Overview of Artificial Intelligence

What is AI ?

- **A**rtificial Intelligence (AI) is a branch of *Science* which deals with helping machines find solutions to complex problems in a more human-like fashion.
- This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way.
- A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behavior appears
- Artificial intelligence can be viewed from a variety of perspectives.
  - ✓ From the perspective of **intelligence** artificial intelligence is making machines "intelligent" -- acting as we would expect people to act.
    - o The inability to distinguish computer responses from human responses is called the Turing test.
    - o Intelligence requires knowledge
    - o Expert problem solving - restricting domain to allow including significant relevant knowledge
  - ✓ From a **business** perspective AI is a set of very powerful tools, and methodologies for using those tools to solve business problems.
  - ✓ From a **programming** perspective, AI includes the study of symbolic programming, problem solving, and search.
    - o Typically AI programs focus on symbols rather than numeric processing.
    - o Problem solving - achieve goals.
    - o Search - seldom access a solution directly. Search may include a variety of techniques.
    - o AI programming languages include:

– LISP, developed in the 1950s, is the early programming language strongly associated with AI. LISP is a functional programming language with procedural extensions. LISP (LISt Processor) was specifically designed for processing heterogeneous lists -- typically a list of symbols. Features of LISP are run- time type checking, higher order functions (functions that have other functions as parameters), automatic memory management (garbage collection) and an interactive environment.

– The second language strongly associated with AI is PROLOG. PROLOG was developed in the 1970s. PROLOG is based on first order logic. PROLOG is declarative in nature and has facilities for explicitly limiting the search space.

– Object-oriented languages are a class of languages more recently used for AI programming. Important features of object-oriented languages include: concepts of objects and messages, objects bundle data and methods for manipulating the data, sender specifies what is to be done receiver decides how to do it, inheritance (object hierarchy where objects inherit the attributes of the more general class of objects). Examples of object-oriented languages are Smalltalk, Objective C, C++. Object oriented extensions to LISP (CLOS - Common LISP Object System) and PROLOG (L&O - Logic & Objects) are also used.

- Artificial Intelligence is a new electronic machine that stores large amount of information and process it at very high speed
- The computer is interrogated by a human via a teletype It passes if the human cannot tell if there is a computer or human at the other end
- The ability to solve problems
- It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence

Importance of AI

- **Game Playing**

  You can buy machines that can play master level chess for a few hundred dollars. There is some AI in them, but they play well against people mainly through brute force computation--looking at hundreds of thousands of positions. To beat a world champion by brute force and known reliable heuristics requires being able to look at 200 million positions per second.

- **Speech Recognition**

In the 1990s, computer speech recognition reached a practical level for limited purposes. Thus United Airlines has replaced its keyboard tree for flight information by a system using speech recognition of flight numbers and city names. It is quite convenient. On the other hand, while it is possible to instruct some computers using speech, most users have gone back to the keyboard and the mouse as still more convenient.

- **Understanding Natural Language**
  Just getting a sequence of words into a computer is not enough. Parsing sentences is not enough either. The computer has to be provided with an understanding of the domain the text is about, and this is presently possible only for very limited domains.

- **Computer Vision**
  The world is composed of three-dimensional objects, but the inputs to the human eye and computers' TV cameras are two dimensional. Some useful programs can work solely in two dimensions, but full computer vision requires partial three-dimensional information that is not just a set of two-dimensional views. At present there are only limited ways of representing three-dimensional information directly, and they are not as good as what humans evidently use.

- **Expert Systems**
  A ``knowledge engineer'' interviews experts in a certain domain and tries to embody their knowledge in a computer program for carrying out some task. How well this works depends on whether the intellectual mechanisms required for the task are within the present state of AI. When this turned out not to be so, there were many disappointing results. One of the first expert systems was MYCIN in 1974, which diagnosed bacterial infections of the blood and suggested treatments. It did better than medical students or practicing doctors, provided its limitations were observed. Namely, its ontology included bacteria, symptoms, and treatments and did not include patients, doctors, hospitals, death, recovery, and events occurring in time. Its interactions depended on a single patient being considered. Since the experts consulted by the knowledge engineers knew about patients, doctors, death, recovery, etc., it is clear that the knowledge engineers forced what the experts told them into a predetermined framework. The usefulness of current expert systems depends on their users having common sense.

- **Heuristic Classification**

  One of the most feasible kinds of expert system given the present knowledge of AI is to put some information in one of a fixed set of categories using several sources of information. An example is advising whether to accept a proposed credit card purchase. Information is available about the owner of the credit card, his record of payment and also about the item he is buying and about the establishment from which he is buying it (e.g., about whether there have been previous credit card frauds at this establishment).

- **The applications of AI are shown in Fig 1.1:**

  ✓ Consumer Marketing

  - Have you ever used any kind of credit/ATM/store card while shopping?
  - if so, you have very likely been "input" to an AI algorithm
  - All of this information is recorded digitally
  - Companies like Nielsen gather this information weekly and search for patterns
    - general changes in consumer behavior
    - tracking responses to new products
    - identifying customer segments: targeted marketing, e.g., they find out that consumers with sports cars who buy textbooks respond well to offers of new credit cards.
  - Algorithms ("data mining") search data for patterns based on mathematical theories of learning

  ✓ Identification Technologies

  - ID cards e.g., ATM cards
  - can be a nuisance and security risk: cards can be lost, stolen, passwords forgotten, etc
  - Biometric Identification, walk up to a locked door
    - Camera
    - Fingerprint device
    - Microphone
    - Computer uses biometric signature for identification
    - Face, eyes, fingerprints, voice pattern

        - This works by comparing data from person at door with stored library
        - Learning algorithms can learn the matching process by analyzing a large library database off-line, can improve its performance.

- ✓ Intrusion Detection
    - o Computer security - we each have specific patterns of computer use times of day, lengths of sessions, command used, sequence of commands, etc
        - would like to learn the "signature" of each authorized user
        - can identify non-authorized users
    - o How can the program automatically identify users?
        - record user's commands and time intervals
        - characterize the patterns for each user
        - model the variability in these patterns
        - classify (online) any new user by similarity to stored patterns
- ✓ Machine Translation
    - o Language problems in international business
        - e.g., at a meeting of Japanese, Korean, Vietnamese and Swedish investors, no common language
        - If you are shipping your software manuals to 127 countries, the solution is ; hire translators to translate
        - would be much cheaper if a machine could do this!
    - o How hard is automated translation
        - very difficult!
        - e.g., English to Russian
        - not only must the words be translated, but their meaning also!

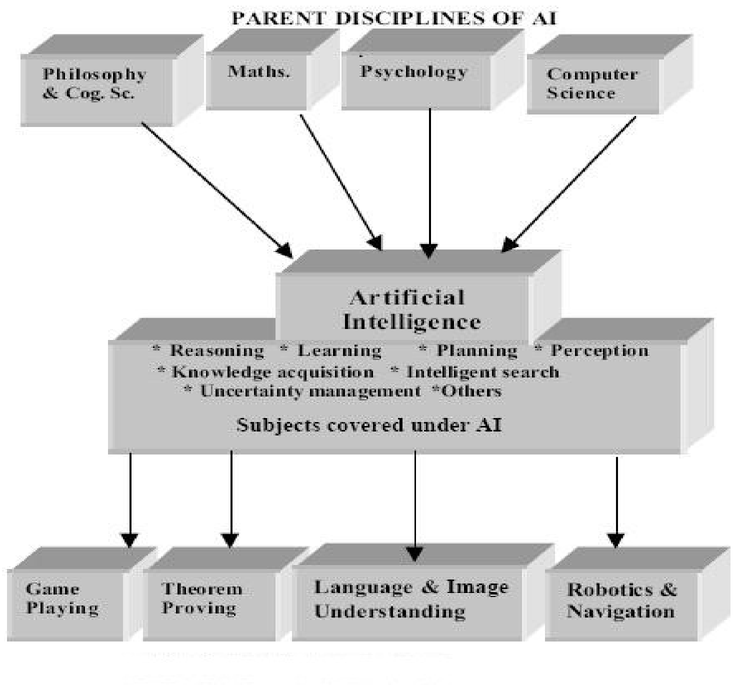

PARENT DISCIPLINES OF AI
Philosophy & Cog. Sc.
Maths.
Psychology
Computer Science
Artificial Intelligence
* Reasoning * Learning * Planning * Perception
* Knowledge acquisition * Intelligent search
* Uncertainty management *Others
Subjects covered under AI
Game Playing
Theorem Proving
Language & Image Understanding
Robotics & Navigation

Fig : Application areas of AI

Early work in AI

- "Artificial Intelligence (AI) is the part of computer science concerned with designing intelligent computer systems, that is, systems that exhibit characteristics we associate with intelligence in human behaviour – understanding language, learning, reasoning, solving problems, and so on."
- Scientific Goal To determine which ideas about knowledge representation, learning, rule systems, search, and so on, explain various sorts of real intelligence.
- Engineering Goal To solve real world problems using AI techniques such as knowledge representation, learning, rule systems, search, and so on.
- Traditionally, computer scientists and engineers have been more interested in the engineering goal, while psychologists, philosophers and cognitive scientists have been more interested in the scientific goal.
- The Roots - Artificial Intelligence has identifiable roots in a number of older disciplines, particularly:
  - ✓ Philosophy
  - ✓ Logic/Mathematics
  - ✓ Computation

- ✓ Psychology/Cognitive Science
- ✓ Biology/Neuroscience
- ✓ Evolution

- There is inevitably much overlap, e.g. between philosophy and logic, or between mathematics and computation. By looking at each of these in turn, we can gain a better understanding of their role in AI, and how these underlying disciplines have developed to play that role.
- Philosophy
  - ✓ ~400 BC Socrates asks for an algorithm to distinguish piety from non-piety.
  - ✓ ~350 BC Aristotle formulated different styles of deductive reasoning, which could mechanically generate conclusions from initial premises, e.g. Modus Ponens

    If A ? B and A then B

    If A implies B and A is true then B is true when it's raining you

    get wet and it's raining then you get wet
  - ✓ 1596 – 1650 Rene Descartes idea of mind-body dualism – part of the mind is exempt from physical laws.
  - ✓ 1646 – 1716 Wilhelm Leibnitz was one of the first to take the materialist position which holds that the mind operates by ordinary physical processes – this has the implication that mental processes can potentially be carried out by machines.
- Logic/Mathematics
  - ✓ Earl Stanhope's Logic Demonstrator was a machine that was able to solve syllogisms, numerical problems in a logical form, and elementary questions of probability.
  - ✓ 1815 – 1864 George Boole introduced his formal language for making logical inference in 1847 – Boolean algebra.
  - ✓ 1848 – 1925 Gottlob Frege produced a logic that is essentially the first-order logic that today forms the most basic knowledge representation system.
  - ✓ 1906 – 1978 Kurt Gödel showed in 1931 that there are limits to what logic can do. His Incompleteness Theorem showed that in any formal logic powerful enough to describe the properties of natural numbers, there are true statements whose truth cannot be established by any algorithm.

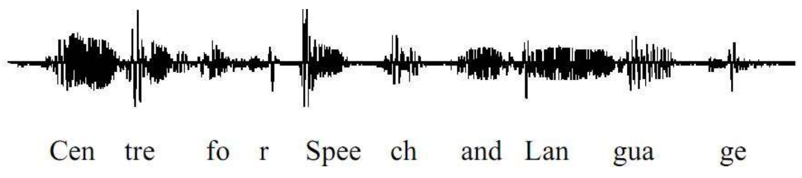
- ✓ 1995 Roger Penrose tries to prove the human mind has non-computable capabilities.

- Computation
  - ✓ 1869 William Jevon's Logic Machine could handle Boolean Algebra and Venn Diagrams, and was able to solve logical problems faster than human beings.
  - ✓ 1912 – 1954 Alan Turing tried to characterise exactly which functions are capable of being computed. Unfortunately it is difficult to give the notion of computation a formal definition. However, the Church-Turing thesis, which states that a Turing machine is capable of computing any computable function, is generally accepted as providing a sufficient definition. Turing also showed that there were some functions which no Turing machine can compute (e.g. Halting Problem).
  - ✓ 1903 – 1957 John von Neumann proposed the von Neuman architecture which allows a description of computation that is independent of the particular realisation of the computer.
  - ✓ 1960s Two important concepts emerged: Intractability (when solution time grows atleast exponentially) and Reduction (to 'easier' problems).
- Psychology / Cognitive Science
  - ✓ Modern Psychology / Cognitive Psychology / Cognitive Science is the science which studies how the mind operates, how we behave, and how our brains process information.
  - ✓ Language is an important part of human intelligence. Much of the early work on knowledge representation was tied to language and informed by research into linguistics.
  - ✓ It is natural for us to try to use our understanding of how human (and other animal) brains lead to intelligent behavior in our quest to build artificial intelligent systems. Conversely, it makes sense to explore the properties of artificial systems (computer models/simulations) to test our hypotheses concerning human systems.
  - ✓ Many sub-fields of AI are simultaneously building models of how the human system operates, and artificial systems for solving real world problems, and are allowing useful ideas to transfer between them.
- Biology / Neuroscience

  - ✓ Our brains (which give rise to our intelligence) are made up of tens of billions of neurons, each connected to hundreds or thousands of other neurons.
  - ✓ Each neuron is a simple processing device (e.g. just firing or not firing depending on the total amount of activity feeding into it). However, large networks of neurons are extremely powerful computational devices that can learn how best to operate.
  - ✓ The field of Connectionism or Neural Networks attempts to build artificial systems based on simplified networks of simplified artificial neurons.
  - ✓ The aim is to build powerful AI systems, as well as models of various human abilities.
  - ✓ Neural networks work at a sub-symbolic level, whereas much of conscious human reasoning appears to operate at a symbolic level.
  - ✓ Artificial neural networks perform well at many simple tasks, and provide good models of many human abilities. However, there are many tasks that they are not so good at, and other approaches seem more promising in those areas.
- Evolution
  - ✓ One advantage humans have over current machines/computers is that they have a long evolutionary history.
  - ✓ Charles Darwin (1809 – 1882) is famous for his work on evolution by natural selection. The idea is that fitter individuals will naturally tend to live longer and produce more children, and hence after many generations a population will automatically emerge with good innate properties.
  - ✓ This has resulted in brains that have much structure, or even knowledge, built in at birth.
  - ✓ This gives them at the advantage over simple artificial neural network systems that have to learn everything.
  - ✓ Computers are finally becoming powerful enough that we can simulate evolution and evolve good AI systems.
  - ✓ We can now even evolve systems (e.g. neural networks) so that they are good at learning.
  - ✓ A related field called genetic programming has had some success in evolving programs, rather than programming them by hand.
- Sub-fields of Artificial Intelligence

- ✓ Neural Networks – e.g. brain modelling, time series prediction, classification
- ✓ Evolutionary Computation – e.g. genetic algorithms, genetic programming
- ✓ Vision – e.g. object recognition, image understanding
- ✓ Robotics – e.g. intelligent control, autonomous exploration
- ✓ Expert Systems – e.g. decision support systems, teaching systems
- ✓ Speech Processing– e.g. speech recognition and production
- ✓ Natural Language Processing – e.g. machine translation
- ✓ Planning – e.g. scheduling, game playing
- ✓ Machine Learning – e.g. decision tree learning, version space learning

- Speech Processing
  - ✓ As well as trying to understand human systems, there are also numerous real world applications: speech recognition for dictation systems and voice activated control; speech production for automated announcements and computer interfaces.
  - ✓ How do we get from sound waves to text streams and vice-versa?



- Natural Language Processing
  - ✓ For example, machine understanding and translation of simple sentences:
- Planning
  - ✓ Planning refers to the process of choosing/computing the correct sequence of steps to solve a given problem.
  - ✓ To do this we need some convenient representation of the problem domain. We can define states in some formal language, such as a subset of predicate logic, or a series of rules.
  - ✓ A plan can then be seen as a sequence of operations that transform the initial state into the goal state, i.e. the problem solution. Typically we will use some kind of search algorithm to find a good plan.
- Common Techniques
  - ✓ Even apparently radically different AI systems (such as rule based expert systems and neural networks) have many common techniques.
  - ✓ Four important ones are:

- Knowledge Representation: Knowledge needs to be represented somehow – perhaps as a series of if-then rules, as a frame based system, as a semantic network, or in the connection weights of an artificial neural network.
- Learning: Automatically building up knowledge from the environment – such as acquiring the rules for a rule based expert system, or determining the appropriate connection weights in an artificial neural network.
- Rule Systems: These could be explicitly built into an expert system by a knowledge engineer, or implicit in the connection weights learnt by a neural network.
- Search: This can take many forms – perhaps searching for a sequence of states that leads quickly to a problem solution, or searching for a good set of connection weights for a neural network by minimizing a fitness function.

AI and related fields

- **Logical AI**
  What a program knows about the world in general the facts of the specific situation in which it must act, and its goals are all represented by sentences of some mathematical logical language. The program decides what to do by inferring that certain actions are appropriate for achieving its goals.

- **Search**
  AI programs often examine large numbers of possibilities, e.g. moves in a chess game or inferences by a theorem proving program. Discoveries are continually made about how to do this more efficiently in various domains.

- **Pattern Recognition**
  When a program makes observations of some kind, it is often programmed to compare what it sees with a pattern. For example, a vision program may try to match a pattern of eyes and a nose in a scene in order to find a face. More complex patterns, e.g. in a natural language text, in a chess position, or in the history of some event are also studied.

- **Representation**

  Facts about the world have to be represented in some way. Usually languages of mathematical logic are used.

- **Inference**

  From some facts, others can be inferred. Mathematical logical deduction is adequate for some purposes, but new methods of *non-monotonic* inference have been added to logic since the 1970s. The simplest kind of non-monotonic reasoning is default reasoning in which a conclusion is to be inferred by default, but the conclusion can be withdrawn if there is evidence to the contrary. For example, when we hear of a bird, we man infer that it can fly, but this conclusion can be reversed when we hear that it is a penguin. It is the possibility that a conclusion may have to be withdrawn that constitutes the non-monotonic character of the reasoning. Ordinary logical reasoning is monotonic in that the set of conclusions that can the drawn from a set of premises is a monotonic increasing function of the premises.

- **Common sense knowledge and reasoning**

  This is the area in which AI is farthest from human-level, in spite of the fact that it has been an active research area since the 1950s. While there has been considerable progress, e.g. in developing systems of *non-monotonic reasoning* and theories of action, yet more new ideas are needed.

- **Learning from experience**

  Programs do that. The approaches to AI based on *connectionism* and *neural nets* specialize in that. There is also learning of laws expressed in logic. Programs can only learn what facts or behaviors their formalisms can represent, and unfortunately learning systems are almost all based on very limited abilities to represent information.

- **Planning**

  Planning programs start with general facts about the world (especially facts about the effects of actions), facts about the particular situation and a statement of a goal. From these, they generate a strategy for achieving the goal. In the most common cases, the strategy is just a sequence of actions.

- **Epistemology**

This is a study of the kinds of knowledge that are required for solving problems in the world.

- **Ontology**
  Ontology is the study of the kinds of things that exist. In AI, the programs and sentences deal with various kinds of objects, and we study what these kinds are and what their basic properties are. Emphasis on ontology begins in the 1990s.

- **Heuristics**
  A heuristic is a way of trying to discover something or an idea imbedded in a program. The term is used variously in AI. *Heuristic functions* are used in some approaches to search to measure how far a node in a search tree seems to be from a goal. *Heuristic predicates* that compare two nodes in a search tree to see if one is better than the other, i.e. constitutes an advance toward the goal, may be more useful.

- **Genetic Programming**
  Genetic programming is a technique for getting programs to solve a task by mating random Lisp programs and selecting fittest in millions of generations.

Search and Control Strategies:

Problem solving is an important aspect of Artificial Intelligence. A problem can be considered to consist of a goal and a set of actions that can be taken to lead to the goal. At any given time, we consider the state of the search space to represent where we have reached as a result of the actions we have applied so far. For example, consider the problem of looking for a contact lens on a football field. The initial state is how we start out, which is to say we know that the lens is somewhere on the field, but we don't know where. If we use the representation where we examine the field in units of one square foot, then our first action might be to examine the square in the top-left corner of the field. If we do not find the lens there, we could consider the state now to be that we have examined the top-left square and have not found the lens. After a number of actions, the state might be that we have examined 500 squares, and we have now just found the lens in the last square we examined. This is a goal state because it satisfies the goal that we had of finding a contact lens.

Search is a method that can be used by computers to examine a problem space like this in order to find a goal. Often, we want to find the goal as quickly as possible or without using too many resources. A problem space can also be considered to be a search space because in order to solve the problem, we will search the space for a goal state.We will continue to use the term search space to describe this concept. In this chapter, we will look at a number of methods for examining a search space. These methods are called search methods.

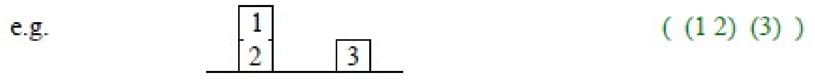
- The Importance of Search in AI
    - It has already become clear that many of the tasks underlying AI can be phrased in terms of a search for the solution to the problem at hand.
    - Many goal based agents are essentially problem solving agents which must decide what to do by searching for a sequence of actions that lead to their solutions.
    - For production systems, we have seen the need to search for a sequence of rule applications that lead to the required fact or action.
    - For neural network systems, we need to search for the set of connection weights that will result in the required input to output mapping.
- Which search algorithm one should use will generally depend on the problem domain? There are four important factors to consider:
    - Completeness – Is a solution guaranteed to be found if at least one solution exists?
    - Optimality – Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?
    - Time Complexity – The upper bound on the time required to find a solution, as a function of the complexity of the problem.
    - Space Complexity – The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.

Preliminary concepts

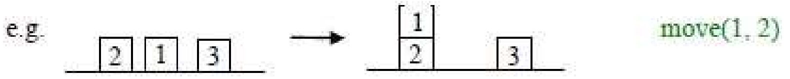
- Two varieties of **space-for-time** algorithms:
    - Input enhancement — preprocess the input (or its part) to store some info to be used later in solving the problem
        - Counting for sorting
        - String searching algorithms
    - Prestructuring — preprocess the input to make accessing its elements easier
        - Hashing

- Indexing schemes (e.g., B-trees)

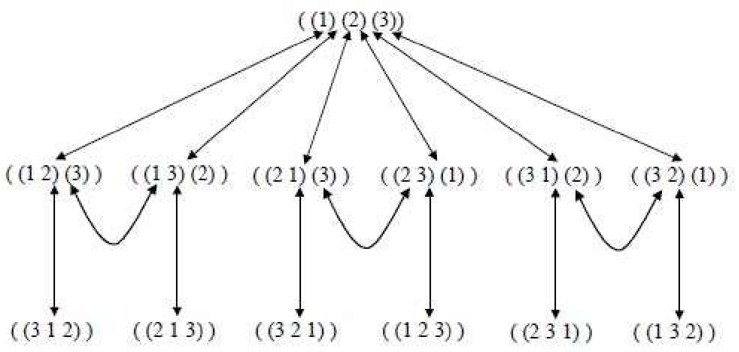
- **State Space Representations:** The state space is simply the space of all possible states, or configurations, that our system may be in. Generally, of course, we prefer to work with some convenient representation of that search space.
- There are two components to the representation of state spaces:
  - Static States



  - Transitions between States

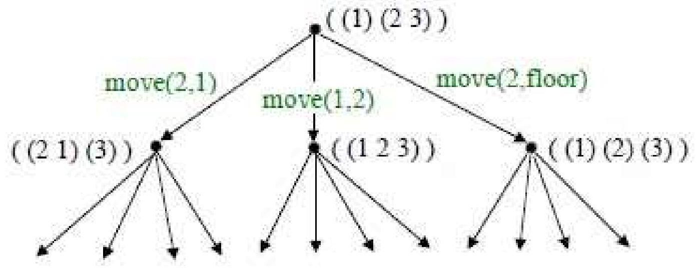


- **State Space Graphs:** If the number of possible states of the system is small enough, we can represent all of them, along with the transitions between them, in a state space graph, e.g.



- **Routes through State Space:** Our general aim is to search for a route, or sequence of transitions, through the state space graph from our initial state to a goal state.
- Sometimes there will be more than one possible goal state. We define a goal test to determine if a goal state has been achieved.
- The solution can be represented as a sequence of link labels (or transitions) on the state space graph. Note that the labels depend on the direction moved along the link.
- Sometimes there may be more than one path to a goal state, and we may want to find the optimal (best possible) path. We can define link costs and path costs for measuring the cost of going along a particular path, e.g. the path cost may just equal the number of links, or could be the sum of individual link costs.

- For most realistic problems, the state space graph will be too large for us to hold all of it explicitly in memory at any one time.
- **Search Trees:** It is helpful to think of the search process as building up a search tree of routes through the state space graph. The root of the search tree is the search node corresponding to the initial state.
- The leaf nodes correspond either to states that have not yet been expanded, or to states that generated no further nodes when expanded.



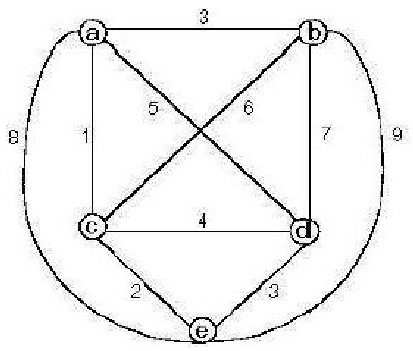
- At each step, the search algorithm chooses a new unexpanded leaf node to expand. The different search strategies essentially correspond to the different algorithms one can use to select which is the next mode to be expanded at each stage.

Examples of search problems

- **Traveling Salesman Problem:** Given *n* cities with known distances between each pair, find the shortest tour that passes through all the cities exactly once before returning to the starting city.
- A lower bound on the length l of any tour can be computed as follows
  - ✓ For each city i, $1 \leq i \leq n$, find the sum s_i of the distances from city i to the two nearest cities.
  - ✓ Compute the sum s of these n numbers.
  - ✓ Divide the result by 2 and round up the result to the nearest integer

$$lb = s / 2$$

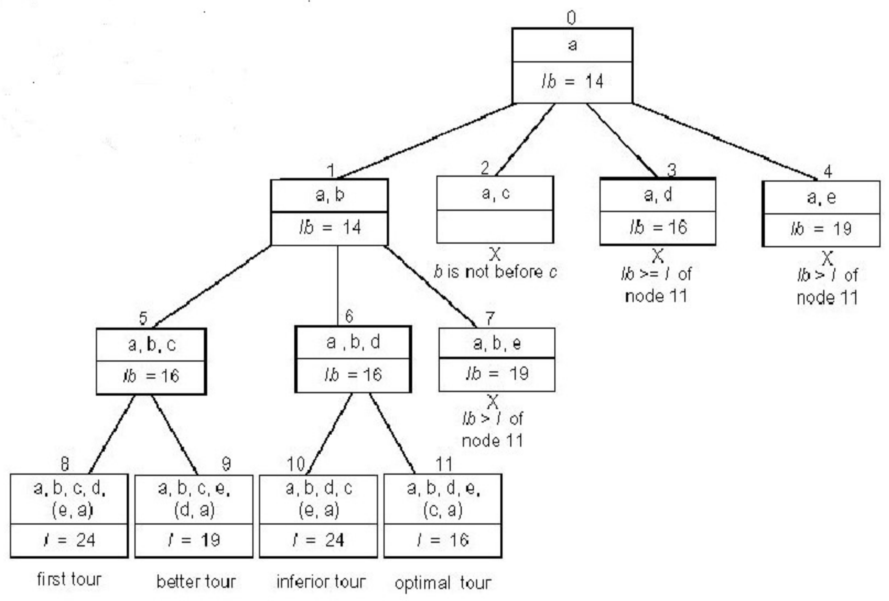
- The lower bound for the graph shown in the Fig 5.1 can be computed as follows:



lb = [(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] / 2 = 14.

- For any subset of tours that must include particular edges of a given graph, the lower bound can be modified accordingly. E.g.: For all the Hamiltonian circuits of the graph that must include edge (a, d), the lower bound can be computed as follows:
  lb = [(1 + 5) + (3 + 6) + (1 + 2) + (3 + 5) + (2 + 3)] / 2 = 16.

- Applying the branch-and-bound algorithm, with the bounding function lb = s / 2, to find the shortest Hamiltonian circuit for the given graph, we obtain the state-space tree as shown below:
- To reduce the amount of potential work, we take advantage of the following two observations:
  - ✓ We can consider only tours that start with a.
  - ✓ Since the graph is undirected, we can generate only tours in which b is visited before c.
- In addition, after visiting n – 1 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one is shown in the Fig 5.2



Root node includes only the starting vertex a with a lower bound of

lb = [(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] / 2 = 14.

- Node 1 represents the inclusion of edge (a, b)
  lb = [(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)] / 2 = 14.
- Node 2 represents the inclusion of edge (a, c). Since b is not visited before c, this node is terminated.
- Node 3 represents the inclusion of edge (a, d)
  lb = [(1 + 5) + (3 + 6) + (1 + 2) + (3 + 5) + (2 + 3)] / 2 = 16.
- Node 1 represents the inclusion of edge (a, e)
  lb = [(1 + 8) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 8)] / 2 = 19.
- Among all the four live nodes of the root, node 1 has a better lower bound. Hence we branch from node 1.
- Node 5 represents the inclusion of edge (b, c)
  lb = [(1 + 3) + (3 + 6) + (1 + 6) + (3 + 4) + (2 + 3)] / 2 = 16.
- Node 6 represents the inclusion of edge (b, d)
  lb = [(1 + 3) + (3 + 7) + (1 + 2) + (3 + 7) + (2 + 3)] / 2 = 16.
- Node 7 represents the inclusion of edge (b, e)
  lb = [(1 + 3) + (3 + 9) + (1 + 2) + (3 + 4) + (2 + 9)] / 2 = 19.

- Since nodes 5 and 6 both have the same lower bound, we branch out from each of them.
- Node 8 represents the inclusion of the edges (c, d), (d, e) and (e, a). Hence, the length of the tour,
  l = 3 + 6 + 4 + 3 + 8 = 24.

- Node 9 represents the inclusion of the edges (c, e), (e, d) and (d, a). Hence, the length of the tour,
  l = 3 + 6 + 2 + 3 + 5 = 19.

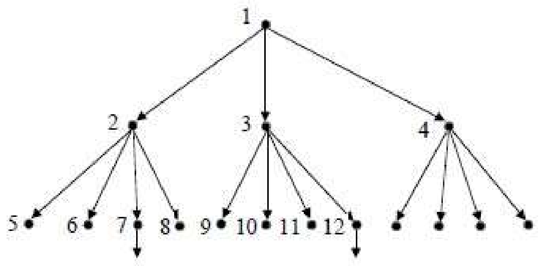
- Node 10 represents the inclusion of the edges (d, c), (c, e) and (e, a). Hence, the length of the tour,
  l = 3 + 7 + 4 + 2 + 8 = 24.

- Node 11 represents the inclusion of the edges (d, e), (e, c) and (c, a). Hence, the length of the tour,
  l = 3 + 7 + 3 + 2 + 1 = 16.

- Node 11 represents an optimal tour since its tour length is better than or equal to the other live nodes, 8, 9, 10, 3 and 4.
- The optimal tour is a → b → d → e → c → a with a tour length of 16.

Uniformed or Blind search

- **Breadth First Search (BFS):** BFS expands the leaf node with the lowest path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple queue ("first in, first out").

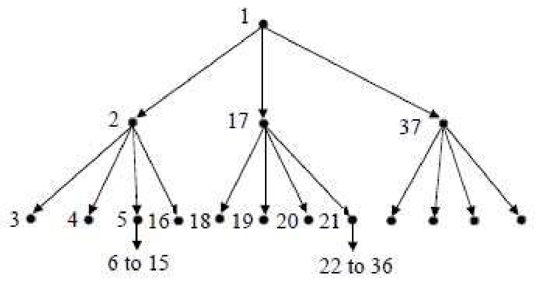


- This is guaranteed to find an optimal path to a goal state. It is memory intensive if the state space is large. If the typical branching factor is b, and the depth of the shallowest goal state is d – the space complexity is O(b^d), and the time complexity is O(b^d).
- BFS is an easy search technique to understand. The algorithm is presented below.

```
breadth_first_search ()
{
    store initial state in queue Q
    set state in the front of the Q as current state ;
    while (goal state is reached OR Q is empty)
    {
        apply rule to generate a new state from the current
        state ;
        if (new state is goal state) quit ;
            else if (all states generated from current states are
            exhausted)
            {
                delete the current state from the Q ;
                set front element of Q as the current state ;
            }
            else continue ;
    }
}
```

- The algorithm is illustrated using the bridge components configuration problem. The initial state is PDFG, which is not a goal state; and hence set it as the current state. Generate another state DPFG (by swapping 1st and 2nd position values) and add it to the list. That is not a goal state, hence; generate next successor state, which is FDPG (by swapping 1st and 3rd position values). This is also not a goal state; hence add it to the list and generate the next successor state GDFP.

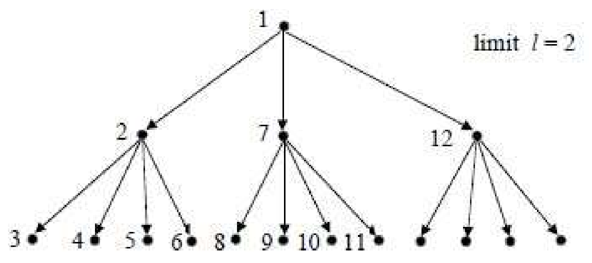
- Only three states can be generated from the initial state. Now the queue Q will have three elements in it, viz., DPFG, FDPG and GDFP. Now take DPFG (first state in the list) as the current state and continue the process, until all the states generated from this are evaluated. Continue this process, until the goal state DGPF is reached.
- The 14th evaluation gives the goal state. It may be noted that, all the states at one level in the tree are evaluated before the states in the next level are taken up; i.e., the evaluations are carried out breadth-wise. Hence, the search strategy is called breadth-first search.
- **Depth First Search (DFS):** DFS expands the leaf node with the highest path cost so far, and keeps going until a goal node is generated. If the path cost simply equals the number of links, we can implement this as a simple stack ("last in, first out").



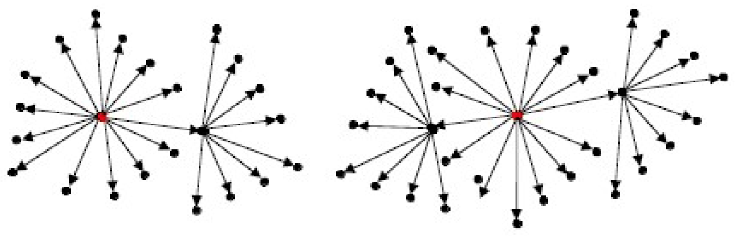
- This is not guaranteed to find any path to a goal state. It is memory efficient even if the state space is large. If the typical branching factor is b, and the maximum depth of the tree is m – the space complexity is O(bm), and the time complexity is O(b^m).
- In DFS, instead of generating all the states below the current level, only the first state below the current level is generated and evaluated recursively. The search continues till a further successor cannot be generated.
- Then it goes back to the parent and explores the next successor. The algorithm is given below.

```
depth_first_search ()

{

        set initial state to current state ;

        if (initial state is current state) quit ;
        else
        {
            if (a successor for current state exists)
            {
                generate a successor of the current state and
                set it as current state ;
            }
            else return ;
            depth_first_search (current_state) ;
            if (goal state is achieved) return ;
            else continue ;
        }
    }
```

- Since DFS stores only the states in the current path, it uses much less memory during the search compared to BFS.
- The probability of arriving at goal state with a fewer number of evaluations is higher with DFS compared to BFS. This is because, in BFS, all the states in a level have to be evaluated before states in the lower level are considered. DFS is very efficient when more acceptable solutions exist, so that the search can be terminated once the first acceptable solution is obtained.
- BFS is advantageous in cases where the tree is very deep.
- An ideal search mechanism is to combine the advantages of BFS and DFS.
- **Depth Limited Search (DLS):** DLS is a variation of DFS. If we put a limit l on how deep a depth first search can go, we can guarantee that the search will terminate (either in success or failure).



- If there is at least one goal state at a depth less than l, this algorithm is guaranteed to find a goal state, but it is not guaranteed to find an optimal path. The space complexity is O(bl), and the time complexity is O(b^l).
- **Depth First Iterative Deepening Search (DFIDS):** DFIDS is a variation of DLS. If the lowest depth of a goal state is not known, we can always find the best limit l for DLS by trying all possible depths l = 0, 1, 2, 3, … in turn, and stopping once we have achieved a goal state.
- This appears wasteful because all the DLS for l less than the goal level are useless, and many states are expanded many times. However, in practice, most of the time is spent at the deepest part of the search tree, so the algorithm actually combines the benefits of DFS and BFS.
- Because all the nodes are expanded at each level, the algorithm is complete and optimal like BFS, but has the modest memory requirements of DFS. Exercise: if we had plenty of memory, could/should we avoid expanding the top level states many times?
- The space complexity is O(bd) as in DLS with l = d, which is better than BFS.
- The time complexity is O(b^d) as in BFS, which is better than DFS.
- **Bi-Directional Search (BDS):** The idea behind bi-directional search is to search simultaneously both forward from the initial state and backwards from the goal state, and stop when the two BFS searches meet in the middle.



- This is not always going to be possible, but is likely to be feasible if the state transitions are reversible. The algorithm is complete and optimal, and since the two search depths are ~d/2, it has space complexity O($b^{d/2}$), and time complexity O($b^{d/2}$). However, if there is more than one possible goal state, this must be factored into the complexity.

- **Repeated States:** In the above discussion we have ignored an important complication that often arises in search processes – the possibility that we will waste time by expanding states that have already been expanded before somewhere else on the search tree.
- For some problems this possibility can never arise, because each state can only be reached in one way.
- For many problems, however, repeated states are unavoidable. This will include all problems where the transitions are reversible, e.g.

$$((1\,2\,3)) \rightarrow ((1)(2\,3)) \rightarrow ((1\,2\,3)) \rightarrow ((1)(2\,3)) \rightarrow ((1\,2\,3)) \rightarrow \ldots$$

- The search trees for these problems are infinite, but if we can prune out the repeated states, we can cut the search tree down to a finite size, We effectively only generate a portion of the search tree that matches the state space graph.
- **Avoiding Repeated States:** There are three principal approaches for dealing with repeated states:
  - ➢ Never return to the state you have just come from

    The node expansion function must be prevented from generating any node successor that is the same state as the node's parent.

  - ➢ Never create search paths with cycles in them

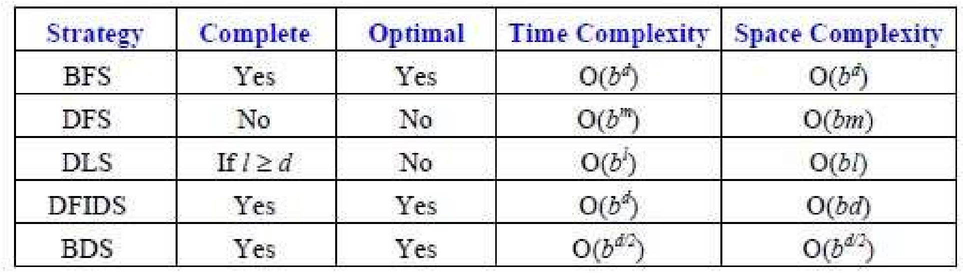
    The node expansion function must be prevented from generating

    any node successor that is the same state as any of the node's ancestors.

  - ➢ Never generate states that have already been generated before

    This requires that every state ever generated is remembered, potentially resulting in space complexity of O(b^d).

- **Comparing the Uninformed Search Algorithms:** We can now summarize the properties of our five uninformed search strategies:



Strategy	Complete	Optimal	Time Complexity	Space Complexity
BFS	Yes	Yes	O(b^d)	O(b^d)
DFS	No	No	O(b^m)	O(bm)
DLS	If $l \geq d$	No	O(b^l)	O(bl)
DFIDS	Yes	Yes	O(b^d)	O(bd)
BDS	Yes	Yes	O($b^{d/2}$)	O($b^{d/2}$)

- Simple BFS and BDS are complete and optimal but expensive with respect to space and time.
- DFS requires much less memory if the maximum tree depth is limited, but has no guarantee of finding any solution, let alone an optimal one. DLS offers an improvement over DFS if we have some idea how deep the goal is.
- The best overall is DFID which is complete, optimal and has low memory requirements, but still exponential time.

Informed search

- Informed search uses some kind of evaluation function to tell us how far each expanded state is from a goal state, and/or some kind of heuristic function to help us decide which state is likely to be the best one to expand next.
- The hard part is to come up with good evaluation and/or heuristic functions. Often there is a natural evaluation function, such as distance in miles or number objects in the wrong position.
- Sometimes we can learn heuristic functions by analyzing what has worked well in similar previous searches.
- The simplest idea, known as greedy best first search, is to expand the node that is already closest to the goal, as that is most likely to lead quickly to a solution. This is like DFS in that it attempts to follow a single route to the goal, only attempting to try a different route when it reaches a dead end. As with DFS, it is not complete, not optimal, and has time and complexity of O(b^m). However, with good heuristics, the time complexity can be reduced substantially.
- **Branch and Bound:** An enhancement of backtracking.
- Applicable to optimization problems.

- For each node (partial solution) of a state-space tree, computes a bound on the value of the objective function for all descendants of the node (extensions of the partial solution).
- Uses the bound for:
  - Ruling out certain nodes as “nonpromising” to prune the tree – if a node’s bound is not better than the best solution seen so far.
  - Guiding the search through state-space.
- The search path at the current node in a state-space tree can be terminated for any one of the following three reasons:
  - The value of the node’s bound is not better than the value of the best solution seen so far.
  - The node represents no feasible solutions because the constraints of the problem are already violated.
  - The subset of feasible solutions represented by the node consists of a single point and hence we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.
- **Best-First branch-and-bound:**
  - A variation of backtracking.
  - Among all the nonterminated leaves, called as the live nodes, in the current tree, generate all the children of the most promising node, instead of generation a single child of the last promising node as it is done in backtracking.
  - Consider the node with the best bound as the most promising node.
- **A* Search:** Suppose that, for each node n in a search tree, an evaluation function f(n) is defined as the sum of the cost g(n) to reach that node from the start state, plus an estimated cost h(n) to get from that state to the goal state. That f(n) is then the estimated cost of the cheapest solution through n.
- A* search, which is the most popular form of best-first search, repeatedly picks the node with the lowest f(n) to expand next. It turns out that if the heuristic function h(n) satisfies certain conditions, then this strategy is both complete and optimal.
- In particular, if h(n) is an admissible heuristic, i.e. is always optimistic and never overestimates the cost to reach the goal, then A* is optimal.

- The classic example is finding the route by road between two cities given the straight line distances from each road intersection to the goal city. In this case, the nodes are the intersections, and we can simply use the straight line distances as h(n).
- **Hill Climbing / Gradient Descent:** The basic idea of hill climbing is simple: at each current state we select a transition, evaluate the resulting state, and if the resulting state is an improvement we move there, otherwise we try a new transition from where we are.
- We repeat this until we reach a goal state, or have no more transitions to try. The transitions explored can be selected at random, or according to some problem specific heuristics.
- In some cases, it is possible to define evaluation functions such that we can compute the gradients with respect to the possible transitions, and thus compute which transition direction to take to produce the best improvement in the evaluation function.
- Following the evaluation gradients in this way is known as gradient descent.
- In neural networks, for example, we can define the total error of the output activations as a function of the connection weights, and compute the gradients of how the error changes as we change the weights. By changing the weights in small steps against those gradients, we systematically minimize the network's output errors.

Searching And-Or graphs

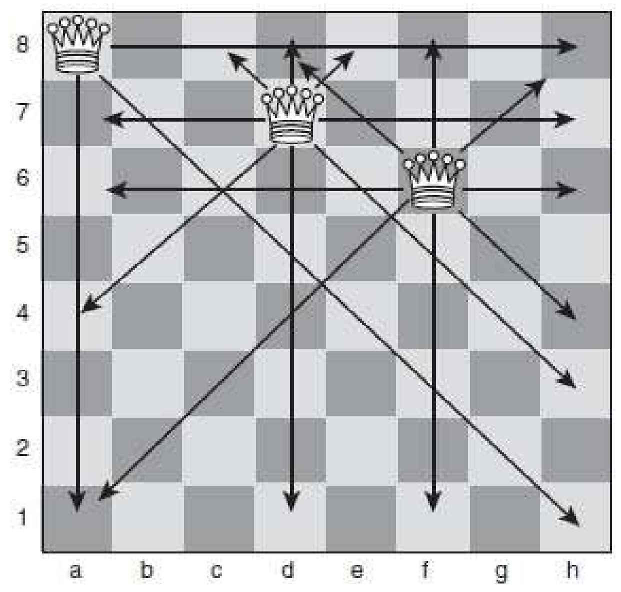
- The DFS and BFS strategies for OR trees and graphs can be adapted for And-Or trees
- The main difference lies in the way termination conditions are determined, since all goals following an And node must be realized, whereas a single goal node following an Or node will do
- A more general optimal strategy is AO* (O for ordered) algorithm
- As in the case of the A* algorithm, we use the open list to hold nodes that have been generated but not expanded and the closed list to hold nodes that have been expanded
- The algorithm is a variation of the original given by Nilsson
- It requires that nodes traversed in the tree be labeled as solved or unsolved in the solution process to account for And node solutions which require solutions to all successors nodes.
- A solution is found when the start node is labeled as solved

- The AO* algorithm
  - Step 1: Place the start node s on open
  - Step 2: Using the search tree constructed thus far, compute the most promising solution tree T_0
  - Step 3:Select a node n that is both on open and a part of $T_{0.}$ Remove n from open and place it on closed
  - Step 4: If n ia terminal goal node, label n as solved. If the solution of n results in any of n's ancestors being solved, label all the ancestors as solved. If the start node s is solved, exit with success where T_0 is the solution tree. Remove from open all nodes with a solved ancestor
  - Step 5: If n is not a solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. If any of n's ancestors become unsolvable because n is, label them unsolvable as well. Remove from open all nodes with unsolvable ancestors
  - Otherwise, expand node n generating all of its successors. For each such successor node that contains more than one subproblem, generate their successors to give individual subproblems. Attach to each newly generated node a back pointer to its predecessor. Compute the cost estimate h* for each newly generated node and place all such nodes thst do not yet have descendents on open. Next recomputed the values oh h* at n and each ancestors of n
  - Step 7: Return to step 2
- It can be shown that AO* will always find a minimum-cost solution tree if one exists, provided only that $h^*(n) \leq h(n)$, and all arc costs are positive. Like A*, the efficiency depends on how closely h* approximates h

Constraint Satisfaction Search

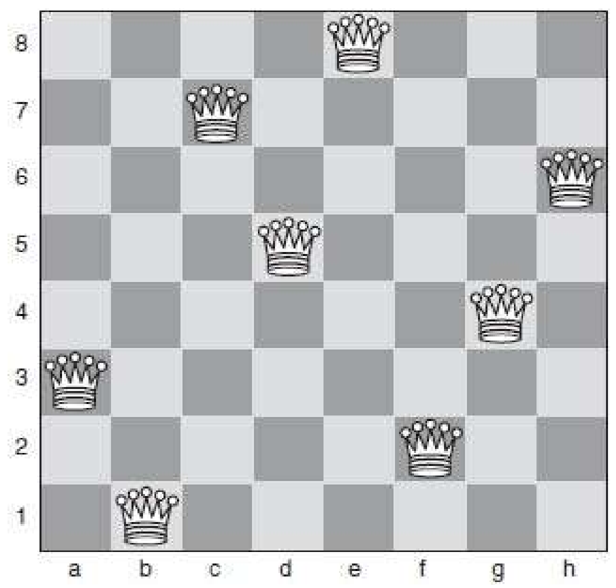
- Search can be used to solve problems that are limited by constraints, such as the eight-queens problem. Such problems are often known as Constraint Satisfaction Problems, or CSPs. I

- n this problem, eight queens must be placed on a chess board in such a way that no two queens are on the same diagonal, row, or column. If we use traditional chess board notation, we mark the columns with letters from a to g and the rows with numbers from 1 to 8. So, a square can be referred to by a letter and a number, such as a4 or g7.
- This kind of problem is known as a constraint satisfaction problem (CSP) because a solution must be found that satisfies the constraints.
- In the case of the eight-queens problem, a search tree can be built that represents the possible positions of queens on the board. One way to represent this is to have a tree that is 8-ply deep, with a branching factor of 64 for the first level, 63 for the next level, and so on, down to 57 for the eighth level.
- A goal node in this tree is one that satisfies the constraints that no two queens can be on the same diagonal, row, or column.
- An extremely simplistic approach to solving this problem would be to analyze every possible configuration until one was found that matched the constraints.
- A more suitable approach to solving the eight-queens problem would be to use depth-first search on a search tree that represents the problem in the following manner:
    - ➢ The first branch from the root node would represent the first choice of a square for a queen. The next branch from these nodes would represent choices of where to place the second queen.
    - ➢ The first level would have a branching factor of 64 because there are 64 possible squares on which to place the first queen. The next level would have a somewhat lower branching factor because once a queen has been placed, the constraints can be used to determine possible squares upon which the next queen can be placed.
    - ➢ The branching factor will decrease as the algorithm searches down the tree. At some point, the tree will terminate because the path being followed will lead to a position where no more queens can be placed on legal squares on the board, and there are still some queens remaining.



In fact, because each row and each column must contain exactly one queen, the branching factor can be significantly reduced by assuming that the first queen must be placed in row 1, the second in row 2, and so on. In this way, the first level will have a branching factor of 8 (a choice of eight squares on which the first queen can be placed), the next 7, the next 6, and so on.

- The search tree can be further simplified as each queen placed on the board "uses up" a diagonal, meaning that the branching factor is only 5 or 6 after the first choice has been made, depending on whether the first queen is placed on an edge of the board (columns a or h) or not.
- The next level will have a branching factor of about 4, and the next may have a branching factor of just 2, as shown in Fig 6.1.
- The arrows in Fig 6.1 show the squares to which each queen can move.
- Note that no queen can move to a square that is already occupied by another queen.



- In Fig 6.1, the first queen was placed in column a of row 8, leaving six choices for the next row. The second queen was placed in column d of row 7, leaving four choices for row 6. The third queen was placed in column f in row 6, leaving just two choices (column c or column h) for row 5.
- Using knowledge like this about the problem that is being solved can help to significantly reduce the size of the search tree and thus improve the efficiency of the search solution.
- A solution will be found when the algorithm reaches depth 8 and successfully places the final queen on a legal square on the board.
- A goal node would be a path containing eight squares such that no two squares shared a diagonal, row, or column.
- One solution to the eight-queens problem is shown in above Fig .
- Note that in this solution, if we start by placing queens on squares e8, c7, h6, and then d5, once the fourth queen has been placed, there are only two choices for placing the fifth queen (b4 or g4). If b4 is chosen, then this leaves no squares that could be chosen for the final three queens to satisfy the constraints. If g4 is chosen for the fifth queen, as has been done in Fig 6.2, only one square is available for the sixth queen (a3), and the final two choices are similarly constrained. So, it can be seen that by applying the

constraints appropriately, the search tree can be significantly reduced for this problem.

- Using chronological backtracking in solving the eight-queens problem might not be the most efficient way to identify a solution because it will backtrack over moves that did not necessarily directly lead to an error, as well as ones that did. In this case, nonchronological backtracking, or dependency-directed backtracking could be more useful because it could identify the steps earlier in the search tree that caused the problem further down the tree.

Forward Checking

- In fact, backtracking can be augmented in solving problems like the eightqueens problem by using a method called forward checking.
- As each queen is placed on the board, a forward-checking mechanism is used to delete from the set of possible future choices any that have been rendered impossible by placing the queen on that square.
- For example, if a queen is placed on square a1, forward checking will remove all squares in row 1, all squares in column a, and also squares b2, c3, d4, e5, f6, g7, and h8.
- In this way, if placing a queen on the board results in removing all remaining squares, the system can immediately backtrack, without having to attempt to place any more queens.
- This can often significantly improve the performance of solutions for CSPs such as the eight-queens problem.

Most-Constrained Variables

- A further improvement in performance can be achieved by using the most-constrained variable heuristic.
- At each stage of the search, this heuristic involves working with the variable that has the least possible number of valid choices.

- In the case of the eight-queens problem, this might be achieved by considering the problem to be one of assigning a value to eight variables, a through h. Assigning value 1 to variable a means placing a queen in square a1.
- To use the most constrained variable heuristic with this representation means that at each move we assign a value to the variable that has the least choices available to it. Hence, after assigning a = 1, b = 3, and c = 5, this leaves three choices for d, three choices for e, one choice for f, three choices for g, and three choices for h. Hence, our next move is to place a queen in column f.
- This heuristic is perhaps more clearly understood in relation to the mapcoloring problem. It makes sense that, in a situation where a particular country can be given only one color due to the colors that have been assigned to its neighbors, that country be colored next.
- The most-constraining variable heuristic is similar in that it involves assigning a value next to the variable that places the greatest number of constraints on future variables.
- The least-constraining value heuristic is perhaps more intuitive than the two already presented in this section.
- This heuristic involves assigning a value to a variable that leaves the greatest number of choices for other variables.
- This heuristic can be used to make n-queens problems with extremely large values of n quite solvable.

Example: Cryptographic Problems

- The constraint satisfaction procedure is also a useful way to solve problems such as cryptographic problems. For example:

  FORTY

  + TEN

  + TEN

  SIXTY

  Solution:

  29786

+ 850

+ 850

31486

- This cryptographic problem can be solved by using a Generate and Test method, applying the following constraints:
    - Each letter represents exactly one number.
    - No two letters represent the same number.
- Generate and Test is a brute-force method, which in this case involves cycling through all possible assignments of numbers to letters until a set is found that meets the constraints and solves the problem.
- Without using constraints, the method would first start by attempting to assign 0 to all letters, resulting in the following sum:

00000

+ 000

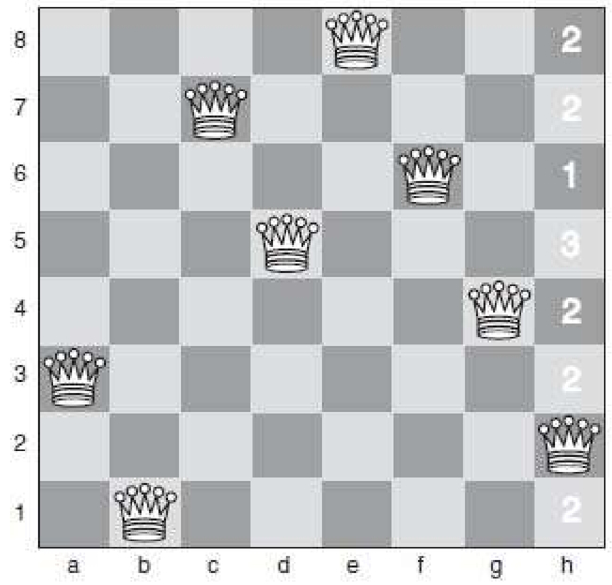
+ 000

00000

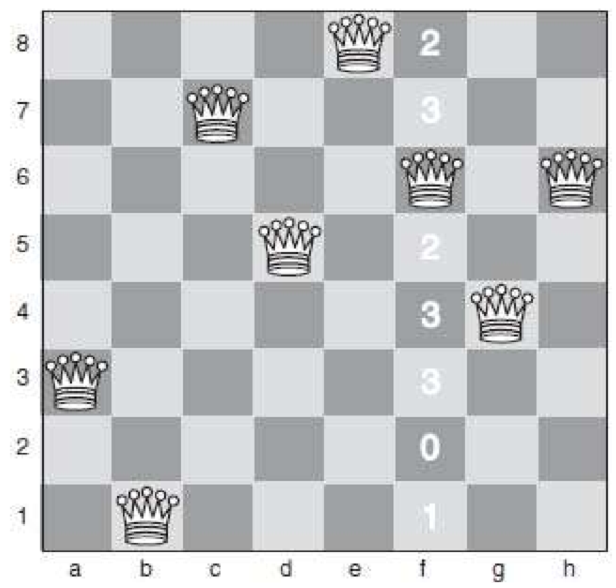
- Although this may appear to be a valid solution to the problem, it does not meet the constraints laid down that specify that each letter can be assigned only one number, and each number can be assigned only to one letter.
- Hence, constraints are necessary simply to find the correct solution to the problem. They also enable us to reduce the size of the search tree.
- In this case, for example, it is not necessary to examine possible solutions where two letters have been assigned the same number, which dramatically reduces the possible solutions to be examined.

Heuristic Repair

- Heuristics can be used to improve performance of solutions to constraint satisfaction problems.
- One way to do this is to use a heuristic repair method, which involves generating a possible solution (randomly, or using a heuristic to generate a position that is close to a solution) and then making changes that reduce the distance of the state from the goal.
- In the case of the eight-queens problem, this could be done using the minconflicts heuristic.
- To move from one state to another state that is likely to be closer to a solution using the min-conflicts heuristic, select one queen that conflicts with another queen (in other words, it is on the same row, column, or diagonal as another queen).
- Now move that queen to a square where it conflicts with as few queens as possible. Continue with another queen. To see how this method would work, consider the starting position shown in Fig 6.3.



- This starting position has been generated by placing the queens such that there are no conflicts on rows or columns. The only conflict here is that the queen in column 3 (on c7) is on a diagonal with the queen in column h (on h2).
- To move toward a solution, we choose to move the queen that is on column h.
- We will only ever apply a move that keeps a queen on the same column because we already know that we need to have one queen on each column.
- Each square in column h has been marked with a number to show how many other queens that square conflicts with. Our first move will be to move the queen on column h up to row 6, where it will conflict only with one queen. Then we arrive at the position shown in below Fig
- Because we have created a new conflict with the queen on row 6 (on f6), our next move must be to move this queen. In fact, we can move it to a square where it has zero conflicts. This means the problem has been solved, and there are no remaining conflicts.
- This method can be used not only to solve the eight-queens problem but also has been successfully applied to the n-queens problem for extremely large values of n. It has been shown that, using this method, the 1,000,000 queens problem can be solved in an average of around 50 steps.
- Solving the 1,000,000-queens problem using traditional search techniques would be impossible because it would involve searching a tree with a branching factor of 10^{12}.



Local Search and Metaheuristics

- Local search methods work by starting from some initial configuration (usually random) and making small changes to the configuration until a state is reached from which no better state can be achieved.
- Hill climbing is a good example of a local search technique.
- Local search techniques, used in this way, suffer from the same problems as hill climbing and, in particular, are prone to finding local maxima that are not the best solution possible.
- The methods used by local search techniques are known as metaheuristics.
- Examples of metaheuristics include simulated annealing, tabu search, genetic algorithms, ant colony optimization, and neural networks.
- This kind of search method is also known as local optimization because it is attempting to optimize a set of values but will often find local maxima rather than a global maximum.
- A local search technique applied to the problem of allocating teachers to classrooms would start from a random position and make small changes until a configuration was reached where no inappropriate allocations were made.
- **Exchanging Heuristics**
    - The simplest form of local search is to use an exchanging heuristic.

- An exchanging heuristic moves from one state to another by exchanging one or more variables by giving them different values. We saw this in solving the eight-queens problem as heuristic repair.
- A k-exchange is considered to be a method where k variables have their values changed at each step.
- The heuristic repair method we applied to the eight-queens problem was 2-exchange.
- A k-exchange can be used to solve the traveling salesman problem. A tour (a route through the cities that visits each city once, and returns to the start) is generated at random. Then, if we use 2-exchange, we remove two edges from the tour and substitute them for two other edges. If this pro duces a valid tour that is shorter than the previous one, we move on from here. Otherwise, we go back to the previous tour and try a different set of substitutions.
- In fact, using k = 2 does not work well for the traveling salesman problem, whereas using k = 3 produces good results.
- Using larger numbers of k will give better and better results but will also require more and more iterations.
- Using k = 3 gives reasonable results and can be implemented efficiently. It does, of course, risk finding local maxima, as is often the case with local search methods.

- **Iterated Local Search**
  - Iterated local search techniques attempt to overcome the problem of local maxima by running the optimization procedure repeatedly, from different initial states.
  - If used with sufficient iterations, this kind of method will almost always find a global maximum.
  - The aim, of course, in running methods like this is to provide a very good solution without needing to exhaustively search the entire problem space.
  - In problems such as the traveling salesman problem, where the search space grows extremely quickly as the number of cities increases, results can be generated that are good enough (i.e., a local maximum) without using many iterations, where a perfect solution would be impossible to find (or at least it would be impossible to guarantee a perfect solution even one iteration of local search may happen upon the global maximum).

- **Tabu Search**
  - Tabu search is a metaheuristic that uses a list of states that have already been visited to attempt to avoid repeating paths.
  - The tabu search metaheuristic is used in combination with another heuristic and operates on the principle that it is worth going down a path that appears to be poor if it avoids following a path that has already been visited.
  - In this way, tabu search is able to avoid local maxima.

Simulated Annealing

- Annealing is a process of producing very strong glass or metal, which involves heating the material to a very high temperature and then allowing it to cool very slowly.
- In this way, the atoms are able to form the most stable structures, giving the material great strength.
- Simulated annealing is a local search metaheuristic based on this method and is an extension of a process called metropolisMonteCarlo simulation.
- Simulated annealing is applied to a multi-value combinatorial problem where values need to be chosen for many variables to produce a particular value for some global function, dependent on all the variables in the system.
- This value is thought of as the energy of the system, and in general the aim of simulated annealing is to find a minimum energy for a system.
- Simple Monte Carlo simulation is a method of learning information (such as shape) about the shape of a search space. The process involves randomly selecting points within the search space.
- An example of its use is as follows: A square is partially contained within a circle. Simple Monte Carlo simulation can be used to identify what proportion of the square is within the circle and what proportion is outside the circle. This is done by randomly sampling points within the square and checking which ones are within the circle and which are not.
- Metropolis Monte Carlo simulation extends this simple method as follows: Rather than selecting new states from the search space at random, a new state is chosen by making a small change to the current state.

- If the new state means that the system as a whole has a lower energy than it did in the previous state, then it is accepted.
- If the energy is higher than for the previous state, then a probability is applied to determine whether the new state is accepted or not. This probability is called a Boltzmann acceptance criterion and is calculated as follows: e(_dE/T) where T is the current temperature of the system, and dE is the increase in energy that has been produced by moving from the previous state to the new state.
- The temperature in this context refers to the percentage of steps that can be taken that lead to a rise in energy: At a higher temperature, more steps will be accepted that lead to a rise in energy than at low temperature.
- To determine whether to move to a higher energy state or not, the probability e(_dE/T) is calculated, and a random number is generated between 0 and 1. If this random number is lower than the probability function, the new state is accepted. In cases where the increase in energy is very high, or the temperature is very low, this means that very few states will be accepted that involve an increase in energy, as e(_dE/T) approaches zero.
- The fact that some steps are allowed that increase the energy of the system enables the process to escape from local minima, which means that simulated annealing often can be an extremely powerful method for solving complex problems with many local maxima.
- Some systems use e(_dE/kT) as the probability that the search will progress to a state with a higher energy, where k is Boltzmann's constant (Boltzmann's constant is approximately 1.3807 _ 10_23 Joules per Kelvin).
- Simulated annealing usesMonte Carlo simulation to identify the most stable state (the state with the lowest energy) for a system.
- This is done by running successive iterations of metropolis Monte Carlo simulation, using progressively lower temperatures. Hence, in successive iterations, fewer and fewer steps are allowed that lead to an overall increase in energy for the system.
- A cooling schedule (or annealing schedule) is applied, which determines the manner in which the temperature will be lowered for successive iterations.
- Two popular cooling schedules are as follows:

  Tnew = Told _ dT

  Tnew = C _ Told (where C < 1.0)

- The cooling schedule is extremely important, as is the choice of the number of steps of metropolis Monte Carlo simulation that are applied in each iteration.
- These help to determine whether the system will be trapped by local minima (known as quenching). The number of times the metropolis Monte Carlo simulation is applied per iteration is for later iterations.
- Also important in determining the success of simulated annealing are the choice of the initial temperature of the system and the amount by which the temperature is decreased for each iteration.
- These values need to be chosen carefully according to the nature of the problem being solved. When the temperature, T, has reached zero, the system is frozen, and if the simulated annealing process has been successful, it will have identified a minimum for the total energy of the system.
- Simulated annealing has a number of practical applications in solving problems with large numbers of interdependent variables, such as circuit design.
- It has also been successfully applied to the traveling salesman problem.
- **Uses of Simulated Annealing**
    - Simulated annealing was invented in 1983 by Kirkpatrick, Gelatt, and Vecchi.
    - It was first used for placing VLSI* components on a circuit board.
    - Simulated annealing has also been used to solve the traveling salesman problem, although this approach has proved to be less efficient than using heuristic methods that know more about the problem.
    - It has been used much more successfully in scheduling problems and other large combinatorial problems where values need to be assigned to a large number of variables to maximize (or minimize) some function of those variables.

Real-Time A*

- Real-time A* is a variation of A*.
- Search continues on the basis of choosing paths that have minimum values of f(node) = g(node) + h(node). However, g(node) is the distance of the node from the current node, rather than from the root node.
- Hence, the algorithm will backtrack if the cost of doing so plus the estimated cost of solving the problem from the new node is less than the estimated cost of solving the problem from the current node.

- Implementing real-time A* means maintaining a hash table of previously visited states with their h(node) values.

Iterative-Deepening A* (IDA*)

- By combining iterative-deepening with A*, we produce an algorithm that is optimal and complete (like A*) and that has the low memory requirements of depth-first search.
- IDA* is a form of iterative-deepening search where successive iterations impose a greater limit on f(node) rather than on the depth of a node.
- IDA* performs well in problems where the heuristic value f (node) has relatively few possible values.
- For example, using the Manhattan distance as a heuristic in solving the eight-queens problem, the value of f (node) can only have values 1, 2, 3, or 4.
- In this case, the IDA* algorithm only needs to run through a maximum of four iterations, and it has a time complexity not dissimilar from that of A*, but with a significantly improved space complexity because it is effectively running depth-first search.
- In cases such as the traveling salesman problem where the value of f (node) is different for every state, the IDA* method has to expand 1 + 2 + 3 + . . . + n nodes = O(n2) where A* would expand n nodes.

Propositional and Predicate Logic

Logic is concerned with reasoning and the validity of arguments. In general, in logic, we are not concerned with the truth of statements, but rather with their validity. That is to say, although the following argument is clearly logical, it is not something that we would consider to be true:

All lemons are blue

Mary is a lemon

Therefore, Mary is blue

This set of statements is considered to be valid because the conclusion (Mary is blue) follows logically from the other two statements, which we often call the premises. The reason that validity and truth can be separated in this way is simple: a piece of a reasoning is considered to be valid if its conclusion is true in cases where its premises are also true. Hence, a valid set of statements such as the ones above can give a false conclusion, provided one or more of the premises are also false.

We can say: a piece of reasoning is valid if it leads to a true conclusion in every situation where the premises are true.

Logic is concerned with truth values. The possible truth values are true and false. These can be considered to be the fundamental units of logic, and almost all logic is ultimately concerned with these truth values.

Logic is widely used in computer science, and particularly in Artificial Intelligence. Logic is widely used as a representational method for Artificial Intelligence. Unlike some other representations, logic allows us to easily reason about negatives (such as, “this book is not red”) and disjunctions (“or”—such as, “He’s either a soldier or a sailor”).

Logic is also often used as a representational method for communicating concepts and theories within the Artificial Intelligence community. In addition, logic is used to represent language in systems that are able to understand and analyze human language.

As we will see, one of the main weaknesses of traditional logic is its inability to deal with uncertainty. Logical statements must be expressed in terms of truth or falsehood—it is not possible to reason, in classical logic, about possibilities. We will see different versions of logic such as modal logics that provide some ability to reason about possibilities, and also probabilistic methods and fuzzy logic that provide much more rigorous ways to reason in uncertain situations.

Logical Operators

- In reasoning about truth values, we need to use a number of operators, which can be applied to truth values.
- We are familiar with several of these operators from everyday language:

    I like apples and oranges.

    You can have an ice cream or a cake.

If you come from France, then you speak French.

- Here we see the four most basic logical operators being used in everyday language. The operators are:
  - and
  - or
  - not
  - if . . . then . . . (usually called implies)
- One important point to note is that or is slightly different from the way we usually use it. In the sentence, "You can have an icecream or a cake," the mother is usually suggesting to her child that he can only have one of the items, but not both. This is referred to as an exclusive-or in logic because the case where both are allowed is excluded.
- The version of or that is used in logic is called inclusive-or and allows the case with both options.
- The operators are usually written using the following symbols, although other symbols are sometimes used, according to the context:

  and $\wedge$

  or $\vee$

  not $\neg$

  implies $\rightarrow$

  iff $\leftrightarrow$

- Iff is an abbreviation that is commonly used to mean "if and only if."
- We see later that this is a stronger form of implies that holds true if one thing implies another, and also the second thing implies the first.
- For example, "you can have an ice-cream if and only if you eat your dinner." It may not be immediately apparent why this is different from "you can have an icecream if you eat your dinner." This is because most mothers really mean iff when they use if in this way.

Translating between English and Logic Notation

- To use logic, it is first necessary to convert facts and rules about the real world into logical expressions using the logical operators
- Without a reasonable amount of experience at this translation, it can seem quite a daunting task in some cases.
- Let us examine some examples. First, we will consider the simple operators, $\wedge$, $\vee$, and $\neg$.
- Sentences that use the word and in English to express more than one concept, all of which is true at once, can be easily translated into logic using the AND operator, $\wedge$.
- For example: "It is raining and it is Tuesday." might be expressed as: $R \wedge T$, Where R means "it is raining" and T means "it is Tuesday."
- For example, if it is not necessary to discuss where it is raining, R is probably enough.
- If we need to write expressions such as "it is raining in New York" or "it is raining heavily" or even "it rained for 30 minutes on Thursday," then R will probably not suffice. To express more complex concepts like these, we usually use predicates. Hence, for example, we might translate "it is raining in New York" as: N(R) We might equally well choose to write it as: R(N)
- This depends on whether we consider the rain to be a property of New York, or vice versa. In other words, when we write N(R), we are saying that a property of the rain is that it is in New York, whereas with R(N) we are saying that a property of New York is that it is raining. Which we use depends on the problem we are solving. It is likely that if we are solving a problem about New York, we would use R(N), whereas if we are solving a problem about the location of various types of weather, we might use N(R).
- Let us return nowto the logical operators. The expression "it is raining inNew York, and I'meither getting sick or just very tired"can be expressed as follows: $R(N) \wedge (S(I) \vee T(I))$
- Here we have used both the $\wedge$ operator, and the $\vee$ operator to express a collection of statements. The statement can be broken down into two sections, which is indicated by the use of parentheses.

- The section in the parentheses is $S(I) \vee T(I)$, which means "I'm either getting sick OR I'm very tired". This expression is "AND'ed"with the part outside the parentheses, which is $R(N)$.
- Finally, the $\neg$ operator is applied exactly as you would expect—to express negation.
- For example, It is not raining in New York, might be expressed as $\neg R(N)$
- It is important to get the $\neg$ in the right place. For example: "I'm either not well or just very tired" would be translated as $\neg W(I) \vee T(I)$
- The position of the $\neg$ here indicates that it is bound to $W(I)$ and does not play any role in affecting $T(I)$.
- Now let us see how the $\rightarrow$operator is used. Often when dealing with logic we are discussing rules, which express concepts such as "if it is raining then I will get wet."
- This sentence might be translated into logic as $R \rightarrow W(I)$
- This is read "R implies W(I)" or "IF R THEN W(I)". By replacing the symbols R and W(I) with their respective English language equivalents, we can see that this sentence can be read as "raining implies I'll get wet" or "IF it's raining THEN I'll get wet."
- Implication can be used to express much more complex concepts than this.
- For example, "Whenever he eats sandwiches that have pickles in them, he ends up either asleep at his desk or singing loud songs" might be translated as

  $$S(y) \wedge E(x, y) \wedge P(y) \rightarrow A(x) \vee (S(x, z) \wedge L(z))$$

- Here we have used the following symbol translations: $S(y)$ means that y is a sandwich. $E(x, y)$ means that x (the man) eats y (the sandwich).

  $P(y)$ means that y (the sandwich) has pickles in it.

  $A(x)$ means that x ends up asleep at his desk.

  $S(x, z)$ means that x (the man) sings z (songs).

  $L(z)$ means that z (the songs) are loud.

- The important thing to realize is that the choice of variables and predicates is important, but that you can choose any variables and predicates that map well to your problem and that help you to solve the problem.
- For example, in the example we have just looked at, we could perfectly well have used instead $S \rightarrow A \vee L$ where S means "he eats a sandwich which has pickles in it," A means "he ends up asleep at his desk," and L means "he sings loud songs."
- The choice of granularity is important, but there is no right or wrong way to make this choice. In this simpler logical expression, we have chosen to express a simple relationship between three variables, which makes sense if those variables are all that we care about—in other words, we don't need to know anything else about the sandwich, or the songs, or the man, and the facts we examine are simply whether or not he eats a sandwich with pickles, sleeps at his desk, and sings loud songs.
- The first translation we gave is more appropriate if we need to examine these concepts in more detail and reason more deeply about the entities involved.
- Note that we have thus far tended to use single letters to represent logical variables. It is also perfectly acceptable to use longer variable names, and thus to write expressions such as the following:

  $$\text{Fish}(x) \wedge \text{living}(x) \rightarrow \text{has\_scales}(x)$$

- This kind of notation is obviously more useful when writing logical expressions that are intended to be read by humans but when manipulated by a computer do not add any value.

Truth Tables

- We can use variables to represent possible truth values, in much the same way that variables are used in algebra to represent possible numerical values.
- We can then apply logical operators to these variables and can reason about the way in which they behave.
- It is usual to represent the behavior of these logical operators using truth tables.
- A truth table shows the possible values that can be generated by applying an operator to truth values.
- **Not**

- First of all, we will look at the truth table for not, $\neg$.
- Not is a unary operator, which means it is applied only to one variable.
- Its behavior is very simple:

  $\neg$ true is equal to false

  $\neg$ false is equal to true

  If variable A has value true, then $\neg$A has value false.

  If variable B has value false, then $\neg$B has value true.

- These can be represented by a truth table,

A	$\neg$ A
true	false
false	true

- **And**
  - Now, let us examine the truth table for our first binary operator—one which acts on two variables:

A	B	A $\wedge$ B
false	false	false
false	true	false
true	false	false
true	true	true

  - $\wedge$ is also called the conjunctive operator.
  - A $\wedge$B is the conjunction of A and B.
  - You can see that the only entry in the truth table for which A $\wedge$B is true is the one where A is true and B is true. If A is false, or if B is false, then A $\wedge$B is false. If both A and B are false, then A $\wedge$B is also false.
  - What do A and B mean? They can represent any statement, or proposition, that can take on a truth value.

- For example, A might represent “It’s sunny,” and B might represent “It’s warm outside.” In this case, A ∧ B would mean “It is sunny and it’s warm outside,” which clearly is true only if the two component parts are true (i.e., if it is true that it is sunny and it is true that it is warm outside).

- **Or**
  - The truth table for the or operator, ∨

A	B	A ∨ B
false	false	false
false	true	true
true	false	true
true	true	true

  - ∨ is also called the disjunctive operator.
  - A ∨ B is the disjunction of A and B.
  - Clearly A ∨ B is true for any situation except when both A and B are false.
  - If A is true, or if B is true, or if both A and B are true, A ∨ B is true.
  - This table represents the inclusive-or operator.
  - A table to represent exclusive-or would have false in the final row. In other words, while A ∨ B is true if A and B are both true, A EOR B (A exclusive-or B) is false if A and B are both true.
  - You may also notice a pleasing symmetry between the truth tables for ∧ and ∨. This will become useful later, as will a number of other symmetrical relationships.

- **Implies**
  - The truth table for implies (→) is a little less intuitive.

A	B	A → B
false	false	true
false	true	true
true	false	false
true	true	true

- This form of implication is also known as material implication
- In the statement A→B, A is the antecedent, and B is the consequent. The bottom two lines of the table should be obvious. If A is true and B is true, then A →B seems to be a reasonable thing to believe.
- For example, if A means "you live in France" and B means "You speak French," then A→B corresponds to the statement "if you live in France, then you speak French."
- Clearly, this statement is true (A→B is true) if I live in France and I speak French (A is true and B is true).
- Similarly, if I live in France, but I don't speak French (A is true, but B is false), then it is clear that A→B is not true.
- The situations where A is false are a little less clear. If I do not live in France (A is not true), then the truth table tells us that regardless of whether I speak French or not (the value of B), the statement A→B is true. A→B is usually read as "A implies B" but can also be read as "If A then B" or "If A is true then B is true."
- Hence, if A is false, the statement is not really saying anything about the value of B, so B is free to take on any value (as long as it is true or false, of course!).
- All of the following statements are valid:

  52 = 25 →4 = 4 (true →true)

  9 _ 9 = 123 →8 > 3 (false →true)

  52 = 25 →0 = 2 (false →false)

- In fact, in the second and third examples, the consequent could be given any meaning, and the statement would still be true. For example, the following statement is valid:

  52 = 25 →Logic is weird

- Notice that when looking at simple logical statements like these, there does not need to be any real-world relationship between the antecedent and the consequent.
- For logic to be useful, though, we tend to want the relationships being expressed to be meaningful as well as being logically true.

- iff

- The truth table for iff (if and only if {↔}) is as follows:

A	B	A↔B
false	false	true
false	true	false
true	false	false
true	true	true

- It can be seen that A ↔ B is true as long as A and B have the same value.
- In other words, if one is true and the other false, then A ↔ B is false. Otherwise, if A and B have the same value, A↔ B is true.

Complex Truth Tables

- Truth tables are not limited to showing the values for single operators.
- For example, a truth table can be used to display the possible values for A ∧(B ∨C).

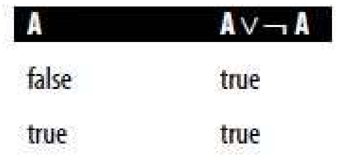
A	B	C	A ∧ (B ∨ C)
false	false	false	false
false	false	true	false
false	true	false	false
false	true	true	false
true	false	false	false
true	false	true	true
true	true	false	true
true	true	true	true

- Note that for two variables, the truth table has four lines, and for three variables, it has eight. In general, a truth table for n variables will have 2^n lines.
- The use of brackets in this expression is important. A ∧(B ∨C) is not the same as (A ∧B) ∨C.
- To avoid ambiguity, the logical operators are assigned precedence, as with mathematical operators.
- The order of precedence that is used is as follows: ¬ , ∧, ∨,→,↔

- Hence, in a statement such as $\neg A \lor \neg B \land C$, the $\neg$ operator has the greatest precedence, meaning that it is most closely tied to its symbols. $\land$ has a greater precedence than $\lor$, which means that the sentence above can be expressed as $(\neg A) \lor ((\neg B) \land C)$
- Similarly, when we write $\neg A \lor B$ this is the same as $(\neg A) \lor B$ rather than $\neg (A \lor B)$
- In general, it is a good idea to use brackets whenever an expression might otherwise be ambiguous.

Tautology

- Consider the following truth table:



A	$A \lor \neg A$
false	true
true	true

- This truth table has a property that we have not seen before: the value of the expression $A \lor \neg A$ is true regardless of the value of A.
- An expression like this that is always true is called a tautology.
- If A is a tautology, we write: $\models A$
- A logical expression that is a tautology is often described as being valid.
- A valid expression is defined as being one that is true under any interpretation.
- In other words, no matter what meanings and values we assign to the variables in a valid expression, it will still be true.
- For example, the following sentences are all valid:

  If wibble is true, then wibble is true.

  Either wibble is true, or wibble is not true.

- In the language of logic, we can replace wibble with the symbol A, in which case these two statements can be rewritten as

  $A \rightarrow A$

$A \vee \neg A$

- If an expression is false in any interpretation, it is described as being contradictory.
- The following expressions are contradictory:

  $A \wedge \neg A$

  $(A \vee \neg A) \rightarrow (A \wedge \neg A)$

Equivalence

- Consider the following two expressions:

  $A \wedge B$

  $B \wedge A$

- It should be fairly clear that these two expressions will always have the same value for a given pair of values for A and B.
- In otherwords, we say that the first expression is logically equivalent to the second expression.
- We write this as $A \wedge B$ _ $B \wedge A$. This means that the $\wedge$ operator is commutative.
- Note that this is not the same as implication: $A \wedge B \rightarrow B \wedge A$, although this second statement is also true.
- The difference is that if for two expressions e1 and e2: e1 _ e2, then e1 will always have the same value as e2 for a given set of variables.
- On the other hand, as we have seen, e1→e2 is true if e1 is false and e2 is true.
- There are a number of logical equivalences that are extremely useful.
- The following is a list of a few of the most common:

  $A \vee A$ _ A

  $A \wedge A$ _ A

  $A \wedge (B \wedge C)$ _ $(A \wedge B) \wedge C$ ($\wedge$ is associative)

  $A \vee (B \vee C)$ _ $(A \vee B) \vee C$ ($\vee$ is associative)

  $A \wedge (B \vee C)$ _ $(A \wedge B) \vee (A \wedge C)$ ($\wedge$ is distributive over $\vee$)

A ∧ (A ∨ B) _ A

A ∨ (A ∧ B) _ A

A ∧ true _ A

A ∧ false _ false

A ∨ true _ true

A ∨ false _ A

- All of these equivalences can be proved by drawing up the truth tables for each side of the equivalence and seeing if the two tables are the same.
- The following is a very important equivalence: A→B _ ¬ A ∨ B
- We do not need to use the →□ symbol at all—we can replace it with a combination of ¬ and ∨ .
- Similarly, the following equivalences mean we do not need to use ∧ or↔:

A ∧ B _ ¬ (¬ A ∨ ¬ B)

A↔ B _ ¬ (¬ (¬ A ∨ B) ∨ ¬ (¬ B ∨ A))

- In fact, any binary logical operator can be expressed using ¬ and ∨ . This is a fact that is employed in electronic circuits, where nor gates, based on an operator called nor, are used. Nor is represented by ↓, and is defined as follows:

A ↓ B _ ¬ (A ∨ B)

- Finally, the following equivalences are known as DeMorgan's Laws:

A ∧ B _ ¬ (¬ A ∨ ¬ B)

A ∨ B _ ¬ (¬ A ∧ ¬ B)

- By using these and other equivalences, logical expressions can be simplified.
- For example, (C ∧ D) ∨ ((C ∧ D) ∧ E) can be simplified using the following rule: A ∨ (A ∧ B) _ A hence, (C ∧ D) ∨ ((C ∧ D) ∧ E) _ C ∧ D

- In this way, it is possible to eliminate subexpressions that do not contribute to the overall value of the expression.

Propositional Logic

- There are a number of possible systems of logic.
- The system we have been examining so far is called propositional logic.
- The language that is used to express propositional logic is called the propositional calculus.
- A logical system can be defined in terms of its syntax (the alphabet of symbols and how they can be combined), its semantics (what the symbols mean), and a set of rules of deduction that enable us to derive one expression from a set of other expressions and thus make arguments and proofs.
- **Syntax**
  - We have already examined the syntax of propositional calculus. The alphabet of symbols, _ is defined as follows

    $$\sum = \{true, false, \neg, \rightarrow, (,), \wedge, \vee, \leftrightarrow, p1, p2, p3, \ldots, pn, \ldots\}$$

  - Here we have used set notation to define the possible values that are contained within the alphabet ∑.
  - Note that we allow an infinite number of proposition letters, or propositional symbols, p1, p2, p3, . . . , and so on.
  - More usually, we will represent these by capital letters P, Q, R, and so on,
  - If we need to represent a very large number of them, we will use the subscript notation (e.g., p1).
  - An expression is referred to as a well-formed formula (often abbreviated as wff) or a sentence if it is constructed correctly, according to the rules of the syntax of propositional calculus, which are defined as follows.
  - In these rules, we use A, B, C to represent sentences. In other words, we define a sentence recursively, in terms of other sentences.
  - The following are wellformed sentences:

    P,Q,R. . .

    true, false

(A)

¬ A

A ∧ B

A ∨ B

A→B

A↔ B

- Hence, we can see that the following is an example of a wff:

  P ∧ Q ∨ (B ∧ ¬ C)→A ∧ B ∨ D ∧ (¬ E)

- **Semantics**
  - The semantics of the operators of propositional calculus can be defined in terms of truth tables.
  - The meaning of P ∧ Q is defined as "true when P is true and Q is also true."
  - The meaning of symbols such as P and Q is arbitrary and could be ignored altogether if we were reasoning about pure logic.
  - In other words, reasoning about sentences such as P ∨ Q∧ ¬ R is possible without considering what P, Q, and R mean.
  - Because we are using logic as a representational method for artificial intelligence, however, it is often the case that when using propositional logic, the meanings of these symbols are very important.
  - The beauty of this representation is that it is possible for a computer to reason about them in a very general way, without needing to know much about the real world.
  - In other words, if we tell a computer, "I like ice cream, and I like chocolate," it might represent this statement as A ∧ B, which it could then use to reason with, and, as we will see, it can use this to make deductions.

Predicate Calculus

- **Syntax**

- Predicate calculus allows us to reason about properties of objects and relationships between objects.
- In propositional calculus, we could express the English statement "I like cheese" by A. This enables us to create constructs such as $\neg$ A, which means "I do not like cheese," but it does not allow us to extract any information about the cheese, or me, or other things that I like.
- In predicate calculus, we use predicates to express properties of objects. So the sentence "I like cheese" might be expressed as L(me, cheese) where L is a predicate that represents the idea of "liking." Note that as well as expressing a property of me, this statement also expresses a relationship between me and cheese. This can be useful, as we will see, in describing environments for robots and other agents.
- For example, a simple agent may be concerned with the location of various blocks, and a statement about the world might be T(A,B), which could mean: Block A is on top of Block B.
- It is also possible to make more general statements using the predicate calculus.
- For example, to express the idea that everyone likes cheese, we might say $(\forall x)(P(x) \rightarrow L(x, C))$
- The symbol $\forall$ is read "for all," so the statement above could be read as "for every x it is true that if property P holds for x, then the relationship L holds between x and C," or in plainer English: "every x that is a person likes cheese." (Here we are interpreting P(x) as meaning "x is a person" or, more precisely, "x has property P.")
- Note that we have used brackets rather carefully in the statement above.
- This statement can also be written with fewer brackets: $\forall x\ P(x) \rightarrow L(x, C)$, $\forall$ is called the universal quantifier.
- The quantifier $\exists$ can be used to express the notion that some values do have a certain property, but not necessarily all of them: $(\exists x)(L(x,C))$
- This statement can be read "there exists an x such that x likes cheese."
- This does not make any claims about the possible values of x, so x could be a person, or a dog, or an item of furniture. When we use the existential

quantifier in this way, we are simply saying that there is at least one value of x for which L(x,C) holds.

- The following is true: $(\forall x)(L(x,C)) \rightarrow (\exists x)(L(x,C))$, but the following is not: $(\exists x)(L(x,C)) \rightarrow (\forall x)(L(x,C))$

- Relationships between $\forall$ and $\exists$
  - It is also possible to combine the universal and existential quantifiers, such as in the following statement: $(\forall x)\ (\exists y)\ (L(x,y))$.
  - This statement can be read "for all x, there exists a y such that L holds for x and y," which we might interpret as "everyone likes something."
  - A useful relationship exists between $\forall$ and $\exists$. Consider the statement "not everyone likes cheese." We could write this as

    $\neg (\forall x)(P(x) \rightarrow L(x,C))$ ------------- (1)

  - As we have already seen, $A \rightarrow B$ is equivalent to $\neg A \vee B$. Using DeMorgan's laws, we can see that this is equivalent to $\neg (A \wedge \neg B)$. Hence, the statement (1) above, can be rewritten:

    $\neg (\forall x) \neg (P(x) \wedge \neg L(x,C))$ ------------- (2)

  - This can be read as "It is not true that for all x the following is not true: x is a person and x does not like cheese." If you examine this rather convoluted sentence carefully, you will see that it is in fact the same as "there exists an x such that x is a person and x does not like cheese." Hence we can rewrite it as $(\exists x)(P(x) \wedge \neg L(x,C))$ ------------- (3)
  - In making this transition from statement (2) to statement (3), we have utilized the following equivalence: $\exists x \cong \neg (\forall x) \neg$
  - In an expression of the form $(\forall x)(P(x, y))$, the variable x is said to be bound, whereas y is said to be free. This can be understood as meaning that the variable y could be replaced by any other variable because it is free, and the expression would still have the same meaning, whereas if the variable x were to be replaced by some other variable in P(x,y), then the meaning of the

expression would be changed: (∀x)(P(y, z)) is not equivalent to (∀x)(P(x, y)), whereas (∀x)(P(x, z)) is.

- Note that a variable can occur both bound and free in an expression, as in (∀x)(P(x,y,z) → (∃y)(Q(y,z)))
- In this expression, x is bound throughout, and z is free throughout; y is free in its first occurrence but is bound in (∃y)(Q(y,z)). (Note that both occurrences of y are bound here.)
- Making this kind of change is known as substitution.
- Substitution is allowed of any free variable for another free variable.

- **Functions**
  - In much the same way that functions can be used in mathematics, we can express an object that relates to another object in a specific way using functions.
  - For example, to represent the statement "my mother likes cheese," we might use L(m(me),cheese)
  - Here the function m(x) means the mother of x. Functions can take more than one argument, and in general a function with n arguments is represented as f(x1, x2, x3, . . . , x_n)

First-Order Predicate Logic

- The type of predicate calculus that we have been referring to is also called firstorder predicate logic (FOPL).
- A first-order logic is one in which the quantifiers ∀ and ∃ can be applied to objects or terms, but not to predicates or functions.
- So we can define the syntax of FOPL as follows. First,we define a term:
- A constant is a term.
- A variable is a term. f(x1, x2, x3, . . . , xn) is a term if x1, x2, x3, . . . , xn are all terms.
- Anything that does not meet the above description cannot be a term.
- For example, the following is not a term: ∀x P(x). This kind of construction we call a sentence or a well-formed formula (wff), which is defined as follows.

- In these definitions, P is a predicate, x1, x2, x3, . . . , xn are terms, and A,B are wff 's. The following are the acceptable forms for wff 's:

  P(x1, x2, x3, . . . , xn)

  ¬ A

  A ∧ B

  A ∨ B

  A→B

  A↔ B

  (∀x)A

  (∃x)A

- An atomic formula is a wff of the form P(x1, x2, x3, . . . , xn).
- Higher order logics exist in which quantifiers can be applied to predicates and functions, and where the following expression is an example of a wff:

  (∀P)(∃ x)P(x)

Soundness

- We have seen that a logical system such as propositional logic consists of a syntax, a semantics, and a set of rules of deduction.
- A logical system also has a set of fundamental truths, which are known as axioms.
- The axioms are the basic rules that are known to be true and from which all other theorems within the system can be proved.
- An axiom of propositional logic, for example, is A→(B→A)
- A theorem of a logical system is a statement that can be proved by applying the rules of deduction to the axioms in the system.
- If A is a theorem, then we write ├ A
- A logical system is described as being sound if every theorem is logically valid, or a tautology.
- It can be proved by induction that both propositional logic and FOPL are sound.
- **Completeness**

- A logical system is complete if every tautology is a theorem—in other words, if every valid statement in the logic can be proved by applying the rules of deduction to the axioms. Both propositional logic and FOPL are complete.

- **Decidability**
  - A logical system is decidable if it is possible to produce an algorithm that will determine whether any wff is a theorem. In other words, if a logical system is decidable, then a computer can be used to determine whether logical expressions in that system are valid or not.
  - We can prove that propositional logic is decidable by using the fact that it is complete.
  - We can prove that a wff A is a theorem by showing that it is a tautology. To show if a wff is a tautology, we simply need to draw up a truth table for that wff and show that all the lines have true as the result. This can clearly be done algorithmically because we know that a truth table for n values has 2n lines and is therefore finite, for a finite number of variables.
  - FOPL, on the other hand, is not decidable. This is due to the fact that it is not possible to develop an algorithm that will determine whether an arbitrary wff in FOPL is logically valid.
- **Monotonicity**
  - A logical system is described as being monotonic if a valid proof in the system cannot be made invalid by adding additional premises or assumptions.
  - In other words, if we find that we can prove a conclusion C by applying rules of deduction to a premise B with assumptions A, then adding additional assumptions $A\neg$ and $B\neg$ will not stop us from being able to deduce C.
  - Monotonicity of a logical system can be expressed as follows:

    If we can prove $\{A, B\} \vdash C$,

    then we can also prove: $\{A, B, A\_, B\_\} \vdash C$.
  - In other words, even adding contradictory assumptions does not stop us from making the proof in a monotonic system.
  - In fact, it turns out that adding contradictory assumptions allows us to prove anything, including invalid conclusions. This makes sense if we recall the line in

the truth table for →, which shows that false → true. By adding a contradictory assumption, we make our assumptions false and can thus prove any conclusion.

Modal Logics and Possible Worlds

- The forms of logic that we have dealt with so far deal with facts and properties of objects that are either true or false.
- In these classical logics, we do not consider the possibility that things change or that things might not always be as they are now.
- Modal logics are an extension of classical logic that allow us to reason about possibilities and certainties.
- In other words, using a modal logic, we can express ideas such as "although the sky is usually blue, it isn't always" (for example, at night). In this way, we can reason about possible worlds.
- A possible world is a universe or scenario that could logically come about.
- The following statements may not be true in our world, but they are possible, in the sense that they are not illogical, and could be true in a possible world:

  Trees are all blue.

  Dogs can fly.

  People have no legs.

- It is possible that some of these statements will become true in the future, or even that they were true in the past.
- It is also possible to imagine an alternative universe in which these statements are true now.
- The following statements, on the other hand, cannot be true in any possible world:

  $A \land \neg A$

  $(x > y) \land (y > z) \land (z > x)$

- The first of these illustrates the law of the excluded middle, which simply states that a fact must be either true or false: it cannot be both true and false.
- It also cannot be the case that a fact is neither true nor false. This is a law of classical logic, it is possible to have a logical system without the law of the excluded middle, and in which a fact can be both true and false.

- The second statement cannot be true by the laws of mathematics. We are not interested in possible worlds in which the laws of logic and mathematics do not hold.
- A statement that may be true or false, depending on the situation, is called contingent.
- A statement that must always have the same truth value, regardless of which possible world we consider, is noncontingent.
- Hence, the following statements are contingent:

  $A \wedge B$

  $A \vee B$

  I like ice cream.

  The sky is blue.

- The following statements are noncontingent:

  $A \vee \neg A$

  $A \wedge \neg A$

  If you like all ice cream, then you like this ice cream.

- Clearly, a noncontingent statement can be either true or false, but the fact that it is noncontingent means it will always have that same truth value.
- If a statement A is contingent, then we say that A is possibly true, which is written $\Diamond$ A
- If A is noncontingent, then it is necessarily true, which is written $\Box$ A
- **Reasoning in Modal Logic**
  - ➢ It is not possible to draw up a truth table for the operators $\Diamond$ and $\Box$
  - ➢ The following rules are examples of the axioms that can be used to reason in this kind of modal logic:

    $\Box A \rightarrow \Diamond A$

    $\Box \neg A \rightarrow \neg \Diamond A$

    $\Diamond A \rightarrow \neg \Box A$

➢ Although truth tables cannot be drawn up to prove these rules, you should be able to reason about them using your understanding of the meaning of the ◊ and □ operators.

Possible world representations

- It describes method proposed by Nilsson which generalizes firtst order logic in the modeling of uncertain beliefs
- The method assigns truth values ranging from 0 to 1 to possible worlds
- Each set of possible worlds corresponds to a different interpretation of sentences contained in a knowledge base denoted as KB
- Consider the simple case where a KB contains only the single sentence S, S may be either true or false. We envision S as being true in one set of possible worlds W_1 and false in another set W_2 . The actual world , the one we are in, must be in one of the two sets, but we are uncertain which one. Uncertainty is expressed by assigning a probability P to W_1 and 1 – P to $W_{2.}$ We can say then that the probability of S being true is P
- When KB contains L sentences, $S_{1,...}$ S_L , more sets of possible worlds are required to represent all consistent truth value assignments. There are 2^L possible truth assignments for L sentences.
- Truth Value assignments for the set {P. P→Q, Q}

Consistent			Inconsistent		
P	Q	P → Q	P	Q	P → Q
True	True	True	True	True	False
True	False	False	True	False	True
False	True	True	False	True	False
False	False	True	False	False	False

- They are based on the use of the probability constraints

  $0 \leq p_i \leq 1$, and $\sum_i p_i = 1$