Solutions to E6 Computer Systems questions, 2001

1. *Bayes' Minimum Error Classifier*

   (a) The Bayes' minimum error rate classifier simply assigns the test data sample to the class with the highest posterior probability. In this case the prior probabilities are equal so the decision rule is simply find the class ω_j which gives

   $$\max_j p(\omega_j|\boldsymbol{x})$$

   (b) In general

   $$\begin{aligned} P_e &= \int P(\text{error}, \boldsymbol{x}) d\boldsymbol{x} \\ &= \int P(\text{error}|\boldsymbol{x}) p(\boldsymbol{x}) d\boldsymbol{x} \end{aligned}$$

   In the two class case we have two regions defined by the decision rule, decide for class ω_1 in $\mathcal{R}_1$ and ω_2 in $\mathcal{R}_2$ and

   $$\begin{aligned} P_e &= P(\boldsymbol{x} \in \mathcal{R}_2, \omega_1) + P(\boldsymbol{x} \in \mathcal{R}_1, \omega_2) \\ &= P(\boldsymbol{x} \in \mathcal{R}_2|\omega_1)P(\omega_1) + P(\boldsymbol{x} \in \mathcal{R}_1|\omega_2)P(\omega_2) \\ &= \int_{\mathcal{R}_2} p(\boldsymbol{x}|\omega_1)P(\omega_1)d\boldsymbol{x} + \int_{\mathcal{R}_1} p(\boldsymbol{x}|\omega_2)P(\omega_2)d\boldsymbol{x} \end{aligned}$$

   Here $P(\omega_1) = P(\omega_2) = 0.5$ and so

   $$P_e = 0.5 \left[\int_{\mathcal{R}_2} p(\boldsymbol{x}|\omega_1)d\boldsymbol{x} + \int_{\mathcal{R}_1} p(\boldsymbol{x}|\omega_2)d\boldsymbol{x} \right]$$

   (c) The maths required for parts (c) and (d) is unseen and not from lectures.

   (i) Sketch should show two Gaussians with a decision boundary half way between the means $x_b = (\mu_1 + \mu_2)/2$ and the error areas under the Gaussians clearly shown. In the equal prior case here for equal variance the error probabilities for the two parts are equal.

   (ii) So for Gaussians since the two error areas are equal and taking the case $\mu_2 > \mu_1$

   $$P_e = 2 \left(0.5 \int_{x_b}^{\infty} \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x - \mu_1)^2}{2\sigma^2} \right) dx \right)$$

   and by change of variable

$$\nu = \frac{x - \mu_1}{\sigma}$$

$$P_e = \int_{\frac{|\mu_2 - \mu_1|}{2\sigma}}^{\infty} \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{\nu^2}{2}\right) d\nu$$

as required.

(d)

(i) In the multi-dimensional case the decision boundary is a hyperplane. As suggested in the question transform the space so that the hyperplane is orthogonal to one of the axes. The error probability can be found by integrating the pdf over the area Note that since the covariance matrix is diagonal.

Assume for a 2-D case that the new axes are z_1 and z_2. If $\boldsymbol{\mu}_1$ is translated to the origin and $\boldsymbol{\mu}_2$ also lies on the z_1 axis the decision boundary will be parallel to one of the axes. The difference in distance between the means in the rotated axes is preserved at $||\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1||/2$. Also scale the space by σ.

The new probability of error can be expressed as

$$P_e = \int_{z_1=a}^{z_1=\infty} \int_{z_2=-\infty}^{\infty} \ldots p(\boldsymbol{z}|\omega_1) dz_1 dz_2 \ldots dz_d$$

All apart from the first one integrate to 1 and hence we are left with the required result where a is the position of the boundary in the transformed space i.e.

$$a = \frac{||\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1||}{2\sigma}$$

(ii) As new features (scaled for constant variance) are added the value of a always increases unless the means between the added features is zero. This implies that P_e reduces or at worst stays the same as new features are added and tends to zero as the number of independent feature tends to infinity. In practice, there are two main problems. First, estimating more parameters as more features are added; seconding extracting independent features and these problems cause generalisation performance to degrade.

E6 2001

2. *Perceptron Algorithm and Binary Classification*

(a) For the original discriminant $g(\tilde{\boldsymbol{y}})$ it would be > 0 for (e.g.) class ω_1 and < 0 for class ω_2.

If this is to be greater than zero for all vectors then we can achieve this by simply reversing the sign of the training data samples for class ω_2. Hence

$$\begin{aligned} \tilde{\boldsymbol{y}} &= \boldsymbol{y} \qquad \text{if } \boldsymbol{y} \in \omega_1 \\ \tilde{\boldsymbol{y}} &= -\boldsymbol{y} \qquad \text{if } \boldsymbol{y} \in \omega_2 \end{aligned}$$

(b) First step is to transform the data as discussed above. Then initialise the weight vector $\boldsymbol{a}$ to some value (this is arbitrary). Then repeatedly cycle through all the transformed data points $\tilde{\boldsymbol{y}}$ in turn and add the value of the data sample to the estimate of $\boldsymbol{a}$ if that the data point is misclassified: i.e.

$$\boldsymbol{a}^{(k+1)} = \boldsymbol{a}^{(k)} + \rho \tilde{\boldsymbol{y}}^n$$

where $\tilde{\boldsymbol{y}}^n$ is a misclassified example (normally cycle through these, although can go through samples in any random order) and ρ is normally set to 1.

This algorithm is guaranteed to converge if the data is linearly separable. Major limitations are that it will oscillate if the data is not linearly separable and that there are no conditions on which vector is chosen within the solution space. Furthermore there may be methods of updating which find a solution vector in fewer steps than the single-sample update perceptron.

(c) Initialise $\boldsymbol{a} = \boldsymbol{0}$. Find the dot product of each transformed vector in turn with $\boldsymbol{a}^{(0)} = [0 \;\; 0 \;\; 0]'$

$$\begin{aligned}
\boldsymbol{a}^{(0)'}.[1 \;\; -2 \;\; 1]' &= 0 \quad \rightarrow \quad \boldsymbol{a}^{(1)} = [1 \;\; -2 \;\; 1]' \\
\boldsymbol{a}^{(1)'}.[1 \;\; -1 \;\; 0]' &= 3 \quad \rightarrow \quad \text{no update} \\
\boldsymbol{a}^{(1)'}.[-1 \;\; -1 \;\; -1]' &= 0 \quad \rightarrow \quad \boldsymbol{a}^{(2)} = [0 \;\; -3 \;\; 0]' \\
\boldsymbol{a}^{(2)'}.[-1 \;\; 1 \;\; -4]' &= -3 \quad \rightarrow \quad \boldsymbol{a}^{(3)} = [-1 \;\; -2 \;\; -4]' \\
\\
\boldsymbol{a}^{(3)'}.[1 \;\; -2 \;\; 1]' &= -1 \quad \rightarrow \quad \boldsymbol{a}^{(4)} = [0 \;\; -4 \;\; -3]' \\
\boldsymbol{a}^{(4)'}.[1 \;\; -1 \;\; 0]' &= 4 \quad \rightarrow \quad \text{no update} \\
\boldsymbol{a}^{(4)'}.[-1 \;\; -1 \;\; -1]' &= 7 \quad \rightarrow \quad \text{no update} \\
\boldsymbol{a}^{(4)'}.[-1 \;\; 1 \;\; -4]' &= 8 \quad \rightarrow \quad \text{no update} \\
\\
\boldsymbol{a}^{(4)'}.[1 \;\; -2 \;\; 1]' &= 5 \quad \rightarrow \quad \text{no update}
\end{aligned}$$

Since all the vectors are correctly classified then $\boldsymbol{a}^{(4)}$ is a solution.

(d) [Not covered in lectures] In this case at each step the update should be

$$\boldsymbol{a}^{(k+1)} = \boldsymbol{a}^{(k)} + \tilde{\boldsymbol{y}}^n$$

if $\tilde{\boldsymbol{y}}^n.\boldsymbol{a}^{(k)} \leq b$. Assuming that b is positive gives a margin for discrimination. It can be shown that this algorithm is guaranteed to converge in a finite number of steps if the data samples are linearly separable.

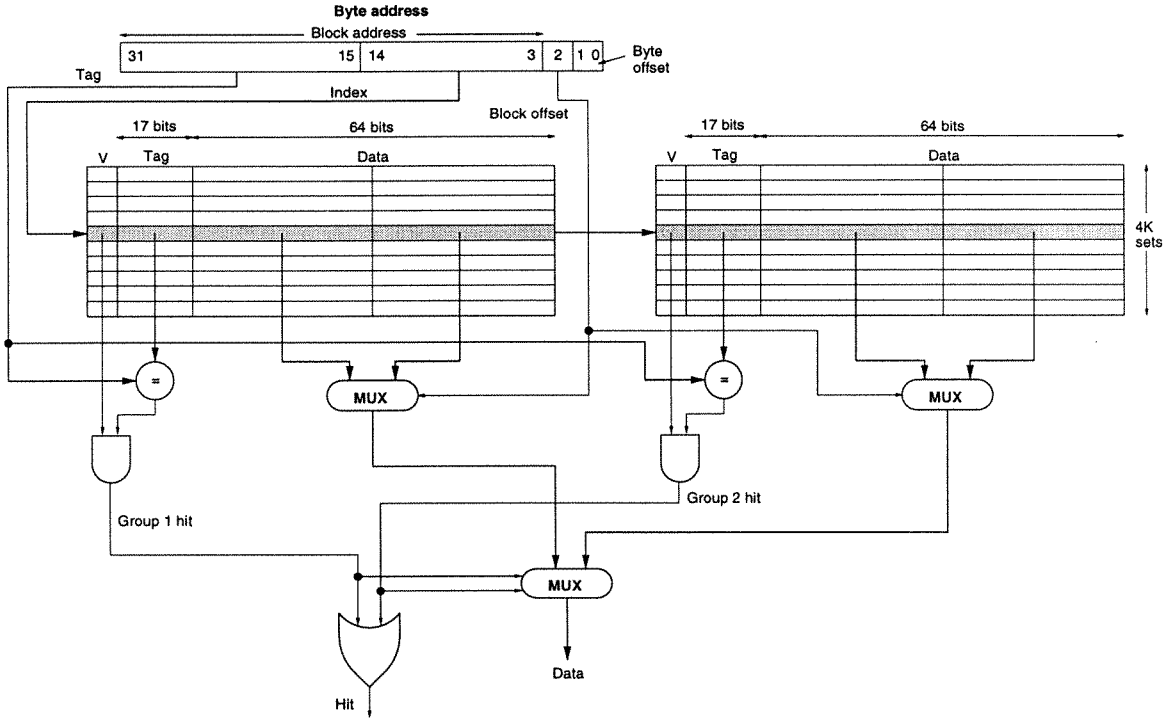
(e) [Note that this is not covered by the lecture material] For multi-class (say N-class) discrimination using perceptron-type 2-class classifiers can have either a set of N 2-class classifiers (1 v. rest) or a set of $N(N-1)/2$ 2-way classifiers. Particular problems of the first case are that they may be several (or no) classifiers which give an output > 0 and if there are several > 0 ties need to be broken. This could be done using the value of the discriminant. In the case of $N(N-1)/2$ classifiers there is a natural way to break ties, but the value of N that can be used is limited by the number of classifiers that would be needed.

E6 2001

3. *Caches and pipeline hazards*

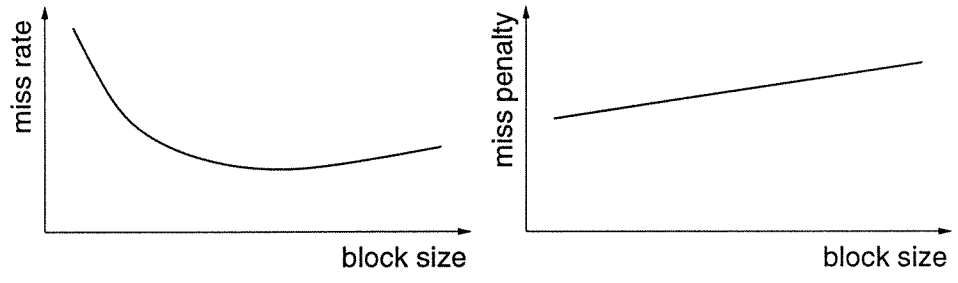
(a) A set-associative cache is a fast memory which holds a copy of selected blocks of a larger, slower, main memory. When the CPU needs to read or write a word in memory, it first checks whether the block containing the word is in the cache. A portion of the memory address is used to index the cache: the index points to a finite number of cache blocks (a *set* of blocks) which all need to be checked for a match. When there is no match (a *cache miss*), the block containing the requested word is fetched from main memory and stored in the cache. Any of the blocks in the indexed set are candidates for replacement. The LRU (least recently used) block replacement strategy takes advantage of temporal locality of reference to reduce the miss rate. Alternatively, the block to be replaced can be selected at random.

(b)

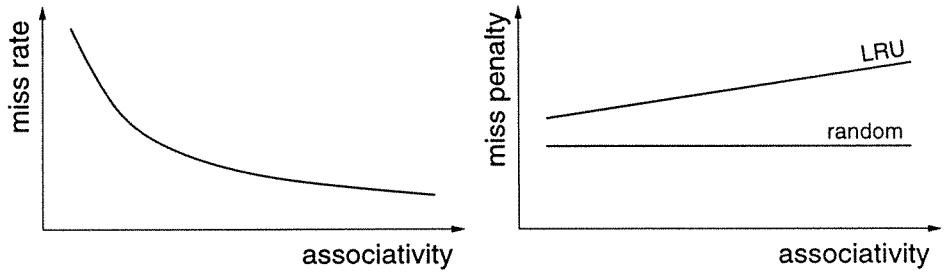


Compared with the direct-mapped case, there is an extra multiplexor at the output (to select between different blocks in each set) which will have a finite propagation delay and therefore increase the hit time.

(c)



Spatial locality of reference means that the miss rate generally decreases with block size, though with very large blocks the miss rate may eventually increase. This is because there will be a small number of blocks in the cache and a great deal of competition for space: a block may be forced out of the cache before many of its words have been accessed. The miss penalty increases with block size, since more words need to be transferred from main memory.



The miss rate decreases with increased associativity, since block replacement strategies like LRU can be used to replace blocks which are unlikely to be needed again soon. Random block replacement results in slightly higher miss rates than LRU. With LRU, the miss penalty usually increases with more associativity, since the LRU algorithm needs to check more access times to decide which block to replace (though this depends on the precise implementation). For random replacement, the miss penalty is likely to be independent of the degree of associativity.

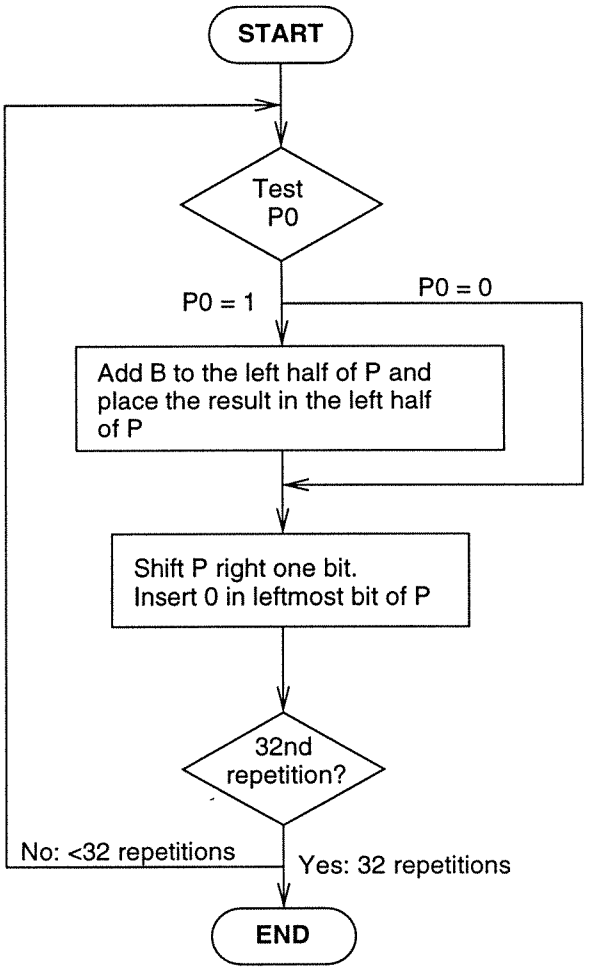
(d) (i) The code, as given in the question, exhibits good spatial locality of reference, since adjacent elements of `x` are checked each time round the inner loop. Switching the order of the loops will destroy this spatial locality of reference and, assuming the block size is greater than one word, increase the cache miss rate.

(ii) The problem with the code given in the question is that there is dependence between adjacent memory access instructions. Each memory read is dependent on the previous write: assuming the compiler unrolls the loops to some extent, this will result in pipeline hazards and consequential stalls, especially with superscalar architectures that issue multiple instructions at the same time. Switching the order of the loops will remove these dependences, allowing the pipe to run with fewer stalls. It is not clear whether switching the loops will improve the execution time or not: this depends on the relative penalties associated with cache misses and pipeline stalls.

E6 2001

4. *Integer multiplication and addition*

(a) Initially, the left half of P should contain zeros.



(b) Consider the 16-bit adders with their two levels of carry-lookahead. At the lower level, propagate and generate signals for the full adders are computed using single gate operations: gi = ai.bi and pi = ai + bi. The low level carry signals are calculated from these using sum-of-product expressions (two gate delays): for example, c2 = g1 + p1.g0 + p1.p0.c0. The full adders produce the sums from the carry and input signals using further sum-of-product expressions.

At the higher level, propagate and generate signals for the 4-bit adders are computed using expressions like P0 = p3.p2.p1.p0 and G0 = g3 + p3.g2 + p3.p2.g1 + p3.p2.p1.g0. Carry signals into each 4-bit adder are calculated using further sum-of-product expressions: for example C2 = G1 + P1.G0 + P1.P0.c0.

A 16-bit adder can therefore produce a sum after 9 gate delays, as shown in the table below.

Gate delay	Signals available	Generated from
0	ai,bi,c0	—
1	gi,pi	ai,bi
3	Pi,Gi	pi,gi
5	Ci	Pi,Gi,c0
7	ci	gi,pi,Ci
9	Sums	ai,bi,ci

Since the two 16-bit adders are connected using ripple carry, the second adder cannot start until the first has finished. The entire 32-bit sum will therefore take $18T$, where T is a typical gate delay.

It is possible to achieve this in $14T$ since the carry to the second adder is available after $5T$ as C4 is available.

(c) When B is negative, the shift-and-add strategy is still valid and the control can still be based on the bits of A as before. The problem is that the sign of the product is corrupted when the zero is shifted in from the left. To correct this, the shift should become an arithmetical right shift, with the sign bit of P being introduced from the left instead of zero.

(d) Taking the identity given in the question, and setting a_{-1} to zero, we obtain

$$\sum_{i=0}^{31} B(a_{i-1} - a_i)2^i \equiv AB$$

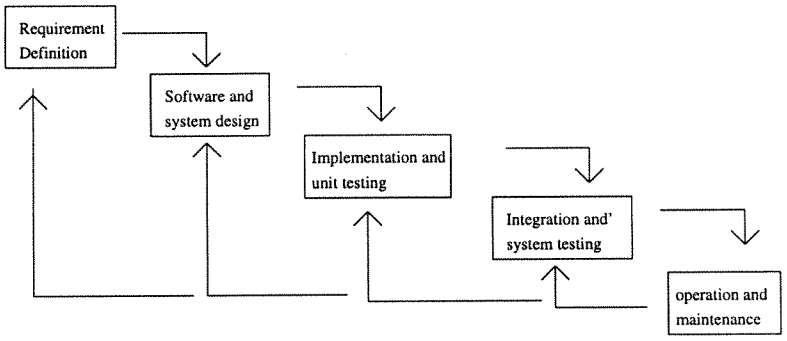
The right hand side is the desired product, the left hand side describes a 32-step shift and accumulate process, where at each step we either add B to the product, subtract B from the product or do nothing, depending on the values of successive *pairs* of bits of A, before shifting the product right:

a_i	a_{i-1}	Action
0	0	shift P right
0	1	add B to left half of P, then shift P right
1	0	subtract B from left half of P, then shift P right
1	1	shift P right

This algorithm (Booth's algorithm) can be implemented on the same hardware as before, with the ALU being used to subtract as well as add, and the control logic considering at each step the rightmost bit of P and the rightmost bit in the previous step (which, in the case of the first step, is $a_{-1} = 0$). The P register will need extending by one bit in order to remember the previous rightmost bit (a_{i-1}).

5. *Software Design and Concurrency*

(a) *Waterfall* model is a methodology for designing systems using a 5-stage sequential approach. see diagram



Formal Specification uses a mathematical framework, e.g. logic statements with pre and post conditions.

Prototyping build a sub-featured (no error correcting, simple frontend etc) version of some aspect of the system typically in a prototyping language (e.g. a scripting language) for evaluation by users prior to implementation of the final product.

(b) *Functional Programming Language*

- global store
- operations are functions on this global store.
- simpler than OOD especially when modifying existing code.

Object-Oriented Design

- local store (within classes/objects)
- operations are methods - little or no global store.
- better for all new systems of any complexity.

(c) (i) radiation cancer treatment - waterfall method appropriate as safety critical. (Prototyping may be appropriate for user interface ...)

(ii) Software for medical clinic (admin) - prototyping as it is hard to get a complete spec.

(iii) Mars explorer - waterfall method - possibly formal methods as cost of failure expensive and hard to fix.

(iv) Word processor - spiral design (repeated prototyping and release and specification of new features.

(d) Two problems exist in the code (1) Non-atomic, so that race condition may occur

E6 2001

```
class DiningPhilosophers
{   private: static int forkinuse[5];
    public:  static void halfhungry(unsigned int n)
             {   while (forkinuse[n % 5]) { /* busy wait */ }
(1) Non-atomic - interruption may occur here
                 forkinuse[n % 5] = 1;
             }
    public:  static void hungry(unsigned int n)
                 while (forkinuse[(n+1) % 5]) { /* busy wait */ }
(1) Non-atomic - interruption may occur here
                 forkinuse[(n+1) % 5] = 1;
                 cout << n << " is eating" << endl;
                 forkinuse[n % 5] = 0;
                 forkinuse[(n+1) % 5] = 0;
             }
};
```

(2) Deadlock may occur where all the dinners take the fork to their left.
Additional problem: (3) the majority of I/O libraries are non-atomic. Thus a sempahore may also be added around the "cout ..." line.

Corrected code for problem (1) requires the use of semaphores. The solution detailed is the simplest. (note that the methods on the semaphore may be lock and unlock).

```
class DiningPhilosophers
{   private: static int forkinuse[5];
             static Semaphore g(1);
    public:  static void halfhungry(unsigned int n)
             {   g.WAIT();
                 while (forkinuse[n % 5]) { /* busy wait */ }
                 forkinuse[n % 5] = 1;
                 g.SIGNAL();
             }
    public:  static void hungry(unsigned int n)
             {   g.WAIT();
                 while (forkinuse[(n+1) % 5]) { /* busy wait */ }
                 forkinuse[(n+1) % 5] = 1;
                 g.SIGNAL();
                 cout << n << " is eating\n";
                 forkinuse[n % 5] = 0;
                 forkinuse[(n+1) % 5] = 0;
             }
};
```

E6 2001

(2) Deadlock is handled by removing symmetry. In this case would use code of the form in both of the methods.

```
if (n == 1) {
     while (forkinuse[(n+1) % 5]) { /* busy wait */ }
     forkinuse[(n+1) % 5] = 1;
} else {
     while (forkinuse[n % 5]) { /* busy wait */ }
     forkinuse[n % 5] = 1;
}
```

Alternative is to use

```
int a=(n%5), b=(n+1)%5;
if (n == 1) {b=1; a=2};
```

The perform forkinuse on a and b.

E6 2001

6. *Transactions*

(a) A database transaction provides a way of implementing concurrent composite operations whilst maintaining database state consistency. They are required to enforce two main aspects of operation composition (a) concurrency control (ensures consistency - cannot interrupt before all operations are performed) and (b) recovery (all or none of the operations are performed)

(b) ACID

- Atomicity: either all or none of the transactions operations are performed
- Consistency: a transaction transforms the system from one consistent state to another
- Isolation: An incomplete transaction cannot reveal its result to other transactions before it commits
- Durability: Once a transaction is committed the system must guarantee that the results of the operations will persist even if there is a subsequent power failure.

(c) Basic design requirements

- Use a classic file system (e.g. FAT or Unix-like) with free blocks but recognising that moving a block of a file to/from the free list needs to be an atomic action protected by semaphone. Moreover the action needs to be added to the log file to protect in case of a crash.
- The log file is a separate file which is only appended to. Before doing an operation to disc, the action should be written to the log file (again serialised by semaphore so that read/modify write of the last block is OK.)
- On resumption the logfile must be scanned to complete or abandon tasks in progress (there are multiple possible answers here as to whether I
  (i) undo an operation and do not respond to the request for it, waiting for it to be re-requested; or
  (ii) complete the operation and issue the 'operation done' packet.

Specification requirements

- Specification of log file access:

```
Operations:
    record-request(r)
       Record the request r on the log file, by scanning to its end
       (zero block at end).
       write the request in final block.
       The above is critical region to be protected by semaphore.
       Returns the request number.
```

E6 2001

```
complete-request(rn)
   Mark the request r as completed in the log file.
   The request may be removed when the log file needs to be compacted.
redo-on-crash()
   [My choice here] standard two-phase locking: redo
   all the transations started in the file which are not finished
   starting in invocation order.
```

- Specification of file system:

```
Directory lives (say) from block 1000-1999, log file 2000-2999,
files are chained through the media (e.g. only 508 bytes used in each
block per file).
Block 999 (say) hold the first block of the free list, again chained
through the media.
All file system operations are locked by semaphore so that
there is no concurrent performance of file operations.
In addition all operations are written to log file at their
start to enable their resumption at the powerfail resumption.
```

- On power resumption:

```
redo-on-crash() is invoked to
1. clear all semaphores;
2. complete existing actions;
3. the log file is then cleared
4. and the system waits for action request.
[This means that action on 'log file full' can just be treated
as pseudo-power-fail!!]
```

- Function Definitions

```
void record-request(r)
{   int i = 2000;
    lock(LOGFILE);
    for (;i<3000;i++)
    {  char v[512];
       readblock(i, v);
       if (space_for_logentry(v))
       {   insert_logentry(v, REQSTART, r);
           writeblock(i, v);
           unlock(LOGFILE);
           return;
       }
    }
    reboot();  /* log file full -- simulate powerfail */
```

E6 2001

```
}


int complete-request(r)
{   int i = 2000;
    lock(LOGFILE);
    for (;i<3000;i++)
    {  char v[512];
       readblock(i, v);
       if (space_for_logentry(v))
       {   insert_logentry(REQEND, v, r);
           writeblock(i, v);
           unlock(LOGFILE);
           return;
       }
    }
    reboot();  /* log file full -- simulate powerfail */
}

int redo-on-crash()
{   /* executed with no concurrent tasks -- hence no semaphore */
    for (i = 2000; i<3000; i++)
    {  char v[512];
       readblock(i, v);
       for (r in v)       /* several requests per block */
       {   if (!completed(r)) redo(r)
           switch (action(r))
           {
case WRITE_BLOCK: write_block();
case READ_BLOCK: read_block();
case CREATE_FILE: create_file();
case DELETE_FILE: delete_file();
           }
           complete-request(r);
       }
    }
}

write_block(char *f, int n, data x)
{   int i;
    lock(FILEACCESS)
    record-request(WRITE, f, n, x);
    i = disc_address(f, n);
```

E6 2001

```
    physical_write_block(i, x);
    network_ack();
    complete-request(WRITE, f, n, x);
    unlock(FILEACCESS)
}

read_block(char *f, int n)
{   int i;
    data x;
    lock(FILEACCESS)
    record-request(WRITE, f, n, x);
    i = disc_address(f, n);
    x = physical_read_block(i, x);
    network_ack();
    complete-request(WRITE, f, n, x);
    unlock(FILEACCESS)
}

create_file(char *f, int n)
{   delete_file(f);       /* just in case exists */
    lock(FILEACCESS)
    record-request(CREATE, f);
    /* usual code to get n blocks from free list, zero them and
       create directory entry */
    network_ack();
    complete-request(CREATE, f);
    unlock(FILEACCESS);
}

delete_file(char *f)
{   lock(FILEACCESS)
    record-request(DELETE, f);
    /* usual code to remove directory entry and
       add disc blocks for file to freelist */
    network_ack();
    complete-request(DELETE, f);
    unlock(FILEACCESS);
}
```

7. *Search techniques*

(a) (i) Map of all paths within a state space is a graph of nodes. Nodes are a bookkeeping data structure used to represent the search tree for a particular problem with a particular algorithm. More than one node in the search tree can correspond to being in the same state but arrived to via a different path.

5 components of node data structure are:

```
- state:       the state in the state space to which node corresponds
- parent node: the node in the search tree that generated the node
- operator: action/operator that was applied to generate state
- depth of node: number of nodes on path from root to this node
- path cost: cost of path from initial state to the state of this node
```

The first is essential. The others depend on the search stategy and pruning to be used.

(ii)Queue is the collection of nodes waiting to be expanded. Operations on queue include:

```
Make-queue
Test-for-empty
Remove-front-element
Add-element
Queuing function to sort queue -- determines different search algorith
(Additional functions for search are goal-test and expand-operator to
 create new node).
```

Need to avoid expanding states that have already been encountered or search trees will be infinite. - Do not return to parent state (i.e. expand function refuses to generate a successor the same as the node's parent. -Do not generate paths with cycles – successor must not be same as any ancestor. -Do not generate any state that has already been created. Need record of all states and hence complex. Done with a hash table that stores all nodes.

(b) (i) For depth first

(a) Form a one element queue, Q, consisting of the root node

(b) Until the Q is empty or the goal has been reached, determine if the first element in the Q is the goal

   i. if it is do nothing

   ii. if it isn't remove the first from the Q and add the first element's children, if any, to the front of the queue.

(c) If the goal is reached, success else failure.

Breadth first is the same other than adding children to the BACK of the Q.

(ii) The cost for breadth first in time and memory is $\mathcal{O}(b^d)$. For depth first the memory of $\mathcal{O}(bd)$ and time $\mathcal{O}(b^d)$.

(iii) Assume that 1000 nodes per second can be expanded and checked against goal-test and that each requires 100 bytes of storage (or up to 1000 these numbers).

Breadth-first and depth first require nodes expanded $\mathcal{O}(b^d) = 10^6$ with time of 16.6 minutes and memory of 100MBytes (breadth-first!!) and 6KBytes (depth). Memory requirements dominate.

(c) (i) Problems where a set of constraints have to be satisfied for a valid solution. Typically have a set of parameter values to find. The set of constraints do not provide a unique global solution o the parameters. Reach global solution with local search.

Apply constraints to a selected node to generate new constraints. If the constraints contain a contradiction then terminate the path (prune tree).

(ii) Depth first and breadth first are uninformed search techniques. No attempt is made to evaluate the usefulness of expanding an open node. Use knowledge to attempt to expand nodes which are judged to be closest to the goal state.

Usually use a heuristic function, h(n), to estimate the cheapest path from the state at node n to a goal state. Algorithms are modified by SORTING children using h(n) before adding to queue based on evaluation.

Modifications lead to hill-climbing. Sorting whole queue leads to Best-first algorithm.

8. *Logic and Inference*

(a) A sentence is VALID if and only if it is true under all possible interpretations in all possible worlds, regardless of the state that is being described.

Form a truth table to test for valid sentences. One row is for each combination of truth values for the proposition symbols of the sentence. If sentence is true in every row then it is valid. For statement 1

E	F	$E \rightarrow F$	$(E \rightarrow F) \wedge F$	S_1
F	F	T	F	T
T	F	F	F	T
F	T	T	T	**F**
T	T	T	T	T

For statement 2

E	F	$E \rightarrow F$	$(E \rightarrow F) \wedge \neg F$	S_2
F	F	T	T	T
T	F	F	F	T
F	T	T	F	T
T	T	T	F	T

(b) Resolution produces proofs by

(a) Converting problems into a canonical form. All axioms are converted to clauses in **Conjunctive Normal Form.**

(b) Using **refutation** — ie, by attempting to show that the negated assertion produces a contradiction with known axioms in the database. This requires using the **resolution inference rule** and **unification.**

The key is to convert all clauses to a standard conjunctive normal form for which it is very easy to apply the resolution inference rule.

Applying the resolution inference rule requires looking for predicate and not-predicate pairs to infer new axioms/clauses which must be true.

Now we need to spot pairs negated pairs like Pred3() and ¬ Pred3() – but before we can immediately resolve them we need look carefully at what the arguments inside are to ensure that the statements are really contradictory. What is required is a method for checking terms and substituting for the variables. This match and substitute process is called *unification.*

The unifier looks for

Predicate(args1) and ¬ Predicate(args2)

E6 2001

and then asks whether *args1* and *args2* are unifiable.

To prove a theorem using resolution:

(a) negate the theorem to be proved and add the result to the list of axioms

(b) put axioms into clause form

(c) Until the empty clause, Nil, is produced or there is no resolvable pair of clauses, find resolvable clauses, resolve them and add the result to the list of clauses

(d) if the empty clause is produced, report that the theorem is TRUE. If there are no resolvable clauses, report that the theorem is FALSE.

Another important component of the proof procedure is to add the negation of the theorem you are trying to prove so that you derive something that is obviously false (proof by refutation or reduction ad absurdum). Formally:

i.e.$KB \wedge P \Rightarrow FALSE$ can be used to deduce that P is TRUE.

(c)(i) Proof by resolution requires the database clauses to be in conjunctive normal form with no existential quantifiers. Skolemization is required to remove the existential quantifier by substituing a constant for function of universal quantifier.

$\exists y \forall x P(x, y)$ becomes $P(x, S)$
$\forall x \exists y P(x, y)$ becomes $P(x, S(x))$

(ii) There is somebody who is loved by everyone.

$\exists y \forall x [loves(x, y)]$ becomes $loves(x, S)$

Everyone loves somebody.

$\forall x \exists y [loves(x, y)]$ becomes $loves(x, S(x))$

Write phrase as Somebody loves John

$\exists x loves(x, John)$

Negate the phrase

$\neg \exists x loves(x, John)$

Hence

$\forall x \neg loves(x, John)$

and add to database. This now consists of

$\neg loves(x_1, John)$
$loves(x_2, S(x_2))$
$loves(x_3, S)$

None of the axioms are resolveablele therefore the theorem is not proven.

(iii) Need to represent change in axoms(i.e. diachronic rules). Situation calculus is a modification to first-order logic by adding a situation argument to the corresponding predicate and changing situations after actions

Use function of form Property or result of action in form Result(action, newsituation). Axioms as a result of actions become effect axioms. Need axioms to show how world stays same and are known as frame axioms.

ENGINEERING TRIPOS PART IIA 2001

ELECTRICAL AND INFORMATION SCIENCES PART I 2001

EXAMINER'S REPORT, PAPER E6 COMPUTING SYSTEMS

Question 1 Attempts 68, mean 11.7/20.
This question examined the candidates understanding of Bayes classifiers and the effects of adding dimensions. A large number of students realised the significance of adding additional dimensions. However, it was disappointing that few could perform the correct algebraic manipulation to derive the probability of error.

Question 2 Attempts 60, mean 12.0/20.
A straightforward question about the perceptron algorithm. A number of the weaker candidates failed to incorporate a bias term into the perceptron algorithm.

Question 3 Attempts 80, mean 10.9/20.
This was a popular question about cache structure and code optimisation for execution on a pipelined processor. There was a broad range of answers with several weak candidates, but a pleasingly high number who understood the last part about pipeline execution.

Question 4 Attempts 59, mean 10.1/20.
This question explored the candidates understanding of the architecture of the carry-look-ahead adder and multiplication. The general standard of the answers was poor, and the question had to be re-marked with a specially adapted generous marking scheme to get the present average. Most candidates were unable to reproduce the book-work on the carry-look-ahead adder and quite a number failed to explain why the multiplier did not work for negative numbers. A very small number of candidates correctly derived Booth's algorithm for the last part of the question.

Question 5 Attempts 42, mean 12.1/20.
This question was about object oriented design and concurrency. It was generally successful and produced a good range of answers, many of which addressed the concurrency problem in the last part with imagination and intelligence.

Question 6 Attempts 20, mean 7.4/20.
This question covered database transactions and the software architecture necessary to maintain a reliable file system in the presence of unreliable hardware. The answers to the simple book-work questions at the start were generally good, but the problem in part (c), which was worth 14 marks, was answered very poorly. Most candidates failed to address satisfactorily the issues of locking, logging, recovery strategy, or to give a plausible structure for the log file. Despite an extremely generous marking scheme, the average mark remained low.

Question 7 Attempts 95, mean 12.1/20.
The most popular questions on the exam paper. It tested knowledge about standard search strategies. In contrast to previous years this question asked about data structures and operations on objects to implement generic search strategies. Many candidates simply described generic search algorithms.

Question 8 Attempts 92, mean 12.4/20.
A popular question with the students. The aim of the question was to investigate knowledge of logic. Some of the weaker students had little understanding of skolemization and how to use it in proving statements.

M.J.F. Gales