

THE UNIVERSITY *of York*

BSc, BEng, MEng and MMath Examinations Examinations ,2008

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

Part A

ALGORITHMS FOR GRAPHICAL MODELS (AGM)

Open Examination

Issued at:

3 December 2008

Submission due:

21 January 2009

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/08/>, 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 Constructing a Bayesian Network and checking for conditional independence (15 marks)

Much current research in graphical models concerns the hard problem of reliably inferring Bayesian networks from data. The *constraint-based* approach to this has two basic steps: conditional independence relations are inferred from the data and then a network fitting the discovered conditional independence relations is constructed.

In this question you are required to do something similar to the second of these steps: you have to construct a Bayesian network involving 6 variables G_1, G_2, \dots, G_6 which meets the following requirements:

1. The Bayesian network is consistent with the following variable ordering:

$$G_1, G_2, G_3, G_4, G_5, G_6$$

so that, for example, G_1 can have no parents.

2. No variable has more than two parents.
3. $\forall i, j \in 1, 2, \dots, 6 : i \neq j \rightarrow G_i \not\perp G_j | \emptyset$
4. $G_5, G_6 \perp G_1, G_2, G_3 | G_4$
5. $G_6 \perp G_5 | G_4$
6. $G_2 \perp G_3 | G_1$

You are not required to write an efficient algorithm to find a suitable BN (that would be machine learning which we have not covered properly). You are free to find the BN however you wish: manually or with the assistance of a computer program. The

method you use is not marked. All that matters is that you find a correct BN and are able to use `gPy` to construct the appropriate ADG object.

As well as constructing this Bayesian network structure you are required to write a function called `ci` which is called like this: `ci(adg, a, b, s)` where:

1. `adg` is a Bayesian network structure (i.e. an ADG object)
2. `a` is a frozenset of variables
3. `b` is a frozenset of variables
4. `s` is a frozenset of variables

`ci(adg, a, b, s)` returns `True` if $a \perp b | s$ in the BN structure `adg` and `False` otherwise.

Specifically, write a Python module which defines an ADG object called `bnq1` which models a BN structure meeting the above requirements. The Python module should also define the function `ci`.

The answer to this question should be submitted in a file called `agm1.py` Marking scheme: Passing all unit tests = 5 marks, Quality of code = 5 marks, Quality of write-up = 5 marks.

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 Making variables independent by (minimally) conditioning (20 marks)

In Question 1 you wrote a function which tested for the conditional independence of sets `a` and `b` *given* a third set `s`. Sometimes it is more important to *find* a set `s` such that two given sets `a` and `b` are conditionally independent given `s`.

Purely to motivate this question, consider the case of *gene regulatory networks* (modelled as BNs). We might wish to find a small set of genes `s` which 'block' any influence between two other sets `a` and `b`. By intervening to artificially set the expression levels of genes in `s` any problems caused by how genes in `a` are being expressed will not affect genes in `b`.

In this question you are required to write a function `find_minimal_separator` which is called like this: `find_minimal_separator(adg, a, b)` where:

1. `adg` is a Bayesian network structure (i.e. an ADG object)

2. a is a variable in adg
3. b is a variable in adg

The function should return, if possible, a *minimal separator*: a set s such that $a \perp b|s$ (according to adg) and such that there is no smaller s' where $a \perp b|s'$. If there are no separators for a and b in adg , then the function should return `None`. If there are several minimal separators it does not matter which is returned. If either a or b is not in adg , then an exception should be raised. (My solution raises a `KeyError`.)

You will not be surprised to learn that you are not the first people to tackle this problem. You are, of course, free to research into and use existing solutions. Be sure to cite any sources

The answer to this question should be submitted in a file called `agm2.py` Marking scheme: Passing all unit tests = 4 marks, Quality of code = 8 marks, Quality of write-up = 8 marks

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 Structure from randomness: the case of the emerging not-entirely-stable checkerboard (15 marks)

Grab the files `run_gibbs.py`, `run1.pck` and `run2.pck` from the assessment directory. Do both

```
python run_gibbs.py run1.pck
```

and then

```
python run_gibbs.py run2.pck
```

from the command line.

In both cases a window will appear. You operate the program by clicking the button marked `Press` in this window. After 100 clicks the program will end (ungracefully!). You will see that in both cases, a checkerboard pattern emerges. Once the pattern is established it is not entirely stable—occasionally squares flip colour but are fixed on the next iteration. Note that if `run1.pck` is used we end up with a white square in the top left hand corner, but using `run2.pck` this square is black.

Look at `run_gibbs.py`. You will see that it reads in an object from the file given as the last command-line argument and calls that object `sample`. `sample` is assumed to be a sequence of sequences. The program also constructs a GUI with 100

`Tkinter.Label` widgets arranged in a 10×10 grid and then adds a `Tkinter.Button` object. Each time the button is pressed the 'next' sequence from `sample` is inspected. This sequence is assumed to be a 100 binary digits. If the i th digit is 1 then the i th widget is set to black, otherwise it is white. Widgets are ordered $(0,0), (0,1), \dots, (0,9), (1,0), \dots, (9,9)$.

The two samples contained in `run1.pck` and `run2.pck` are in fact the output of 100 iterations of Gibbs sampling from a distribution which you have to define. These two files were produced as follows:

```
python agm3.py run1.pck
python agm3.py run2.pck
```

where `agm3.py` is my model answer. To help you along here are the last few lines of my model answer:

```
import sys, cPickle
from gibbs import gibbs_sample
from gPy.Models import FR
fr = FR(factors)
sample = gibbs_sample(fr,100,0)
cPickle.dump(sample,open(sys.argv[1],'w'))
```

So the programming part of this question boils down to constructing the object `factors` which is then used to create a probability distribution `fr` from which we can then sample. The file `gibbs.py` containing the function `gibbs_sample` is available from the assessment directory. Note that the unittest provided merely checks that the file produced by your code has the right format.

Evidently, due to the random nature of Gibbs sampling your program is not expected to generate either `run1.pck` or `run2.pck` exactly. It's enough that it has the same qualitative behaviour: convergence to one of the two possible checkerboards, with occasional fixable mutations once it gets there.

In your write-up discuss how well Gibbs sampling is working here. Does it provide a good way of sampling from the distribution I called `fr` in my model solution?

The answer to this question should be submitted in a file called `agm3.py` Marking scheme: Passing all unit tests = 2 marks, Quality of code = 6 marks, Quality of write-up = 7 marks.

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`

