

THE UNIVERSITY *of York*

BSc, BEng, MEng and MMath Examinations ,2010

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

Part A

ALGORITHMS FOR GRAPHICAL MODELS (AGM)

Open Examination

Issued at:

**8 December 2010**

Submission due:

**26 January 2011**

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 four files called `agm1.py`, `agm2.py`, `agm3.py` and `agm4.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/10/>, 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 Part-of-speech tagging (15 marks)

Part-of-speech (PoS) tagging is a rudimentary form of natural language processing. The problem is to ‘tag’ each word in a sentence of natural language (e.g. English) with its correct part of speech. Examples of parts of speech are: noun, verb, adjective, determiner, etc. The problem is non-trivial since many words are ambiguous. For example, the word *saw* can be a noun or a verb.

Your task in this question is to construct a probabilistic graphical model to do a very restricted form of PoS tagging. The restrictions are as follows:

1. Each sentence to be tagged contains at most 5 words.
2. No words other than the following will be found in sentences: *I, love, you, saw, bank, the.*
3. Only the following PoS tags are to be used: noun, verb, det. (det is short for determiner.)

Given a sentence, your probabilistic graphical model should allow the computation of, for each word in that sentence, a (marginal) probability distribution over each possible tag for that word. Let  $W_i$  be the (observed) word at position  $i$  in the sentence and  $T_i$  be the (unobserved) correct tag for that sentence. Your model should obey the following conditional independence relations:

1.  $W_i$  should be independent of  $W_j$  and  $T_j$  for all values of  $j \neq i$  conditional on  $T_i$ .
2. For all  $i$ ,  $T_{i+1}$  is independent of all  $T_j$  where  $j < i$ , conditional on  $T_i$ .

The distribution defined by your model should have the following (conditional) marginals:  $P(T_1 = \text{noun}) = 0.4$ ,  $P(T_1 = \text{verb}) = 0.1$ ,  $P(T_1 = \text{det}) = 0.5$

$$P(T_{i+1} = \text{noun} | T_i = \text{noun}) = 0.2$$

$$P(T_{i+1} = \text{det} | T_i = \text{noun}) = 0.1$$

$$P(T_{i+1} = \text{verb} | T_i = \text{noun}) = 0.7$$

$$P(T_{i+1} = \text{noun} | T_i = \text{verb}) = 0.4$$

$$P(T_{i+1} = \text{det} | T_i = \text{verb}) = 0.4$$

$$P(T_{i+1} = \text{verb} | T_i = \text{verb}) = 0.2$$

$$P(T_{i+1} = \text{noun} | T_i = \text{det}) = 0.8$$

$$P(T_{i+1} = \text{det} | T_i = \text{det}) = 0.1$$

$$P(T_{i+1} = \text{verb} | T_i = \text{det}) = 0.1$$

$$P(W_i = \textit{the} | T_i = \text{det}) = 1$$

$$P(W_i = \textit{I} | T_i = \text{noun}) = 0.3$$

$$P(W_i = \textit{you} | T_i = \text{noun}) = 0.2$$

$$P(W_i = \textit{love} | T_i = \text{noun}) = 0.2$$

$$P(W_i = \textit{saw} | T_i = \text{noun}) = 0.2$$

$$P(W_i = \textit{bank} | T_i = \text{noun}) = 0.1$$

$$P(W_i = \textit{love} | T_i = \text{verb}) = 0.4$$

$$P(W_i = \textit{saw} | T_i = \text{verb}) = 0.4$$

$$P(W_i = \textit{bank} | T_i = \text{verb}) = 0.2$$

In your probabilistic graphical model you should use the following naming convention: variables  $W_1, W_2, W_3, W_4, W_5$  should be represented by the strings  $w1, w2, w3, w4, w5$ , respectively. Similarly,  $T_1, T_2, T_3, T_4, T_5$  should be represented by the strings  $t1, t2, t3, t4, t5$ , respectively. The probabilistic graphical model itself must be called `pgm`. (If you don't respect this convention your solution will fail the associated unittests.)

Note that you have been asked to produce a single probabilistic graphical model to cope with sentences of varying length, but not told how to do this. Note also that the particular type of graphical model has not been specified. In both cases you have choices to make, be sure to justify them.

**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 Out of clique marginals in a join tree (15 marks)

Once a join tree has been calibrated it is easy to compute the marginal distribution of a set of variables, *as long as all the variables are contained in one of the cliques of the join tree*. We just find that clique and sum out any extra variables. In this question you are required to write a function that computes marginal distributions from calibrated join trees even when the variables in the marginal are not all together in some clique.

Specifically, define a class called `MyJFR` which extends `gPy.JFR` and defines a single method called `marginal`. Your new method `MyJFR.marginal` should behave exactly like the original `gPy.JFR.marginal` method except without the restriction previously discussed.

**The answer to this question should be submitted in a file called `agm2.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 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 Data structure for ‘loopy’ belief propagation (10 marks)

There is an algorithm called ‘loopy’ belief propagation which is similar to join tree calibration except that the underlying data structure does not have to be a tree. In loopy belief propagation, there are a number of possible message passing schedules and the marginals which result are generally only approximations to the true marginals. On the other hand, loopy belief propagation can be much faster than join tree calibration, since one can often get away with smaller cliques.

In this question you are not required to implement loopy belief propagation or investigate its properties, instead you are required to construct a data structure which could be used for loopy belief propagation. You are required to write a function which takes a hypergraph  $H$  as input and returns a graph  $G$  and a set of separators  $S$  as output with the following properties:

1.  $G$  should be undirected (like a join tree).

2. Each vertex of  $G$  is a subset of the variables of  $H$  (like a join tree).
3. Each separator  $s_e$  is associated with an edge  $e$  of  $G$  and is a subset of the variables of  $H$  (like a join tree).
4. For each hyperedge in  $H$  there should be a corresponding vertex in  $G$  (like a join tree for a decomposable hypergraph).
5. If  $c_1$  and  $c_2$  are two vertices in  $G$  connected by an edge  $e$  then  $s_e \subseteq c_1 \cap c_2$  (in a join tree  $s_e = c_1 \cap c_2$ ).
6. (A version of the running intersection property:) Let  $x$  be a variable of  $H$ , then if  $x \in c_1$  and  $x \in c_2$  where  $c_1$  and  $c_2$  are two vertices of  $G$  then there is a single path between  $c_1$  and  $c_2$  such that  $x \in s_e$  for every edge  $e$  in the path.

Specifically, write a function called `q3fun` in a Python module called `agm3.py` which takes a `gPy.Hypergraph.SimpleHypergraph` object  $H$  as input and outputs a pair  $(g, \text{seps})$  where  $g$  is a `gPy.Graphs.UGraph` object  $G$  and `seps` is a dictionary mapping edges of  $g$  to sets of variables of  $H$ . The edges of  $g$  should be implemented as `frozenset` objects (of size 2, of course).

**The answer to this question should be submitted in a file called `agm3.py`** Marking scheme: Passing all unit tests = 4 marks, Quality of code = 4 marks, Quality of write-up = 2 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`

## 4 Monte Carlo game position scoring (10 marks)

Some recent advances in game playing (e.g. chess, go) have exploited the surprising fact that random play from a given position provides a good evaluation of the strength of that position. One can use this to choose good moves: move to a position which has been evaluated to be a strong position.

Specifically, the strength of a position is the probability that you win from that position with both you and your opponent choosing uniformly at random between all legal moves at each turn. Since this probability is too difficult to compute exactly, it has to be approximated.

Your task here is to implement a program which uses an appropriate approximate approach to evaluating the strength of positions in noughts and crosses. Specifically, write a module called `agm4.py` which defines a function `xo_eval` which takes two

arguments: the first represents a noughts and crosses position and the second is a positive integer. The output is an estimate of the probability of you winning given that

1. you are in the input position,
2. you are playing crosses (X),
3. you make the next move and
4. both you and your opponent play randomly.

The second argument determines how much computing effort is used: larger values will generally result in better estimates, but the estimates will take longer to produce. A position is represented as a tuple of tuples. This position:

```

X | X |
-----
  | O | X
-----
O | O |

```

should be represented as follows:

`(( 'X', 'X', None), (None, 'O', 'X'), ('O', 'O', None))`. Your solution need not check that the input position is one that could be reached in a game.

**The answer to this question should be submitted in a file called `agm4.py`** Marking scheme: Passing all unit tests = 3 marks, Quality of code = 4 marks, Quality of write-up = 3 marks.

How to run the unit test for this question:

1. Copy `agm4_test.py` from the assessment directory to the directory containing your answer called `agm4.py`.
2. Type `python agm4_test.py`

