Engineering Part IIA and EIST Part I 1998

Paper E6 (Computing Systems) Solutions

RC 6/98

II A + EIST I
Paper E6 1998

21). Computer Architecture

. MIMD parallel processors

(a) A MIMD parallel architecture comprises a number of processors, each of which fetches its own instructions and operates on its own data. The processors are often off-the-shelf microprocessors. MIMD stands for *multiple instruction streams, multiple* (2) *data streams.*

(b) A small number of processors can be connected together as in Machine A. Since each processor has its own cache, the single bus and memory system can serve the needs of all the processors, as long as there are not too many of them (up to a few tens of processors).

With more processors, the single bus and memory system becomes a bottleneck, and the need for a physically longer bus also reduces the bus' bandwidth and increases its latency. So architectures like Machine B tend to be used for larger MIMD machines. The memory is distributed amongst the nodes, so local processor-memory traffic can proceed at a high rate, independent of the number of processors. Inter-processor (3) communication, however, is over a network and slower than in a single bus design.

(c) Machine B can use either shared memory or message passing. Just because the memory is distributed doesn't mean it cannot be shared: the physically separate memories can be addressed as one unified address space. Alternatively, each processor's memory can be completely private and inaccessible to remote processors: the (2) processors will then have to communicate by message passing.

(d) In a shared memory MIMD machine, multiple caches can simultaneously hold copies of the same data. It is important to ensure that the multiple copies are consistent, or, in other words, that the caches are coherent. This is the essence of cache coherency.

Bus snooping caches monitor the bus and react to reads and writes on other caches. Coherency can be enforced in two ways:

Write-invalidate: a writing processor broadcasts an invalidate signal to all other caches. If they have a copy of the same block, they invalidate it immediately.

Write-update: a writing processor broadcasts the new data over the bus. If other (4) caches have a copy of the same block, they update it immediately.

(e) This is a straightforward application of Amdahl's Law. We are enhancing a fraction of the computation (the part that will run in parallel mode): this fraction of the application will now go 10 times faster. Amdahl's Law says:

$$t_{\text{new}} = t_{\text{old}} \left((1 - \text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}} \right)$$

$$\Rightarrow \text{speedup}_{\text{program}} = \frac{t_{\text{old}}}{t_{\text{new}}} = \frac{1}{(1 - \text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{\text{speedup}_{\text{enhanced}}}}$$

Substituting $\text{speedup}_{\text{program}} = 8$ and $\text{speedup}_{\text{enhanced}} = 10$ gives

$$\begin{aligned} 8 &= \frac{1}{(1 - \text{fraction}_{\text{enhanced}}) + \frac{\text{fraction}_{\text{enhanced}}}{10}} \\ \Leftrightarrow 0.125 &= (1 - \text{fraction}_{\text{enhanced}}) + 0.1 \times \text{fraction}_{\text{enhanced}} \\ \Leftrightarrow 0.9 \times \text{fraction}_{\text{enhanced}} &= 0.875 \\ \Leftrightarrow \text{fraction}_{\text{enhanced}} &= 0.972 \end{aligned}$$

So less than 2.8% of the computation can be serial. The chance of achieving this depends on the nature of the application itself. It would be near impossible to take a typical, serial application and convert it into a parallel processing program with 97.2% parallelism. On the other hand, if our application involved running 10 or more independent processes, then we could achieve this sort of parallelism.

(5)

(f) We can only expect a 10 times speed up if each of the processors runs as fast in the MIMD machine as it did in the uniprocessor machine, without increased stalling. Unfortunately this is not likely to be the case. The coherence protocol will significantly increase the cache miss penalty, resulting in more CPU stall cycles. Also, the MIMD machine's processor-memory bus is likely to have an increased latency, since it has more potential bus masters (the processors) requiring more arbitration. The processors will need to stall while waiting for control of the bus. Another factor we have not considered is I/O: if the 10 processes require significant I/O activity, they may saturate the I/O channels when running in parallel, limiting any potential speedup.

(4)

Q1. 2. Virtual memory

(a) There are two main requirements that motivate the adoption of a virtual memory system. The first is the desire to be able to write programs without having to worry about the amount of physical memory installed in the computer. The second is the need for the CPU to execute multiple processes separately: each process should be unaware of, and protected from, the others.

(2)

(Key points: multiple processes and process protection, ability to write programs without regard for physical memory size.)

(b) A virtual memory system satisfies these requirements by allowing programs to use virtual memory addresses. These are then translated into physical addresses at run time.

Virtual memory and main memory are both divided into chunks called pages: the translation is between a virtual page number and a physical page number. The translations are stored in a structure called a page table, which is indexed by the virtual page number and contains the physical page number.

If a program accesses more virtual memory than can be fitted into the computer's physical memory, then the excess pages are stored on disk. When these pages are required they are fetched into main memory, overwriting another page which is stored back on disk. Hence a program can be designed and implemented without regard for the amount of physical memory in the machine.

Separate processes running simultaneously on the same CPU will have separate page tables. These are managed by the operating system to ensure that independent virtual pages map to disjoint physical pages. Since a user process is not allowed to modify its own page table, it follows that one process cannot access another process' data: this is the essence of process protection.

(Key points: division of memory into pages, virtual page numbers translated into physical page numbers, translations stored in page table, disk acts as backing store for pages that won't fit into main memory, one page table per process, OS manages page tables, user process not allowed to modify page tables.)

(3)

(c) The need to look up address translations in the page table (which is large and therefore stored in main memory) undermines the existence of the cache. So the CPU typically contains a small, fast cache called the translation lookaside buffer which caches recently used page table entries. Locality of reference to the page table means that the TLB miss rate is very low.

(Key points: page table lookup undermines cache, TLB caches page table entries, TLB lookup is very fast.)

(2)

(d) The cost of a disk access is enormous (typically millions of clock cycles). The virtual memory system is designed to minimize page faults by allowing fully associative placement of pages in main memory, with a least recently used (LRU) replacement strategy (implemented in software as part of the operating system). In addition, write back is used to replace a page on disk, and the page table includes a "dirty" bit which is set the first time the page is modified in main memory. If a page is not marked "dirty" then it does not have to be written back to disk when it is replaced.

(4)

(Key points: enormous cost of disk access, fully associative page placement, LRU replacement strategy implemented in software, write back, "dirty" bit prevents unnecessary write backs.)

(e) A large page size results in (i) a smaller page table, (ii) more efficient use of disk transfers, since most of the disk transfer time is overhead (seek time, rotational latency, controller overhead), so a 32K page can be fetched from disk in much less than twice the time it takes to fetch a 16K page, and (iii) fewer TLB misses, since more memory is mapped through a fixed number of TLB entries. All these factors motivate the selection of a large page size.

(3)

(Key points: smaller page tables, more efficient disk transfers, fewer TLB misses.)

One factor which suggests a smaller page size is *internal fragmentation.* This is the term given to the wasted memory which is an inevitable consequence of a fixed page size. Since a process is forced to monopolise an integral number of pages, on average half a page will not be needed by the process and therefore wasted[1].

(Key points: amount of wasted memory increases with page size.)

(2)

When the CPU switches from executing process A to executing process B, the TLB will contain some of A's page table entries. To maintain process protection, B must not be allowed to use these translations. There are two ways to handle this. The first is to reset all the TLB's valid bits on a process switch, so that all entries are invalidated. The second is to extend the TLB entries with a unique process identifier. The first scheme is the easier to implement, though the second will result in better overall performance. To see this, consider two processes running concurrently on a CPU. If the processes are sufficiently small, the TLB could hold simultaneously all the translations required by both process A and process B. However, the invalidating strategy will flush the TLB every process switch, and the translations will have to be re-fetched from the page table in main memory. This cost will not be incurred by the second scheme.

(Key points: invalidate all TLB entries on process switch, extend TLB with process identifier, first scheme simpler, second more efficient since it prevents unnecessary re-fetching of translations from the page tables.)

(4)

[1] In fact, processes typically use three separate memory segments, the *text*, *heap* and *stack*. The average amount of wasted memory is therefore 1.5 pages per process. This detail is included here for completeness: candidates are not expected to know this.

Question 3

a) An abstract datatype exhibits abstraction in the sense that only those parts of the datatype which need to be visible to the rest of the software system are visible; detail which need not be visible is hidden. In most cases the visible interface is provided via a set of procedures, functions and datatypes whose definition is visible but whose detailed implementation is hidden. The major benefit of this is that changes to the detailed implementation can be made without affecting the rest of the system.

This decomposition of a large system into small self-contained manageable components with a clearly defined interface specification is valuable throughout the software lifecycle. The initial specification of a system can frequently be in terms of a few top level abstract datatypes. The use of a common representation in the specification, design and implementation removes the need for translation and consequently the amount of work and risk of errors. The more detailed design and implementation can be based on stepwise refinement of the top level datatypes. The existence of rigorous interface specifications for the datatypes ensures that the individual components can be integrated.

The high level of isolation between abstract datatypes aids the splitting of the task between a team of software engineers throughout the design, implementation and testing. It is also helpful in reducing the impact of changes once the system is running and hence of maintenance costs.

b) Pascal Module Illustration

```
MODULE cardpack;

EXPORT

PROCEDURE initialise;
FUNCTION put_card (c : card) : boolean;
FUNCTION get_card (VAR c : card) : boolean;
PROCEDURE shuffle;

IMPLEMENT

TYPE pack = RECORD
                cards : ARRAY [1..52] OF card;
                num : integer;
              END;
VAR thepack : pack;

PROCEDURE initialise;
BEGIN
  VAR i : integer;
      s : suit;
      v : value;

  i := 1;
  FOR s := spade TO club DO
    FOR v := two TO ace DO BEGIN
```

```
                thepack.cards[i].s = s;
                thepack.cards[i].v = v;
                i := i + 1
        END
    thepack.num = 52;
END;

FUNCTION put_card (c : card) : boolean;
BEGIN
    IF thepack.num < 52 THEN BEGIN
        thepack.cards[thepack.num] := c;
            thepack.num := thepack.num + 1;
            put_card := true
    END
    ELSE put_card := false
END;

FUNCTION get_card (VAR c: card) boolean;
BEGIN
    VAR i : integer;
    IF thepack.num > 0 THEN BEGIN
            c := thepack.cards[1];
            thepack.num := thepack.num - 1;
            FOR i := 1 TO thepack.num DO
          thepack.cards[i] := thepack.cards[i+1]
        get_card := true
    END
    ELSE get_card := false
END;

PROCEDURE shuffle;
BEGIN
    VAR i : integer;
        j : integer;
            c : card;

    FOR i := 1 TO thepack.num DO BEGIN
        j := (random MOD thepack.num) + 1;
        c := thepack.cards[i];
            thepack.cards[i] := thepack.cards[j];
        thepack.cards[j] := c
    END
END;

END.
```

c) In general, object oriented methods provide a better basis for abstract datatypes than that provided by modules. Much of the improvement stems from there being a package per data item rather than for the type as a whole. This is perhaps most obviously valuable in the case of automatic initialisation: a module can only provide this once for the type as a whole; an object oriented scheme can provide it per object. Additionally, object oriented languages usually provide inheritance mechanisms allowing a base type to be customised to a derived type which shares the data structure (possibly with additional members) and can redefine the methods (the procedures and functions). Since a derived type can be compiled separately from the base type, this further helps modularisation and later modification of a system.

In this example, the module supports only a single pack. More packs could be supported by including the pack as an argument to each procedure and function and making the existence (though not the structure) of the datatype visible. However, any subsequent changes to the data structure would then require recompilation of other modules declaring objects of this type (to account for changes in the storage required).

Partial packs for games requiring fewer than 52 cards could be implemented using the module by suitable calls to get_card (and put_card to return those not to be removed). This is however messy and does not help the clarity of the program. The use of a derived type in which the initialisation routine is redefined would provide a cleaner clearer solution.

Similarly alternative shuffling strategies could be supported using put_card and get_card but redefinition of the method in a derived type is a much better solution.

Another problem exhibited in the above is that of error handling in put_card and get_card. The solution used, that of returning false in the case of an error, tends to lead to poor program structure. Most object oriented languages (and some others) also provide an exception mechanism which allows the error to be communicated back immmediately to the part of the program which can handle it.

Question 4

a) If more than one process concurrently attempts to modify the same data, the result may not reflect what is intended unless it can be guaranteed that the operation for one completes before the other starts. As an example of what can go wrong, consider the case of two processes A and B both trying to increment an integer variable i stored in memory. Given the following sequence:

A reads i
B reads i
A increments its in-CPU copy of i
B increments its in-CPU copy of i
A stores i
B stores i

the result is that i is incremented by one not two.

Correct operation can be ensured by making the critical operation (eg incrementing i) indivisible (ie once started it completes before anything else can start). Special mechanisms like semaphores and monitors can be placed around critical sections of code to ensure this. For reliable operation they must ensure:

- Mutual exclusion - ie only one process at a time can execute the code in the critical region.
- No deadlock - If one or more processes is trying to enter the critical region, at least one must succeed.
- No starvation - each process trying to enter the critical region must eventually succeed.
- In the absence of contention, the single process must succeed, ideally with minimal delay
- Immunity to failure - the failure of one process outside the critical region should not jeopardise the others.

Mechanisms like semaphores and monitors interact with the scheduler in the operating system since a process should be suspended while waiting to enter the critical region. The state of the critical region (whether or not in use) must be indicated by a variable which can be tested and set in a single, indivisible in hardware terms, operation.

b) *putinlist* and *getfromlist* will not operate reliably if two processes access the same list concurrently since all four pointers must be updated before the list is again in a consistent state.

For example, consider putinlist and two concurrent processes A and B attempting to insert itemA and itemB respectively. If process A executes item^.next := head^.prev^.next and then B executes the same statement before A executes the next statement, both itemA and itemB will appear to be followed by the same next item in the list. Similarly problems occur if the other statements are interleaved between A and B.

To avoid this a semaphore (preferably per queue to avoid unecessary exclusion between processes accessing different lists) must be added. This would most naturally be added to the listhead type, eg

```
TYPE listhead = RECORD
                    p : listptr;
                    sem : semaphore;
                END;
```

and the routines become

```
PROCEDURE putinlist(head:listhead; item:listptr);
BEGIN
    P(head.sem);
    item^.next := head.p^.prev^.next;
    head.p^.prev^.next := item;
    item^.prev := head.p^.prev;
    head.p^prev := item;
    V(head.sem)
END;

FUNCTION getfromlist(head:listhead) : listptr;
BEGIN
    VAR  p : listptr;
    P(head.sem);
    IF listempty(head) THEN
            p := NIL
    ELSE BEGIN
            p := head.p^.next;
            head.p^.next := head.p^.next^.next;
            head.p^.next^.prev := p^.prev;
            p^.next = NIL; p^.prev = NIL
    END;
    V(head.sem);
    getfromlist := p
END;
```

Note that the in getfromlist, the mutual exclusion must include the test for the list being empty to avoid the possibility of two processes both trying to extract the last item.

Monitors and semaphores perform a very similar function, the main difference being that monitors provide mutual exclusion within a program element much like a module and thus avoid the possibility of programming errors causing unmatched P and V operations. In this case there is little to be gained from doing this as the code using the semaphores is simple and written once. It would be hard to provide monitor condition variables (the equivalent of the semaphore variable) on a per list basis so some functionality would probably be lost.

ii) There are a number of other problems with the code as it stands:

a) Unless the data from the IO devices is always ready, there is a serious risk of deadlock with proc1 waiting for list2 to fill and proc2 waiting for list1. Conversely if data is always available from the devices, the lists will never be emptied. A more sophisticated mechanism for waiting for data from the list OR the device is required - somthing along the lines of the SELECT mechanism used with interprocess communication channels would be suitable.

b) It is possible that proc1 will try to use list2 before proc2 has initialised it (or proc2 use proc1). The obvious solution here is for the initialisations to be done before either process is started.

c) No attempt is made to limit the amount of data stored in either list. In a practical system, this could result in there being insufficient memory.

Q5a). i) $p(\omega_A)$: prior probability; the frequency of adverse outcomes one would expect for this problem

$$\simeq \frac{N_A}{N_A + N_B}$$

ii) $p(x|\omega_A)$: p.d.f. of the feature variable (x) amongst the population having an adverse outcome.

- Model as a parametric form e.g. Gaussian and estimate parameters from a set of data
- alternative is to use a non-parametric form, e.g. histogram to represent the distributions, $p(x|\omega_A)$.

(Require a collection of training data to fit a model and also a validation set (unseen data).

iii). $p(x)$, the denominator, is a normalising constant

$$p(x) = p(\omega_A)\, p(x|\omega_A) + p(\omega_B)\, p(x|\omega_B)$$

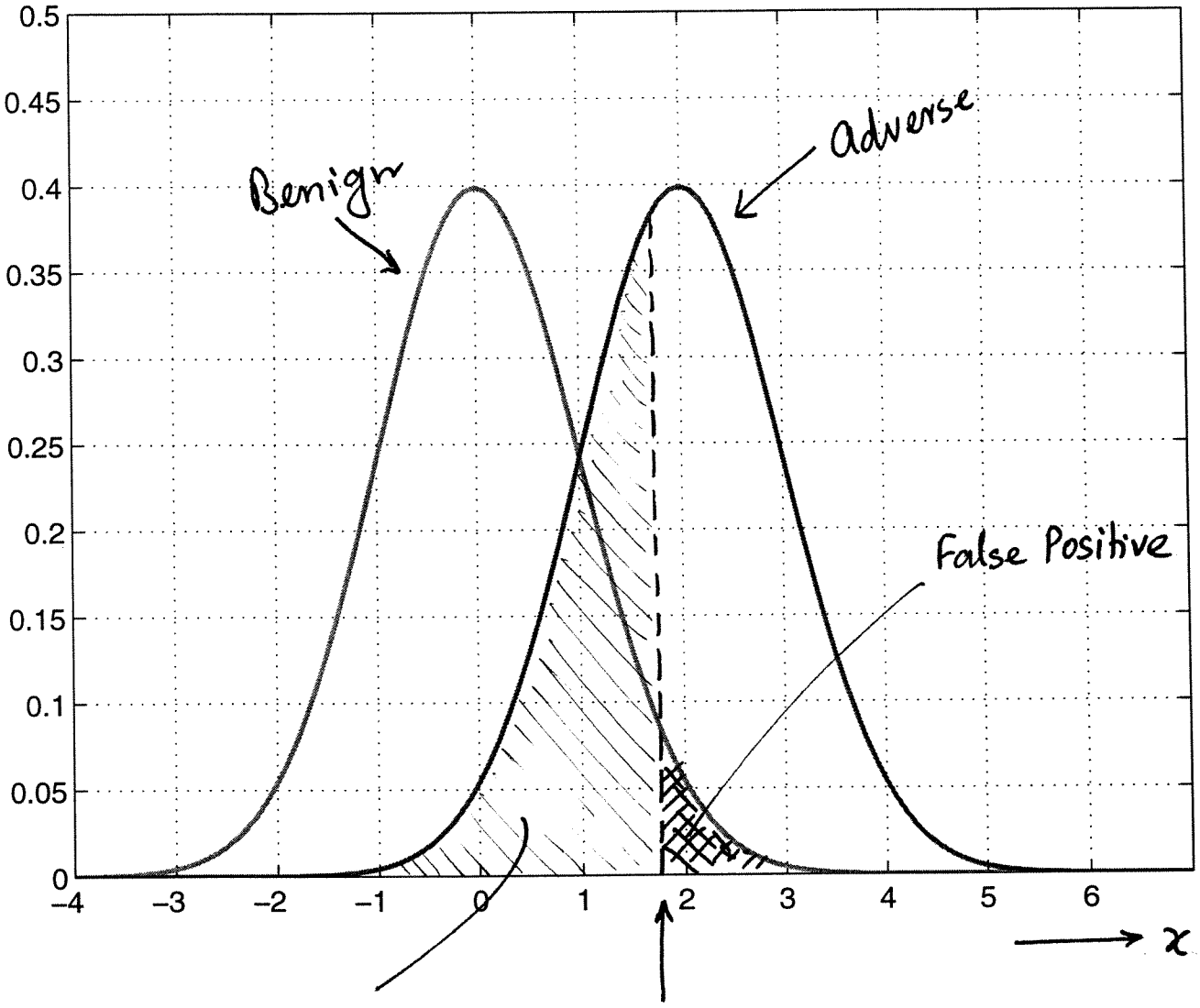
(6) If we are only interested in comparing posterior probabilities $p(\omega_A|x)$ and $p(\omega_B|x)$ then we do not need to compute $p(x)$.

b). Two types of misclassification error: "True Negative" and "False Positive."
(see figure)

$$P(\text{error}) = P(\omega_A)\int_0^\infty \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}\,dx + P(\omega_B)\int_{-\infty}^0 \frac{1}{\sqrt{2\pi}} e^{-\frac{(x-2)^2}{2}}\,dx$$

In example $P(\omega_A) = P(\omega_B) = \frac{1}{2}$

$$P(\text{error}) = \frac{1}{2}\Phi(0) + \frac{1}{2}\left[1 - \Phi(0-2)\right] \quad \text{where } \Phi(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi}} e^{-\frac{t^2}{2}}\,dt$$



True Negative given by this area.

Say θ is set to this value

Q5bii). Misclassification error is minimized by $x = \theta$ such that:

$$p(\omega_A)\, p(x/\omega_A) = p(\omega_B)\left(p(x|\omega_B)\right)$$

where $\frac{dP_{error}}{d\theta} = 0$

$$\therefore \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{(\theta-2)^2}{2}\right\} \times \underbrace{0.1}_{p(\omega_A)} = \frac{1}{\sqrt{2\pi}} \exp\left\{-\frac{\theta^2}{2}\right\} \cdot \underbrace{0.9}_{p(\omega_B)}$$

$$-\frac{(\theta-2)^2}{2} + \frac{\theta^2}{2} = \log_e 9$$

$$\frac{4\theta - 4}{2} = \log_e 9$$

$$\underline{\theta = 1 + \log_e 3 = 2.10}$$

5b) In a practical setting the cost of misclassification for the 2 types of error may differ. If this is known at the time of designing the classifier, one can set θ to minimize the expected loss (ie. costs weighted by posterior probabilities).

Q6 a) k NN rule is a non parametric pattern classification method. It computes k nearest neighbours to the test pattern. Amongst these neighbouring patterns, we find which class label is represented most and assign the test pattern to that class.

Say in a small volume V in feature space there are n_i examples from each of the C classes. $\sum_{i=1}^{C} n_i = n$.

Say k_i examples of the k examples nearest to the test pattern belong to class i.

From a histogram type argument

$$p(\underline{x}) = \frac{k/n}{V}$$

$$p(x \mid \text{class } i) = \frac{k_i/n_i}{V}$$

a suitable prior is n_i/n

Bayes' rule gives

Posterior probability $P(\omega_i \mid \underline{x}) = \dfrac{p(x \mid \omega_i) \cdot P(\omega_i)}{p(\underline{x})}$

$$= \frac{k_i/n_i V \cdot n_i/n}{k/nV} = \underline{\underline{k_i/k}}$$

Largest representation $\Longrightarrow$ maximum posterior probabili

Linear discriminant analysis is a parametric method in which one only needs to store as many parameters as the input dimension. (for a two class problem). Computing a linear discriminant function is very easy, at test.

Nearest neighbour needs no training computation (LDA needs solution of a least squares problem) but at test stage one has to compute a large number of distances, and store all the training patterns. at high dimensions (such as in character recognition tasks) this difference can be very large.

b) Collect a set of labelled data in the form

$$\{x_i, f_i\}_{i=1}^{N}$$ N data points.

↑ Some real values to represent class labels.

e.g. $f_i = \begin{cases} 0 & \text{class } 1 \\ 1 & \text{" } 2. \end{cases}$

Minimise the total squared error over this training data

$$E = \sum_{i=1}^{N} \{f_i - g(x_i)\}^2$$

$$= \sum_{i=1}^{N} \{f_i - [w_0 + w_1 x + w_2 x^2]\}^2$$

differentiate and set to zero leading to a least squares solution.

Or in matrix form we get

$$E = |Y \underline{w} - \underline{f}|^2$$

Where Y = $N \times 3$ matrix, rows of which are $[1 \quad x_i \quad x_i^2]$

$$\underline{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$$

$\underline{f}$ = $N \times 1$ vector of f_i values.

The least squares solution to this is

$$\underline{w} = (Y^T Y)^{-1} Y^T \underline{f}$$

[This derivation is seen in the lectures in the context of a linear classifier, but connecting it to polynomial (quadratic) is new]

Question 7

a) This is a slightly open-ended question and credit should be given for plausible or insightful answers.

From the course, three ways in which people solve problems are:

i) Search.
ii) Analogy.
iii) Knowledge gained from past experience.

b)

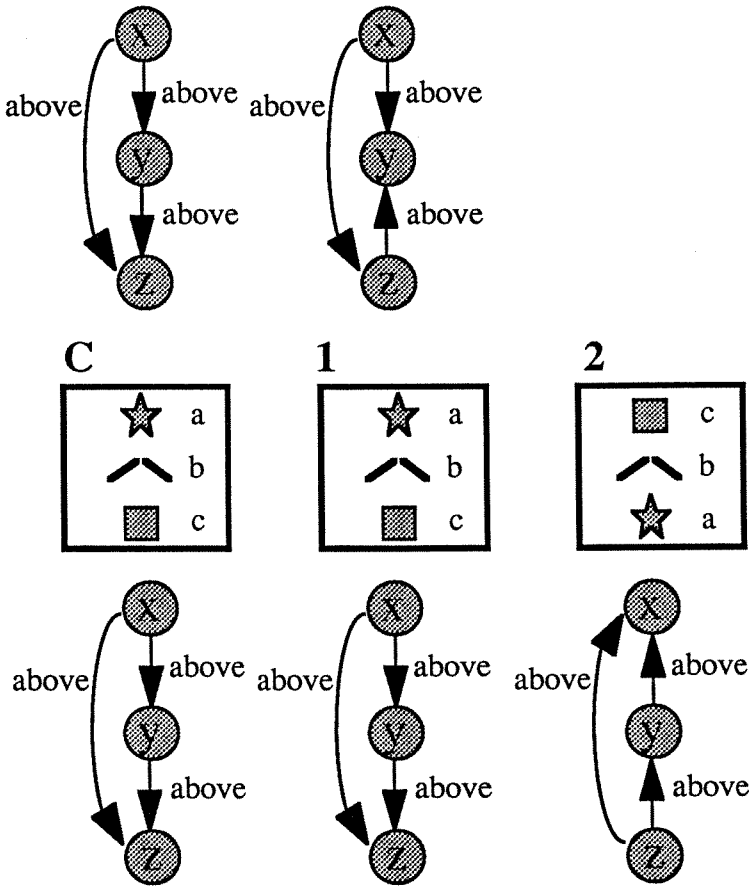
i) See the figure.

ii) The computer uses the describe and match method to solve the analogy problem. Firstly, a semantic net is constructed that describes the features and their relationships. Secondly, rules are developed to show how A becomes B etc. Thirdly, a match is performed between these rules and those between C and the answers.

c) In the example, the items are related in the following way: x -> a, y -> b, z -> c. The A to B and C to 1 rules have more elements in common than the C to 2 rules. Hence the computer picks answer 1. The difference between what the person and computer may be explained by realising that a person concludes that A relates to B and C to 2 due to the swapping of the square and the star. The computer has probably not developed the idea of swapping. People have greater knowledge of the behaviour of shapes and can bring this to bear on the problem.

d) The problem is an example of problem solving by analogy and, as with many AI problems, the key to its successful solution lies in the choice of a pertinent representation of the key problem features. The question has focused only on the reasoning stage and not the earlier, but very important *feature extraction* phase where the various shapes, their orientation and relationships are ascertained.

The fully automatic extraction of features can require sophisticated methods involving vision and spatial reasoning algorithms - which are often a more difficult problem than the rule-based comparison mechanism shown here. Practically speaking the two issues of speed and discrimination determine how the features are used in particular situations.


above
above
above
X
Y
Z
above
above
above
X
Y
Z
C
a
b
c
1
a
b
c
2
c
b
a
above
above
above
X
Y
Z
above
above
above
X
Y
Z
above
above
above
X
Y
Z

Question 8

a) The four criteria are:

i) Completeness: is the strategy guaranteed to find a solution?
ii) Time complexity: how long does finding a solution take?
iii) Space complexity: how much memory is required?
iv) Optimality: will the best solution be found?

Give credit for plausible similar criteria.

b) The attached figure shows the network in the question re-written as a search tree. The line shows one way of undertaking depth-first exhaustive search.

Other (e.g. right-to-left) forms of depth-first search are also acceptable.

c) The tree above shows that, for this simple network, 17 nodes have to be searched. If each node involves 0.1 seconds calculation time, then a complete route calculation takes 1.7 seconds.

Given that even the longest route takes only 320 milliseconds to transmit a 1KB message, a better solution would simply be to send the message down any route and not waste time calculating the optimal route - this is not a good way to solve this problem!

What would need to change to make this method useful are, for example:

i) Each node would need a much higher store-and-forward time due, possibly, to a significantly increased packet size.
ii) The calculation time would need to decrease significantly.

Give credit for other plausible points - for example, suggesting the use of another more efficient search strategy or discussions about how varying traffic conditions would affect transit times.

d) A* is a combination of branch and bound with dynamic programming and a heuristic cost function.

This technique will, like the exhaustive search earlier, probably find the optimal path but will do so without exhaustively searching every path.

The search algorithm proceeds as follows:

i) The algorithm starts with a usual blind search manner and identifies the cost of reaching the first few nodes in the tree.

ii) When two identical nodes are identified, the one with the higher path cost is, together with its sub-tree, removed from the process (pruned) and no longer considered. This is because it is pointless to get to a node via a path when the same node can be reached by a cheaper route.

In the diagram produced earlier, the two nodes corresponding to 'd' are reached and the path costs determined. The one with the higher path cost of 80 is pruned. The one with the path cost of 70 is left to have its sub-tree considered further. The advantage is that the need to consider four sub-nodes, with the corresponding computational overhead, is removed.

iii) The heuristic part of the A* algorithm works by making an estimate of the remaining path cost between the current node and the goal node. The existence of the information sent back after successful transmission would have the advantage of allowing the routing algorithm to develop knowledge of the state of the network. This knowledge can be used to estimate the likely path cost of sending a packet down a particular route.

The evaluation function for A* is: $f^*(n) = g^*(n) + h^*(n)$ where:

$g^*(n)$ is the estimate of the minimum cost of a path from the start node to a node n,
$h^*(n)$ is the estimate of the minimum cost of a path from node n to a goal node, and
$f^*(n)$ is the estimate of a path passing through node n.

Inevitably, the knowledge about the network is always out of date, and for best functioning of A*, the estimate should be equal to or less than the real value (the admissibility criteria).

This d node and its sub tree gets 'pruned' out by the A* algorithm in favour of the lower-cost node next to it (of cost 7).