generation**
 - Producing output in the natural language from some internal representation.
 - Different level of synthesis required:
deep planning (what to say),
syntactic generation
- NL Understanding is much harder than NL Generation. But, still both of them are hard.

The difficulty in NL understanding arises from the following facts:

- Natural language is extremely rich in form and structure, and **very ambiguous**.
 - How to represent meaning,
 - Which structures map to which meaning structures.
- One input can mean many different things. Ambiguity can be at different levels.

- Lexical (word level) ambiguity -- different meanings of words
- Syntactic ambiguity -- different ways to parse the sentence
- Interpreting partial information -- how to interpret pronouns
- Contextual information -- context of the sentence may affect the meaning of that sentence.
- Many input can mean the same thing.
- Interaction among components of the input is not clear.

The following language related information are useful in NLP:

- **Phonology** – concerns how words are related to the sounds that realize them.
- **Morphology** – concerns how words are constructed from more basic meaning units called morphemes. A morpheme is the primitive unit of meaning in a language.
- **Syntax** – concerns how can be put together to form correct sentences and determines what structural role each word plays in the sentence and what phrases are subparts of other phrases.
- **Semantics** – concerns what words mean and how these meaning combine in sentences to form sentence meaning. The study of context-independent meaning.
- **Pragmatics** – concerns how sentences are used in different situations and how use affects the interpretation of the sentence.
- **Discourse** – concerns how the immediately preceding sentences affect the interpretation of the next sentence. For example, interpreting pronouns and interpreting the temporal aspects of the information.
- **World Knowledge** – includes general knowledge about the world. What each language user must know about the other's beliefs and goals.

13.1.1 Ambiguity

I made her duck.

- How many different interpretations does this sentence have?
- What are the reasons for the ambiguity?
- The categories of knowledge of language can be thought of as ambiguity resolving components.
- How can each ambiguous piece be resolved?
- Does speech input make the sentence even more ambiguous?
 - Yes – deciding word boundaries
- Some interpretations of : **I made her duck.**

1. I cooked *duck* for her.
 2. I cooked *duck* belonging to her.
 3. I created a toy duck which she owns.
 4. I caused her to quickly lower her head or body.
 5. I used magic and turned her into a *duck*.
- duck – morphologically and syntactically ambiguous:
noun or verb.
 - her – syntactically ambiguous: dative or possessive.
 - make – semantically ambiguous: cook or create.
 - make – syntactically ambiguous:
 - Transitive – takes a direct object. => 2
 - Di-transitive – takes two objects. => 5
 - Takes a direct object and a verb. => 4

Ambiguities are resolved using the following methods.

- *models* and *algorithms* are introduced to resolve ambiguities at different levels.
- **part-of-speech tagging** -- Deciding whether duck is verb or noun.
- **word-sense disambiguation** -- Deciding whether make is create or cook.
- **lexical disambiguation** -- Resolution of part-of-speech and word-sense ambiguities are two important kinds of lexical disambiguation.
- **syntactic ambiguity** -- her duck is an example of syntactic ambiguity, and can be addressed by probabilistic parsing.

13.1.2 Models to represent Linguistic Knowledge

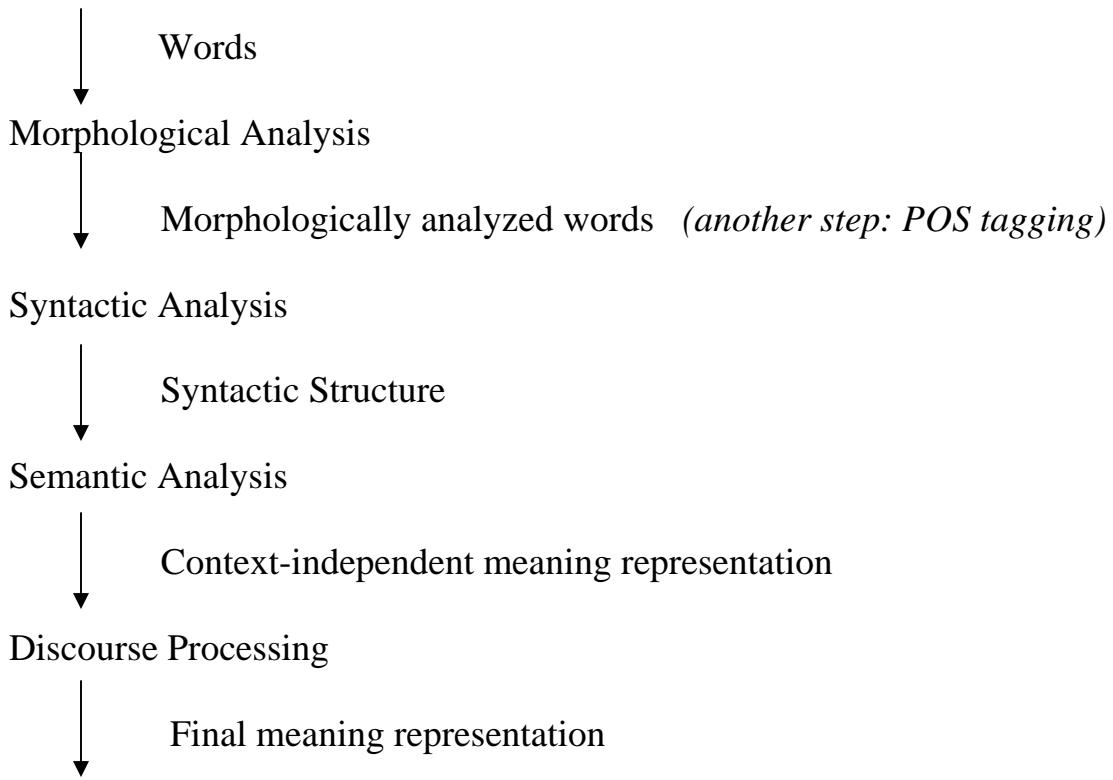
- We will use certain formalisms (*models*) to represent the required linguistic knowledge.
- **State Machines** -- FSAs, FSTs, HMMs, ATNs, RTNs
- **Formal Rule Systems** -- Context Free Grammars, Unification Grammars, Probabilistic CFGs.
- **Logic-based Formalisms** -- first order predicate logic, some higher order logic.
- **Models of Uncertainty** -- Bayesian probability theory.

13.1.3 Algorithms to Manipulate Linguistic Knowledge

- We will use *algorithms* to manipulate the models of linguistic knowledge to produce the desired behavior.
- Most of the algorithms we will study are **transducers** and **parsers**.
 - These algorithms construct some structure based on their input.
- Since the language is ambiguous at all levels, these algorithms are never simple processes.
- Categories of most algorithms that will be used can fall into following categories.
 - state space search
 - dynamic programming

13.2 Natural Language Understanding

The steps in natural language understanding are as follows:



Module 13

Natural Language Processing

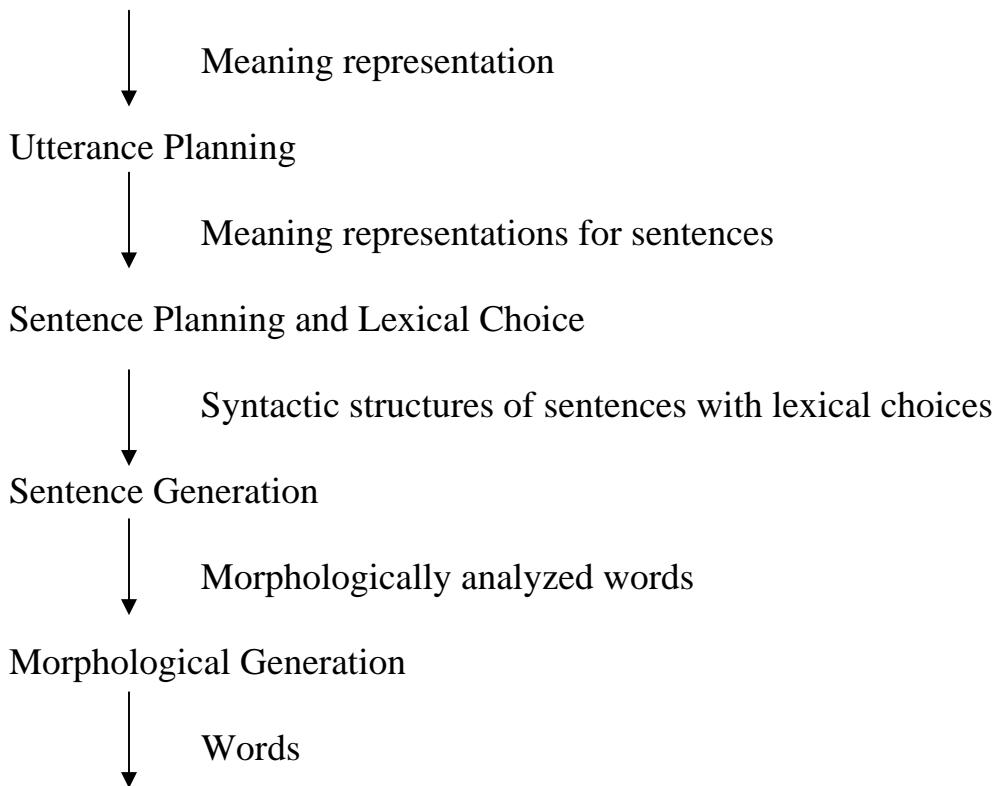
Version 2 CSE IIT, Kharagpur

Lesson 41

Parsing

13.3 Natural Language Generation

The steps in natural language generation are as follows.



13.4 Steps in Language Understanding and Generation

13.4.1 Morphological Analysis

- Analyzing words into their linguistic components (morphemes).
- Morphemes are the smallest meaningful units of language.

cars	car+PLU
giving	give+PROG
geliyordum	gel+PROG+PAST+1SG - I was coming

- Ambiguity: More than one alternatives

flies	flyVERB+PROG	
	flyNOUN+PLU	
adam	adam+ACC	- the man (accusative)
	adam+P1SG	- my man
	ada+P1SG+ACC	- my island (accusative)

13.4.2 Parts-of-Speech (POS) Tagging

- Each word has a part-of-speech tag to describe its category.
- Part-of-speech tag of a word is one of major word groups (or its subgroups).
 - **open classes** -- noun, verb, adjective, adverb
 - **closed classes** -- prepositions, determiners, conjunctions, pronouns, participles
- POS Taggers try to find POS tags for the words.
- duck is a verb or noun? (morphological analyzer cannot make decision).
- A POS tagger may make that decision by looking at the surrounding words.
 - Duck! (verb)
 - Duck is delicious for dinner. (noun)

13.4.3 Lexical Processing

- The purpose of lexical processing is to determine meanings of individual words.
- Basic methods is to lookup in a database of meanings – **lexicon**
- We should also identify non-words such as punctuation marks.
- Word-level ambiguity -- words may have several meanings, and the correct one cannot be chosen based solely on the word itself.
 - bank in English
- Solution -- resolve the ambiguity on the spot by POS tagging (if possible) or pass-on the ambiguity to the other levels.

13.4.4 Syntactic Processing

- **Parsing** -- converting a flat input sentence into a hierarchical structure that corresponds to the units of meaning in the sentence.
- There are different parsing formalisms and algorithms.
- Most formalisms have two main components:
 - **grammar** -- a declarative representation describing the syntactic structure of sentences in the language.
 - **parser** -- an algorithm that analyzes the input and outputs its structural representation (its parse) consistent with the grammar specification.

- CFGs are in the center of many of the parsing mechanisms. But they are complemented by some additional features that make the formalism more suitable to handle natural languages.

13.4.5 Semantic Analysis

- Assigning meanings to the structures created by syntactic analysis.
- Mapping words and structures to particular domain objects in way consistent with our knowledge of the world.
- Semantic can play an import role in selecting among competing syntactic analyses and discarding illogical analyses.
 - I robbed the bank -- bank is a river bank or a financial institution
- We have to decide the formalisms which will be used in the meaning representation.

13.5 Knowledge Representation for NLP

- Which knowledge representation will be used depends on the application -- Machine Translation, Database Query System.
- Requires the choice of representational framework, as well as the specific meaning vocabulary (what are concepts and relationship between these concepts -- ontology)
- Must be computationally effective.
- Common representational formalisms:
 - first order predicate logic
 - conceptual dependency graphs
 - semantic networks
 - Frame-based representations

13.6 Discourse

- Discourses are collection of coherent sentences (not arbitrary set of sentences)
- Discourses have also hierarchical structures (similar to sentences)
- **anaphora resolution** -- to resolve referring expression
 - Mary bought a book for Kelly. She didn't like it.
 - She refers to Mary or Kelly. -- possibly Kelly
 - It refers to what -- book.
 - Mary had to lie for Kelly. She didn't like it.

- Discourse structure may depend on application.
 - Monologue
 - Dialogue
 - Human-Computer Interaction

13.7 Applications of Natural Language Processing

- Machine Translation – Translation between two natural languages.
 - See the Babel Fish translations system on Alta Vista.
- Information Retrieval – Web search (uni-lingual or multi-lingual).
- Query Answering/Dialogue – Natural language interface with a database system, or a dialogue system.
- Report Generation – Generation of reports such as weather reports.
- Some Small Applications –
 - Grammar Checking, Spell Checking, Spell Corrector

13.8 Machine Translation

- Machine Translation refers to converting a text in language A into the corresponding text in language B (or speech).
- Different Machine Translation architectures are:
 - interlingua based systems
 - transfer based systems
- Challenges are to acquire the required knowledge resources such as mapping rules and bi-lingual dictionary? By hand or acquire them automatically from corpora.
- Example Based Machine Translation acquires the required knowledge (some of it or all of it) from corpora.

Questions

1. Consider the following short story:

John went to the diner to eat lunch. He ordered a hamburger. But John wasn't very hungry so he didn't finish it. John told the waiter that he wanted a doggy bag. John gave the waiter a tip. John then went to the hardware store and home.

Each inference below is based on a plausible interpretation of the story. For each inference, briefly explain whether that inference was primarily based on syntactic, semantic, pragmatic, discourse, or world knowledge. (Do not answer world knowledge unless none of the other categories are appropriate.)

- (a) John is the person who ordered a hamburger.
- (b) John wasn't just stating a fact that he desired a doggy bag, but was requesting that the waiter bring him a doggy bag.
- (c) John went to the hardware store and then went to his house. (As opposed to going to a hardware store and a hardware home.)
- (d) John gave the waiter some money as a gratuity. (As opposed to giving him a suggestion or hint.)
- (e) John was wearing clothes.

2. Identify the thematic role associated with each noun phrase in the sentence below:

Mary went from Utah to Colorado with John by bicycle.

Solutions

1.a. Discourse knowledge. The inference comes from coreference resolution between "John" and "He" in the first and second sentences.

1.b. Pragmatics. Most people would assume that John was making a request of the waiter and not merely stating a fact, which is a pragmatic issue because it reects the purpose of John's statement.

1.c. Syntactic knowledge. This inference reflects one syntactic parse: ((hardware store) and (home)), as opposed to an alternative parse: (hardware (store and home)).

1.d Semantic knowledge. Most people would assume that "tip" means gratuity, as opposed to other meanings of the word "tip", such as suggestion or hint.

1.e. World Knowledge. There is nothing stated in the story that mentions clothes, but in our culture people virtually always wear clothes when they leave their house. So we make this assumption.

2. The roles are

agent = Mary

source (from-loc) = Utah

destination (to-loc) = Colorado

co-agent = John

instrument = bicycle