MATH0071—Applications of Logic

Russell Bradford[1]

1 Intro

> "The question of whether a computer can think is no more interesting than the question of whether a submarine can swim." E. W. Dijkstra

This course is called "Applications of Logic," which is a decidedly non-trendy thing to call it. This course is *described* as being a course on Artificial Intelligence, but that term is so over-used, and so hyped that we like to get back to the basics of the subject. Another title might be "Knowledge Engineering," but that is too presumptuous to be true. This course is about using logic (and other things) to solve problems.

So what is artificial intelligence? You can argue the philosophical twists and turns for ages (not least that some regard the phrase as being an oxymoron), but for our purposes we are going to use some simple definitions. You can choose any. AI is

1. the attempt to automate what we perceive as intelligent behaviour in humans (this avoids defining intelligence)
2. the study of how to make computers do those things that they are currently bad at (so AI wishes to make itself redundant)
3. the study of how to extrapolate beyond your experience (!)

And these even avoid defining what we mean by artificial.

There was a definition of intelligence, given by Turing, in the famous Turing test. To test an entity for intelligence do the following: have it sit (metaphorically) in front of a terminal, and a human examiner sit at another connected terminal, out of sight of the entity. The examiner does not know if the connection is to the entity or another human somewhere else. The examiner is allowed to ask any question of the entity (formal or informal) and the entity may answer in any way it pleases. If the examiner can not determine whether there is a human or not on the other terminal, we decree the entity to have intelligence. The entity may answer questions truthfully or not, correctly or not, or evade questions, and in general do anything to make the examiner believe that it is a human.

This, some believe, is is the true end-product of AI–and is the Hollywood version of AI—but in fact there is a lot more to it than that. These days AI encompasses a large range of problems:

1. Mundane tasks
   (a) perception: vision, hearing, speech
   (b) natural language: understanding, generation, translation (the computer in Star Trek)
   (c) common sense reasoning
   (d) robot control

[1] These notes include text from Mike Dewar

2. Formal tasks
   (a) games: chess, backgammon, chequers, go
   (b) mathematics: geometry, logic, integral calculus
3. Expert tasks
   (a) engineering: design, fault finding, planning
   (b) scientific analysis
   (c) medical diagnosis
   (d) financial analysis

The last bunch are collected together under the umbrella of "expert systems," where the aim is to reproduce the abilities of an expert engineer or doctor.

Not everyone agrees on the content of AI: personally I believe doing integral calculus is an algorithmic task: see MATH0070—Computer Algebra.

2 This Course

The content of this course will generally follow the book

> "Artificial Intelligence," by E Rich and K Knight, McGraw Hill

which is a reasonable book as these things go. You might want to peruse

- "Introduction to Artificial Intelligence," by Charniak and McDermott,
- "Artificial Intelligence," by P Winston,,
- "Artifician Intelligence," by Luger and Stubblefield,

but these are more involved with detail, and lose the overview.

You should also know some Lisp, as it will be needed for the assignments to come later in the course: this I will cover if demands are great enough.

Thus we find:

- What is AI?
- Searching
- Knowledge and Predicate Logic
- Uncertainty, Statistical reasoning
- Knowledge representation
- Game playing

- Planning
- Natural Language
- Expert systems

In particular we are going to miss out perception (see C86-Speech, and MATH0080 Computer Vision), mathematics (see MATH0070 Computer Algebra), and control (see Engineering).

This course includes COURSEWORK! This is 25% of the marks for the course, and will appear probably about half-way through the semester. It is very likely to involve programming some AI technique in Lisp.

3 What is an AI Technique?

There needs to be some description of what distinguishes an "AI technique" from any of the other methods for solving problems that you have come across in the last few years. Most of the problems you have seen so far are (in principle) well-defined, amenable to a specific solution, and static (in the sense that the problem does not change as you work on it). Roughly speaking, if you remove these constraints, you get an AI problem. Namely it is

- ill-defined, or
- no algorithmic solution is evident (or possible), or
- it is constantly changing or being updated.

For example: "Read this paper and report on the salient points" it an ill-defined problem, any number of reports might be acceptable as solutions. "Write a chess program" is a specific problem, but no algorithmic solution that is guaranteed to win or draw would be likely to run in a reasonable amount of time. "Drive this car to London" involves many changing situations, and requires a lot of adaptivity to solve.

It will become apparent that to solve a problem intelligently requires a lot of *knowledge*. Unfortunately, knowledge has a number of somewhat undesirable properties, namely there being a lot of it, it is hard to collect and characterise accurately, and it is constantly changing and being updated. Knowledge is not the same as data, because it is usually structured in some way. That structure is designed to reflect the way it will be used.

We now illustrate the difference between "algorithmic" and "AI" techniques using the game of noughts and crosses.

3.1 Example: 3 programs to play noughts and crosses

This example is taken from Rich & Knight, though being Americans they call the game "tic-tac-toe". They also give another example which you may want to look at.

3.1.1 Program 1: The "brute-force" or precomputed approach

This program requires two data structures. First, a 9-element vector which represents the board as follows:

1	2	3
4	5	6
7	8	9

An entry in the vector will be 0 if the square is blank, 1 if it has an X and 2 if it has an O. We then construct a "move table", each of whose elements is a possible configuration of the board. There are clearly 3^9 possible configurations (since we ignore the facts that some configurations are actually unattainable, and that by symmetry many positions are actually identical) i.e. 19683 entries, each of which is a 9-vector. We construct the move table with great care so that the following algorithm will work:

1. View the vector representing the board as a ternary number (which works since the elements are in the range 0 to 2).
2. Use this number as an index into the move table.
3. We have arranged that the vector stored at this location represents the current board *after the best possible move is made.* So replace the current board with this vector and, unless this is an end position, go back to step 1.

The advantages of this approach is that it is very fast, and provided we constructed the move table carefully, could play an optimal game of noughts and crosses. It does, however, have several disadvantages:

- it takes a lot of space, since every possibility is stored explicitly;
- an expert player will need to do a tremendous amount of work to specify the entries in the table;
- the chances are that errors will be introduced into the move table, and be very hard to detect;
- the program will only work for noughts and crosses — we would have to start from scratch to develop e.g., a 3-D version (and in fact that would overwhelm present computer memories).
- the "best possible" next move may very well depend on the behaviour of the opponent, and so is not amenable to pre-computation.

The problem is really that the strategy we are employing has been worked out in advance, and is implicit in the contents of the move table. Lets see if we can do better by specifying the strategy in a more abstract way.

3.1.2 Program 2: Explicit strategy

We will use the same representation for the board, and keep track of what turn it is using a simple variable which will range form $1 \ldots 9$.

Our algorithm will use three procedures as follows:

`Make2` Returns a 5 if the centre square is blank, otherwise it returns the index of any non-blank corner square.

`PossibleWin(p)` Returns 0 if player p cannot win on his or her next move, otherwise the number of the winning square.

`Go(n)` Makes a move in square n. Also increment the turn counter.

The algorithm has a specific strategy for each turn. If it is X it plays the odd-numbered moves, otherwise it plays the even-numbered ones. The strategy is as follows:

1. `Go(1)`
2. if `empty? Board(5)` then `Go(5)` else `Go(1)`
3. if `empty? Board(9)` then `Go(9)` else `Go(3)`
4. if `PossibleWin(X)` then `Go(PossibleWin(X))` else `Go(Make2)`
5. if `PossibleWin(X)` then `Go(PossibleWin(X))`
   else if `PossibleWin(O)` then `Go(PossibleWin(O))`
   else if `empty? Board(7)` then `Go(7)`
   else `Go(3)`
6. if `PossibleWin(O)` then `Go(PossibleWin(O))`
   else if `PossibleWin(X)` then `Go(PossibleWin(X))`
   else `Make2`
7. if `PossibleWin(X)` then `Go(PossibleWin(X))`
   else if `PossibleWin(O)` then `Go(PossibleWin(O))`
   else `Go(`anywhere blank`)`
8. if `PossibleWin(O)` then `Go(PossibleWin(O))`
   else if `PossibleWin(X)` then `Go(PossibleWin(X))`
   else `Go(`anywhere blank`)`
9. if `PossibleWin(X)` then `Go(PossibleWin(X))`
   else if `PossibleWin(O)` then `Go(PossibleWin(O))`
   else `Go(`anywhere blank`)`

This approach is slower than the last one because of all the conditions we have to check, but it is a lot more compact. Although both programs probably use the same strategy, it is much clearer as represented here. Because it is easier to understand, the strategy can be checked or changed with ease. However the strategy is still laid down in advance by the programmer, and the strategy is still specific to playing 2-D noughts and crosses.

3.1.3 Program 3: The minimax algorithm

For this approach we need a data structure which contains the current board position, together with a list of all the possible positions which could result from the next move. In addition, we have a variable which is an estimate of how good the current position is for the player whose turn it is.

How do we estimate how good our position is? The procedure is as follows:

1. See if it is a win. If it is, give it the highest rating.

2. Otherwise, consider all the possible moves that our opponent could make next, and find which is the worst for us (by recursively calling this procedure). Assume that our opponent will make this move. Whatever rating it has, assign it to the current move.
3. The best node is that with the highest rating.

This algorithm looks for sequences of moves which lead to a win. It attempts to maximise our chance of winning while at the same time assuming that our opponent is trying to minimise that likelihood, This is a standard technique used in AI game-playing programs, and is called the *minimax* method.

This process is slower than the others, but the approach is generic: it could be adapted to playing many other games. It is extremely easy to adapt: you might wish to ignore some "unpromising" avenues to make it go faster, for example. Alternatively, you might only look a few moves ahead.

It odes have problems, though: it assumes that the opponent will always make their best move. It may be that we are in a losing position, but the opponent does not realise that and makes a sub-optimal move that opens up the possibility of our winning. This could be helped along by our making a confusing move rather than an optimal move. On the other hand the minimax method could still apply if we could factor the opponents sense of confusion into the evaluation function!

Although this technique will be less efficient than the others for small problems, it demonstrates some important features of the methods we will be studying:

- It uses *search* to look for solutions.
- It uses *knowledge* about the problem domain to rate solutions.
- It exploits *abstraction* to isolate the domain-independent features of the problem-solving process from the domain-dependent ones.

It is clear that AI solutions will require the ability to perform efficient searches over knowledge: this we shall now study in detail.

3.1.4 A Learning Method

A whimsical approach to this problem uses matchboxes with numbered markers inside to make decisions. A number of matchboxes are prepared with pictures of play positions on the front (we can make good use of symmetry here). Inside is a marker for each possible move from this position.

To teach the system we play many games. At each position take a random marker from the appropriate matchbox and play that move. At the end of the game if we lost, remove the marker of the last move we made. Thus we can never make that losing move again.

Repeat many times. Ultimately the system learns never to make losing moves. If we come across an empty matchbox, we know we are in a losing position, and we should remove the marker from the matchbox that led us to this position.

An alternative learning strategy is to reward wins instead of penalising losses. At the end of a game if we won add a new marker that is a copy of the one that led us to the winning position. With this strategy the system gradually prefers winning moves over losing ones, though it can still lose.

We can use both strategies simultaneously, of course! The system will not eliminate losing moves as fast, since it is less likely to make them, but eventually will do so.

4 Problem Spaces and Searching

To search through knowledge we need first to determine over what we are searching: these are *problem spaces*.

We are motivated by the desire to construct programs which can solve particular problems. To build such programs we need to do four things:

1. Define the problem as precisely as possible, paying particular attention to the start situation and what constitutes a solution.
2. Analyse the problem, to gain an understanding of how we can solve it.
3. Identify the knowledge needed to solve the problem, and codify it in some way.
4. Choose the most appropriate problem-solving techniques, and apply them to the problem in hand.

The first of these can be regarded as saying "we are here, but we want to be somewhere else", the last as a strategy for getting there. We will concentrate on this first, before looking at how we represent knowledge later in the course.

4.1 Problem Definition

We need to think briefly about how we can represent problems on a machine.

Earlier, we designed a program to play noughts and crosses. Writing systems to play games has always been a favourite area of AI research for a number of reasons:

- The problems tend to be well specified: we know what the starting and finishing positions are.
- There is a clear goal: to beat the opponent!
- The rules of the game provide well-defined methods for getting from the starting point to this goal.

We can think of each possible configuration of the board as a *state*. The set of all possible (legal) configurations is called the *state space*. The legal moves provide us with a way of moving between states. These moves need to be expressed in some way. We could simply enumerate all the individual possibilities, but this is in practice not possible. There are at most 9! possible moves in noughts and crosses, but imagine the possibilities in a game like chess, which has roughly 10^{120} different board configurations! Clearly no person could ever produce a complete set of rules for chess, and no computer could easily handle them. Thus it is essential to encapsulate each legal move in as general a way as possible. Supposing we represent a chess board in the usual way:

Black

8								
7								
6								
5								
4								
3								
2								
1								
	a	b	c	d	e	f	g	h

White

then the rules for moving a white pawn can be written as:

1. Pawn at $(x, 2)$ and $(x, 3)$ empty and $(x, 4)$ empty then move to $(x, 4)$.
2. Pawn at (x, y) and $y < 7$ and $(x, y + 1)$ empty then move to $(x, y + 1)$.
3. Pawn at $(x, 7)$ and $(x, 8)$ empty then move to $(x, 8)$ and queen.
4. Pawn at (x, y) and $(x - 1, y + 1)$ has a black piece then move to $(x - 1, y + 1)$ and remove black piece.
5. Pawn at (x, y) and $(x + 1, y + 1)$ has a black piece then move to $(x + 1, y + 1)$ and remove black piece.

If we defined all the legal moves of chess like this, and we defined the *start state* in the obvious way then we nearly have enough information to write a chess-playing program. We also need to define the goal state as e.g. a position where the opponents king is in check and he or she has no legal move (checkmate), and the possible final states as checkmate for either player or a stalemate. We have now reduced the problem of playing chess to that of moving around in a state space. Notice that this is equivalent to teaching a human how to set up the board and what the legal moves are — we haven't yet learned how to play chess well! In fact, what we have done is simply produced a formal definition of the game, which we can manipulate easily. In this case it was easy — chess is an artificial problem which has a clear structure, for "real life" problems coming up with a complete, formal description is not anything like as easy. In summary, our description comprises the following components:

1. A state space containing all the possible configurations of the board (implicit in the above).
2. One start state.
3. A number of goal states (checkmates).
4. A set of rules describing how to move from one state to another.

The problem of winning at chess then reduces to a search for a legal sequence of states leading from the start state to a goal state. While we can use other approaches as well, search is usually fundamental to the problem-solving process. In fact, real chess programs tend to use table lookup to play the highly-stylised openings and endgames, and search to conduct the skirmishing in between. In this case the goal states are in fact those which are known to lead to a winnable endgame.

4.2 Production Systems

Since search is so important it would be nice if we had a framework which enabled us to describe and perform the search process. It is common to use a framework called a *production system* for this. These are sometimes delivered in a completely problem-independent way (a so-called *expert system shell*) and consist of:

- A mechanism for defining rules with two parts: a *pattern* and an *action*. The pattern determines when the rule is applicable, in the earlier example:

  Pawn at $(x, 2)$ and $(x, 3)$ and $(x, 4)$ empty then move to $(x, 4)$

  the pattern is the fact that the pawn is in the second rank with two blank squares in front of it, while the action is the actual movement of the piece.

- A knowledge base which contains structured information about the task in hand. The knowledge base may change during the solution process.
- A *control strategy* which determines in which order the rules will be examined, and what to do if several patterns match simultaneously.
- A means of applying (or *firing*) a rule, and possibly updating the knowledge base.

In our chess-playing example, we defined a set of rules which defined the legal moves, and we defined the starting state. This is our initial knowledge base. As the game progresses we will update the knowledge base to reflect the current configuration of the board. The process of updating the board will be carried out when we fire a rule. We must now address the problem of deciding which of several possible rules to apply at any given time.

4.3 Problem Characteristics

Before we describe some search methods, we should note some things about problems and their descriptions.

Different problems can be solved by different techniques. There is one particularly important characteristic which has a big impact on which method we choose: are we "planning" or "doing"?

Often when trying to solve a problem we are just *planning* a solution, and we can modify the plan if we want. This has an important consequence: if we reach a position we don't like, we can *backtrack* to a previous position and try a different path. In fact, we can backtrack as far as we want. Now suppose that we are playing chess. Once we have made a move, we are committed to that move, if our opponent does not behave as expected we cannot just roll back and try again.

If the problem space is static we can know exactly what the outcome of a particular action will be. Many problems, however, involve chance or incomplete information (e.g., poker). Now it is not possible to search a problem space as that space is changing, so we might need to introduce probabilities into our computations. We can make plans, but we might have to revise them in the light of new knowledge.

Another important thing to consider is the direction of the search. We have implicitly assumed that we start from some initial state and reason towards a goal. This is not always the right way to tackle a problem: for example in proving a mathematical theorem it is often helpful to start from the goal state and work backwards.

In deciding whether to use *forward reasoning* or *backwards reasoning* there are various characteristics of the search space to be born in mind. If there are a large number of goal states and one start state then reasoning in a forward direction looks a better choice. However supposing there is only one start state and one goal state, then the best direction to go is the one with the smallest *branching factor*, i.e., the one with the fewest choices at each node. Sometimes it is easier to solve a maze by starting at the centre and working outwards!

Finally, we have talked so far about searching a tree, but in many cases we might be better off searching a graph, since this will prevent us repeating moves incessantly. This is more memory intensive, as it requires us to note when we have seen a state before. Occasionally this can be a good thing, because many paths can lead to the same state, thus reducing the problem space we have to search.

4.3.1 Control Strategies

Often we have several rules we could fire, and we have to choose one of them. We could select one at random, but this may not be the best strategy: we want to make sure we are always moving towards our goal, and we want to be systematic: in other words we don't want to repeat the same sequence of operations several times. This is easy to state but in practice it can be hard to achieve.

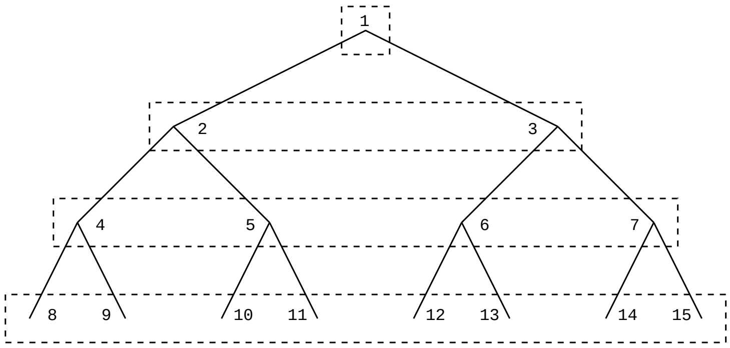
It is useful to consider the possible subsequent states as a tree with the current state as the root. Each potential rule gives rise to a separate subtree. We wish to search this tree until we find a goal state. The series of states which leads us to this goal state from our current position can be thought of as a *plan*. By following this plan we will be successful.

4.3.2 Breadth-First Search

We start with two methods for searching: these are general "computer science" methods, not restricted to use in AI.

The idea of breadth-first search is that we move down successive levels of the tree until we find a solution.

1. set the variable `nodelist` to contain the initial state
2. while `nodelist` is not empty do
   (a) remove the first element, E, from `nodelist`
   (b) generate all the children nodes of E
   (c) if any is a goal state, return it and end
   (d) else append the new states to the `nodelist`



Advantages: guaranteed to find a solution if it exists (will not get trapped down long blind alleys); will find a minimal solution; finds all minimal solutions simultaneously.

Disadvantages: uses a lot of memory (to store the whole of the tree computed so far)

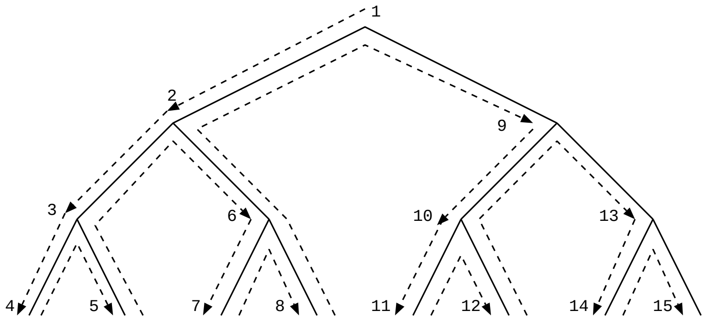
4.3.3 Depth-First Search

The idea of this strategy is that we follow a sequence of successive states until we find one which yields success. The recursive algorithm can be expressed as follows:

1. If we are in a goal state then we have succeeded.
2. Otherwise, repeat until success or failure is signalled:
   (a) Generate a new child of the initial state. If there are no more children then signal failure.

(b) recursively call the depth-first search with this new state as the initial state.

(c) if success is returned, return success, otherwise continue this loop



The action of returning fail and crawling back up the tree until we find a new branch to explore is called *backtracking*.

Advantages: requires relatively little memory (since we only store that part of the tree we are currently searching);

Disadvantages: might get stuck down a "blind alley" where we will search forever without finding a solution; might not find a minimal solution; finds only a single solution (but can be extended to find more solutions if needed)

Both breadth-first and depth-first searches need modifying if there is a possibility of loops in the search space.

4.3.4 Heuristic searches

Both depth- and breadth-first search strategies exhaustively search the state space, looking for a solution. They use no domain-specific methods to find a solution, and are often known as *strong* strategies. In practice it is often unrealistic to do this, since the number of nodes which has to be searched is too great. For example, consider the *Travelling Salesman Problem.* A salesman has a list of n cities, each of which he must visit exactly once. Every pair of cities is connected by a direct road. The problem is to determine the shortest possible round trip starting from any given city. It is clear that the number of possible routes is $(n-1)!$. Exhaustive search will be fine for small n, but as n grows the time taken to explore the total search space will grow much faster — a phenomenon known as *combinatorial explosion.* In this case both depth- and breadth- first search fare equally badly, since the problem is to find the optimal route and so all possibilities must be explored. (Incidentally, the travelling salesman is one of a class of problems that are hard to solve exactly, but good techniques exist that solve it to a good accuracy.)

Since the time taken to look at one route will be proportional to n, we will say that the total time to search all possible routes will be proportional to $n!$. If we have 10 cities, we will take 3628800 steps. Double the number of cities to 20, and the time taken shoots up to $2432902008176640000 \approx 2 \times 10^{18}$, a factor of the order of 10^{11} (there are fewer seconds in 3000 years!). An improvement is to use the *nearest neighbour heuristic*, which at each step selects the closest unvisited city. This takes about N^2 time, and produces a solution that is boundedly far from the exact answer. But in general this sort of improvement is hard to find.

Thus we are lead to consider *heuristic* search methods, that is methods that are not guaranteed to be best, or fast or whatever, but nevertheless generally produce a good enough answer. These are also known as *weak* strategies, perhaps misleadingly as they might find solutions that the strong strategies cannot through lack of time of space. Unfortunately, heuristics are not always so successful, or not always amenable to proofs of how good (or bad) they are.

For example, the *branch and bound* technique works by generating complete paths, but giving up when the length of a path exceeds the shortest one found so far. We are guaranteed to find the shortest path, but this algorithm still takes exponential time for the travelling salesman.

Heuristics are valuable, because we often do not need the absolutely optimal solution, but only need a good approximation (this appears to be how humans solve problems, too). Also bad cases that confound heuristics rarely arise in the real world: the real world is only a tiny subset of the mathematical world.

There are two major ways that heuristic knowledge can be incorporated into rule-based searches:

1. in the rules themselves (perhaps only heuristically "sensible" rules are included)
2. as a heuristic function that evaluates how good a problem state is.

Heuristic functions map from problem states to measures of goodness, usually numerical. The hard part is getting a good map that accurately reflects the state of problem. Sometimes simple functions are adequate (the length of the path so far, perhaps), but often complex functions are required (e.g., for chess). The heuristic function should indicate which is the (probably) the best path to follow when there is a choice of more than one, but care must be taken that the time spent evaluating the function does not outweigh the effort saved in the search process: a perfect heuristic function would be to search all paths to their end, but clearly this is what we are trying to avoid.

4.4 Heuristic Strategies

These are strategies which attempt to find a good solution, rather than the optimal one. Many heuristics apply in only one problem domain and exploit knowledge which is specific to that domain. We will briefly consider some general heuristics.

We have already see depth- and breadth- first searching. Strictly speaking these are not heuristics, but generally they have heuristics added to them to make them usable, for example, a depth-first search may stop after searching to a certain depth in order to avoid chasing down blind alleys.

4.4.1 Generate-and-test

This follows the following

1. generate a possible solution (this may be a state, or a path from the start state to another state)
2. test to see if this is a solution
3. if so, return it, else repeat from step 1.

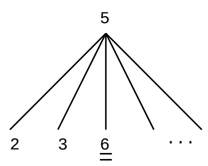
This will find a solution if it exists, and if the generation is done systematically. On the other hand, it is clearly a poor strategy without some extra help. The generation is generally done in a depth-first manner with backtracking.

4.4.2 Hill Climbing

Hill climbing adds to generate-and-test by helping the decision of which way to move in the problem space by use of information returned from the test function. Generate-and-test is a special case of hill climbing with the test returning a single bit (yes/no).

A heuristic way of finding the top of a mountain is to walk uphill. Similarly, in search space, we evaluate possible successor nodes until we find one that is "better" than the current one. We choose this and continue recursively. Choosing the best successor node may involve a lot of detailed knowledge, this is where the heuristic comes in.

1. Evaluate the initial state. If it is the goal, we are finished.
2. Loop until there are no more successor states:
   (a) select a successor state, and evaluate it
   (b) if it is a goal, we are finished
   (c) else if it is better than the current state make it the current state



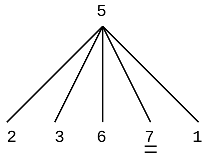
This uses information from the evaluation to guide the search. It does, however, rely on a good evaluation function which might be hard to find.

There are problems with this method which we shall describe in a moment, after giving a modification.

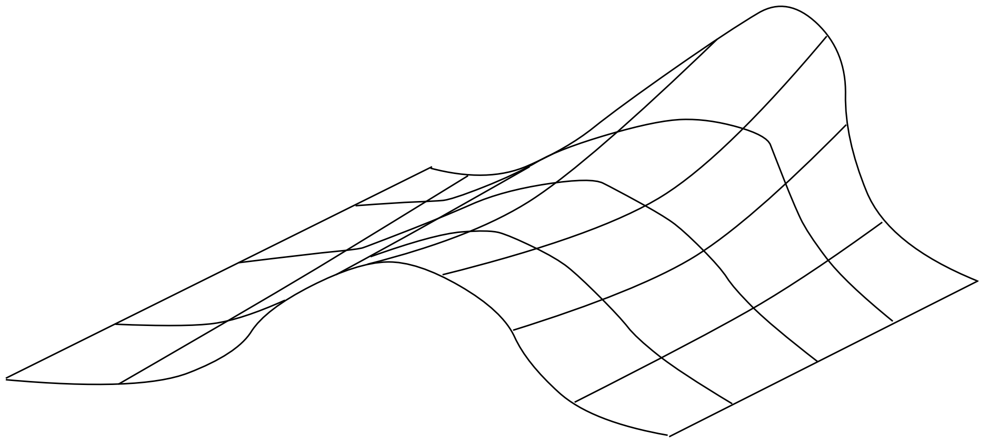
4.4.3 Steepest Ascent

This is a simple variation on the previous strategy: at every node in the search space we analyse all the possible subsequent nodes and select the best one, continuing recursively until we reach a goal.

1. Evaluate the initial state. If it is the goal, we are finished.
2. Loop until there are no more successor states:
   (a) let SUCC be the current state
   (b) loop through all successor states:
      (i) evaluate the new state.
      (ii) If this it is a goal we are finished.
      (iii) else if it is better that SUCC, set SUCC to be this state
   (c) if SUCC is better than the current state set the current state to be SUCC



The technique is also called called *gradient search* because it is like climbing a hill by always taking the steepest path. The theory is that by doing this you cover less ground to reach the summit. Unfortunately both forms of hill climbing run into problems if there are *local maxima*, where we only climb a foothill and not the mountain. Similarly, there can be *plateaux*: sets of points of equal value, or *ridges*: these have steep sides so we get to the top of the ridge easily, and they have gentle slopes along the ridge up towards a maximum. However, it the allowable moves don't take us precisely along the crest of the ridge, we are stuck.



If this happens, we can try several things.

1. backtrack to a previous state, and try something else. This helps with local maxima.
2. make a large jump in some direction to try to get off a plateau.
3. move two or more times before doing evaluations. This helps with ridges.

Hill climbing is critically dependent on a good evaluation function: the value must generally point in the right direction, and not fall away too quickly as we move away from the goal. It is a *local* strategy, as it only looks at local states, but can be helped by evaluation functions that look deeper into the space.

4.4.4 Simulated Annealing

Another variant on hill climbing that derives from the physical annealing of metals. This hopes to explore enough of the space early on to avoid the problems of local maxima and the like.

It is conventional to call the heuristic function the "objective function," and to try to minimise it (instead of maximising). In physical annealing, a metal is repeatedly heated and cooled to a form particular crystalline structure. The energy of the atoms is the objective function, and it naturally decreases with time. However, there is a chance that a transition to a higher energy state will occur, and this happens with probability

$$p = e^{-\Delta E/kT},$$

where ΔE is the positive energy change, T the temperature, and k is Boltzmann's constant. So generally the energy will decrease, but with small probability it will increase; larger jumps have lesser probability, and jumps are less likely at lower temperatures.

To apply this to searching, we set ΔE to be the change in the objective function, and rewrite kT as a single parameter T (as scales are arbitrary).

1. Evaluate the initial state. If it is the goal, we are finished.
2. Set BEST-SO-FAR to be the current state
3. Initialise T (discussed below)
4. Loop until there are no more successor states:
   (a) select and evaluate a successor state
   (b) if it is a goal we are finished
   (c) else set $\Delta E =$ value of current $-$ value of new
   (d) if new state is better than current, make it the current, and also make it BEST-SO-FAR
   (e) otherwise make it the current state with probability $p = e^{-\Delta E/T}$ (hop to a worse state with a certain low probability).
5. decrease T

So this descends valleys, but occasionally hops back up again. Larger hops are less likely, and hops generally become less likely as T decreases.

It remains to describe the way we set and decrease T: this is called the *annealing schedule*. Unfortunately, no general schedule can be given as it appears the best you can do is to try a few examples on your specific problem (i.e., search the problem space of annealing schedules!) You should note, though, that as T gets close to 0, hops become very unlikely, and simulated annealing degenerates to the hill climbing method.

4.4.5 Best-First Search

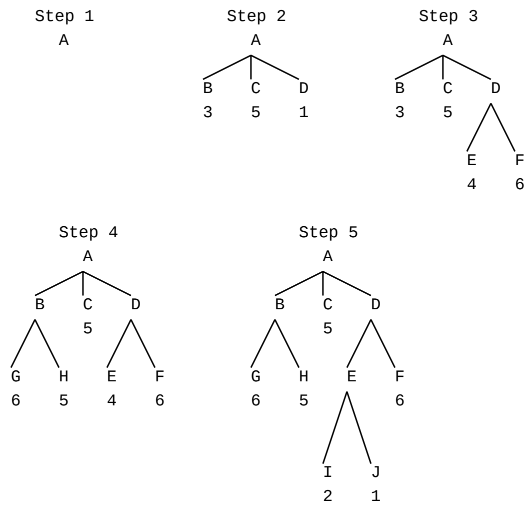
This is a method for combining both the advantages of depth-first search and breadth-first search. The general idea is that at each stage you explore the path that looks most promising from *all* the paths expanded so far. The result of this is that if an initially promising path begins to look dodgy, you can go back to an earlier path which now looks better in retrospect. Old branches are never forgotten, so this technique can be expensive in terms of memory.

This procedure differs from gradient search in a number of ways: in gradient search we select the best move and reject all the other candidates permanently, in best-first search we select the best move, but that may not be a successor state of the one just explored. This provides a way of dealing with local maxima, plateaux etc.

This method keeps a list OPEN of nodes that look promising, but have yet to be explored. Often this is kept as an ordered queue, with the best nodes first. Also, this methods works if there are loops in the state space, i.e., more than one path to a particular state.

1. Start with OPEN containing the initial state

2. Loop until there are no states left in OPEN
   (a) pick the best node from OPEN
   (b) for each of its successors do
      (i) if it is new node, evaluate it, add it to OPEN, and record its parent
      (ii) if it has been seen before, change the parent to the current if the current is better than the old one, and in that case update the cost of this and its parent nodes to reflect the new cost (i.e., ensure the cost to this node is counted along this new, cheaper, path).



4.4.6 The A* Algorithm

This method looks more closely at the node evaluation function. Call this function f. We decompose this into two parts: $f = g + h$. Here, g is the cost of getting to the current node (this is known exactly as the sum of the individual steps of getting from the initial to the current state), and h is an estimate of the additional cost of getting from here to the goal: in this we encode our knowledge of the problem.

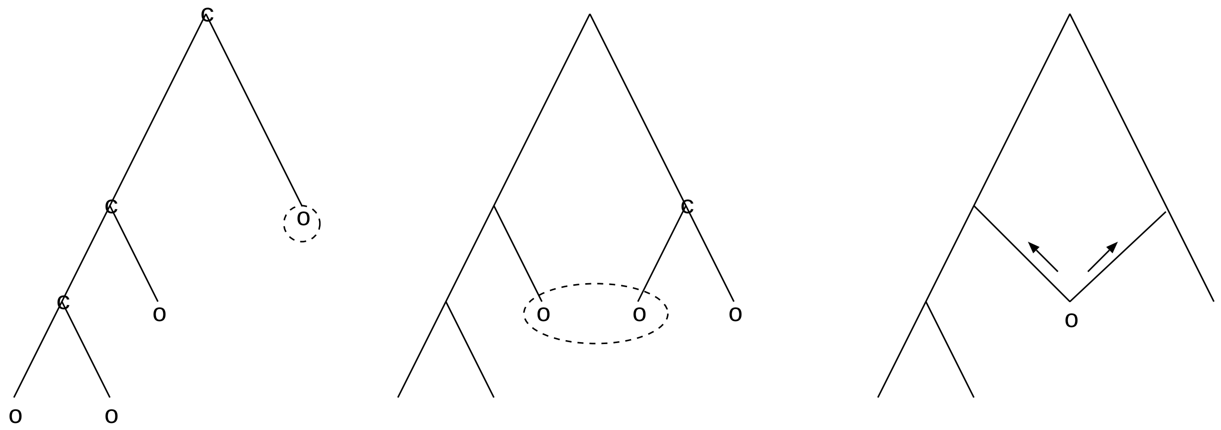
The A* algorithm (so called) is a development of the best-first method above (actually, the best-first is a simplification of the A*). It uses a set OPEN of nodes yet-to-be-explored, and CLOSED, a set of nodes already examined.

1. Start with OPEN containing the initial state. Set that node's g value to 0, and its f value to its h value. CLOSED is empty.
2. Loop until there are no states left in OPEN
   (a) remove a node from OPEN with the lowest f and call it BEST-SO-FAR. Put it in CLOSED. If it is a goal, we are finished. Otherwise generate the successors, and for each SUCCESSOR do the following
      (i) Set the parent of SUCCESSOR to be BEST-SO-FAR.
      (i) Compute $g(\text{SUCCESSOR}) = g(\text{BEST-SO-FAR}) +$ cost of getting from BEST-SO-FAR to SUCCESSOR.

(ii) If SUCCESSOR is in OPEN (we have seen it before, but not yet explored it further), set SUCCESSOR to be the old version from OPEN, and similarly replace it in the list of BEST-SO-FAR's successors. If, now, the new path to SUCCESSOR is better than the existing path, set the parent of SUCCESSOR to be BEST-SO-FAR, and update the values of f and g to reflect this.

(iii) If it was not in OPEN, see if it is in CLOSED. If so, set SUCCESSOR to be the old version from CLOSED, and similarly replace it in the list of BEST-SO-FAR's successors. As before, check to see if the new path is better than the old one, and if so, set the parent of SUCCESSOR to be BEST-SO-FAR, and update the values of f and g to reflect this; also propagate this change to all of SUCCESSOR's children (which we have already computed). This is done by a simple depth-first traversal, terminating a branch when there are either no successors, or a node is still in OPEN. (In fact, need only update the a child if the new path to it is better than the old.)

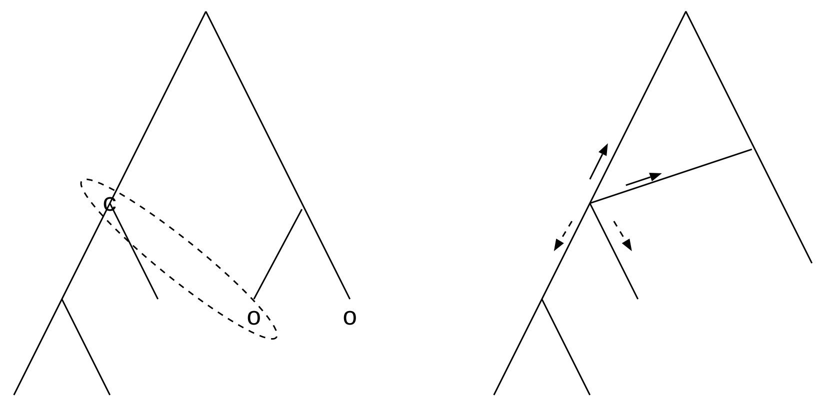
(iv) If SUCCESSOR was neither in OPEN or CLOSED, put it on OPEN, and add it to BEST-SO-FAR's successor list. Compute $f = g + h$.



o open, c closed
expand node

new node already seen as
open, merge

choose best path, update
cost of o if necessary



new node already seen as
closed, merge

choose best path, update
cost of node and child nodes

By splitting the heuristic function f into g and h this method allows us to choose nodes to follow based not only on how good it looks itself (h), but also on how good the path to the node was (g).

If we set $g = 0$ everywhere, we get a method that merely finds a solution, not a "minimal" solution. Setting $g = 1$ will find the shortest path to a solution. (Compare the problem of finding the shortest road route between two points vs. the problem of finding the fastest route.)

On the other hand, if h is a perfect measure of the distance of to a goal, then this method will converge immediately with no searching. The better the h, the closer to this approach we get. If $h = 0$ everywhere, the search is controlled by g: if $g = 1$ this becomes a breadth-first search, if $g = 0$, the search is random.

In general we can say this: if h is never an overestimate of the true value, then the A^* algorithm is guaranteed to converge to a g-minimal solution it it exists. An algorithm that is so guaranteed is called *admissible*. This situation is hard to arrange, but all is not lost: if h rarely overestimates by more than δ, the the A^* algorithm will rarely find a solution whose solution is more than δ greater than the cost of the optimal solution.

This method restricts simply to trees: simply never check to see if a node is in the lists OPEN or CLOSED.

Aside: the choice of heuristic evaluation function can be a problem in itself. There is a story of a chess-playing program written at MIT: there was a bug in the evaluation function that caused it to return (say) 1 for bad and 0 for good, instead of the other way about. Thus the system was programmed to try to lose. However, it was not obvious that this was happening: the program did not make "bad" moves, rather it made complex and incisive moves that seemed quite reasonable. But these moves were designed to *force* the opponent to win: forcing yourself to lose whatever the opponent is doing is quite difficult too!

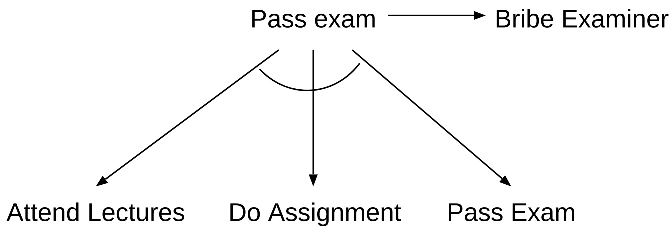
4.4.7 Others

There are several other search methods:

- Agendas. This regards the problem as a list of tasks that might be performed (each task may spawn several subtasks, etc.). Associated with each task there is a list of reasons why it is proposed that this node should be executed (*justifications*), and also an overall rating representing the weight of evidence for this task. Searching involves executing likely tasks, and recomputing ratings and justifications.

  The agenda system allows the importance of tasks to vary with time: something may not be important now, but might become more so later.

- The AO^* method and Problem reduction. So far we have assumed branches in problem space are independent: technically such a state space is called an *OR-graph* since we take one path or another. However in practice we often want to satisfy several subgoals together. For example, consider the task of passing an exam. There are two ways to do this: the first is by attending the lectures, doing the assignment, revising hard and sitting the exam. The second is to bribe the examiner. We represent this as follows:

  The arcs connect paths which must all be satisfied together. The AO^* method is an adaption of the A^* method to cope with this, and *problem reduction* is a simplification (c.f., A^* and best-first).

- Constraint Satisfaction Problems (CSP). Some problems are naturally expressed as a collection of constraints:

```
 SEND
+MORE
-----
MONEY
```

  This is a cryptarithmic problem, where each letter stands for a digit (no repeats, etc.) The constraints are like "$D + E = Y$ or $D + E = Y + 10$", or perhaps "$D + E = Y \bmod 10$". If we deduce that $E = 0$, we can propagate this to find "$D = Y$ or $D = Y + 10$." We can make guesses for values, follow the consequences through, and backtrack if we get a contradiction.

- Means-Ends analysis. This allows a mixing of forward and backward searching in the same solution. Means-ends analysis firstly identifies a difference between the current state and the goal. It then searches for a way of reducing the difference, but perhaps the rule it finds cannot be applied directly to the current state. So a subproblem is generated to discover how to get from the current state to one in which the rule can be applied (backward reasoning from the subgoal to the current). Once found we can then proceed forward reasoning again with the subgoal as the new current state (we hope that the new current is better than the old).

  It appears that people use this technique to solve problems. For example, to get to London, we might decide that going to the Railway station is a good idea. So we can then take the bus to the station, and the train to London.

- Neural Nets. A trendy new method, modelled on the brain. Mostly used as a recogniser, i.e., a method of classifying objects.

  A NN is a collection of simple processors (*units*), possibly having a little memory each, connected by unidirectional channels (*connections*) that carry numerical data. A unit can process local data and the data it receives on its connections. An output is some simple weighted function of this data.

  A neural network is expected to recognise certain patterns of inputs, e.g., pictures of faces, and typically answers "yes" that is a face or "no" it is not.

  Most neural networks are subjected to *training* where the weights of connections are adjusted on the basis of given examples. In other words, neural networks are supposed to "learn" from examples, just like people learn to recognize faces from many examples of faces.

  NNs are Turing complete, i.e., they can compute any computable function, but are best at problems where there are lots of example data; there needs to be tolerance for error; simple rules do not work. NNs are bad at symbolic computation.

  The literature for NNs is huge: there have been many examples tried (not all successful), and many "improvements" given, particularly on how to change weights in the training phase. For example, there is *supervised learning* vs. *unsupervised learning*. In supervised learning a teacher is present to tell the NN what the appropriate response to a set of inputs should be. In unsupervised learning the NN is supposed to deduce for itself what it is looking for from the set of examples. This is useful if the implementor of the NN is not sure of what they are looking for in the first place.

- Genetic Algorithms.

  Another trendy method, inspired by evolutionary processes. Points in the problem space are regarded as individuals in a population. An individual is represented as a finite string of symbols, often binary, called the *genome*. An individual has an associated *fitness*, i.e., evaluation function.

  At each stage we have a (often fixed size) collection of individuals, called a *generation*. From these we make a new generation. Individuals with a high fitness are more likely to make it to the next generation, i.e., we apply a form of evolutionary selection. An individual's relative fitness decides how many copies of it might appear in the next generation: if A has twice the fitness of B, then it is twice as likely to be selected to succeed.

So far, so good. But this does not introduce new individuals into the population. To do this, with a small probability, an individual *mutates*. A symbol in its representation is randomly replaced. Further, again with a small probability, two individuals *cross over*. A chunk of symbols from A are swapped with the corresponding chunk from B.

Thus we get new individuals that we hope have a larger fitness.

Thus:

1. initialise the population P (often at random)
2. while not done do
3. evaluate the fitnesses of the members of P
4. select the new population from the old with regard to relative fitness
5. do crossovers
6. do mutations
7. done

A small example. We encode individuals as genomes of eight bits:

	genome	fitness
A	00000110	2
B	11101110	6
C	00100000	1
D	00110100	3

our fitness is the number of 1 bits in the genome.

Next, select the next generation with regard to relative fitness: B, D, B, C, say.

	genome	fitness
B	11101110	6
D	00110100	3
B	11101110	6
C	00100000	1

We crossover B and D, say, by swapping the tails after the randomly chosen first bit position, producing new individuals E and F.

	genome	fitness
E	10110100	4
F	01101110	5
B	11101110	6
C	00100000	1

Next mutate, say, E (6th bit) and B (first bit)

	genome	fitness
E'	10110000	3
F	01101110	5
B'	01101110	5
C	00100000	1

And repeat. Notice that the maximum fitness has decreased, while the average fitness has increased. Eventually we would generate an individual with the goal fitness, 8.

Typical genomes are 50-1000 bits, and we have to go through many hundreds of thousands of generations. Clearly this can become very slow and expensive. GAs are used when the search space is large, complex or poorly understood. For example, the travelling salesman problem; protein structure prediction; robot control; economic prediction; generating neural networks.

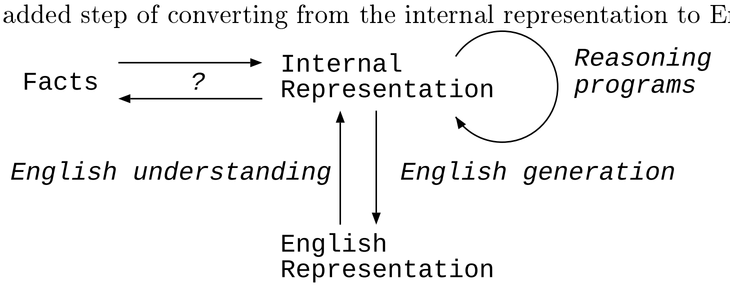
5 Knowledge Representation

We now move on to a different topic: that of knowledge representation. This is concerned with how to encode knowledge about problem spaces in a problem-solving program, and ways to combine problem-solving techniques with knowledge to aid the solution of problems. In particular we shall address the vague notions above of heuristic functions that evaluate "goodness," and representations of the problem space that allow us to search or generate successors to nodes.

"Knowledge" in the AI sense is different from the real-world usage of the word, i.e., facts, but is a representation in a program of these facts. There should be a clear mapping between the two:

- Facts = truths in some relevant world
- Representations of facts in some formalism

The nature of such representations is important (the *representation mappings*), but there is a very common one, namely English (or other) language. Being able to express results in English is a great help in understanding the output of a program. Generally, though, internal program representations are not done in English, so there is an added step of converting from the internal representation to English.



As a simple example consider (the fact represented by) the English sentence

All dogs have tails

this might be represented as

$$\forall x : \text{dog}(x) \Rightarrow \text{hastail}(x)$$

in predicate logic. Then from "Spot is a dog," or dog(Spot), we can use deductive logic to conclude hastail(Spot), or translating back, "Spot has a tail."

There are a number of properties a good system for knowledge representation needs:

1. *Representational Adequacy* The ability to represent all forms of knowledge about a problem.
2. *Inferential Adequacy* The ability to manipulate existing knowledge so as to derive "new" knowledge.
3. *Inferential Efficiency* The ability to add extra information to the knowledge structure to focus the inference mechanism on promising areas.
4. *Acquisitional Efficiency* The ability to acquire new information easily.

In practice we have to compromise between these four criteria, hence the many different knowledge representation techniques used.

5.1 Relational Knowledge

This is essentially what a database uses. Groups of facts are collected into structures which are indexed in some way. For example the University central computer might have a database containing information about their students as follows:

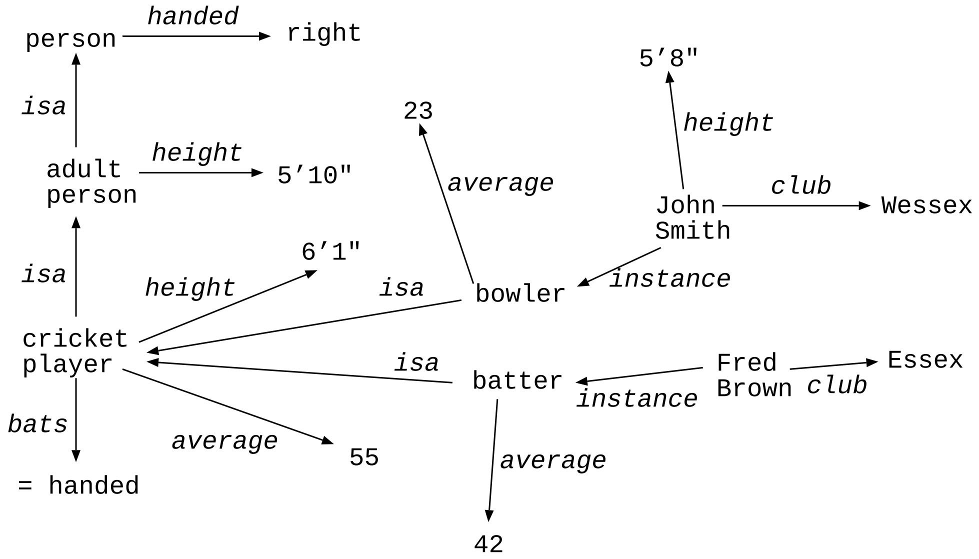
Name	**Sex**	**D.O.B.**	**Course**
Allen, Woody	male	5.9.43	Child Psychology
Bassinger, Kim	female	3.2.65	CST
Clarke, Ken	male	2.6.51	Politics & Economics

This allows us to answer simple questions. For example if we want to know what degree Kim Bassinger is studying for, we look her up in the index and inspect the corresponding field in her record. We could also answer questions about the average age of CST students, or the number of students on a particular course, but that is stretching this technology to the limit. A more sophisticated structure is needed for knowledge in an AI system.

5.2 Inheritable Knowledge

This kind of representation has a lot in common with inheritable behaviour as implemented in object oriented systems. Namely objects can inherit knowledge from "super-objects," or they can shadow or redefine knowledge for their own special case.

To support such an idea, objects must be arranged into classes, and those classes into a hierarchy.



Nodes represent objects, and arrows represent attributes of those objects. This structure is an example of something variously called a *semantic network*, a *slot-and-filler structure*, or a collection of *frames*. An individual frame represents the collection of attributes and values of a particular node:

Cricket player
isa: adult person
bats: (= handed)

height: 6-1
average: 55

Points to note:

1. there are two kinds of attributes: specific ones like *height* and *average*, and two general ones, namely *isa* and *instance*. The latter two are of special significance, and appear in most semantic nets. The first, *isa*, denotes class inclusion (subclass relationship), and the second, *instance*, denotes class membership (c.f., $\subset$ and $\in$).
2. some values are code, i.e., something to be executed: in this case the node (= handed) specifies an action to be taken to determine a value, namely the value of the bats attribute is the same as the value of the handed attribute.
3. some attributes override the more general values: the height of a general cricket player is 6-1, whereas the height of a general person is 5-10, and the height of John Smith is 5-8.
4. the average for a batter is given as 42, and that for a bowler is 23, whereas the average for a general cricketer is 55. This kind of inconsistency can be hard to detect.

To look up a value of an attribute follow this algorithm. To retrieve a value for attribute A of an instance O

1. Find O in the net
2. If there is a value for the attribute A, return it
3. Otherwise, see if there is an *instance* link, if not fail
4. Move up the *instance* link, and look for a value for attribute A, return it if found
5. Otherwise, repeat this loop while there is an *isa* link
   (a) move up the *isa* link
   (b) look for a value for attribute A, return it if found

This fails to deal with the problems that arise if there is more than one *instance* or *isa* link at a node (this is the multiple inheritance problem). Perhaps Jim Evans is an all-rounder: both a bowler and a batter.

Thus:

- John Smith's club is Wessex. This is directly stored.
- Fred Brown's average is 42. We follow an *instance* link.
- Fred Brown's height is 6-1. We follow an *instance* link, then an *isa* link.
- John Smith bats right handed. Here we follow an *instance* and an *isa* link. to get the value (= handed), which is code. Running this code we discover we need to determine the handedness of John Smith. To get this we follow up the *instance* link from John Smith, then 3 *isa* links before finding a node with a *handed* link.

The purpose of semantic nets is to encode general information shared by many objects in a single place. Some computation is then required to find the relevant value. Special cases can be included that override the general case.

5.3 Procedural Knowledge

The above methods are relatively static, and are *declarative.* Another way of doing things is to use procedural knowledge, that is a collection of actions to perform to produce the information required. Generally these are represented in the form of programs.

Consider the frame for a cricket player

Cricket player
isa: adult person
bats:

```
(labels
  ((loop (x)
     (cond ((has-attribute? 'handed x)
            (get-attribute 'handed x))
           ((has-attribute? 'isa x)
            (loop (get-attribute 'isa x)))
           (t 'fail))))
  (loop self))
```

height: 6-1
average: 55

This encodes directly the search up the *isa* links for the handed attribute.

Procedural knowledge can also be encoded as a collection of production rules

1. if has-attribute(x, handed) then get-attribute(x, handed)
2. if has-attribute(x, isa) then get-attribute(get-attribute(x, isa), handed)
3. ...

Poor in terms of inferential adequacy (it is hard to argue about a program's behaviour), and also in terms of acquisitional efficiency (the process of writing and maintaining large pieces of code is difficult).

5.4 Inferential Knowledge

Another declarative method.

Two forms of deduction system which are well-known to mathematicians are the *predicate calculus* and the *propositional calculus.* These allow us to take statements and rules of the form:

$$\begin{array}{l}\text{lectureTheatre(1W3.7)}\\ \forall U, \text{lectureTheatre}(U) \Rightarrow \text{room}(U)\end{array}$$

and manipulate them to answer questions like "is 1W3.7 a room?".

Recall that a *proposition* is a statement which can be either true or false in a given world, e.g. "it is raining". Propositions are represented by capital letters (by convention $P, Q, \ldots$), and can be combined using the *connectives* $\wedge, \vee, \sim, \Rightarrow, =$. A *sentence* in the propositional calculus is any well-formed combination of propositions and connectives. An *interpretation* of a sentence is a sentence where each proposition has been replaced by true or false, and the whole sentence evaluated by the usual rules to yield true or false.

In the propositional calculus there is no way of breaking a proposition down: it is a straightforward assertion which is either true or false. So we cannot see the relationship between e.g. "John is tall" and "Mike is tall". Thus this particular method of knowledge representation may well fail the test of inferential adequacy described earlier.

The predicate calculus provides a method of viewing statements as relationships between things. For example in the propositional calculus we say things like "John's hair is brown" whereas in the predicate calculus we might say `hair(john,brown)`. In this case the predicate function is `hair`, and it indicates some kind of relationship between the *constants* `john` and `brown`. Note that all predicates evaluate to a constant value, but not necessarily to true or false. In the predicate calculus, we can also work with variables and quantifiers: for example the sentence $\forall x : \text{hair}(x, \text{brown})$ says that for all values of the variable x, the sentence $\text{hair}(x, \text{brown})$ evaluates to true. The two quantifiers are $\forall$ (called *the universal quantifier*) and $\exists$ (called *the existential quantifier*).

To use this representation of knowledge we need some means of manipulating it, in particular we need some inference procedure that can take propositions ($\forall x : \text{dog}(x) \Rightarrow \text{hastail}(x)$, $\text{dog}(\text{spot})$) and deduce new information from them ($\text{hastail}(\text{spot})$). A common method is *resolution*, which we shall come to later.

Note that logic statements would fit in well in a semantic net as the value of a node: these things are not mutually exclusive.

5.5 Problems in Knowledge Representation

Having seen a few methods for representing knowledge, we now address a few general issues.

- Basic attributes: can we determine any attributes that *all* (or almost all) problems use? If so they must be treated appropriately.
- Are there any relationships between attributes?
- How large should chunks of knowledge be: are there "atomic" or primitive pieces of knowledge?
- How to represents collections of objects.

5.5.1 Important Attributes

We have already seen two important attributes, namely *isa*, and *instance*. These represent the basic notions of subset and membership (or subclass and instance), and so have widespread use. Some mechanisms involve these directly (semantic nets), while others have to encode them: for example in logical form $\forall x : \text{dog}(x) \Rightarrow \text{animal}(x)$, and $\text{dog}(\text{spot})$; alternatively $\text{instance}(\text{spot}, \text{dog})$ with general rules like

$$\forall x, y, z : \text{instance}(x, y) \wedge \text{isa}(y, z) \Rightarrow \text{instance}(x, z).$$

5.5.2 Relationships between Attributes

It is worthwhile thinking about attributes as they are themselves entities that we need to represent.

1. Inverses. Some relationships have "inverses": for example, if John Smith's club is Wessex, then Wessex has John Smith as a member. So in this case we have inverse relationships of "club and "member". Others, for example *sibling*, are self-inverse. It would be useful to make this connection between attributes explicit. Logical assertions

$$\text{club}(\text{John Smith}, \text{Wessex})$$

can be interpreted in either direction, thus unifying the two attributes in a single statement. On the other hand, semantic nets and frame systems have to keep pairs of links: a *club* link from John Smith to Wessex, and a *member* link the other way. Normally these are added in to the net as a single operation, so there is no scope for losing links.

2. Hierarchy of Attributes. Attributes can also be placed in a hierarchy, e.g., *height* is a specialisation of *physical attribute*. Hierarchies of attributes are useful in exactly the same way as hierarchies of values, as they encode information more efficiently (at the cost of a more sophisticated search).

3. Attributes of Attributes. We can classify attributes in various ways that can be useful: it is single valued? Does it have a particular type? It it code that must be executed? And so on.

5.5.3 Granularity

The detail at which we represent facts is important: the more detail we store the better we can answer questions, but the more memory we use and the harder it will be to find the answer we seek.

For example, suppose we have (represented) the fact $\text{spotted}(\text{John}, \text{exam paper})$. If we have just this, we cannot determine whether $\text{saw}(\text{John}, \text{exam paper})$, as we have no connection between "spotted and "saw". We might add a fact that $\forall x, y : \text{spotted}(x, y) \Rightarrow \text{saw}(x, y)$, and infer the answer. On the other hand, we could choose a more primitive representation for seeing, one that involves a time interval, so we might say

$$\text{saw}(\text{John}, \text{exam paper}, \text{briefly}).$$

There are several problems in representing everything at such a low level.

1. High-level facts become very memory intensive when broken down into small parts; and also because of the repetition of similar facts in slightly different environments.

2. The translation to and from English language becomes problematical.

3. The choice of primitives is very difficult. Making the set of primitives complete and more-or-less independent can be impossible. Similarly there is the problem of unique representation, as a set of primitives ideally will represent a single entity in a single way: consider the representations of the planet Venus by "evening star", and "morning star". The *objects* these phrases represent are the same, but the meaning associated are different.

5.5.4 Collections

Often we need to represent sets of objects ("the members of the current government"), and the way we represent them can make a difference. We can write sets *extensionally*, that is list all the elements

$$\{\text{John Major}, \text{Douglas Hurd}, \ldots\},$$

or we can express a set *intensionally*, as a collection of properties that members of that set have

$$\{x : \text{MP}(x) \wedge \text{sleazy}(x)\}.$$

The difference is in the ways we can manipulate the set, as the extensional description allows us to deduce more facts about the members of the set. Furthermore, testing two extensional sets for equality is trivial, whereas two intensional descriptions can be very difficult. On the other hand, intensional sets can easily be infinite, and can depend on parameters that change with time (e.g., $\text{MP}(x)$ appears highly variable).

5.6 The Frame Problem

There are some facts that never change. For example, in the state space of a robot that moves about a room, in every state it is true that the ceiling is above the floor, or $\text{above}(\text{ceiling}, \text{floor})$. If we were to record this information in every node, we would be using a huge amount of space, and generally to little effect. On the other hand, some facts must be associated with a particular node, e.g., $\text{above}(\text{table}, \text{robot})$. The problem of representing those facts that change and those that don't (and determining which are which) is called the *frame problem*. This can be non-trivial: consider a window above a plant on a table. If we move the table the plant moves too, but the window doesn't.

Some systems use an explicit set of axioms to support this kind of reasoning, called *frame axioms*. These describe all those facts that do not change when we move from one node in state space to another, e.g.,

$$\text{colour}(x, y, s_1) \wedge \text{move}(x, s_1, s_2) \Rightarrow \text{colour}(x, y, s_2),$$

meaning if x has colour y in state s_1, and "move" changes our state from s_1 to s_2, then x has colour y in state s_2.

Again, this requires a large number of axioms and lot of memory, so some choose to make all such rules implicit: so unless explicitly mentioned (or is derivable from something mentioned explicitly), no property of any object changes. If we never say anywhere that colour changes with position, we can assume that it doesn't. This is a powerful technique, and partially solves the frame problem. We may come back to this later.

6 Logic

We now consider one of the above representations in more detail: the inferential method, using propositional and predicate logic.

We have already seen that propositional logic is too clumsy to provide an adequate representation, so we will be concentrating on predicate logic.

Here is a collection of facts

1. Marcus was a man
2. Marcus was a Pompeian
3. All Pompeians were Romans
4. Caesar was a ruler
5. All Romans were either loyal to Caesar or hated him
6. Everyone is loyal to someone
7. People only try to assassinate rulers they are not loyal to
8. Marcus tried to assassinate Caesar

These can be represented in predicate logic as

1. $\mathrm{man}(\mathrm{Marcus})$
2. $\mathrm{Pompeian}(\mathrm{Marcus})$
3. $\forall x : \mathrm{Pompeian}(x) \Rightarrow \mathrm{Roman}(x)$
4. $\mathrm{ruler}(\mathrm{Caesar})$
5. $\forall x : \mathrm{Roman}(x) \Rightarrow \mathrm{loyalto}(x, \mathrm{Caesar}) \vee \mathrm{hate}(x, \mathrm{Caesar})$
6. $\forall x : \exists y : \mathrm{loyalto}(x, y)$
7. $\forall x : \forall y : \mathrm{person}(x) \wedge \mathrm{ruler}(y) \wedge \mathrm{tryassassinate}(x, y) \Rightarrow \neg \mathrm{loyalto}(x, y)$
8. $\mathrm{tryassassinate}(\mathrm{Marcus}, \mathrm{Caesar})$

Though there are other possible interpretations of the English. For example, in fact 5, the (probably) exclusive "or" was translated into an logical inclusive "or". Possibly a better translation would be

$$\begin{array}{l}\forall x : \mathrm{Roman}(x) \Rightarrow \\ (\mathrm{loyalto}(x, \mathrm{Caesar}) \vee \mathrm{hate}(x, \mathrm{Caesar})) \wedge \neg\, (\mathrm{loyalto}(x, \mathrm{Caesar}) \wedge \mathrm{hate}(x, \mathrm{Caesar})).\end{array}$$

Similarly, statement 6 is ambiguous, it might mean

$$\exists y : \forall x : \mathrm{loyalto}(x, y),$$

i.e., there is a single person that everyone is loyal to.

Now we can try to answer a question: was Marcus loyal to Caesar? By looking at the statements we have, it looks likely that the answer is no, and we should be able to prove

$$\neg \mathrm{loyalto}(\mathrm{Marcus}, \mathrm{Caesar})$$

We can do this by searching a proof tree, that is the tree of all possible consequence of the above statements. This is generally an AND-OR graph. So we need to reason backward from the above statement we wish to prove to try to get back to the given facts. One attempt goes like this

- $\neg$loyalto(Marcus, Caesar) (the goal)

  (7), substitution

- person(Marcus) $\wedge$ ruler(Caesar) $\wedge$ tryassassinate(Marcus, Caesar)

  (4)

- person(Marcus), tryassassinate(Marcus, Caesar)

  (8)

- person(Marcus)

We cannot get any further as we cannot satisfy the last subgoal. Clearly if we had

9. $\forall x : \text{man}(x) \Rightarrow \text{person}(x)$

we could complete the proof.

This raises several points about using logic as a formalism

- English is ambiguous, finding correct interpretations can be hard
- There is a question of how much information to encode; we have totally overlooked the problem of time and tenses in the above (man(Marcus) might mean "Marcus is a man", of "Marcus was a man", and the other statements are similar). Similarly, "tryassassinate" is a simple representation of a very complicated action.
- it is hard to contain *all* the information we might need: forgetting that being a man implies being a person is easy to do. Many trivial but necessary facts tend to be overlooked.
- We "knew" in advance that we wanted to prove $\neg$loyalto(Marcus, Caesar). This cannot be done in general, e.g., "is Marcus loyal to Caesar?" So we need some strategy for deciding what to do to answer a question. We might use

  1. forward reasoning, directly try to prove or disprove the statement. But the branching factor is so huge this is impossible
  2. Use a heuristic to decide to prove or disprove a statement, and follow it up, possibly giving up and trying the other way if nothing useful happens.
  3. Do both: try to prove and disprove.

  Even the last may not work, due to incompleteness. An attempt to prove a statement can fail for two reasons. Firstly, it is not true. Secondly, is is not provable, i.e., the facts are incomplete (as we had above).

6.1 Inheritance

We saw earlier how useful inheritance of data is in a semantic net. We can get the same sort of effect in logic, although it is more work.

1. man(Marcus)

2. Pompeian(Marcus)
3. $\forall x : \text{Pompeian}(x) \Rightarrow \text{Roman}(x)$
4. ruler(Caesar)
5. $\forall x : \text{Roman}(x) \Rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$

This is the encoding we had before. It states explicitly that Pompeians are a subset of Romans. We are using predicates "Roman" and so on to indicate set membership. So we can rewrite the above to

1. instance(Marcus, man)
2. instance(Marcus, Pompeian)
3. $\forall x : \text{instance}(x, \text{Pompeian}) \Rightarrow \text{instance}(x, \text{Roman})$
4. instance(Caesar, ruler)
5. $\forall x : \text{instance}(x, \text{Roman}) \Rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$

But we can go further, and use *isa*

1. instance(Marcus, man)
2. instance(Marcus, Pompeian)
3. isa(Pompeian, Roman)
4. instance(Caesar, ruler)
5. $\forall x : \text{instance}(x, \text{Roman}) \Rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$
6. $\forall x : \forall y : \forall z : \text{instance}(x, y) \wedge \text{isa}(y, z) \Rightarrow \text{instance}(x, z)$

Now we need an extra rule for the relationship between *isa* and *instance*.

This kind of representation could clearly be continued, but it gets very cumbersome. Either kind of representation may be used (implicit or explicit membership), but it is not a good idea to mix the two.

There is another facet to inheritance used in semantic nets: the overriding of values by more specific ones. Again, it is possible to encode this in logic, but it is very hard. For example, suppose we wish to add the following to our original set of statements

10. Pompeian(Paulus)
11. $\neg\,(\text{loyalto}(\text{Paulus}, \text{Caesar}) \vee \text{hate}(\text{Paulus}, \text{Caesar}))$

so Paulus is an exception to the general rule of Romans attitude to Caesar. These two statements cannot simply be added to the general database, as they would produce contradictions (meaning we could prove any fact). We need to update old rule 5

5. $\forall x : \text{Roman}(x) \wedge \neg(x = \text{Paulus}) \Rightarrow \text{loyalto}(x, \text{Caesar}) \vee \text{hate}(x, \text{Caesar})$

(Here $(x = \text{Paulus})$ is a shorthand for the predicate $eq(x, \text{Paulus})$) So every exception to a general rule appears in two places: once as the statement, and again in the general rule. This is difficult to maintain, and can be very inefficient.

6.2 Computable Knowledge in Logic

In a semantic net we could have nodes that computed values. A related sort of thing can happen in logic representations. Some facts are too difficult to represent in their entirety, either because they are too complex, or because they are too large. If we wished to encode all the greater-than relationships between the integers we have a problem:

$$\ldots, \mathrm{gt}(0,-1), \mathrm{gt}(1,0), \mathrm{gt}(2,1), \mathrm{gt}(2,-1), \ldots$$

What we need is a predicate *gt* whose value we can compute, maybe from a general specification. Then we could determine the truth of statements like "$2+3$ is greater than $3+4$", or $\mathrm{gt}(2+3, 3+4)$.

Here is an example of the use of computable predicates

1. Marcus was a man
2. Marcus was a Pompeian
3. Marcus was born in 40 AD
4. All men are mortal
5. All Pompeians died in the volcanic eruption of 79 AD
6. No mortal lives for more than 150 years
7. The date is 2001 AD
8. Alive is the same as not dead
9. If someone dies, then they are dead at all later times

which becomes

1. $\mathrm{man}(\mathrm{Marcus})$
2. $\mathrm{Pompeian}(\mathrm{Marcus})$
3. $\mathrm{born}(\mathrm{Marcus}, 40)$
4. $\forall x : \mathrm{man}(x) \Rightarrow \mathrm{mortal}(x)$
5. $\mathrm{erupted}(\mathrm{volcano}, 79)$

5′. $\forall x : \mathrm{Pompeian}(x) \Rightarrow \mathrm{died}(x, 79)$

6. $\forall x : \forall t_1 : \forall t_2 : \mathrm{mortal}(x) \wedge \mathrm{born}(x, t_1) \wedge \mathrm{gt}(t_2 - t_1, 150) \Rightarrow \mathrm{dead}(x, t_2)$
7. $\mathrm{now} = 2001$
8. $\forall x : \forall t : (\mathrm{dead}(x,t) \Rightarrow \neg\mathrm{alive}(x,t)) \wedge (\neg\mathrm{dead}(x,t) \Rightarrow \mathrm{alive}(x,t))$
9. $\forall x : \forall t_1 : \forall t_2 : \mathrm{died}(x, t_1) \wedge \mathrm{gt}(t_2, t_1) \Rightarrow \mathrm{dead}(x, t_2)$

Points to note:

5, 5′: A single English sentence generates two logical statements. But even more is implicit, like the assumption that the volcano caused the deaths.

7: Equal quantities can be substituted for each other. Note, though, there *is* a difference between "now" and "2001", for otherwise we would not have two different ways of expressing them.

8: But "not alive" can also mean "not living", i.e., inanimate.

Now prove that $\neg$alive(Marcus, now).

- $\neg$alive(Marcus, now) (the goal)

  (8), substitution
- dead(Marcus, now)

  (6), substitution
- mortal(Marcus) $\wedge$ born(Marcus, t_1) $\wedge$ gt(now $- t_1$, 150)

  (4), substitution
- man(Marcus) $\wedge$ born(Marcus, t_1) $\wedge$ gt(now $- t_1$, 150)

  (1)
- born(Marcus, t_1) $\wedge$ gt(now $- t_1$, 150)

  (3)
- gt(now $-$ 40, 150)

  (7)
- gt(2001 $-$ 40, 150)

  compute difference
- gt(1961, 150)

  compute *gt*

In this we have statements of the form $a \wedge b \Rightarrow c$, which must be solved as a and b two subgoals. But these subgoals may not be independent if they share variables. In the above, we can satisfy born(Marcus, t_1) by binding t_1 to the value 40. But this means that the t_1 in gt(now $- t_1$, 150) must also be 40.

We need to develop a systematic method of exploring the tree of logical consequences that can keep track of such bindings, and can ensure that bindings are uniform throughout the proof. However, as always, there are problems. It takes many steps even to prove trivial things: there is no good way of chunking data into larger facts. Also, we need ways of dealing with more complicated logical statements, such as those with more than one term on the RHS.

6.3 Resolution

This is a computational method for automating proofs that was first developed in the 60s. It proceeds by refutation, or proof by contradiction. To show a statement is true we assume its negation, and try to show that his leads to a contradiction with the existing statements. First, though, we we need to cast our statements into a normal form that the resolution method can use.

6.3.1 Clause Form

Statements can be complicated:

$$\forall x : (\text{Roman}(x) \wedge \text{know}(x, \text{Marcus})) \Rightarrow (\text{hate}(x, \text{Caesar}) \vee (\forall y : \exists z : \text{hate}(y, z) \Rightarrow \text{thinkcrazy}(x, y))) \,,$$

or, "All Romans who know Marcus either hate Caesar or think that anyone who hates anyone in crazy".

We need to simplify this, by reducing the amount of nesting, and somehow removing the quantifiers. The form we seek is *conjunctive normal form:* statements are expressed as a conjunction of (list of) disjunctions ($\vee$s) of predicates or negations of predicates. The CNF for the above is

$$\neg\text{Roman}(x) \vee \neg\text{know}(x, \text{Marcus}) \vee \text{hate}(x, \text{Caesar}) \vee \neg\text{hate}(y, z) \vee \text{thinkcrazy}(x, y)$$

The essence of this translation is as follows:

1. Eliminate $\Rightarrow$. This is easy, as $a \Rightarrow b$ is equivalent to $\neg a \vee b$.
2. Replace $\neg(\neg a)$ by a, and use DeMorgan's laws to distribute $\neg$ over terms: $\neg(a \vee b) \equiv \neg a \wedge \neg b$, and $\neg(a \wedge b) \equiv \neg a \vee \neg b$. Also we need $\neg\forall x : P(x) \equiv \exists x : \neg P(x)$, and $\neg\exists x : P(x) \equiv \forall x : \neg P(x)$.
3. Rename variables, so that no two bound variable have the same name. For example $(\forall x : P(x)) \wedge (\exists x : Q(x))$ can become $(\forall x : P(x)) \wedge (\exists y : Q(y))$. This avoids clashes later.
4. Move all quantifiers to the left. So $(\forall x : P(x)) \wedge (\exists y : Q(y))$ becomes $\forall x : \exists y : P(x) \wedge Q(y)$.
5. Eliminate $\exists$. Saying that $\exists x : P(x)$ is equivalent to saying that $P(S_1)$, where S_1 is a symbol that names one of those objects for which P of it is true. If an existential quantifier is within the scope for a universal quantifier, the introduced symbol may depend on the universally quantified one: $\forall x : \exists y : P(x) \wedge Q(y)$ must become $\forall x : P(x) \wedge Q(S_1(x))$. These are *Skolem functions*, or *Skolem constants.*
6. Drop the universal quantifiers. The universal quantification is now implicit in the use of variables. $\forall x : P(x) \wedge Q(S_1(x))$ becomes $P(x) \wedge Q(S_1(x))$.
7. Convert to a conjunction of disjuncts. This means distribute the $\vee$s over the $\wedge$s. So $a \vee (b \wedge c)$ becomes $(a \vee b) \wedge (a \vee c)$.
8. Create a separate clause for each disjunct. So $(a \vee b) \wedge (a \vee c)$ becomes the two statements $a \vee b$, $a \vee c$. So the original statement is true if *all* the separate clauses are true.
9. Change the variable names in each clause generated in the previous step, so that no two clauses contain the same names. Again, this prevents clashes later in the resolution algorithm. So $P(x) \vee Q(x, y)$, $R(x)$ becomes $P(x) \vee Q(x, y)$, $R(z)$.

The process of removing quantifiers is called *Skolemisation.*

6.3.2 Resolution

We can now start the resolution. This proceeds by taking together clauses in pairs, and producing a new clause from their unification. If we have two clauses

$\text{kermit} \vee \text{piggy}$
$\neg\text{kermit} \vee \text{frog}$

This is really a conjunction of the two: $(\text{kermit} \vee \text{piggy}) \wedge (\neg\text{kermit} \vee \text{frog})$, which is equivalent to $\text{piggy} \vee \text{frog}$ if we expand and simplify (they are equivalent to $\neg\text{piggy} \Rightarrow \text{kermit}$ and $\text{kermit} \Rightarrow \text{frog}$, giving $\neg\text{piggy} \Rightarrow \text{frog}$, or $\text{piggy} \vee \text{frog}$).

We can do this mechanically: take the two clauses, delete opposing pairs of terms (kermit and ¬kermit in the above), and join the remainder using $\vee$.

If we get the empty clause, then the original clauses were inconsistent

kermit
¬kermit

This is a contradiction, and contradictions are what the resolution algorithm hopes to find.

An algorithm for resolution: to determine if a statement P is true

1. Convert all statements to clause form, including the negation of P
2. Repeat until contradiction, or we are exhausted
   (a) Select two clauses, the *parents*.
   (b) Resolve them together. This many require *unification*, see later. The new clause is called the *resolvent:* this contains all the terms in the original two clauses (possibly after some unification of variables), but with terms T and $\neg T$ cancelled.
   (c) If the resolvent is empty, we are done, (P is proved). If not, add it to the set of clauses, and continue.

An example: suppose we wish to prove the statement "Rover will die" from the statements "Rover is a dog", "all dogs are animals", and "all animals will die".

Predicate Form	**Clause Form**
dog(Rover)	dog(Rover)
$\forall x : \text{dog}(x) \Rightarrow \text{animal}(x)$	$\neg\text{dog}(x) \vee \text{animal}(x)$
$\forall x : \text{animal}(x) \Rightarrow \text{die}(x)$	$\neg\text{animal}(y) \vee \text{die}(y)$

Now we negate the thing we wish to prove, giving: ¬die(Rover).

- Resolve $\neg\text{dog}(x) \vee \text{animal}(x)$ and $\neg\text{animal}(y) \vee \text{die}(y)$ by substituting y for x to give $\neg\text{dog}(y) \vee \text{die}(y)$.
- Resolve this with dog(Rover) by substituting Rover for y to give die(Rover).
- Resolve this with ¬die(Rover) to yield the empty clause, the goal.

Thus poor Rover will die after all!

6.4 Unification

In the above we matched various clauses against each other, for example ¬dog(x) ∨ animal(x) against ¬animal(y) ∨ die(y). To do this we had to note that with a change of variables the "animal" predicates match. Similarly, we see that ¬dog(y) ∨ die(y) matches dog(Rover) by substituting a value for a variable. This actions of matching and substituting is called *unification*.

Unification is a formal matching procedure for comparing two literals in the predicate calculus and trying to find a set of substitutions which will make them identical. Given two literals, the first thing we need to do is check whether their predicates match. If not then they cannot be unified. If the predicates are the same then we need to look at the arguments and try and unify them. Different constants or predicates cannot match but identical ones can. Variables can match other variables or constants or even predicates, provided only that any predicate expression contains no instances of the variable being matched.

The complication is that all substitutions must be consistent. It is no good trying to unify $P(x,x)$ and $P(y,z)$ by substituting y for x and then z for x. In this case we want to do something like substitute y for x and then substitute z for y, i.e., compose the two substitutions. We write this as:

$$(z/y)(y/x)$$

So in the above example where we resolved ¬dog(y) ∨ die(y) and dog(Rover) we performed the substitution Rover$/y$.

In this case we substituted a constant for a variable. This was necessary to get a match, but in theory given the pair of sentences ¬dog(x) ∨ animal(x) and ¬animal(y) ∨ die(y) we could have done the substitution (Rover$/x$)(x/y) even though it wasn't strictly necessary. In practice we try to make the most general substitution possible, i.e, make as *few* substitutions as possible, to help the resolution procedure.

We must also avoid substituting an expression containing a variable for that variable (i.e., $f(x)/x$) which leads to infinite recursion.

6.4.1 The Unification Algorithm

This algorithm will return either the composition of the substitutions needed to unify two sentences L_1, L_2 or FAIL.

1. If either of L_1, L_2 is a variable or constant, then:
   (a) If $L_1 = L_2$ then return NIL (empty list of substitutions).
   (b) Else, if L_1 is a variable, then if L_1 occurs in L_2 then return {FAIL} else return (L_2/L_1).
   (c) Similarly if L_2 is a variable.
   (d) Else return {FAIL}.
2. If the initial predicates for L_1, L_2 differ return {FAIL}
3. (If L_1, L_2 have different numbers of arguments return {FAIL}.)
4. Initialise the variable SUBS to NIL
5. for each argument in L_1
   (a) unify it against the corresponding argument in L_2, and place the result in S (this is a recursive call).

(b) if S contains FAIL then return {FAIL}

(c) if S is not NIL then:

i. apply the substitutions S to the remainder of L_1 and L_2

ii. prepend S to SUBS.

6. Return SUBS.

The check in steps 1(b) and 1(c) are to ensure that an expression containing a variable is not unified against that variable. This prevents unifications like

$$f(x, x)$$
$$f(g(x), g(x))$$

where $g(x)$ is substituted for x. This would lead to an infinite descent of replacing xs by $g(x)$s.

6.4.2 Unification in Resolution

Now we can clarify step 2(b) in the resolution algorithm above. The resolvent of two clauses is the disjunction of all the terms in the original clauses with one exception: if there is a term T_1 in one parent clause, and a term $\neg T_2$ in the other parent, and T_1 is unifiable with T_2, then cancel T_1 and $\neg T_2$ from the resolvent, and apply the substitution produced from the unification to rest of the resolvent. If there is more than one pair of unifiable complementary terms, choose just one pair.

In the case of Rover, we unified the second term of $\neg\text{dog}(x) \vee \text{animal}(x)$ against the first term of $\neg\text{animal}(y) \vee \text{die}(y)$, i.e., $L_1 = \text{animal}(x)$ against $L_2 = \text{animal}(y)$.

Step 1, 2 and 3 are not applicable.

We look at the argument x of L_1. We need to unify against argument y of L_2

Recursively: to unify x against y we apply step 1(b), giving substitution (y/x).

SUBS now is the set $\{(y/x)\}$, so we apply step 5(c), and the terms become $L_1 = \text{animal}(y)$ and $L_2 = \text{animal}(y)$.

We are finished, with SUBS $\{(y/x)\}$

So now, back in the resolution algorithm, we get the resolvent $\neg\text{dog}(y) \vee \text{die}(y)$, as the x has been transformed to an y, and the "animal" terms cancel.

6.4.3 Resolution Strategies

We need to make systematic the way we choose the pairs of clauses to resolve, otherwise we might take a long time doing nothing.

- Eliminate clauses that are not useful, in particular tautologies (statements that are always true, e.g., $P \Rightarrow P$), and clauses that are subsumed by others, e.g., given $P \vee Q$ and P we can drop $P \vee Q$. We might do this as a one-off preprocessing pass.

- Only resolve clauses that have unifiable complementary terms. Otherwise we will not get shorter clauses: we are aiming for the goal of a zero length resolvent.
- If possible, resolve clauses that contain part of the statement we are trying to prove, or clauses generated from such clauses. This is the *set-of-support strategy.*
- If possible, resolve against clauses with a single term. This is also to produce a shorter clause than we started with. This is the *unit-preference strategy.*

The resolution algorithm can help toward the problem mentioned earlier, that was if the question P? is asked, do we try resolving against P or $\neg P$? If there is no clause with term matching P in it, then we know we can't resolve away to the empty clause if we enter $\neg P$. Similarly, if there are no $\neg P$ terms, it will be useless to enter P. This may not be obvious at the start, but may become clearer as we proceed. And similarly the problem of knowing whether a fact is unprovable from given data.

6.4.4 Filling in Answers

The technique of resolution can do more than simply answer "true" or "false" to a question, it can also provide data. Suppose, in the context of the facts about Rover, we ask the question "is there something that will die?", or $\exists x : \mathrm{die}(x)$. Taking the negation, $\neg\exists x : \mathrm{die}(x)$ or $\forall x : \neg\mathrm{die}(x)$ we get the normal form $\neg\mathrm{die}(z)$.

Resolve $\neg\mathrm{die}(z)$ against $\neg\mathrm{animal}(y) \vee \mathrm{die}(y)$ to get $\neg\mathrm{animal}(z)$ with substitution (z/y).

Resolve this against $\neg\mathrm{dog}(x) \vee \mathrm{animal}(x)$ to get $\neg\mathrm{dog}(z)$ with substitution (z/x).

Resolve this against dog(Rover) to get the empty clause, with substitution (Rover/z).

So we have proved this statement true, with the substitution $z =$ Rover, i.e., there is something that will die, and it is Rover.

This method will work in general. A question $\exists x : P(x)$ will either contradict a statement $\forall y : P(y)$, meaning there are many values for which $P(x)$ is true; or it will contradict a more specific statement $P(S)$, for some constant S, and S is a value we seek (it may contradict other specific values, too).

6.5 Comments on Logic

Predicate calculus plus resolution seem quite a powerful combination, but there are problems, of course. There is the *completeness* problem, namely whether it is guaranteed that a search will find a contradiction when one exists. As presented above there is no guarantee, but some changes can be made so there is (though these changes are not normally included in theorem-proving systems). Another problem is the time it takes to find contradictions: extra heuristics are needed to direct the search, or extra structure is placed on the data.

6.6 Natural Deduction

There is a problem with Logic Representation and resolution as a proof procedure (other than the difficulty with scaling to large problems), namely the very simplicity of the representation loses heuristic information present in the original representation of the facts. For example, we may believe that all judges who are not crooked are well-educated, or

$\forall x : \text{judge}(x) \wedge \neg\text{crooked}(x) \Rightarrow \text{educated}(x)$, or
$\neg\text{judge}(x) \vee \text{crooked}(x) \vee \text{educated}(x)$

in clause form. However, the latter also can be interpreted as saying that someone who is not crooked and not educated is a not judge. This may be so, but trying to prove someone is not a judge by showing they are not crooked and not educated is probably not a good approach. There is a lot of implicit information in "all judges who are not crooked are well-educated" that is not translated.

Another problem is that resolution is not a human-friendly method: interacting with a resolution prover is difficult. Often a question is asked of the system, and we get back a yes or no answer. Then the next question is "why?", and getting a human-comprehensible answer is difficult (an answer along the lines "because this clause resolved against that one" are not very helpful). Thus some are led to consider techniques known as *Natural Deduction*, a term that covers a multitude of methods used in combination. They attempt to model more closely the processes used in human deduction.

A major difficulty is that many kinds of facts are not easily represented in terms of predicates. Many sorts of information involve a degree of equivocation in their description, like

- "it is very cold today" temperature has a continuous scale of values
- "blond-haired people often have blue eyes" a fact that has a degree of truth
- "if there is no evidence to the contrary, assume that any adult you meet knows how to read" one fact is to be inferred from the lack of unspecified other facts
- "I think I understand mathematics, but others think I don't" multiple beliefs to be reconciled

Some of these problems are addressed later in this course (Bayesian Networks, Fuzzy Logic, non-monotonic reasoning, etc.)

7 Rule-Based Systems

Moving on from logic for a moment, we turn to the use of rules to represent knowledge. Rule-based systems have played a large rôle in the development of AI systems, and many commercial systems are of this type (or strongly based on rules).

7.1 Declarative Knowledge

We have briefly mentioned procedural and declarative systems earlier. The logic representation we used earlier was a *declarative* system. That is, knowledge is declared, but no action or use of the knowledge is given. To use declarative knowledge we need a program to manipulate it (e.g., resolution). On the other hand we can represent knowledge as the programs themselves: we might consider an implication "$a \Rightarrow b$" as a program to compute b given a (or perhaps a program to convert proofs of a into proofs of b). Then logic statements form a graph, with possible execution paths leading from the start to the goal. To use a procedural representation we need an interpreter to run the programs.

The real difference between declarative and procedural systems is where the control lies: consider

man(Marcus)
man(Caesar)
person(Cleopatra)
$\forall x : \text{man}(x) \Rightarrow \text{person}(x)$

where we wish to answer whether $\exists x : \text{person}(x)$.

If we view the statements declaratively, there is no particular order on them, and we might get answers $y = \text{Marcus}$ or $y = \text{Caesar}$ or $y = \text{Cleopatra}$, perhaps non-deterministically.

If we regard them as procedural, then there is the implicit ordering from top to bottom, and we would get a single answer $y = \text{Cleopatra}$.

Reordering the above to

man(Marcus)
man(Caesar)
$\forall x : \text{man}(x) \Rightarrow \text{person}(x)$
person(Cleopatra)

the declarative system sees the same set of statements, but the procedural system will now probably return $y = \text{Marcus}$. Or maybe something else if the system happens to search the rules in some other order.

There is no definite answer to whether declarative or procedural systems should be used (even if we could clearly determine which was which).

7.2 Logic Programming (Prolog)

Viewing logic as a procedural rule system has been done for a long time. The most popular example of such a system is Prolog.

A (pure) Prolog consists of a set of *Horn clauses*. This is a conjunctive normal form clause that has at most one positive literal, e.g., p, $\neg p \vee q$ and its equivalent $p \Rightarrow q$. This is clearly a restriction, but has important consequences: firstly efficiency, as the uniform and restricted form that clauses can take means you can write a better interpreter. Secondly, the logic of Horn clause systems is decidable (unlike full first-order logic).

Take a set of statements

$\forall x : \text{pet}(x) \wedge \text{small}(x) \Rightarrow \text{goodpet}(x)$

$\forall x : \text{cat}(x) \vee \text{dog}(x) \Rightarrow \text{pet}(x)$

$\forall x : \text{poodle}(x) \Rightarrow \text{dog}(x) \wedge \text{small}(x)$

poodle(Rover)

this becomes in Prolog

1. `goodpet(X) :- pet(X), small(X)`
2. `pet(X) :- cat(X)`
3. `pet(X) :- dog(X)`

4. `dog(X) :- poodle(X)`
5. `small(X) :- poodle(X)`
6. `poodle(rover)`

There are several points

1. constants are lower case, while variables start with upper case
2. *facts* contain only constants, while *rules* contain variables
3. all variables are implicitly quantified
4. $\wedge$ is represented by a comma, but there is no $\vee$: this must be represented as a collection of statements (2 and 3 above)
5. everything is written backwards: $p \Rightarrow q$ becomes `q :- p` ("q if p")
6. there are no negative literals
7. all left-hand-sides of `:-` have just one literal

To convert CNF to Prolog form

1. if there are no negative literals (i.e., it is just a single positive literal) use it as it is
2. otherwise rewrite as an implication, with the single positive literal as the rhs, and the conjunction of the negative literals as the lhs
3. reverse the order of the rhs and lhs
4. any variables that were in negative literals are regarded as existentially quantified (and are now on the rhs), while those now in the lhs are universally quantified.

This reverses some of the steps of the conversion to CNF. For example, $\neg\text{pet}(x) \vee \neg\text{small}(x) \vee \text{goodpet}(x)$ becomes $\text{pet}(x) \wedge \text{small}(x) \Rightarrow \text{goodpet}(x)$, which is `goodpet(X) :- pet(X), small(X).`

Similarly, $\forall x \exists y : Q(x, y) \Rightarrow P(x)$ becomes `P(X) :- Q(X,Y).`

The input to a Prolog program is (the database and) the the goal to be proved. The system uses backward reasoning, reading the program from top to bottom and left to right, searching depth first with backtracking. It looks for facts that prove the goal directly, and rules whose head matches the goal. Unification is used on matching: bindings are propagated as we search (note that we have to undo bindings as we backtrack). Usually Prolog programs are written with more specific facts or rules before more general ones to help this search.

With the goal

`goodpet(X)`

This matches rule 1, and so we try to prove `pet(X)` and `small(X)`, doing the `pet(X)` first.

`pet(X)` matches rule 2, so try `cat(X)`

`cat(X)` fails as there are no rules for cats `pet(X)` matches rule 3, so try `dog(X)`

`dog(X)` matches rule 4, so try `poodle(X)`

`poodle(X)` matches rule 6, with `X` bound to `rover`

We must now match `small(rover)`

and so on

There is no explicit way to represent negation in Prolog

$$\forall x : \mathrm{dog}(x) \Rightarrow \neg\mathrm{cat}(x)$$

cannot be represented. Instead the idea of *negation as failure* is used, i.e., this fact is represented as a lack of an assertion, and using a *closed world assumption*, which states that we know all the facts we need. This means any unprovable statement is false.

If we relax some of Prolog's restrictions we get more flexible rule-based systems.

7.3 Forward and Backward Reasoning

Prolog searches backwards from the goal. It is also possible to search forward from the start state

forward starting from the initial states as root build a tree by applying all rules with matching left hand sides at each level

backward starting from the goal state(s) as root build a tree by applying all rules with matching right hand sides at each level

Depending on the problem, it may be better to go in one direction than the other

1. number of start or goal states: it is better to move from a smaller set to a larger (easier to find) set
2. branching factors: find the smallest number of branches
3. answer justification: if the program will be required to justify its answer, it might be better to move in the direction that the user will want
4. what sort of problems to solve: if we need to determine the consequences of a new fact, move forward; if a query to to answered, move backward.

It is easier to travel home from a strange destination than to travel to it (locations close to home are easier to recognise than locations close to the destination). It has long been recognised that the best way to generate a proof of a theorem is by working from the result back to the premises.

Of course we can mix the two: when there is a large branching factor in both directions it can be worthwhile starting at the ends and working toward the middle (the *bidirectional search*): this can reduce the number of branches taken significantly if the number of nodes at each step grows quickly. On the other hand, there is a additional difficulty of getting the two searches to meet in the middle (c.f., building the Channel Tunnel). In means-ends analysis effort is directed to reducing the intermediate gap, but in general we have to be careful.

Sometimes we might (say) search forward most of the time, with the occasional bit of backwards searching. Perhaps a rule is nearly matched: most of the terms on the lhs match the current state except maybe one. Then we could use backward chaining to try to prove that last condition before moving forward again.

7.4 OPS5 and Rete

Forward search is useful when we want to react to new data: driving on the road we react to things as they happen. However, we have to be slightly more careful in a forward search: if a combination of states fire a particular rule, it is quite likely this combination is still valid after the firing. Thus we must ensure that we don't immediately fire the same rule again, ad infinitum.

In such systems it is estimated that as much as 90% of the time is spent in matching the lhs of rules: a particular system, OPS5, implements a very efficient matcher, using the *Rete* algorithm.

Given a set of rules, OPS5 will *compile* them into code to execute when we wish to do some matching. This has several advantages, not least we are running compiled code instead of interpreting rules, but principally there are two things we can take advantage of

- Temporal redundancy: each match-use cycle will typically only change a small part of the system state. So we keep note of those rules that matched this time round to try again in the next cycle.
- Structural similarity: many rules share some preconditions. To save time we can share the matching amongst those rules: if we have

  $$\forall x : \mathrm{big}(x) \wedge \mathrm{blue}(x) \Rightarrow \mathrm{ibm}(x)$$
  $$\forall x : \mathrm{big}(x) \wedge \mathrm{red}(x) \Rightarrow \mathrm{kawasaki}(x)$$

  then we can partially match both rules in a single step. Any condition is matched at most once, regardless of how many rules it appears in.

At each cycle the algorithm determines which rules it can fire: generally there are more than one, the *conflict set* of rules. Then we need a *conflict resolution strategy* to pick a rule to fire.

7.4.1 Conflict resolution

There are three basic ways of addressing conflict resolution

1. preference based on rules: for example, the order of the rules in the database (as in Prolog), or more use specific rules before more general ones, etc.
2. preference based on objects: the objects have an order of importance attached to them, for example when driving a car hitting the brakes to avoid an accident is more important than turning on the wireless.
3. preference based on states: use a heuristic function to evaluate a good rule to fire (this is best first search).

A classic example of a pattern matching system is ELIZA, an imitation of a psychotherapist (1966). (Implemented in Emacs: type `M-x doctor`). Rules are like

```
(X me Y) -> (Y you Y)
(I remember X) -> (Why do you remember X just now?)
```

where the rhs is an "appropriate" response to the lhs. Rules were indexed by keywords in the lhs to share matching, and used preference based on objects: certain words like "I" and "mother" are important and invoke rules in preference to others. Some rules had no lhs and were used when no matches could be found to generate replies like "Tell me something about your family."

8 Reasoning Under Uncertainty

The logic systems described above have all lived in a world that is complete, consistent, and unchanging. However, this is not flexible enough to describe many problems: the information may be incomplete or uncertain, and may change with time. Propositional and Predicate Calculi are examples of *monotonic reasoning*. Adding new facts to the knowledge base does not invalidate existing facts, and the whole system remains consistent. In other words if it is possible to infer a sentence P from the knowledge base, it will always be possible to infer P no matter how the knowledge base grows. Often, though, we are not reasoning directly with facts, but with *beliefs* that can change. A new belief added to the knowledge base may cause us to abandon an earlier belief, and this in turn will almost certainly change the set of inferences that can be derived from it.

In some cases we can reason on the basis of the absence of a piece of information. For example you might say "since I haven't been told otherwise, I will assume that there is an Applications of Logic lecture on Tuesday". This is an example of *non-monotonic reasoning*. Of course the addition of a new piece of knowledge may render this inference invalid.

In the presence of such reasoning we have to address several issues:

1. incompleteness: how to make inferences on the basis of lack of knowledge as well as its presence? For example, "innocent until proven guilty". We should distinguish between

   - we know $\neg P$
   - we cannot determine if P

   Monotonic systems can only use the former, while non-monotonic systems can use the latter. This is why they are called "non-monotonic": as more facts are entered, it may be the case that the number of "true" things decreases. On the other hand, in a monotonic system, adding facts always increases the number of provable things.

2. update: what to do when we add a new fact to the system? This may make some old proofs (called *justifications*) and their conclusions invalid, so they will have to be withdrawn.

3. inconsistency: what do do with contradictory inferences? Often, there are parts of the database that are locally self-consistent, but globally inconsistent. We must have a means of choosing what to believe.

8.1 Logics for Non-monotonic reasoning

We cannot work directly with first-order logic, as it is inherently monotonic. There are several kinds of non-monotonic logics proposed (no single one seems to solve all the problems). Such logics need to provide solutions to the difficulties mentioned above, and

- some way to indicate preference of one belief over another
- a practical implementation
- some correspondence to our own intuition of reasoning

8.1.1 Default Reasoning

One possibility is to reason on the basis of what is "most likely" to be true. This is often called *default reasoning*.

NML

One system that is a basis for default reasoning is NML (for non-monotonic logic). This takes the ordinary first-order logic, and adds a modal operator M. This operator means "is consistent".

$$\mathrm{pasta}(\mathrm{spaghetti}) \wedge \mathrm{M}\,\mathrm{likesPasta}(\mathrm{Socrates}) \Rightarrow \mathrm{philosopher}(\mathrm{Socrates})$$

can be interpreted as "if spaghetti is a form of pasta and it is consistent with everything else to believe that Socrates likes pasta then we conclude that Socrates is a philosopher". We can use all the usual rules for manipulations of logical expressions: from $A \wedge \mathrm{M}\,B \Rightarrow B$ with $\neg A \wedge \mathrm{M}\,B \Rightarrow B$ we can deduce $\mathrm{M}\,B \Rightarrow B$.

We need to define what "is consistent" means. Now consistency is an undecidable notion, so we need some heuristic to help us. Often the Prolog idea of negation as failure is used: to show P is consistent, try to prove $\neg P$. If we fail, then we assume $\neg P$ is false, so P is consistent. This is fine as far as it goes, but negation as failure is only really usable in Prolog as Prolog is restricted to Horn clauses (a decidable theory): in full first-order logic we might have to regard consistency as "failure to prove inconsistency within a certain amount of time".

Another problem is what happens when we have a set of non-monotonic statements which are individually consistent with the knowledge base, but collectively inconsistent. For example:

$\forall x : \mathrm{Scotsman}(x) \wedge \mathrm{M}\,\neg\mathrm{teeTotal}(x) \Rightarrow \neg\mathrm{teeTotal}(x)$
$\forall x : \mathrm{Presbyterian}(x) \wedge \mathrm{M}\,\mathrm{teeTotal}(x) \Rightarrow \mathrm{teeTotal}(x)$
$\mathrm{Scotsman}(\mathrm{Jimmy})$
$\mathrm{Presbyterian}(\mathrm{Jimmy})$

Clearly if we apply the first assertion then the second cannot apply, but if we apply the second assertion then the first cannot apply.

The answer lies in defining the set of theorems which can be derived from a set of axioms to be the intersection of all the possible sets of theorems that could be derived. Thus in this example no conclusion about Jimmy's drinking habits can be reached. This theory is very conservative about theoremhood. In the original NML, the semantics of M were unclear. A newer system *Autoepistemic Logic* fixed some of those problems.

DL

Another system is DL (Default Logic). This adds a new class of allowable inferences. We have inference rules of the form

$$\frac{A : B}{C}$$

which we read as "if A is provable, and it is consistent to assume B, then conclude C".

This is similar to the M operator of NML, but there are differences. Firstly, these new inferences generate *plausible extensions* to the given knowledge base: this means that from a inference such as above we deduce that C is plausible. A theorem in this logic is anything that is valid in any extension. Some other mechanism is needed to decide amongst the extensions.

Secondly, the non-monotonic nature is in the *inferences*, not the *expressions*. Given

$$\frac{A : B}{B} \qquad \frac{\neg A : B}{B}$$

in the absence of anything about A, we cannot deduce anything about B.

Abduction

We are used to the idea of deduction, e.g., from the axioms:

$\forall x : \text{politician}(x) \Rightarrow \text{liar}(x)$
$\text{politician}(\text{Waldegrave})$

we can *deduce* that Waldegrave is a liar. However suppose we knew that Waldegrave was a liar. Is it safe to assume that he is a politician? Now suppose we have a rule like:

$\forall x : \text{flatBattery}(x) \Rightarrow \text{wontStart}(x)$

This says that if our car has a flat battery it won't start. But suppose we observe that the car won't start, is is safe to assume that we have a flat battery?

In neither case will conventional logic licence such conclusions. However sometimes this technique, called *abductive reasoning*, provides a reasonable way of proceeding. Formally: given two sentences $A \Rightarrow B$ and B, if it is consistent to assume A then do so. Some deeper understanding of the relation between A and B (e.g., does A *cause* B or vice-versa) will help to decide whether abduction is a valid mechanism.

It would be nice if we could assign some sort of confidence level to these assumptions. For example if somebody is a liar that doesn't necessarily mean that they are a politician. On the other hand if your car won't start there is a pretty good chance that your battery is flat.

Abduction is not a kind of logic in the same sense as NML or DL, but is useful nevertheless. In fact it can be described using either of them:

$$((A \Rightarrow B) \wedge B \wedge \mathrm{M}\, A) \Rightarrow A$$

Inheritance

We can now revisit the problem of inheritance and logic. Earlier we saw the problems associated with inheritance in predicate logic: the overriding of a general value by a more specific one became very messy. We can use DL to fix this.

We write the rule for determining the height of a cricket player as a DL inference

$$\frac{\text{cricket-player}(x) : \text{height}(x, 6'1")}{\text{height}(x, 6'1")}$$

which says the height of a cricket player is 6'1" in the absence of contradictory information. If we also assert bowler(John Smith), we can deduce that JS is a cricket player, and so his height is 6'1". But we might also have asserted $\text{height}(\text{John Smith}, 5'8")$ we could deduce both that JS's height is 5'8", and it is also plausible that his height is 6'1". Adding a rule

$$\forall x, y, z : \text{height}(x, y) \wedge \text{height}(x, z) \Rightarrow y = z$$

now eliminates the default rule. Thus an explicit values overrides a general one.

8.1.2 Minimalist Reasoning

These logics are based on "there are fewer true statements than false ones, so we should assume anything true and relevant is in our knowledge base: the *only* true statements are those that can be deduced from our data."

Prolog uses this in its negation as failure system for Horn clauses. What happens when we remove the Horn clause restriction?

CWA

The *Closed World Assumption* is a minimalist system: we assume all the data we need, we have. A airline database will contain all the flights for that company, so absence of a route directly means that route does not exist.

There are two problems with CWA: firstly the real world is not closed. Secondly, there is an imbalance between positive predicates and negative ones. If our knowledge base contains just

$$P(\text{Rover}) \vee Q(\text{Rover})$$

we can deduce from the CWA both $\neg P(\text{Rover})$ and $\neg Q(\text{Rover})$, as neither of the individual $P(\text{Rover})$ and $Q(\text{Rover})$ can be deduced. But extending our database by these conclusions

$$P(\text{Rover}) \vee Q(\text{Rover})$$
$$\neg P(\text{Rover})$$
$$\neg Q(\text{Rover})$$

we get an inconsistency. The CWA extends our database by $\neg P$ whenever that is consistent. Now, when translating from English, it is a matter of chance whether we define P as a description of an event, or define $\neg P$, but it makes a difference in what we can deduce from the CWA.

If we choose a predicate "hot", with

hot(curry)
hot(chilli)

we can deduce ¬hot(ice cream). On the other hand, if we had chosen predicate "cold", with

¬cold(curry)
¬cold(chilli)

we can deduce ¬cold(ice cream).

8.2 Implementation

There are many problems with non-monotonic deduction, both in the systems themselves, and in their implementations.

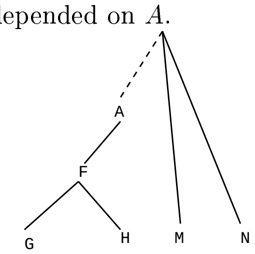
- management of conclusions: restricting our attention to those things relevant to the problem in hand

- management of the database: as we prove new statements we have to update our database. But this might have far-reaching consequences on other conclusions
- multiple extensions: we can have many plausible conclusions, even mutually contradictory ones
- computational effectivity: these logics are not decidable. What's more, they are not efficient.

We can classify search techniques into depth-first and breadth-first:

- depth-first: we follow a single most-likely path until we get a new piece of information that forces us to give up this path
- breadth-first: we consider all paths equally likely, and consider as a group. We eliminate paths as facts become available, and we hope to end up with just one (or a few) conclusion.

In depth-first, we use *dependency-directed backtracking* rather than *chronological backtracking*. Suppose we assume A, and derive F. Then we derive G and H from F. Next we derive M and N independent of A. If A turns out to be false, we need not discard (the proofs of) M and N, even though we must discard G and H. With chronological backtracking we discard everything after A (and A itself). With dependency-directed backtracking we discard everything that depended on A.



Systems such as JTMS (justification-based truth maintainance system) manage this by making networks of dependencies of *justifications* of statements, and traversing this. Another example is LTMS (logic-based truth maintainance system).

For breadth-first, typified by ATMS (assumption-based truth maintainance system) alternative paths are explored in parallel. Backtracking is avoided by maintaining multiple contexts, and as reasoning proceeds the collection of contexts is pruned as contradictions are found.

9 Statistical Reasoning

Non-monotonic reasoning deals with cases where we either believe a fact to be true or false, or we have no evidence which allows us to reach a definitive conclusion. Sometimes though we want to describe beliefs which are not certain, but for which we have supporting evidence.

In some cases there is an element of randomness, for example in playing dice or card games. We cannot predict things absolutely, but can use probability to give a reasonable estimate of what will happen.

A large class of problems fall into a related category: the world is fairly predictable unless certain exceptions occur. We can predict the likelihood of such exceptions occurring under certain circumstances. Many expert reasoning tasks fall into this category, for example the task of predicting the likelihood of finding certain ores at a given location given certain geological characteristics.

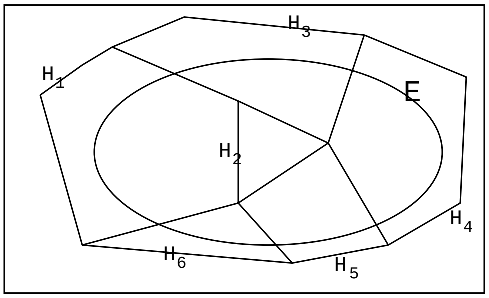
9.1 Bayes' Theorem

In reasoning, we often collect a body of evidence which points to a conclusion. We need some mechanism to allow us to assess what conclusion a given piece of evidence points to. Bayesian statistics is just such a mechanism which employs conditional probability as follows:

- Recall the notion of a conditional probability $P(H|E)$, which represents the probability that H is true given that E is true. This is just $P(H|E) = P(H \wedge E)/P(E)$.
- We can think of this as the probability of a given hypothesis being true when we observe a certain piece of evidence.
- We also need $P(H)$, the so called *prior probability* that H is true, i.e., the probability we would assign to H before accumulating any extra evidence.
- We also need the probability that we will observe E if H is true, i.e., $P(E|H)$.
- Bayes' Theorem states that if we are trying to discriminate between k different hypotheses, then the effect of observing a particular piece of evidence is as follows:

$$P(H_i|E) = \frac{P(E|H_i)\,P(H_i)}{P(E)} = \frac{P(E|H_i)\,P(H_i)}{\sum_{j=1}^{k} P(E|H_j)\,P(H_j)}$$

Here the H_i are all mutually independent.



This is a numerical analogue of abduction: it turns probabilities of a hypothesis given some evidence into probabilities of the evidence given some hypotheses.

This mechanism was used in the PROSPECTOR expert system. This was a system used to find mineral deposits by examining data about the geology of an area. To locate a possible deposit we need to know:

- The prior probability that any mineral will be found.
- The probability that certain physical characteristics will be present when various minerals are present.

Then we can work out how likely it is a particular mineral is present when we observe certain physical characteristics.

An example PROSPECTOR rule (converted into English) looks like:

```
IF   : magnetite or pyrite in disseminated or veinlet form is present
THEN : (2,-4) there is favourable mineralization and texture for the
              propylitic stage
```

The 2 represents the extent to which the presence of the evidence suggests the hypothesis, while the -4 indicates the extent to which the lack of the evidence suggests that the conclusion is not valid.

There is a major assumption implicit in all this: that all relationships between evidence and hypotheses ($P(E|H_i)$) are independent. This is in practice quite a hard thing to determine, and so often we need to look at the joint probabilities of collections of pieces of evidence. This collection of probabilities soon grows very large.

As an example of this, consider the problem of medical diagnosis. Suppose we observe that a patient has a fever and spots, and we wish to infer from that the fact that he or she has measles. We have to take into account the fact that spots and fever are not independent events by breaking the evidence spaces up into smaller pieces: here we would need probabilities for "spots without fever", "spots with fever" and "fever without spots". This number of cases grows as 2^n where n is the number of symptoms being considered.

There are other problems too:

- Whenever we discover a new relationship between hypothesis and evidence we need to rebuild all the probability relationships. In a constantly changing area like medicine this is very hard.
- The space requirement for all the probabilities is huge
- Getting the probabilities in the first place is hard: this must be done by expert people, and there is a lot of evidence that people are bad at estimating probabilities.

Bayesian statistics are sufficiently useful that several ways of making them useful and tractable have been explored.

9.2 Certainty Factors

The MYCIN system (late 1970s) was designed to help provide diagnoses for patients with meningitis-type infections. This is a difficult problem because by the time a doctor sees a patient suffering from these diseases, they are usually very ill and have to be treated immediately. The doctor must form a diagnosis based on the visible symptoms without waiting for lab tests to be performed (which can give an unambiguous diagnosis, but take several days). The program was designed to do what a doctor does in such a situation—come to a safe conclusion based on incomplete and possibly unreliable information. For reasons such as those just given it could not use pure Bayesian methods, instead it employed a more practical compromise.

The basic idea is that the presence of a piece of evidence effects how confident we are in the validity or invalidity of a particular hypothesis. So for example we might say "the fact that the lecturer is looking tired and pale suggests that he was out drinking late last night". The observed evidence could of course suggest other things, such as the fact that the lecturer has 'flu for example. To discriminate between these two hypotheses we might look for other evidence, and then we would need a mechanism for combining the two pieces to produce a new measure of our certainty in our conclusion.

A certainty factor consists of two components: a measure of belief (MB) and a measure of disbelief (MD) as follows:

- MB$[h, e]$ is a measure (between 0 and 1) of the extent to which evidence e supports hypothesis h.
- MD$[h, e]$ is a measure (between 0 and 1) of the extent to which evidence e supports the negation of hypothesis h.

Then the certainty factor is defined as:

$$\mathrm{CF}[h,e] = \mathrm{MB}[h,e] - \mathrm{MD}[h,e]$$

Since no evidence can support both h and $\neg h$ we see that one of MB and MD is always zero, and so the value of CF determines both MB and MD.

This method was developed for use in the MYCIN system at Stanford. MYCIN was designed to diagnose treatments for bacterial blood infections such as meningitis. It consisted of a number of rules (coded in Lisp) of the form:

```
PREMISES: (1) the morphology of the organism is coccus
          (2) the organism grows in clumps
ACTION  : conclude there is suggestive evidence [0.7] that the organism
          is staphylococcus
```

The initial certainty factors were entered by medical experts. However during the reasoning process it is necessary to combine certainty factors in different ways. It is important to note that in all cases the premise parts were independent of each other.

We can collect evidence in any order, so our mechanism for combining certainty factors must be commutative and associative. Until we are 100% sure of a conclusion, additional supporting evidence should increase our measure of belief and similarly for our measure of disbelief. Finally if we make an inference as a consequence of uncertain information, then the final CF should be less than if we made it on certain grounds.

Collecting Evidence Suppose we know $\mathrm{CF}[h,e_1]$ and $\mathrm{CF}[h,e_2]$ and we wish to calculate $\mathrm{CF}[h,e_1 \wedge e_2]$, then we need to consider the measures of belief and disbelief separately. Considering the measure of belief, this will clearly be zero when the measure of disbelief given both pieces of evidence is 1. Otherwise we take $\mathrm{MB}[h,e_1]$ and "reinforce it" by adding a proportion of $\mathrm{MB}[h,e_2]$ to it. The maximum value we can add is $1 - \mathrm{MB}[h,e_1]$, which we want to do in the case that $\mathrm{MB}[h,e_2] = 1$. So we end up with the formula:

$$\mathrm{MB}[h,e_1 \wedge e_2] = \begin{cases} 0, & \text{if } \mathrm{MD}[h,e_1 \wedge e_2] = 1; \\ \mathrm{MB}[h,e_1] + (1 - \mathrm{MB}[h,e_1])\,\mathrm{MB}[h,e_2] & \text{otherwise.} \end{cases}$$

and similarly:

$$\mathrm{MD}[h,e_1 \wedge e_2] = \begin{cases} 0, & \text{if } \mathrm{MB}[h,e_1 \wedge e_2] = 1; \\ \mathrm{MD}[h,e_1] + (1 - \mathrm{MD}[h,e_1])\,\mathrm{MD}[h,e_2] & \text{otherwise.} \end{cases}$$

Example: suppose we have $\mathrm{CF}[h,s_1] = 0.3$ (so that $\mathrm{MB}[h,s_1] = 0.3$ and $\mathrm{MD}[h,s_1] = 0$). If we now find that $\mathrm{CF}[h,s_2] = 0.2$ we can compute

$$\begin{aligned} \mathrm{MB}[h,s_1 \wedge s_2] &= 0.3 + 0.7 \times 0.2 \\ &= 0.44 \end{aligned}$$

so that $\mathrm{MD}[h,s_1 \wedge s_2] = 0$ and $\mathrm{CF}[h,s_1 \wedge s_2] = 0.44$. So two pieces of evidence s_1 and s_2 combine to give a higher confidence for h.

Combining Hypotheses Suppose we know $\mathrm{CF}[h_1,e]$ and $\mathrm{CF}[h_2,e]$ and we wish to calculate either $\mathrm{CF}[h_1 \wedge h_2,e]$ or $\mathrm{CF}[h_1 \vee h_2,e]$ then the formulas are:

$$\begin{aligned} \mathrm{MB}[h_1 \wedge h_2,e] &= \min(\mathrm{MB}[h_1,e],\mathrm{MB}[h_2,e]) \\ \mathrm{MB}[h_1 \vee h_2,e] &= \max(\mathrm{MB}[h_1,e],\mathrm{MB}[h_2,e]) \end{aligned}$$

with analogous rules for measures of disbelief.

Chaining rules Suppose we have $\mathrm{CF}[h, e_1]$, but we also have $\mathrm{CF}[e_1, e_2]$ (perhaps because e_1 is experimental data for example). Then we need some way to reflect our original confidence in e_1 in our confidence in h. If $\mathrm{MB}'[h, e_1]$ is our measure of belief in h where we are 100% certain e_1 is valid, then:

$$\mathrm{MB}[h, e_1] = \mathrm{MB}'[h, e_1] \max(0, \mathrm{CF}[e_1, e_2]) = \mathrm{MB}'[h, e_1]\, \mathrm{MB}[e_1, e_2]$$

The initial confidence factors are entered by an expert, and reflect the amount by which matching that rule will affect the system's belief in a given hypothesis. There is an implicit assumption of independence between the rules, but preserving that independence is the responsibility of the builder of the rule base.

So why is are CFs better than the Bayesian method? A CF represents a contribution to the belief of a hypothesis so it is related to the value $P(H|E)$. But this is the evidence for H given E and nothing else: given extra evidence joint probabilities must be considered. This is why CF combination formulas must have all rules independent Consider this example:

S: garden sprinkler was on last night
W: the grass is wet
R: it rained last night

If the sprinkler was on last night then this is good evidence that the grass will be wet this morning: this might have $\mathrm{MB}[W, S] = 0.9$. Similarly, if the grass is wet this morning then this is good evidence that it rained last night: $\mathrm{MB}[R, W] = 0.8$. But now chaining these two we get

$$\mathrm{MB}[R, S] = \mathrm{MB}[R, W]\mathrm{MB}[W, S] = 0.8 \times 0.9 = 0.72$$

so we have good evidence that it rained if the sprinkler was on. This is because $\mathrm{MB}[R, W]$ is actually dependent on $\mathrm{MB}[W, S]$.

MYCIN was a fairly successful system (the rules were carefully chosen to avoid the above problems), and was developed into EMYCIN, a generalised system that could form the basis for other kinds of expert behaviour.

More sophisticated methods of statistical reasoning include Bayesian networks. This attempts to solve the problems of the direct Bayesian approach by using modularity: not every event is conditionally dependent on every other, and so we can restrict our attention to cluster of events that interact. This builds networks of events where links are causalities, and uses a variety of algorithms for traversal to compute probabilities.

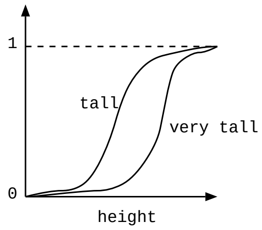
9.3 Fuzzy Set Theory and Logic

All the above systems take set theory and predicate logic and add little bits in a variety of ways. An alternative approach is to alter set theory itself.

Conventional set theory has a definite notion of membership: an element is either within a set, or it is not. Compare this with the need to represent

John is very tall
I am half asleep
This is nearly impossible to understand
This is almost, but not quite, entirely unlike tea

and so on. Traditional set theory cannot express degrees of membership of elements in a set. Fuzzy set theory ascribes a probability distribution instead of the usual boolean in/out.



This distribution describes you someone might be called tall as their heights increase: there is a lower probability of being called "very tall" than simply "tall".

A fuzzy set is a pair (S, σ) where S is a normal set and $\sigma : S \to [0,1]$. So $\sigma(s)$ is the degree of membership of s in S: if $\sigma(s) = 1$ then s is a full member, while $\sigma(s) = 0$ says s is not in S at all. For example, the fuzzy set of positive integers $(N, n \mapsto 1)$ where every element is definitely there (a *crisp* set). The fuzzy set of small positive integers could be (S, σ) where $\sigma(1) = 1$, $\sigma(2) = 1$, $\sigma(3) = 0.99$, ..., $\sigma(100) = 0.0001$, and so on.

A fuzzy subset of (T, τ) is a fuzzy set (S, σ) where $S \subseteq T$, and $\sigma \leq \tau$ everywhere on S. For example, the fuzzy set of small positive integers is a fuzzy subset of the fuzzy set of all positive integers.

A fuzzy function f between fuzzy sets (S, σ) and (T, τ) is a function $f : S \to T$ with $\sigma \leq \tau \circ f$. This means that the degree of membership of s in (S, σ) cannot exceed that of $f(s)$ in (T, τ). Thus every fuzzy subset $S \subseteq T$ relation has an associated fuzzy embedding $S \hookrightarrow T$.

There are problems with fuzzy sets that lead true logicians to use other kinds of partial membership in preference. Consider the following two fuzzy subsets of (S, σ). The first is $(\emptyset, \emptyset)$, the second $(S, s \mapsto 0)$. The first is the empty set, while the second is the set in which every element is not there. Neither set contains any elements, but these two must be considered distinct fuzzy sets (they have different fuzzy subsets). This leads to difficulties with power sets (taking power sets transforms membership into equality, so without fuzzy equality you cannot make fuzzy power sets).

10 Slot-and-Filler Structures

We move away from logic for the moment to consider two other structures for knowledge representations: conceptual dependency, and scripts. These have specific notions of what kinds of objects and relations between those objects may exist.

10.1 Conceptual Dependency

This (CD) is a way of encoding knowledge found in natural language descriptions. It has two aims: to allow us to draw inferences from the descriptions, and to be independent of the language in which the descriptions were actually written. CD uses a set of primitives which it hopes will be adequate to cover meanings for all natural languages. This is like a semantic net, but with a specified granularity and a specified set of primitives.

A typical set of primitives is

ATRANS: transfer of an abstract relationship (e.g., give)

PTRANS: transfer of the physical location of an object (e.g., go)

PROPEL: application of physical force to an object (e.g., push)

MOVE: movement of a body part by its owner (e.g., kick)

GRASP: grasping of an object by an actor (e.g., clutch)

INGEST: ingestion of an object by an animal (e.g., eat)

EXPEL: expulsion of something from the body of an animal (e.g., cry)

MTRANS: transfer of mental information (e.g., tell)

MBUILD: building new information out of old (e.g., decide)

SPEAK: production of sounds (e.g., say)

ATTEND: focusing of a sense organ towards a stimulus (e.g., listen)

There is a set of allowable dependencies amongst the conceptualisations in a sentence:

ACT: actions

PP: objects (picture producers)

AA: modifiers of actions (action aiders)

PA: modifiers of PPs (picture aiders)

(See Rich & Knight, p. 280, for a range of dependencies.) Furthermore, dependency structures can themselves be used as conceptualisations, and thus used as components of larger structures.

Finally, there are modifiers that can provide information about tense or mood of a form. These include

p: past

f: future

t: transition

t_s: start transition

t_f: end transition

k: continuing

?: interrogative

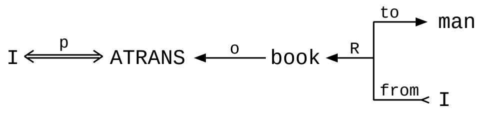
/: negative

nil: present

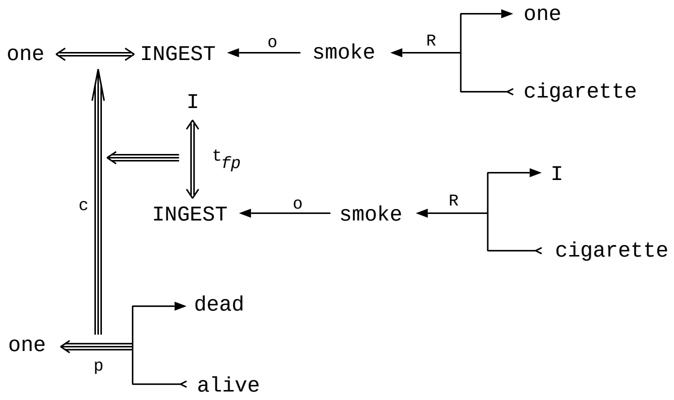
delta: timeless

c: conditional

A translation of "I gave the man a book" might be



Arrows indicate direction of dependency; "o" indicates object of the concept; "R" indicated recipient of the concept. Now, "Since smoking can kill you, I stopped" becomes



The vertical causality link indicates smoking kills. But the "c" modifier notes that this is conditional, that smoking doesn't definitely kill. The modifier t_{fp} indicates that smoking (the object link into INGEST) has stopped, and did so in the past.

The benefits of CD include these:

1. Everything is in terms of the given primitives, so fewer inference rules are needed. Rules need only be represented for each primitive: all of "give", "take", "steal", "donate" indicate transfer of ownership, and are covered by the same inference rules.
2. some inferences can be drawn directly from the representation itself
3. as we select CD structures to represent a natural language statement they will contain holes that need filling in: this directs attention to what is required to understand the natural language sentence fully.

Arguments against CD include the granularity of the primitives: representing even quite simple statements can take a huge number of CD forms. Further, CD only properly models events. Extensions or adaptions of CD are required to model other kinds of knowledge, such as social actions, and of physical objects.

10.2 Scripts

CD represents events: we also need to represent sequences of events. A *script* is a structure that describes a typical sequences of events appening in a certain context. A script is a set of slots, where each slot has associated information that describes what sort of values the slot may contain, default values, and so on. Additionally the script has certain other components:

entry conditions: conditions that generally must be satisfied before the events in the script can occur

result: conditions that generally will be true after the events in the script props: slots representing objects that are involved in the events

roles: slots representing people involved in the events

track: variations on the general pattern of events represented by this script

scenes: the sequences of events that can occur, represented in CD

The objects and people represented in "props" and "roles" are present even if not explicitly mentioned in the natural language description: we can infer their presence.

Scripts aim to imitate naturally occurring patterns of events, and chain together causally related things. The events in a script form a *causal chain*, stretching from the entry conditions to the results. Scripts are particularly good at inferring and predicting events that were not explicitly mentioned, and at describing how events are connected.

The Restaurant Script (Figure 1): this describe the things that might happen of someone went to the restaurant. Suppose we are given this story:

> Susan went out to lunch. She sat down at a table and called the waiter. The waiter brought her a menu and she ordered a hamburger.

We can ask the question "What did Susan eat?" This is not directly answered by the description, but as the details match the restaurant script, we can *activate* the script and deduce things from it. Activating the script involves filling in the slots with values we know, thus the customer S is Susan, and the food F is hamburger. Other things, such as the identity of the waiter, are left open.

Following the script, we can deduce that Susan did eat something, namely a hamburger from the "S INGEST F" event. We can also ask "why did the waiter bring the menu to Susan?" Going backward from this event in the script we find the event that caused the action of bringing the menu, namely "because Susan asked her to". On the other hand we can move forward to find what the event entails, and answer "so Susan could decide what to eat".

Scripts also help us to identify unusual events.

> John went to the restaurant. He was shown to the table. He ordered a large steak. He sat there and waited for a long time. He got angry and left.

This departs from the normal course of events, so we cannot deduce anything that happens after he ordered, such as "did John eat"? or "did John pay?"

Scripts are very effective for representing event-structured knowledge, but not so good at other kinds of knowledge.

10.3 Summary of Representation techniques

1. semantic nets
2. frames
3. predicate logic
4. production rules

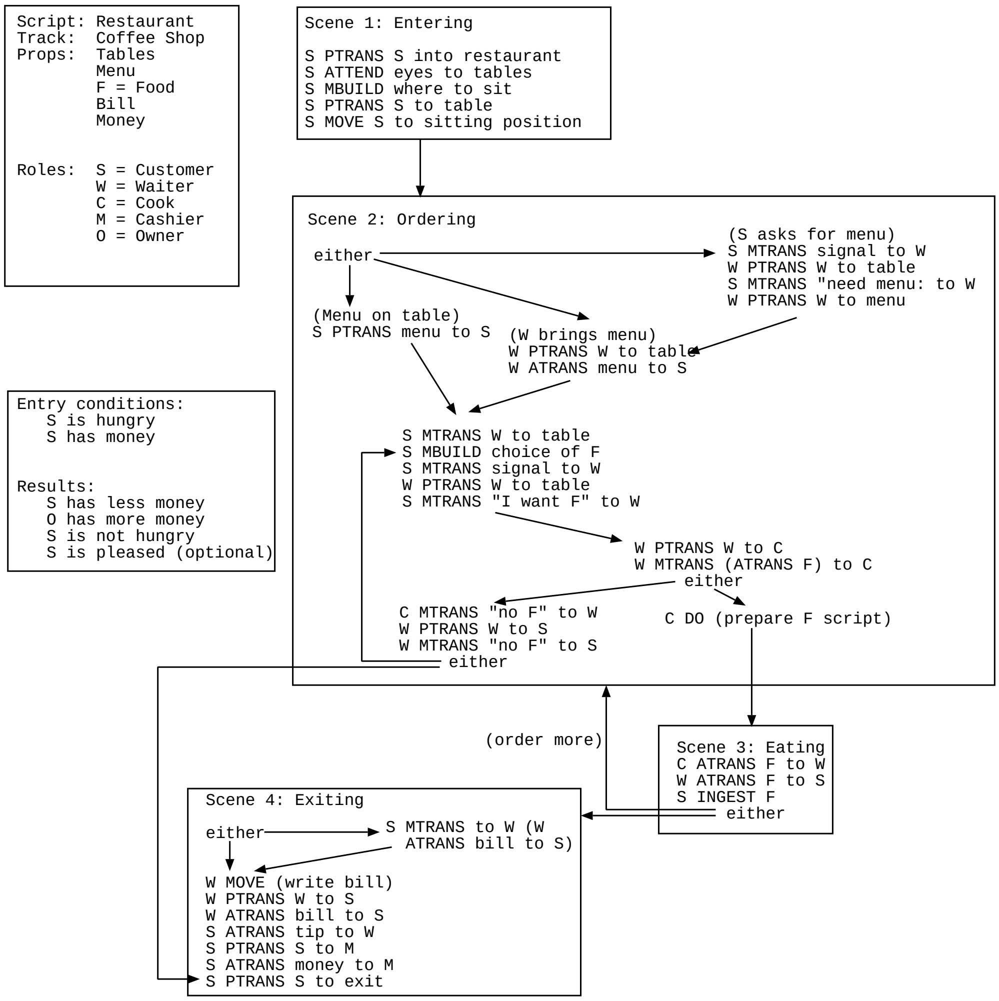


Figure 1: Restaurant Script

5. non-monotonic systems
6. statistical reasoning systems
7. conceptual dependency
8. scripts
9. . . . and there are many more.

11 Game Playing

Games (chess, chequers, go, poker, and so on) have been long been studied as AI problems. Originally, this was for two reasons:

1. they provide a easy to understand task, with well-defined goals

2. they do not obviously require a large amount of knowledge, and so are solvable by straightforward search techniques.

Unfortunately, the latter reason is now known not to be valid. Consider chess.

1. Average branching factor about 35
2. average game length of 50 moves for each player

This means to explore a complete game tree we need to examine about $35^{100} \approx 10^{154}$ nodes. So the brute force search idea will not work, and heuristics are required. Two ways to improve a search-based game player are

1. improve the generation of moves, so that only "good" moves are generated
2. improve the testing of paths so that the best moves are recognised and are explored first

The first aims to reduce the 35 next possible moves in chess, while the second aims to pick the best one to look at first.

Typically, game players use *plausible-move generators* which try to generate a few good-looking moves, with the hope that these will actually turn out to be good. For example, in playing go there is something on the order of 300 legal moves at any particular point: reducing this to a small number of plausibly good moves is essential. With a small number of moves to examine, the heuristic that checks paths can be more thorough.

Some games, in particular chess, have very stylised beginnings and endings, so it is valuable in such cases to have a simple look-up table of pre-computed good moves. A complete game will contain both searching and non-searching parts.

Searching methods that continue until a goal is found (e.g, depth first) will not work, so instead the game tree is only searched to some reasonable depth. It is usually only possible to search about ten or twenty moves (called *ply*) deep. At the tips of this search, a *static evaluation function* is used that does its best to evaluate the situation given the position on the board.

Static evaluation functions have figured large in the development of game playing systems. Turing suggested that a chess position could be evaluated as (sum of values of white's pieces)/(sum of values of black's pieces). The chequer playing program written by Samuel used a (mostly) linear combination of several simple functions

$$c_1 f_1(p) + c_2 f_2(p) + \cdots$$

for position p, and constants c_i. Furthermore, as each game was played, information was fed back to change the values of the c_i, to increase the values on those things that seem important, and decrease the values on those that seem unimportant. However, this is a problem in itself: how do we know which parts are important? This is the *credit assignment problem.*

The two elements of a good game player are

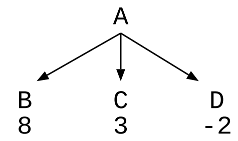
1. a good plausible move generator
2. a good static evaluation function

For a one-player game (or puzzle) the A* algorithm is enough: go as deep into the problems space as time allow, then use the heuristic estimate h at the terminal nodes, and propagate the values back up. But for two player games this does not work, as players alternate. What is good for player A on (say) odd numbered moves will be bad for player B on the even numbered moves.

A popular method used for game problems is the *minimax* procedure. This also work for searching problem spaces.

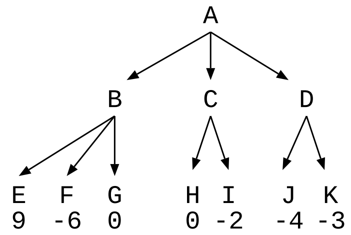
11.1 Minimax

The minimax procedure is breadth-first and depth-limited. Suppose we have a static evaluation function that we wish to maximise. From the current position we generate plausible successors. Now we can use the static evaluation function to estimate the successor nodes, and pass back the best value to be the value of the current node. As we are maximising, the best value is the largest value.



Here we will choose B as the best, and assign the value 8 to A.

Now, we could search deeper to get a more accurate estimation of A. For a two ply search we might get



But now this represents moves for the opponent, who will presumably wish to minimise the evaluation value. If we had played B, the opponent would probably reply with F. Thus the actual score for move B is -6. Similarly, if we play C, the reply will be I; and if we play D, the reply will be J. So the values of B, C, and D are -6, -2 and -4. We have the choice of B, C, or D, and of these C is the best (least worst!).

Thus, in general, we maximise on odd plys, and minimise on even plys.

The Minimax algorithm. It is invoked as Minimax(position, depth, player) to indicate the current position, the current depth of search, and which player.

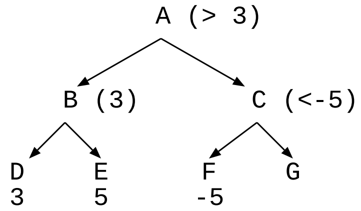
1. if DeepEnough(position, player) then return the value StaticEval(position, player), and an empty path (i.e., the static evaluation of this node, and there is no more path from this node).
2. Set SUCCESSORS to be the set of plausible next moves from this position for this player.
3. IF SUCCESSORS is empty (a winning or losing position), return StaticEval(position, player) and an empty path.
4. Otherwise, initialise BEST-SCORE to the minimum value that StaticEval can return. For each SUCC in SUCCESSORS do

   (a) Set RESULT-SUCC to Minimax(SUCC, depth + 1, Opposite(player)) (i.e., a recursive call from the point of view of the other player).
   (b) Set NEW-VALUE to the negative of the value of RESULT-SUCC.
   (c) If NEW-VALUE > BEST-SCORE then we have a new best node, so
      i. Set BEST-SCORE to NEW-VALUE
      ii. Set BEST-PATH to the path from RESULT-SUCC prefixed by SUCC

5. Return the value BEST-SCORE, and the path BEST-PATH.

11.1.1 Alpha-Beta Cutoff

Minimax is good, but it can be improved. We can sometimes not bother with branches of searches because we know that the best this branch can produce is worse than what we have found already. This is called *α-β pruning*, where we cut off unnecessary branches.



Here we have used minimax on the left branch B, and are currently considering the right branch C. After looking at F, we know that the value of C is -5 or less (a minimising ply). This is compared to the B branch, where we are guaranteed at worst 3. Thus we don't even need to look at (or generate) G, and in a deeper search we don't need to look at G or its successors.

The implementation of the α-β requires two bounds α and β to be kept which record the best-so-far and worse-so-far at each stage.

The effectiveness of α-β pruning depends greatly on the order we search the nodes: if we examine the worse paths first we gain nothing. On the other hand, if we search in a better order (but remember we don't have control of this) there is on average an effective doubling of the depth of search possible for a given amount of space and time. This is a significant improvement.

11.1.2 Better than Minimax

Other improvements to minimax include *waiting for quiescence*, where we search down a branch until there are no drastic changes from one ply to the next.

Also, there is *secondary search*, where after a normal search we choose a good move and explore more deeply from it. We may not be able to explore all the space to such a depth, but we can follow up a good-looking path more accurately.

11.2 Other Techniques

An alternative technique to minimax is *iterative deepening.* This generates a single ply and applies the static evaluation function; then generates the next ply and applies the function; then the third ply and so on.

This allows the system to search under time constraints (a chess game might be limited to 2 hours), as at any stage the search can be stopped and the best result so far returned. Also α-β pruning helps eliminate branches as we search.

Iterative deepening can be used with the A* algorithm, too. The *iterative deepening A**.

1. Set THRESHOLD to be the heuristic evaluation of the start state.
2. Do a depth-first search, pruning a branch when its total cost $g + h$ exceeds THRESHOLD. (If we happen on a solution path, return it.)
3. Otherwise, increment THRESHOLD by the minimum amount it was exceeded above, and repeat from step 2.

11.3 End Game

Chess is a good game to study in that it provides a good example of the trade-off between knowledge and search. Human chess players use a lot of knowledge and relatively little search (perhaps examining about 100 branches per move). In contrast, a computer can evaluate millions of branches, and uses the inferior knowledge encoded in the static evaluation function.

The most successful chess machines use deep searching to a much greater extent than they use knowledge: some use custom built parallel hardware to speed up plausible move generation and examination.

On the other hand, go playing machines are currently much worse off, as they cannot possibly examine enough nodes by simple brute force. The human game is almost entirely knowledge-based (perhaps based on recognising patterns?).

Another problem is that game playing machines generally assume that the opponent is perfect, and they will choose their best possible move at each play. This is not true, of course. A human player can take advantages of the failings of another and lead them into traps or deliberately mislead them to win more quickly than a machine could.

12 Natural Language Processing

Human languages are by far the most common way of transmitting information between people. In fact, speech forms the greater part of this transmission, while written language plays a less important rôle. We shall consider the processing and understanding of the written form only. There are many interesting problems with speech understanding, enough for an entire course of its own (see C86).

12.1 The Problems

There are many problems with understanding well-formed natural language sentences. We shall not even consider understanding non-syntactic statements, e.g., newspaper headlines: though these are easily understandable if they are known to be headlines (perhaps humans have a headline parser that is different from the normal parser?).

1. English sentences are incomplete descriptions of what they purport to convey. "Do you want to go to the coffee bar?" implies that you might want to drink some coffee. This allows a greater flow of information using fewer words, as much can be left unsaid.

2. There are multiple meanings that depend on the context. "Where's the water?" requests pure water in a chemistry lab; drinkable water in a house; undrinkable water when investigating a leak in the car radiator. On the other hand, we can describe an infinite number of things using a finite language.

3. Natural language is always changing. "I'll fax it to you" is a new construction on old words; also the meanings of words can change. For example, in Britain "mad" used to mean angry, but now means "insane". The old meaning is the one that was taken to the USA by the early settlers and is still used there.

4. There are multiple representations of the same fact: "Mary was born on October 11th" and "Mary's birthday is October 11th."

5. Sometimes statements are not what they seem. "Do you want to go to the coffee bar?" may be answered by "I've got to write my lecture notes." There is an implied answer of "no".

There is the further problem in that we haven't made it clear by what it means to "understand" a piece of natural language. This, too, depends on context, as it might only require the extraction of gross features (where "John kicked the dog" can become "kicked(John,dog)") or more subtle information may be required (e.g., John dislikes dogs, or John dislikes this particular dog, or John has a violent disposition, etc.) The target of the representation (logic, CD etc.) can make a difference in how we translate things, too.

12.2 Big Steps

Generally, language understanding is divided into five phases. It is important to note that these phases are not disjoint, and may be done simultaneously or in sequence, and they may pass information between themselves.

1. Morphological analysis. Determining what are words and what is punctuation. Note that punctuation is important, compare

   Can factories produce pollution?
   Can factories produce pollution.

2. Syntactic analysis. The linear sequence of words are converted to a structure (e.g., tree) that reflects the syntax of the sentence. There may be several candidate structures in the case of ambiguity. Examples include "I saw the man on the hill with the telescope," or the famous "time flies like an arrow." This might spot non-syntactic sentences: "The kicked John dog". (Note that "The kicked dog John" is a meaningful sentence!)

3. Semantic analysis. Mappings are made from the syntax structures to objects in the target domain. There may be syntactically correct sentences with no such mapping. A famous example is "colourless green ideas sleep furiously". This is *semantically anomalous.*

4. Discourse integration. Sentences may depend on and may influence the preceding and following sentences. In "John wanted it. He always had." the "it" depends on something earlier, while the "John" influences the "he".

5. Pragmatic analysis. The sentence is reinterpreted to find its actual meaning: "Do you know what time it is?" is actually a request for the time.

Syntactic analysis depends a lot on semantic analysis. In "kicked John the dog" we would need to know that "John" is not a verb, and "kicked" is not a noun in order to determine syntactic incorrectness.

12.3 Example of Processing

As an example we will try to understand

> I want to print John's .profile file.

This is actually a request, not a statement.

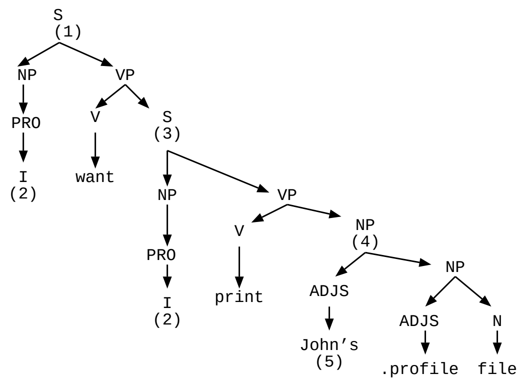
12.3.1 Morphological Analysis

This will recognise "John's" as a word "John" with a possessive modifier "'s". Also this should spot the dot in ".profile" is not an end of sentence, but forms part of the word ".profile".

12.3.2 Syntactic Analysis

This is also called *parsing*, and on the face of it, has a great deal in common with parsing of computer languages. On the whole, though, computer languages are context free (Chomsky type 2), which is not thought to be adequate for natural languages. So most of the theory learnt in computer science courses is not 100% applicable, though can be used to some extent.

Our sentence might be parsed as



The *reference markers* in parentheses correspond to entities in the sentence, and may be matched against entities in, say, a frame system. This parse we might get from a grammar:

S $\rightarrow$ NP VP

NP $\rightarrow$ the NP1 | PRO | PN | NP1

NP1 $\rightarrow$ ADJS N

ADJS $\rightarrow$ ϵ | ADJ ADJS

VP $\rightarrow$ V | V NP

N → file — printer

PN → John

PRO → I

ADJ → short | long | fast

V → printed | created | want

Here we interpret

S	sentence
NP	noun phrase
VP	verb phrase
N	noun
PN	proper noun
PRO	pronoun
ADJ	adjective
V	verb

This may be done by top-down or bottom-up parsing (c.f., forward and backward searching), whichever is more appropriate.

Dealing with Ambiguity

Some sentences may admit more than one parse. In fact, many do, but humans are so adept at choosing the correct interpretation they often do not notice the alternatives. There are four principal ways of handling multiple parses:

1. Follow all paths. Build all possible interpretations as we read the sentence, killing off those that become infeasible. This can be inefficient.
2. Best path with backtracking. Follow a reasonable interpretation until it becomes unreasonable, then backtrack. This requires saving enough state at every stage to enable the backtrack. Also, one section may be analysed many times (could use dependency-directed backtracking, but this is complex).
3. Best path with patch-up. Follow one path, but fix it up when we go wrong. This is more efficient than the above, but the patching can be difficult.
4. Wait and see. Follow one path, but don't assign syntactic classes until absolutely necessary. This has problems with *garden path sentences.*

   The old man the boats.
   The horse raced past the barn fell.
   The cotton clothing is made from grows in Mississippi.
   I told the boy the dog bit Sue would help him.
   The store has assembled models, as well as kits, in stock.

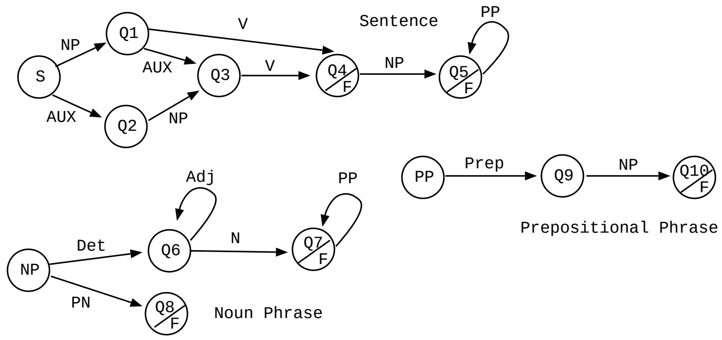
   These start off with one reasonable interpretation that later turns out to be wrong, though humans have exactly the same problem.

Augmented Transition Networks

An ATN is is top-down parsing procedure that incorporates knowledge into the parse to improve efficiency. They are used in many systems. An ATN is essentially a finite state machine, where the transitions between states are labelled. They may have a combination of

- specific words, like “in”
- word categories, like “noun”
- references to other networks that are used to recognise significant components, e.g., a prepositional phrase recogniser may refer to a noun phrase recogniser
- procedures that perform tests on the current state and sentence components that have already been recognised
- procedures that form the structures that comprise the parse

A fragment of an ATN for parsing English:



We try to parse the sentence

> The long file has printed.

The execution reads in the sentence a word at a time, starting with “the”

1. Begin in state S.
2. Push to NP.
3. Do a category test to see if “the” is a determiner.
4. This test succeeds. Move to state Q6.
5. (We look for a list of adjectives) Do a category test to see if “long” is a adjective.
6. This test succeeds, so return to state Q6.
7. Do a category test to see if “file” is a adjective. This fails.
8. Do a category test to see if “file” is a noun. This succeeds, we set NOUN to “file”, go to state Q7.
9. Push to PP.
10. Do a category test to see if “has” is a preposition. This fails. We can go no further, so pop back, and return fail.
11. We can go no further from state Q7, so pop and return what we have constructed so far namely (NP FILE (LONG) DEFINITE-DETERMINER).
12. We return to state Q1. Do a category test to see if “has” is a verb.

13. This test succeeds. Save "has" in V. Go to state Q4.
14. Push to NP. This tests "printed" for determiner or a proper noun, both of which fail. So we return to state Q4.
15. The only alternative for Q4 is to halt. But more input remains, so we have to backtrack.
16. We had a choice a state Q1, so we can unwind to there, undoing all the constructions we made from that point (namely that we thought "has" was the verb).
17. (Choose the other path) Do a category test to see if "has" is an auxiliary.
18. This test succeeds, go to state Q3.
19. Do a category test to see if "printed" is a verb.
20. This test succeeds, go to state Q4.
21. There is no more input, and we are in a acceptable final state. Pop and return

```
(S (NP FILE (LONG) DEFINITE-DETERMINER)
   HAS
   (VP PRINTED))
```

12.3.3 Semantic Analysis

This stage tries to get meanings for words. To "understand" a sentence to find a mapping from the natural language sentence to the *target language*, be it predicate logic, CD or whatever. There are, broadly speaking, two kinds of target language favoured in natural language processing:

- When the natural language itself is to be understood, and perhaps reponses to questions are required, the target language can be chosen to reflect those distinctions normally made in language. An example is CD.
- When the natural language is used as input to another system, e.g., a knowledge base or expert system, the target language must be legal input to the system. So the particular target system may influence the way we process the natural language.

Lexical Processing

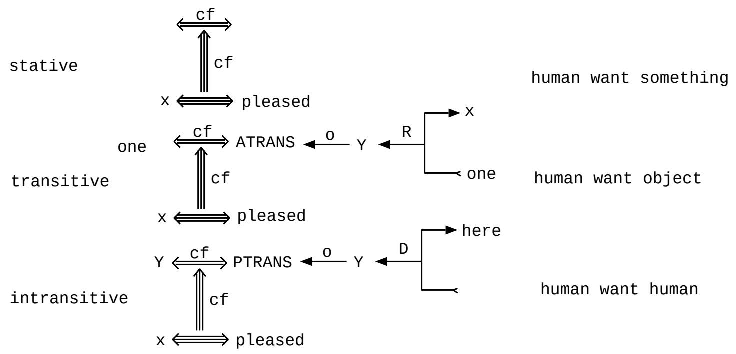
This looks up words in a dictionary (or *lexicon*) to get their meanings. Unfortunately, there is grand scope for ambiguity in a lot of words and often context is needed to choose the correct meaning. The process of choosing the right meaning is called *word sense disambiguation* or *lexical disambiguation*. Each word in the lexicon has contexts associated with it, and these point out the probable meaning in a given context. For example, consider "I'll meet you in the square" The word "square" might have the marker LOCATION, and "in" requires a word that has a LOCATION attribute. So we know that "square" does not mean "conventional person" in this context.

Useful attributes include PHYSICAL-OBJECT, ANIMATE-OBJECT, ABSTRACT-OBJECT.

The problem with this approach is that it has difficulty with metaphor: "My car hates the cold", where "hate" is usually associated with animate objects. Furthermore, to get a decent amount of accuracy of lexical disambiguation the number of attributes needs to be very large.

There are a number of other techniques used for semantic analysis, including

- Semantic grammars. A set of rules akin to syntactic grammars. The rules include semantic information, and semantic actions. (This works as syntax grammar is based on semantic concepts.)
- Case grammars. The syntax parser contains some (but not total) semantic information.
- Conceptual parsing. Syntactic and semantic knowledge are combined in a single system, driven by the semantic knowledge. Syntactic parsing is secondary, and only provides hints. Conceptual dependency structures are used to direct the processing: e.g., three CD diagrams associated with the word "want" could be
  1. wanting something to happen
  2. wanting an object
  3. wanting a person



When processing a sentence the PTRANS and ATRANS etc., can help to determine the correct interpretation. This has some of the problems of lexical processing in that a large amount of semantic information is required to make an accurate translation.

12.3.4 Discourse Integration

This phase will need to assign meanings to parts like "I" and "John". These would probably be translated as specific userids on some computer, e.g., "John" might be user `jsmith`. From knowledge of "John" we can determine the actual file referred to by ".profile".

12.3.5 Pragmatic Analysis

The presumable intended effect of this sentence is to get a printout of the named file. It is expressed as a simple statement, and not request. The reinterpretation would regard the statement of wanting as a request of action, and generate the wanted action, namely `lpr ~jsmith/.profile`.