THE UNIVERSITY *of York*

BSc, BEng, MEng and MMath ExaminationsExaminations ,2007

COMPUTER SCIENCE, COMPUTER SYSTEMS AND SOFTWARE ENGINEERING, COMPUTER SCIENCE AND MATHEMATICS

Part A

ALGORITHMS FOR GRAPHICAL MODELS (AGM)

Open Examination

Issued at:
**26 November 2007**

Submission due:
**18 January 2008**

Your attention is drawn to the Guidelines on Mutual Assistance and Collaboration in the Student's Handbook.

All queries on this assessment should be addressed to **James Cussens**.

**Your examination number must be written on the front of your submission. You must not identify yourself in any other way.**

## Instructions

- All questions should be attempted.

- A submission after the deadline will be marked initially as if it had been handed in on time, but the Board of Examiners will normally apply a lateness penalty.

- Your write-up must be typeset and submitted to the General Office of the Department of Computer Science.

- The programs you write should be submitted electronically through the web page http://www.cs.york.ac.uk/submit. Your programs should be submitted as three files called `agm1.py`, `agm2.py` and `agm3.py`.

- You are permitted to use `gPy` modules in your answers. Only use the version of these modules found in this directory: `/usr/course/agm/gPy/gPy`.

- All extra materials for this assessment can be found at http://www-users.cs.york.ac.uk/~jc/teaching/agm/assessment/07/, henceforth know as the *assessment directory*.

- All solutions must be justified, all programs fully documented. Discuss any possible time versus space compromises. Inefficient solutions will be penalised.

- Unit tests are provided to help you check your programs. Programs which pass the given tests get credit for doing so. To get further marks you need to write programs that are:

  1. efficient in both time and space. (If you decide to effect a trade-off between time and space, explain your reasoning.);
  2. well documented; and
  3. which also work on inputs other than those supplied with the tests you have!

# 1 Statistical Genetics (30 %)

In this question you have to write a Python program whose sole task is to construct a Bayesian network which provides a simplified model of the process of genetic inheritance for the 14 individuals represented by the family tree in Fig 1.1. As is customary in statistical genetics circles represent females and squares men. So, for example, in Fig 1.1 the man 9 and the woman 10 have produced two daughters 13 and 14.

There is genetic variance between individuals causing, for example, variation in eye colour and blood type. The different possible forms of a gene are called *alleles*. For example, there are three possible alleles for the blood type gene: A, B and O. Each individual has two copies of each gene, one inherited from their mother and one from their father—this constitutes their *genotype*. So in the case of the blood type gene there are six possible genotypes: AA, AO, AB, BB, BO and OO.

Your genotype is only probabilistically dependent on that of your parents. Recall that your father has *two* copies of each gene. *One* of these is chosen with probability 1/2 and passed down to you. Similarly, a randomly chosen copy is passed down from your mother. This is how your two copies are created. For example, suppose your mother had blood genotype AO and your father BB, then only genotypes AB and BO are possible, both with probability 1/2.

This model of genetic inheritance (called *Mendelian segregation*) allows one to compute genotype probabilities for an individual conditional on that individual's parents' genotypes. For individuals with unknown parents, genotype probabilities are estimated from genotype frequencies in the relevant human population. For the blood alleles assume that population frequency for the 6 genotypes is as in Table 1.1. (These are made up numbers.)

Specifically, you have to write a Python program which creates a Bayesian network (a `gPy.Models.BN` object) called `pedigree` with the following properties:

1. `pedigree` has 14 random variables called G1, G2,...G14 representing the genotypes of each of the 14 individuals in Fig 1.1.

2. `pedigree` can be used to compute the probability of each possible genotype

| Genotype | AA | AO | AB | BB | BO | OO |
|---|---|---|---|---|---|---|
| Probability | 0.2 | 0.2 | 0.3 | 0.1 | 0.1 | 0.1 |

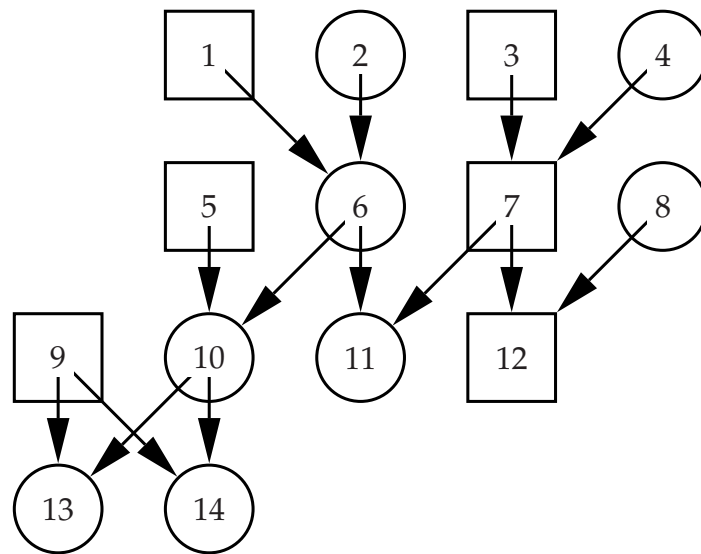Table 1.1: Genotype probabilities for those with unknown parents

3

Figure 1.1: Family tree for 14 (fictional) individuals

for each possible individual—perhaps conditional on information specifying the genotype of other individuals. (See the accompanying unittest for some example questions and the answers.)

(Further details on using Bayesian networks for statistical genetics can be found in [2], but don't feel you need to read this paper.)

**The answer to this question should be submitted in a file called `agm1.py`** Marking scheme: Passing all unit tests = 30 %, Quality of code = 40 %, Quality of write-up = 30 %

How to run the unit tests for this question:

1. Copy `agm1_test.py` from the assessment directory to the directory containing your answer called `agm1.py`.

2. Type `python agm1_test.py`

# 2 Inference for singly-connected, no evidence BNs (30 %)

There are inference algorithms for computing marginals in Bayesian networks which, when the BN meets certain requirements, can take advantage of those requirements to be more efficient than general-purpose algorithms like join forest calibration. In this question you are required to implement an algorithm which

1. takes a *singly-connected* BN

2. with no instantiated variables

as input, and returns the marginal distribution of each variable in the BN.

A singly-connected BN is one where there is at most one directed path between any two variables in the associated acyclic directed graph. So, for example, the 'Asia' BN is *not* singly-connected since there are two directed paths from Smoking to Dyspnea. On the other hand, the BN in Fig 1.1 is singly-connected. Singly-connected networks are also called *polytrees*.

Your algorithm should take advantage of the two given restrictions on its expected input to compute the marginals efficiently. Here are some hints on how to do this.

- Observe that the marginal for a variable without parents is always immediately available, and that at least one such variable exists in any BN.

- Think about what the two conditions imply for the parents of any variable.

- As specified below, your algorithm is to be implemented as a function called `forward_inference`, this name was chosen due to similarities with forward sampling.

Be sure to include in your write-up a *proof* that your approach computes the correct marginals.

Specifically you are asked to write a Python module which defines a class `myBN` which is a subclass of `gPy.Models.BN` and which has a method `forward_inference` defined for it. `forward_inference` takes a single argument—`self` a `myBN` object—and returns the marginals of all variables in that `myBN` object as a dictionary mapping each variable to a `gPy.Parameters.CPT` object representing its marginal.

Your code is not required to check that the BN under consideration is singly-connected and has no instantiated variables, and if passed such an object it can behave arbitrarily.

**The answer to this question should be submitted in a file called `agm2.py`** Marking scheme: Passing all unit tests = 20 %, Quality of code = 40 %, Quality of write-up = 40 %

How to run the unit test for this question:

1. Copy `agm2_test.py` from the assessment directory to the directory containing your answer called `agm2.py`.

2. Type `python agm2_test.py`

# 3 Robot tracking (40 %)

Robots, like ourselves, live in an uncertain world: their motors can fail to take them where they want to go and their sensors can be faulty. In this question you are required to use a sampling approach to help a (fictional) robot estimate where it is likely to be (and how it got there).

Our robot lives in an unbounded grid world: at any time point its position is given by a grid location: $(x, y)$ where $x$ and $y$ are both integers. At each time step, the robot can choose to attempt to move in one of four directions:

**LEFT** $(x, y) \rightarrow (x - 1, y)$

**RIGHT** $(x, y) \rightarrow (x + 1, y)$

**UP** $(x, y) \rightarrow (x, y + 1)$

**DOWN** $(x, y) \rightarrow (x, y - 1)$

Unfortunately, the robot has unreliable motors. In each of the 4 cases an attempt to move may fail, in which case the robot stays where it is. The probability of failure in each of the 4 cases is 0.2.

Each grid location has a colour which is either red, blue or green. After each attempted move the robot uses a sensor in an attempt to determine the colour of the grid location where it is located. However what is seen by the robot may not be the actual colour. The probabilities relating seen colours to actual colours are given by the CPT in Fig 3.1.

Your basic task is as follows. Given a sequence of robot actions, together with robot observations, *estimate* the correct probability distribution over the sequence of true

```
Actual |          Seen
       |    blue   green    red
------ | -------------------
blue   |    0.60    0.20   0.20
green  |    0.20    0.60   0.20
red    |    0.20    0.20   0.60
```

Figure 3.1: $P(\texttt{Seen}|\texttt{Actual})$ for robot sensing of colours

positions for the robot, *assuming that it starts from position (0,0)*. Fig 3.2, which was produced by running the associated unit test, should give you the general idea. For example, the first part of this output shows that 6 trajectories have been estimated to have non-zero probability given that the robot went LEFT, UP, DOWN, LEFT and observed blue, green, green, red.

Now to be more specific. The colour for each grid location is computed by the `colour` function defined in the Python module `agm3_colour` given by the file `agm3_colour.py` available from the assessment directory. Evidently, your solution will need to import this function.

You are required to write a single function called `sample_trajectories` which takes exactly two arguments. The first argument is a 'history', a tuple each element of which is a pair: (attempted move, noisy observation). The second argument is a sample size. With a sample size of $n$, your code should sample $n$ possible trajectories using *normalised importance sampling*. This will produces $n$ weighted sampled trajectories which should be used to estimate the true distribution over state trajectories. This estimate is what your `sample_trajectories` returns. It should be a dictionary mapping each trajectory to its estimated probability, whenever that estimate is positive.

Note that you have not been told from which distribution to sample the trajectories, nor how to compute the weights, nor how to produce the final probability estimates. This is for you to decide. You are looking for a solution which provides a reasonable estimate of the true distribution, but does this reasonably quickly. Discuss any trade-offs. To give you a rough idea of how a solution should perform, Fig 3.3 shows how long (18 secs) a simple, non-optimised solution takes to produce a million (weighted) samples for a history of length 6. Perhaps you'll do better!

(The problem given here is rather artificial in that normally one cares only about the location of a robot at a particular time. A nice account of this area is given in [1], but don't feel you need to read this paper.)

**The answer to this question should be submitted in a file called `agm3.py`** Marking scheme: Passing all unit tests = 20 %, Quality of code = 40 %, Quality of write-up = 40 %

How to run the unit test for this question:

1. Copy `agm3_test.py` from the assessment directory to the directory containing your answer called `agm3.py`.

2. Type `python agm3_test.py`

```
test_basic (__main__.Testagm3) ... Using this history:  (('LEFT',
'blue'), ('UP', 'green'), ('DOWN', 'green'), ('LEFT', 'red'))
With this many samples 10
((-1, 0), (-1, 0), (-1, -1), (-1, -1)) 0.0714285714286
((0, 0), (0, 0), (0, -1), (-1, -1)) 0.0714285714286
((0, 0), (0, 1), (0, 0), (-1, 0)) 0.0714285714286
((-1, 0), (-1, 1), (-1, 0), (-2, 0)) 0.285714285714
((-1, 0), (-1, 1), (-1, 1), (-2, 1)) 0.428571428571
((-1, 0), (-1, 1), (-1, 0), (-1, 0)) 0.0714285714286
**********************************************************
Using this history:  (('LEFT', 'blue'), ('UP', 'green'),
('DOWN', 'green'), ('LEFT', 'red'))
With this many samples 100
((-1, 0), (-1, 0), (-1, -1), (-1, -1)) 0.00675675675676
...
((-1, 0), (-1, 0), (-1, -1), (-2, -1)) 0.162162162162
**********************************************************
Using this history:  (('LEFT', 'green'), ('UP', 'green'),
('UP', 'green'), ('LEFT', 'red'))
With this many samples 1000
((0, 0), (0, 0), (0, 1), (0, 1)) 0.0230769230769
....
((-1, 0), (-1, 0), (-1, 1), (-2, 1)) 0.403846153846
**********************************************************
Using this history:  (('LEFT', 'green'), ('UP', 'green'), ('UP',
'green'), ('LEFT', 'red'), ('LEFT', 'green'), ('UP', 'green'),
('UP', 'green'), ('LEFT', 'red'))
With this many samples 1000
((-1, 0), (-1, 0), (-1, 0), (-2, 0), (-3, 0), (-3, 1), (-3, 1), (-4,1))
0.0159791328745
....
((0, 0), (0, 1), (0, 2), (-1, 2), (-2, 2), (-2, 3), (-2, 4), (-2, 4))
4.9318311341e-05
**********************************************************
ok


----------------------------------------------------------------
Ran 1 test in 0.061s

OK
```

Figure 3.2: Example output produced by running python agm3_test.py. The output has been edited so that many lines have been deleted and line breaks have been added.

10

```
Python 2.4.3 (#1, Jul 26 2006, 20:13:39)
[GCC 3.4.6] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from agm3 import sample_trajectories
>>> x=(('LEFT','blue'),('UP','green'),('DOWN','green'),
('LEFT','red'),('LEFT','blue'),('UP','green'))
>>> from timeit import Timer
>>> Timer('sample_trajectories(x,1000000)',
          'from __main__ import sample_trajectories,x').timeit(1)
18.445687055587769
```

Figure 3.3: Timing a solution

# Bibliography

[1] Dieter Fox, Sebastian Thrun, Wlofram Burgrad, and Frank Dallaert. Particle filters for mobile robot localization. In Arnaud Doucet, Nando de Freitas, and Neil Gordon, editors, *Sequential Monte Carlo Methods in Practice*, pages 401–428. Springer, 2001.

[2] Steffen L. Lauritzen and Nuala A. Sheehan. Graphical models for genetic analyses. *Statistical Science*, 18(4):489–514, 2003.