Exam 2 March 2015 Functional Programming and Intelligent Algorithms

Duration 4 hours **IE501614**

Calculators and other accessories are not permitted.
Include **all intermediate calculations** necessary to justify your answer.

Problem 1 . . . . *(6%)*

Evaluate the following Haskell expressions:

(a) `[ x^2 | x <- [1..4] ]`

(b) `map abs [-1,0,1]`

(c) `zipWith g [-1,0,1] [3,2,0]`

```
    where g x y = x^y
```

Problem 2 . . . . *(12%)*

This question concerns features of and techniques for functional programming.

(a) Addition is denoted by `+`. In Haskell, `(+2)` is a valid expression. What is the type of `(+2)`?

(b) Show how you can use recursion to define your own version of the `map`-function.

(c) Explain why I/O (input/output) poses particular challenges for functional programming. What techniques are applied to overcome the challenges in Haskell?

(d) Using pseudo-random numbers in Haskell can be cumbersome. Without global state the pseudo-random generator has to be passed as an argument to every function needing it, and the updated generator must be returned. Explain briefly how monads can be used to solve to simplify the use of pseudo-random numbers.

Problem 3 . . . . *(15%)*

This problem concerns the neuron and its key properties. In Haskell we could use the following data type and recall function for a neuron:

```haskell
type Neuron = [Double]
recall n xs = threshold $ sum $ zipWith (*) n ((-1):xs)
```

(a) Explain what we mean by a neuron and how it can be used.

(b) Rewrite the definition of `recall` using common mathematical notation.

(c) Give a suitable type declaration for the `recall` function.

(d) Explain what we mean by *linear classifier*. Feel free to draw figures to support your explanation.

(e) Demonstrate that the neuron is a linear classifier.

Problem 4 . . . . *(15%)*

(a) Define *underfitting* and *overfitting* of classifiers. Explain how you can identify cases of underfitting and of overfitting through testing.

(b) What are the typical causes of underfitting?

(c) What are the typical causes of overfitting?

(d) What do we mean by *feature selection*? Why is feature selection useful?

(e) Describe one method for feature selection, including a step by step description of how it is performed.

Problem 5 . . . . *(17%)*

When we design a neural network to solve a particular classification problem, we are often adviced to split the available data into three sets. Marsland calls them *training set*, *validation set*, and *test set*.

(a) Explain how each of the three sets are used at different stages of the design process. Please emphasise the differences between the roles of the different data sets.

(b) Why is it important to keep the three sets disjoint? Explain what ill effects we risk if we were to reuse items in several of the sets.

(c) What advantage is gained by increasing the size of the training set?

(d) What advantage is gained by increasing the sizes of the test and validation sets?

Problem 6 . . . . *(12%)*

A simple algorithm for optimising 1D functions is implemented in the code below. Since the search space is 1D and equivalent to a line, the algorithm can search along the line until it finds an optimum. If the goal is to find a maximum (minimum), the algorithm moves up (down) the line until it cannot get any further, and the stopping point is the solution.

```haskell
f0, f1 :: Double -> Double
f0 x = x^2 + 2*x
f1 x = - (f0 x)

fopt :: (Double -> Double) -> Double -> Double -> (Double, Double, Int)
fopt f x0 delta | x0 - delta < l || x0 + delta > u = (x0, f x0, 0)
                | f (x0 - delta) > f x0            = (a, b, c + 1)
                | f (x0 + delta) > f x0            = (a', b', c' + 1)
                | otherwise                        = (x0, f x0, 0)
                  where l = -20
                        u = 20
                        (a,b,c)    = fopt f (x0 - delta) delta
                        (a',b',c') = fopt f (x0 + delta) delta
```

Consider the function `fopt` in the code above and the following example evaluation:

```
*Exam> fopt f1 5 0.01
(-0.9999999999999384,1.0,600)
```

(a) What does the three items of the returned tuple of the function evaluation above mean?

(b) Without doing any calculations, explain why evaluating the expression `fopt f0 5 0.01` would not yield the correct solution? (Hint: how are `f0` and `f1` related?)

(c) How could you implement a function `fopt'` similar to `fopt` such that `fopt' f0 5 0.01` would indeed yield the correct solution?

(d) For 1D functions with multiple optima, what is a fundamental limitation of algorithms like the one implemented here?

Problem 7 . . . . *(5%)*

Receding horizon control can be used to solve dynamic optimisation problems, that is problems which have to be solved repeatedly at every time step.

Explain the principles of receding horizon control. Illustrate your answer with a brief example.

Problem 8 . . . . *(18%)*

The code below shows some types and functions typically found in a binary GA.

```haskell
type Gene = [Int]
type Chromosome = [Gene]
type Population = [Chromosome]

isEqual :: (Eq a) => a -> a -> Int
isEqual a b | a == b = 0
            | otherwise = 1

geneCost :: Gene -> Gene -> Int
geneCost g1 g2 = sum $ zipWith isEqual g1 g2

chromCost :: Chromosome -> Chromosome -> (Int, Chromosome)
chromCost target c = (sum $ zipWith (geneCost) target c, c)

evalPop :: Chromosome -> Population -> [(Int, Chromosome)]
evalPop target pop = map (chromCost target) pop

-- Some test data
g1,g2,g3,g4 :: Gene
c1,c2,c3,c4,t :: Chromosome
p :: Population
g1 = [1,0,1,0,1,0]; g2 = [0,1,0,1,0,1]; g3 = [0,0,0,0,0,0]; g4 = [1,1,1,1,1,1]
c1 = [g1,g2]; c2 = [g3,g4]; c3 = [g1,g3]; c4 = [g2,g4]
p  = [c1,c2,c3,c4]
t = [[0,1,0,1,0,1],[1,1,1,1,1,0]] -- target chromosome
```

The functions do the following:

- `isEqual` compares the values of two parameters and returns zero they are equal, or one otherwise.
- `geneCost` compares two genes bit by bit and returns a cost equal to the total number of unequal bits.
- `chromCost` compares a target chromosome with a candidate chromosome and finds a cost equal to the total number of unequal bits. The cost and the candidate chromosome is returned in a tuple.
- `evalPop` evaluates all the chromosomes in a population by comparing them to a target chromosome. It returns a list of tuples, where each tuple consists of a cost and the chromosome that evaluates to that cost.

(a) For GAs in general, explain the terms (i) chromosome; (ii) mutation; and (iii) cost function.

(b) An example function evaluation and output is given below:

```
*Exam> evalPop t p
[(10,[[1,0,1,0,1,0],[0,1,0,1,0,1]]),
 (4,[[0,0,0,0,0,0],[1,1,1,1,1,1]]),
 (11,[[1,0,1,0,1,0],[0,0,0,0,0,0]]),
 (1,[[0,1,0,1,0,1],[1,1,1,1,1,1]])]
```

Based on this output, give the correct sorted order of the four chromosomes `c1`, `c2`, `c3`, and `c4`, where the best chromosome is listed first.

(c) How would you implement a function for sorting a population? You should use the following type signature:

```haskell
sortPop :: [(Int, Chromosome)] -> [(Int, Chromosome)]
```

(d) For the chromosomes `c1` and `c2`, perform mating (determine the two offspring chromosomes) by using two-point crossover with crossover points between bits 3 and 4 for the first gene and between bits 4 and 5 for the second gene.