➢ 1st October 1999

Introduction

The **Turing** Test has never yet been passed — where a computer tries to be a *human*. Modern AI is more interested in creating "*partners*" for people.

We need to make **inferences** — extract new facts from old. Inference saves **storage**. Knowledge representation is a big part of the course. *A semantic net* is an example of this. A problem solving method is describe and mesh. Consider the problem of the *farmer*, *fox*, *goose* and *grain*. The farmer wants to move himself and the others **across** a river, but only 2 things fit in the boat. Unattended, the **fox** eats the goose and the **goose** eats the grain. What should he do?

Stage 1: Draw *nodes* for every legal configuration. There are $2^4 = 16$ configurations. There are *6 illegal* configurations e.g. Farmer and fox on one side, Goose and Grain on the other. Stage 2: Make **links** for every transition (legal) between legal configurations/nodes. There are 10×9 possible links. Draw a **semantic net** showing all possible links. Throw away *surplus* information. The node-link representation is quite compact (object, relations). A good description of a problem opens the **door** to its solution. *"Once a problem is described using appropriate representation, it is almost solved"*.

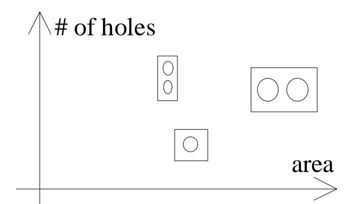
Representations consist of (1) *Lexical* part — describing what the **symbols** are in the representation vocabulary. (2) *Structural* part — describes constraints on how the symbols can be **arranged**. (3) *Procedural* part — specifying **procedures** for creating descriptions, **modifying** them and answering **questions** about them. (4) *Semantic* part — associates **meaning** with the descriptions.

A **semantic net** is a representation. Basically, there are *nodes*, *links* and *link labels*. Structurally, the links connect **tail** nodes to **head** nodes. Semantically, the nodes and links denote *application specific* entities. Procedurally, there are constructors for making nodes, links (labelling), readers (produces a list of all links leading/arriving at a node; produces a tail node/head node/link label **given** a link).

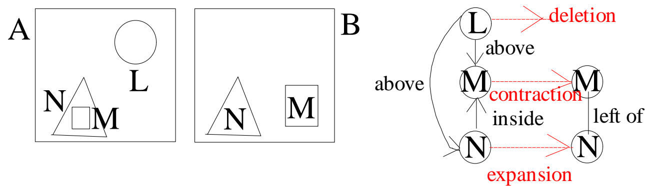
➢ 8th October 1999

Semantic Networks: Describe and Match

To **identify** an object, describe the object in a *suitable* representation. Match that description against a library of descriptions until there is a satisfactory match or until there are no **more** descriptions in the library. Announce the answer if there is one — success/failure. Suppose some feature you wish to match is not all-or-nothing but is described by some numbers. You might plot on a graph the location of an object e.g. switch face plates. Features: area of a face plate, number of rectangular holes.

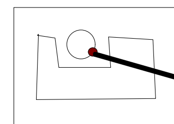


An **analogy** problem is: "A is to B as C is to ?". For analogy problems, use a geometric analogy net. It is a semantic net for which there are *nodes* denoting e.g. dots, circles, squares,... There will be *links* denoting relations among figure objects. Specifically inside, above, to left, ...

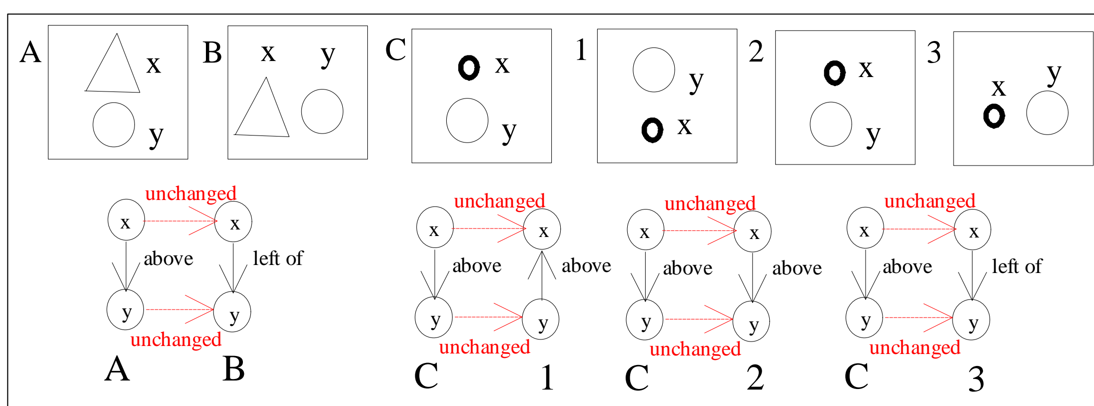


Other links describe how figure objects are *transformed*: addition, deletion, expansion, contraction, rotation, reflection.

How do we know objects are inside/outside each other? Pick a **point** on the surface of one object, follow to the *boundary*, counting the collisions with the second object. If there is an even number of crossings, then the 1st object is outside the 2nd. Conversely, if the number of crossings is odd, then the 1st object is inside the 2nd. Avoid **tangential** paths.



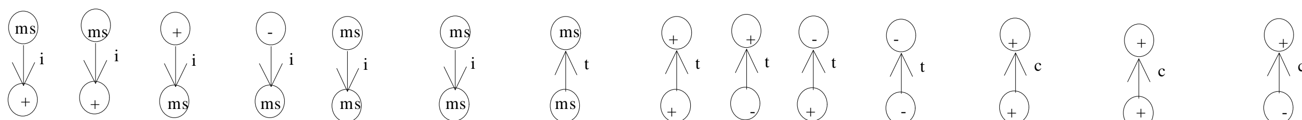
How do we know if objects are left/above? Calculate the centre-of-area for the 2 objects (45°). Make diagonal lines. It is better if the objects are **convex**, and the aspect ratio is not too different from each other. A *trial* problem is shown below. "A is to B what C is to _____". By constructing diagrams of C to all the numbers, we see that "*C is to 3*" is the correct answer.

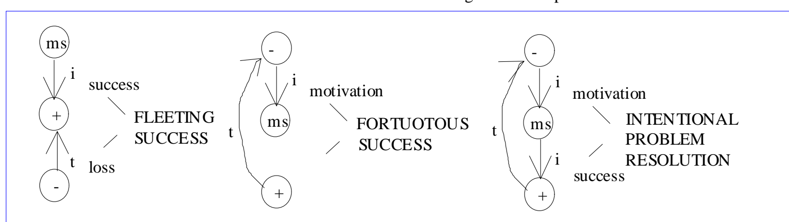


Label Assignment, Rotations

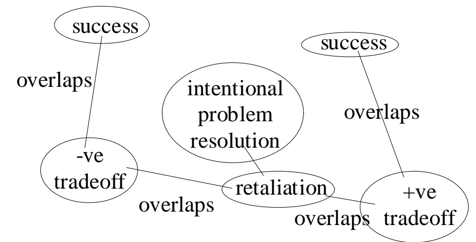
For n objects, there are n! ways of *labelling* objects. Rotations — the red arrows above could represent e.g. "Rotate 45°" if the 2nd object (e.g. the square) was rotated 45°. But beware with shapes like squares — if in the 2nd picture the square is unchanged, it could be represented on the red line as "unchanged" **OR** by "rotated by 90º".

Recognising abstractions in story parts: an *example*. Define a simple set of nodes: a character's mental state (ms), positive part (+), negative part (-); then link labels: I = initiates; t = terminates, c = coferers. Then we have the *following* base units shown below. These can be used to **build** stories, as shown in the blue box. (Read **Chapter** 2 of the book).





One process could be seen as *positive* from one person's perspective, but *negative* from another person's perspective. Diagrams can be drawn showing how **events** are seen from 2 people's perspective, and how events affect each other. The diagram on the right shows a situation where 2 people *form* a company because they have respect for each other. 1 is absent minded so the other reduces his duties; then we have retaliation...

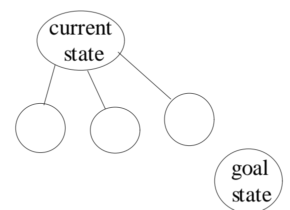


We have pattern recognition with **top** level design or description — this is where we have the *processes* e.g. intentional problem resolution in **linked** bubbles. Such analysis is helpful in e.g. **explaining** what the story is about. The *central* abstraction for this story as seen in the diagram is retaliation. The result of this is a loss for Thomas (-ve trade off) and a +ve trade off for Albert. Does a **particular** abstraction appear in a story? Does it *reappear* in another i.e. does e.g. success appear in another story?

3 Problem Solving Methods

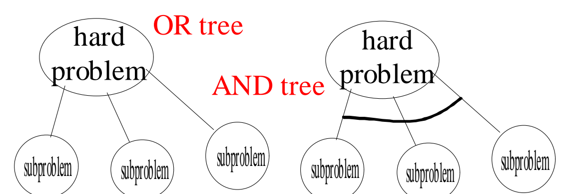
(1) **Generate and Test**. Two modules are used. A *generator* of trial solutions and a *tester* to check potential solutions. > until a satisfactory solution is found or until there are no more candidate solutions > generate a candidate solution; > test a candidate solution. if a match is made, announce it, otherwise announce failure. The generator produces **hypothesis**. Properties of good generators: ***completeness*** (eventually produces all candidate solutions) ***non-redundant*** (never produces the same "solution" twice) ***informal*** (they should use relevant information to restrict the solutions they try).

(2) **Means-ends analysis.** *State space* representation. A state is a description of the current situation. State-space: it has *nodes* representing states, and *links* which denote one-step transitions between states. Typically, we have a **current** state & a **goal** state. You evaluate a difference between the two states and then use it to generate a step which is likely to *reduce* the difference. If you generate a "bad step" which takes you further away from the goal step, do you go with it **anyway**? > until the goal is reached or unti there are no more steps available > describe the current state goal state; difference between them; > use the difference to select a promising step; > make that step and update the current state, > announce success or failure. Usually in means-ends analysis, we can produce a *difference* procedure table as shown on the left.

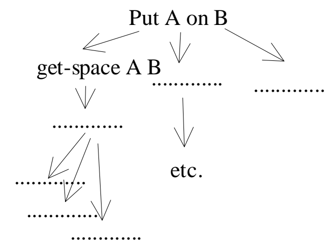


distance	Plane	Train	Car
>300mi	Y		
100-300mi		Y	
<100mi			Y

(3) **Problem resolution**. In computer programming, we often use *top-down methods*. Reduce the general problem into subgroups. This may be arranged into a **goal** tree. These may be "**OR**" trees or "**AND**" trees, as shown.



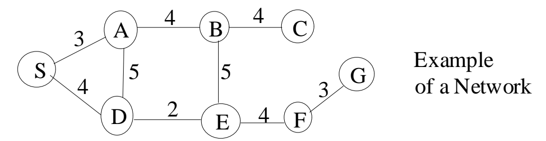
Example Goal: "Put Block A on Block B" or put <block> on <block>. We would also need *subroutines* or *subgoals*, e.g. put-on <...> <...>. *get-space, make-space, grasp, clear-top, get-rid-of, ungrasp* and *move* could be other subgoals. We could then produce a **goal tree** as shown which we could traverse. Some advantages of goal trees: allow **introspection**; "**How**" questions — by satisfying my immediate *subgoals*; "**Why**" questions — to satisfy my immediate *super goals*.

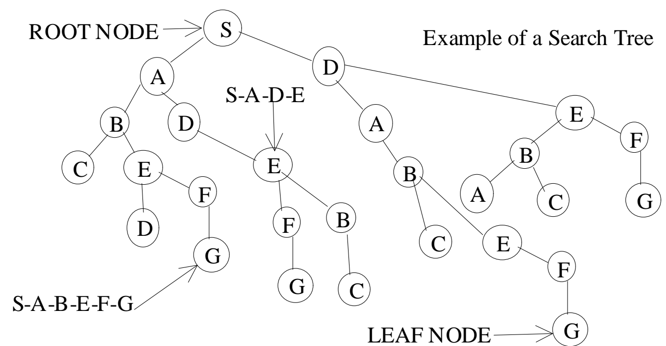


➢ 22nd October 1999

Searching through Trees

Consider **blind searching** first. We start with a network, an example of which is shown, and convert our network into a **search tree**. We ignore all paths with *loops*. This search tree is also shown. Formally, a search tree is a **semantic** tree. Each **node** represents a path. **Branches** connect paths by one-step path extensions. We can produce path descriptions from the tree. Typically, search trees will *explode exponentially* in size as the depth of the tree increases.





A **<u>depth first search</u>** dives into the tree and backtracks when it reaches leaf nodes, until it finds an *untried path*. The it continues. (Graphically, we think of it as evolving from **left** to **right** through the tree). Implementation: put the *child nodes* of a node at the front of a queue and develop these.

To **conduct** a depth-first search, **>** form a one-element queue, consisting of a zero-length path — containing therefore only the root node (our start node). **>** Until the first path in the queue terminates at the goal node or until the queue is empty, **>** Remove the first path from the queue, **>** create new paths by extending the path we just removed to all its neighbours, **>** reject all paths with loops, **>** add the new paths (if any) to the front of the queue. **>** If the goal is found, announce success, otherwise announce failure.

So in our example, we **could** have the following happening: (Note: **Bold** means it is being <u>deleted</u>): {[S]}, {[**SA**], [SD]}, {[**SAB**], [SAD], [SD]}, {[**SABC**], [SABE], [SAD], [SD]}, {[**SABE**], [SAD], [SD]}, {[**SABED**], [SABEF], [SAD], [SD]}, {[**SABEF**], [SAD], [SD]}, {[SABEFG], [SAD], [SD]}. Now the first element in the queue *reaches the goal node* so we announce <u>success</u>!

<u>Breadth-first search.</u> Search all nodes of depth 1, then move on to nodes of depth 2, then.... The algorithm is *similar* but we add new paths (if any) to the **<u>BACK</u>** of the queue instead of to the front. **>** form a 1-element queue for the start (root) node. **>** until the 1st path in the queue terminates at the goal or until the queue is empty, take the first path from the queue, **>** create from it child paths which extend it by one step; **>** reject all paths with loops, **>** put these new paths at the back of the queue. **>** Finish off.

So in **our** example, we could have the following *happening*: {[S]}, {[SA], [SD]}, {[SD], [SAB], [SAD]}, {[SAB], [SAD], [SDA], [SDE]}, {[SAD], [SDA], [SDE], [SABC], [SABE]}, {[SDA],...,[SABE], [SADE]}, {[SDE], [SABC], [SABE], [SADE], [SDAB]}, etc., until we get a [S....G] at the **start** of the queue.

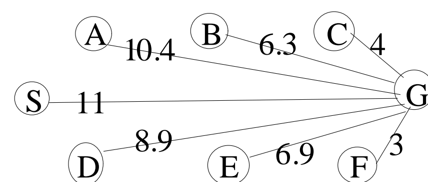
Suppose you wanted to find the *goal* with a path that had the fewest number of steps. In this case, use *breadth-first*. If memory is critical, you would want to use *depth-first*. Variations: Run depth-first with a depth **limit**. *Another* variation is called iterative deepening — do depth-first to some depth-limit. If there is no success, increase the depth-limit and start from scratch. The **length** of the paths and the **branching factor** is important in choosing *between* depth/breadth.

Non-Deterministic Search

If at each stage we **randomise** the queue, we get a complete search, but in a *non-deterministic* manner.

➢ 29th October 1999

Heuristic Searches



Let us now try to **improve** our search for (G) using *"as-a-crow-flies distances"* In Hill Climbing, we have a scoring function to help us decide between alternative choices. To conduct hill climbing, > construct a **one** element queue of the "root" node > Until the first path in the queue terminates at the *goal* node or until the queue is *empty*, > remove the first path from the queue; create new paths extending n to all its neighbourhood; reject paths with loops; > sort the new paths, if any, by the scoring function, placing the **best** choice first; > add the new paths to the front of the queue. > Announce *success* and the path, or announce failure.

Example on the **network** on page 4: {[S]}. [SA]$^{10.4}$ [SD]$^{6.7}$ so {[SD], [SA]}. [SDA]$^{10.4}$ [SDE]$^{6.9}$ so {[SDE], [SDA], [SA]}. [SDEB]$^{6.7}$ [SDEF]^{3} so {[SDEF], [SDEB], [SDA], [SA]}. [SDEFG]^{0} so {[SDEFG],....}. etc. **Parameter Based hill climbing**: Try small steps in all directions. Use the scoring function to pick one direction, and *commit* to it. (i.e. no queue — just too big a search space).

Beam Search: Like breadth first, however it only uses the *best w paths* at any level. **Best first search**. Like hill-climbing, but you sort the *entire* queue, not just the new paths.

Summary: Depth-first is good when *unproductive* paths are never too long. Breadth-first is good when the *branching factor* is never too big. Non-deterministic is good if you can't *decide* whether depth or breadth-first are better. Hill-climbing is good if there's a natural scoring function and when a *good* path appears to be good at each choice point. Beam-search is good if there is a scoring function and a *good* path is likely to be covered by the beam at all levels. Best-first is good when there is a scoring function and *incorrect* paths soon play out.

Optimal Search

Search for the best (e.g. **shortest**) path regardless of the time spent. (1) **<u>The British Museum procedure</u>**. e.g. depth or breadth-first search. **<u>Branch-and-Bound</u>**. Example: An oracle tells you that [SDEFG] is the shortest path (with length 13). *Compare* to [SDAB] (13). More generally, branch and bound keeps track of all <u>partial</u> paths. Only the **shortest** is extended to the next level.

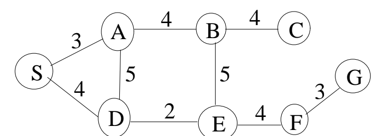
Because we are always *extending* shortest paths, we are <u>likely</u> to end up with the optimal one. To turn this into **certainty**, extend all partial paths until they are *all at least as long* as the complete path. You terminate where the **shortest** partial path is longer than the shortest complete path. Procedure for branch & bound: > form {[S]}. > until the 1st path in the queue is the *goal* or until the queue is *empty*, > remove the first path and create loop-free extensions. > add new *paths* to the queue. > sort the *entire* queue with the shortest paths at the front. > if the goal is found, announce success, else announce failure.

Branch & bound uses **exact** information about path lengths and is **not** heuristic based. Example using the network as before: $\{[S]_0\}$, $\{[SA]_3, [SD]_4\}$. $[SAB]_7$ $[SAD]_8$ $[SD]_4$ so $\{[SD]_4$ $[SAB]_7$ $[SAD]_8\}$. $[SDA]_9$ $[SDE]_6$ $[SAB]_7$ $[SAD]_8$ so $\{[SDE]_6$ $[SAB]_7$ $[SAD]_8$ $[SDA]_9\}$ and *so on* for 11 steps.

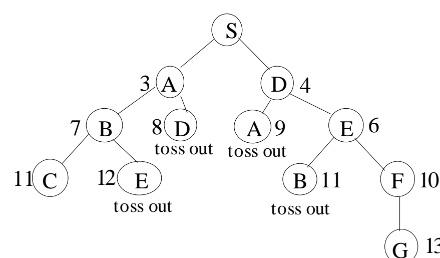
<u>Improve Efficiency by using underestimates</u>. Suppose we have some estimating function e() = estimate of. So *e(total path length) = d(already travelled) + e(distance remaining)*. If the estimates allow <u>over</u> estimates, we may **miss** the optimal solution. However, underestimates *cannot* cause the right path to be overlooked. So write u's **instead** of the e's above (u is an *underestimate*). **<u>Summary</u>**: *branch and bound search:* take the last <u>exact</u> path at each stage (sort the entire queue based on the exact cost so far). *Best-first search:* take the best <u>looking</u> path based on a heuristic.

➢ <u>2nd November 1999</u>

Redundant Paths (Dynamic Programming)



Consider the **diagram** as before. The dynamic programming principle: The best way **through** a particular intermediate place is the best way *to it from the starting point,* followed by the best way from it to the goal. There is no need to look at any **other** paths to or from this intermediate place.

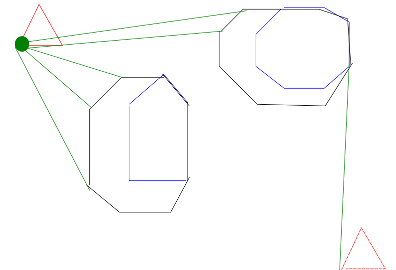


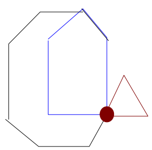
Procedure for **branch-and-bound** with **dynamic programming**: > form the *initial* queue with the **root** node. > Until the first path in the queue *terminates* at the goal or until the queue is *empty*, > remove the first path from the queue, creating one-step extensions, > **reject** paths with loops, > add the remaining paths to the queue, > <u>(dynamic programming part)</u> if two or more paths reach a *common* node, delete all those paths except the one that reaches this common node with the **minimum** cost, > <u>(branch & bound part)</u> sort the entire queue with the **best** cost paths first. > announce success or failure as normal. If you were to *draw* your progress through the algorithm for the network above, the drawing above is what you would get. Here, we **announce** success: SDEFG, total cost 13 (miles?).

Even more efficient is to include both **underestimates** of the remaining distance and **dynamic programming**. This is called the A* procedure. In the procedure, the dynamic programming part stays the *same* (use exact cost so far here). Now sort the queue using the **underestimate** for the total path = *distance travelled so far plus the underestimate for the remaining distance.*

Configuration Space

Consider the diagram shown to the *right*. We need to move the **red** triangle to the dotted location. What is the shortest path?, especially if there are the two **blue** obstacles as shown. What we do is to construct the *configuration space representation* for each obstacle. We track a point on the triangle and draw its path as the triangle goes around the object (without rotating). In the main diagram we will then move the triangle to one of the vertices of the configuration space representation and then use one of the methods described on the previous pages to calculate the **minimum** distance.





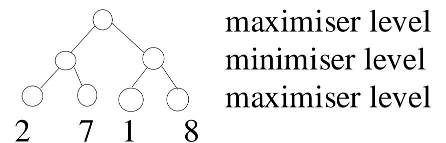
➢ 5th November 1999

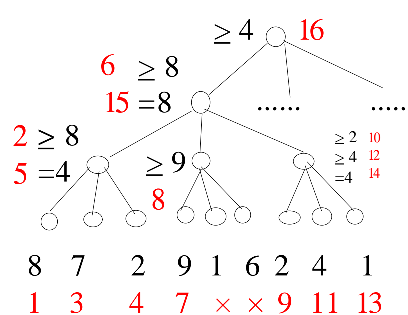
Read **chapter** 8 in the book after understanding *rule based systems in chapter 7* (we will not go over chapter 8 in the course). Tips: When writing out a **search** tree based on e.g. a network, write out the queue involved e.g. by using the *depth-first algorithm*, alongside the actual search tree. Try not to give **vertical** paths — always use slanted ones.

Trees and Adversarial Search

The *minimax* procedure aims to make the differences between moves (in adversarial games like chess, checkers, etc.) much **larger** by using a static evaluator after several moves. Suppose that the position at any time is *evaluated* by a static evaluation score. The player hoping for positive scores is called the **maximiser**. (negative scores — **minimiser**).

The minimax procedure (**MINIMAX**). > If the **limit** has been reached, compute the static value of the *current position* (relative to the appropriate player). Report the result. > otherwise, if the level is a **minimising** level, use MINIMAX on the children of the **current** position. Report the **minimum** of the results. > otherwise, if the level is a **maximising** level, use MINIMAX on the children of the current position. Report the **maximum** of the results.



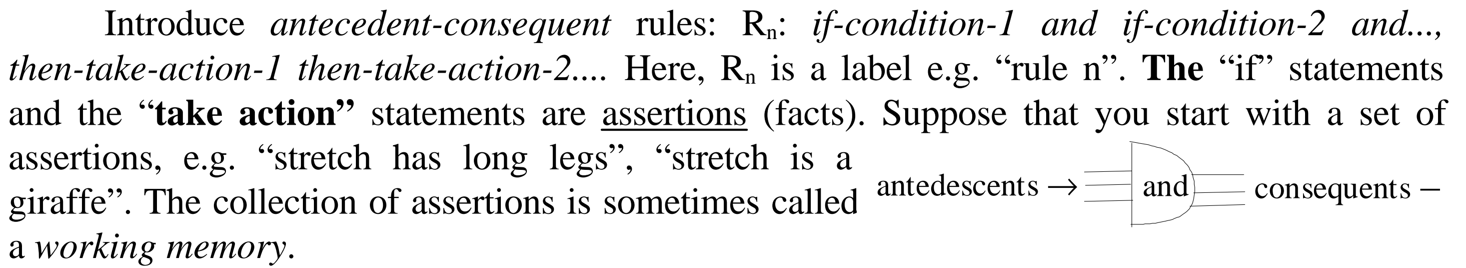


Minimax with alpha-beta pruning. The alpha-beta principle: if you have an idea that is surely **bad**, do not take the time to see how truly **awful** it is. See the book (page 110). We apply the principle to the *tree* on the left. The numbers shown in **red** denote the *order* of execution.

➢ 8th November 1999

Rule Based Deduction

Introduce *antecedent-consequent* rules: R_n: *if-condition-1 and if-condition-2 and..., then-take-action-1 then-take-action-2....* Here, R_n is a label e.g. "rule n". **The** "if" statements and the "**take action"** statements are assertions (facts). Suppose that you start with a set of assertions, e.g. "stretch has long legs", "stretch is a giraffe". The collection of assertions is sometimes called a *working memory*.



Sometimes, instead of an assertion for a consequence, we have an **action**, then the rule based system is called a *reaction system*. Usually, though, it is called a *deduction system*. The "*motion*" from the 'if' patterns to the 'then' patterns is called **forward chaining**. In forward chaining, whenever an if pattern is observed to match an *assertion*, (a fact), then the antecedent is said to be satisfied. Whenever **all** the if patterns of a rule are satisfied, then the rule is triggered. Whenever a triggered rule establishes a new assertion or performs an action, it is **fired**.

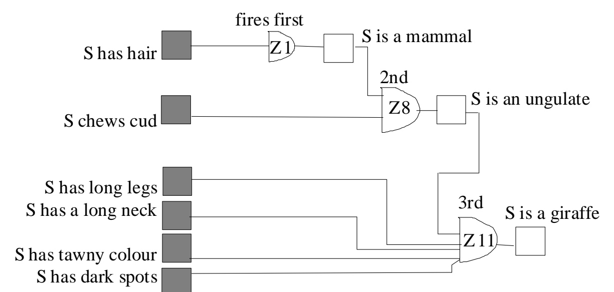
Example: *Deduction of animal identities*. Consider that we have the following working memory: *"Stretch has hair", "Stretch chews cud", "Stretch has long legs", "Stretch has a long neck", "Stretch has a tawdry colour", "Stretch has dark spots"*. We have a collection of (14) rules e.g. $\mathbf{Z_1}$: If ?x has hair then ?x is a mammal; $\mathbf{Z_2}$: If ?x gives milk then ?x is a mammal; $\mathbf{Z_3}$: If ?x has feathers then ?x is a bird; $\mathbf{Z_4}$: If ?x flies and ?x lays eggs then ?x is a bird; $\mathbf{Z_5}$: If ?x is a mammal and ?x eats meat then ?x is a carnivore....

For the working memory, we go through *all the rules* $\mathbf{Z_1,...,Z_{14}}$ and see if we trigger any of them. For example, analysing rule $\mathbf{Z_1}$, Stretch *does have hair* so $\mathbf{Z_1}$ is fired and *"Stretch is a mammal"* is added to the working memory. Similarly, when $\mathbf{Z_8}$ fires, *"Stretch is an ungulate"* is added to the working memory. Finally, $\mathbf{Z_{11}}$ fires, giving *"Stretch is a giraffe"*.

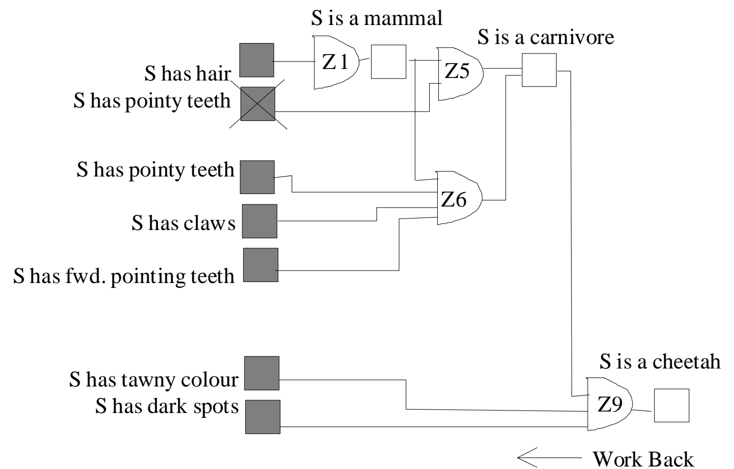
➢ 12th November 1999

To identify an animal using **forward chaining,** > until **no** rule produces a new assertion, or until the animal is *identified*, > for each rule, > try supporting each of the rule's *antecedents* by matching them to known facts, > if all the rule's antecedents are supported, assert the consequence, unless there is an *identical* one already. > Repeat for all matching and instantiation alternatives.

?x, X: When we **match** ?x to "Stretch", for example, we say that ?x is *bounded* to stretch and we say that the pattern x (?x, X) is instantiated. Schematically, this forms an inference net as shown.



Let's suppose "**Robbie**" has a hypothesis about an animal "*Swifty*": Swifty is a cheetah. To verify this, Robbie considers rule $\mathbf{Z_9}$, that is "S is a cheetah if S is a *carnivore*, has *tawny colour* and *dark spots"*. Consider that we have the following working memory: Swifty has *forward-pointing eyes, has claws, has pointed teeth, has hair, has tawny colour and has dark spots*. We then construct a diagram as shown on the top of the next page.



Explanation of the diagram: We start at "S is a cheetah". We analyse our *list of rules* and see what rule needs to be satisfied so that "S is a cheetah" is fired. Here it is only fired by $\mathbf{Z_9}$.

We see what is **required** for $\mathbf{Z_9}$ to fire. First, S must be a *carnivore*. For this to be true, then a rule must be fired because it is not in the working memory. $\mathbf{Z_5}$ fires this, but all of the conditions for $\mathbf{Z_5}$ are not met. So we look at $\mathbf{Z_6}$ which also fires "S is a carnivore". The conditions for this *are* met, so we can say that S *is* a carnivore. We then return to look if S has tawny colour and has dark spots. It does, so we conclude that S is **indeed** a cheetah based on the information in the *working memory*.

Back-chaining (for animal identification). > Until **all** hypothesis have been tried and either *none* have been supported, or until the animal is *identified*, > for each hypothesis, > for each rule whose consequent *matches* the current hypothesis, > Try to support each of the rule's antecedents by matching it to assertions in **working** memory, or by backward chaining through another rule, creating a new hypotheses. (Check **all** matching and instantiation alternatives). > If all the rule's antecedents are supported, announce *success* and conclude that the hypothesis is true.

In chapter 7, read about the **MYCIN** and **XCON** expert systems. We've described two ways to use *antedescent-consequence* rules: forward and backward chaining. Arguments for *backward* chaining: Is there a high degree of fan-out; are there many potential **conclusions** from the facts? Arguments for *forward* chaining: if a typical hypothesis can lead to many questions, the system exhibits a high degree of **fan-in**.

If neither dominates and if you have *no facts*, but the ability to inquire from the world, then **backward** chaining can allow you to efficiently test hypothesis. Alternatively, if you have all the facts at hand and *everything* you are interested in can be concluded from these, use **forward** chaining.

Suppose you have a *fleeting glimpse of an animal*. Here, backward chaining would lead to many **wasted** questions. It is better to use forward chaining. Consider that we have the working memory *"Comet is a horse", "Prancer is a horse", "Comet is a parent of Dasher", "Comet is a parent of Prancer", "Prancer is Fast", "Dasher is a parent of Thunder", "Thunder is fast", "Thunder is a horse" and "Dasher is a horse".*

Also we have the **rule** R_1: If ?x is a horse, if ?x if a parent of ?y, and if ?y is fast then ?x is *valuable*. For forward chaining *cycle* through all the alternatives.

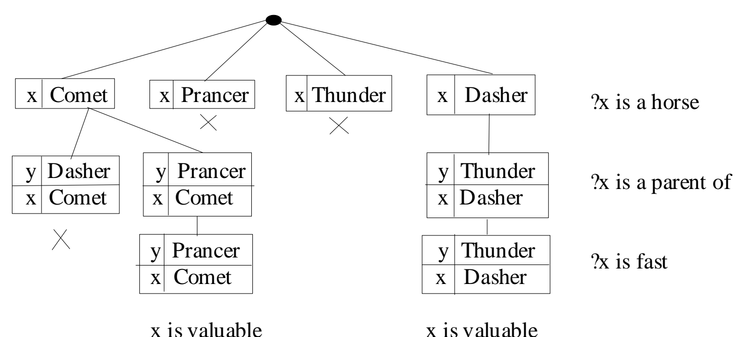
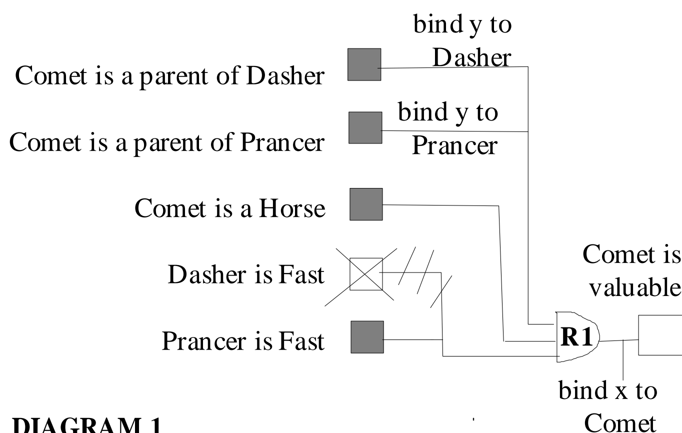
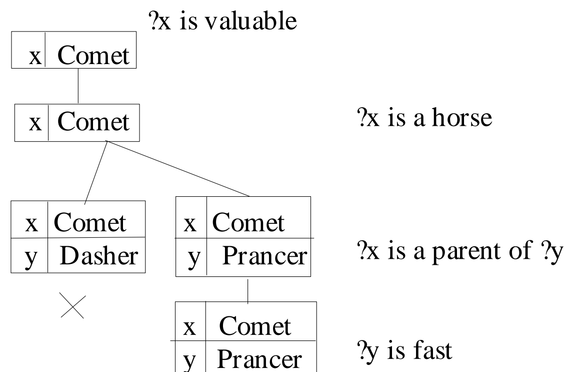


Diagram (1): Consider **backward** chaining. Hypothesis: Comet is valuable. Use an influence net. **Diagram (2): Backward** Chaining Arranged as a *Tree*.





Consider that we add *another rule:* **W$_1$**: If ?w is a winner then ?w is fast. We also add the facts *"Dasher is a winner" and "Prancer is a winner"*. How does this change the list of valuable horses?

➢ 26th November 1999

The exam will cover material from the first **eight** chapters of the book. Remember to read up on the *expert* systems. They (1) Gather information; (2) Use backward chaining to test hypotheses; (3) Rank Hypotheses based on probabilities and then make a decision. The two systems in the book are **Mycin** and **XCon**.

Variation

Variation: change "*If ?x is a horse, ?x is a parent of ?y, ?y is fast, ?x is fertile, then ?x is valuable*" to "*If ?x is a horse, ?x is a parent of ?y, ?y is fast, then ?x is valuable provided ?x is fertile*". **Show Me Mode:** Treat providing assumptions as ordinary antecedents. **Ask Questions Later Mode:** Ignore all "*providing*" assumptions.

We could also change: *"If ?x is a horse, ?x is a parent of ?y, ?y is fast, ?x is alive then ?x is valuable"* to "*If ?x is a horse, ?x is a parent of ?y, ?y is fast then ?x is valuable unless ?x is dead"*. **Show Me Mode:** Treat unless assumptions as **ordinary** antecedents (but use Negation). **Ask Questions Later Mode:** *Ignore* unless assumptions.

Decision maker mode: assume all *providing* assumptions are *true* and the *unless* assumptions are *false,* unless there is **direct** evidence in the working memory. **Trusting-Skeptic mode:** Assume that the *providing* assumptions are true and that the *unless* assumptions are *false*, **unless** we can show otherwise with a single rule.

Progressive Reliability mode: Work in *"Ask questions later"* mode and as time permits, then explore more and more of the providing and unless assumptions. This is like *progressive* (iterative) *deepening*. An expert system will typically be *limited* to some domain of expertise. It is the task of the **knowledge engineer** to try to formalise the information into a set of rules.

➢ 3rd December 1999

Summary of Course

1. **Representations of Networks**

   Semantic Networks

   Good representations are the key to good problem solving e.g. farmer, fox, goose

   Describe and Match method for identification of a feature based object identification e.g. electrical box covers

   Describe & Match for analogy problems

   Algorithms for inside/outside, for left of/above e.g. trial analogy problems, ...

2. **Problem Solving**

   Generate and test

   Good generators are complete, non-redundant and informed

   Means ends analysis

   Step towards the goal, evaluating a difference

   Often implemented using a different procedure table e.g. deciding on mode of transport

   Problem Reduction Method, e.g. moving blocks

   Goal trees: OR tree, AND tree

   Introspection with a goal tree. How questions, Why questions

3. **Basic Searches**

   Blind Searches

   Search tree — discard paths revisiting nodes e.g. S → G; paths between cities on a map

   Root node, leaf nodes, parent nodes, child nodes, branching factor

   Depth / Breadth first search (described by queues)

   Heuristic Searches (informed)

   Hill climbing — when can you get stuck?

   Variations of depth first — iterative deepening (should be in blind?)

   Beam Search (Breadth first with a limited beam width)

   Non deterministic searches

   Best first search (should be in blind?)

4. **Optimal search**

   Best Path

   British Museum procedure

   Branch-and-bound — queue description

   Added underestimates for efficiency

   Redundant paths

   Discard redundant paths

   Dynamic programming principle

   A* procedure

   Example: a configuration space to move objects in 2-D

5. **Trees & Adversarial Search**

Algorithmic Methods

Nodes represent board positions, links are individual moves

Exhaustive search typically impossible e.g. chess

Minimax procedure; Alpha - Beta pruning: descriptions of implementations involve recursive algorithms

Heuristic Methods

Progressive Deepening

6. **Rules and Rule Chaining**

Rule based deduction systems, e.g. identifying animals

Working memory, rule base

Forward & Backward chaining and procedures for their implementation (depth first), e.g. a valuable horse

Inference Nets

Which to Choose — Forward vs. Backward (fan-in, fan-out, fleeting glimpse,...)

Examples: Mycin, Xcon

7. **Rules & Reasoning**

Introspection using inference nets

Relaxing *modes* of reasoning (show me mode, ask questions later mode, decision maker mode, trusting sceptic mode, progressive reliability mode)

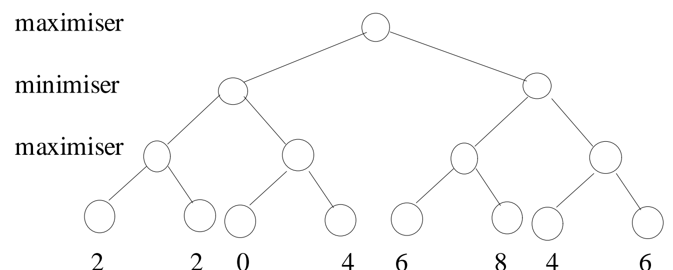
A mention of *probabilities* and reasoning

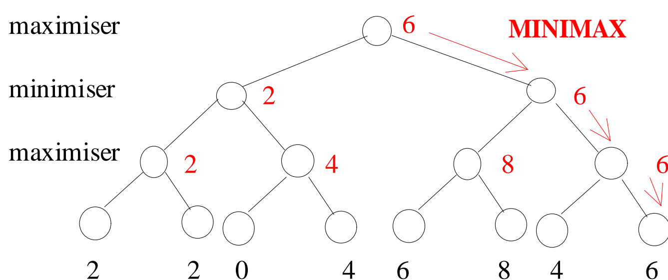
Heuristics of knowledge engineering

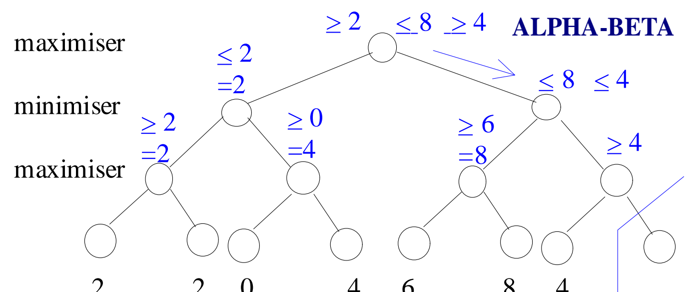
Ask lots of questions watching a domain expert. Ask about apparently indistinguishable situations — treated differently?

Tutorial

For the tree shown, use **minimax** to score each node. Which move is optimal? Then explore the tree with **alpha-beta pruning**. Indicate all parts of the tree that are cut off. Indicate the winning path(s).







Exam Paper: January 2000

Attempt TWO questions from four.

(1) For **the problem** described below:

(a) By means of a tree diagram with explanatory notes, describe the form of the **search space** for the problem. (You are not expected to describe the whole search space).
[9 marks]

(b) Describe how the problem can be solved using:
(i) **Depth-first** search
(ii) **Breadth-first** search. **[10 marks]**

(c) Briefly describe the advantages and disadvantages of these two search methods.
[6 marks]

The Problem:

"A hungry monkey finds himself in a room in which a bunch of bananas is hanging from the ceiling. The monkey, unfortunately, cannot reach the bananas. However, in the room there are also a stick and a chair. The ceiling is just the right height so that a monkey standing on a chair could knock the bananas down with the stick. The monkey knows how to move around (which may be denoted as 'M'; this includes moving to where the stick, chair or bananas are and it includes climbing on the chair), carry things around ('C'), reach for the bananas ('R'), and wave a stick in the air ('W'). How can the monkey get the bananas?"

(2) (a) Identify the main components of a typical expert system. **Briefly** explain the purpose and function of each component. **[5 marks]**

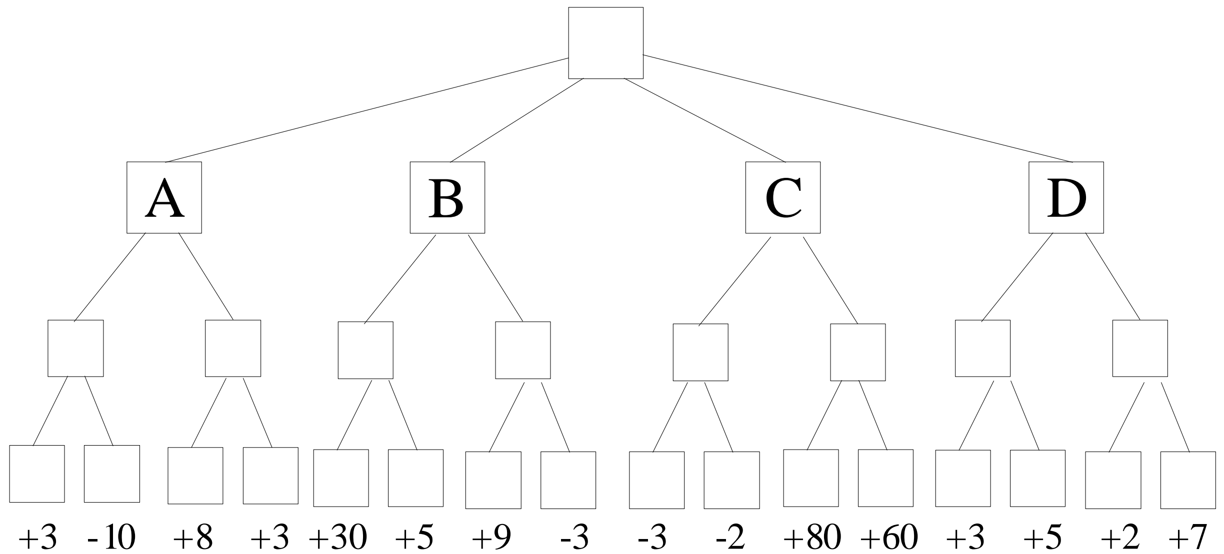
(b) What do you understand by the term 'antecedent-consequent rule'? Give one or more examples of this kind of rule. **[5 marks]**

(c) Using examples, explain the meaning of the terms 'forward chaining' and 'backward chaining'. **[7 marks]**

(d) Describe any ONE application of an expert system and assess the strengths and weaknesses of expert systems for this problem. **[8 marks]**

(3) (a) For a competitive game between two players describe how to use
(i) the minimax search procedure
(ii) minimax with alpha-beta pruning
to find the 'best' move amongst several alternatives. **[8 marks]**

(b) Consider the following game tree:



Use minimax to find the score of the top node.
[It is your turn to move].
What move is optimal, A, B, C or D? **[6 marks]**

(c) Repeat this search with alpha-beta pruning.
Show which leaf nodes of the game tree are **not** evaluated?
Would you still pick the same move as in part (b)? **[11 marks]**

(4) (a) Suppose that you have just installed a new carpet in your bedroom. You discover that the bedroom door will not open until you trim between $^1/_4$ and $^1/_{16}$ inch off the bottom. Construct a **possible** difference-procedure table relating a saw, a plane, and a file for the precise amount of wood you need to trim off. **[5 marks]**

(b) You have learned that rules with "unless" and "providing" assumptions are similar to ordinary antecedent-consequent rules.

(i) Recast the following as ordinary antecedent-consequent rules:

Fly Rule:

If	?x	is_a_bird
then	?x	files
unless	?x	is_a_penguin
providing	?x	is_alive

Bird Rule:

If	?x	flies
then	?x	is_a_bird
unless	?x	is_a_bat
providing	?x	is_alive

[12 marks]

(ii) Explain how the given rules differ from the recast rules from the perspective of a reasoning system operating in ask-questions-later mode. **[8 marks]**

<u>(Questions done: 1, 3)</u>