

THE UNIVERSITY *of York*

BSc, BEng, MEng and MMath Examinations Examinations ,2009

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

Part A

ALGORITHMS FOR GRAPHICAL MODELS (AGM)

Open Examination

Issued at:

9 December 2009

Submission due:

20 January 2010

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/09/>, 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 (Weighted) satisfiability (10 marks)

Propositions in propositional logic can be viewed as binary variables taking either the value true (1) or false (0). A *clause* in propositional logic puts a constraint on permissible instantiations of these binary variables. For example the clause $x_1 \vee \neg x_2 \vee x_3$ states that either x_1 is true or x_2 is false or x_3 is true. In the *satisfiability problem for propositional logic*, usually just called SAT, the problem is to decide whether there is any joint instantiation of the propositional variables that satisfies all the given clauses. Such an instantiation is called a *model* (Don't confuse this with a statistical model!) For example, the set of clauses in Fig 1 is satisfiable by setting $x_1 = 1, x_2 = 0, x_3 = 1$ (so this is a model for the set of clauses). On the other hand the set of clauses in Fig 2 is not satisfiable (has no models).

$$\begin{aligned}
 &x_1 \vee \neg x_2 \vee x_3 \\
 &\neg x_2 \vee \neg x_3 \\
 &\neg x_2 \vee x_3 \\
 &\neg x_3 \vee x_1
 \end{aligned}$$

Figure 1: A satisfiable set of clauses

A satisfiable set of clauses defines a probability distribution over all instantiations of the propositional variables. Let Z be the number of models of the clauses, then for any given instantiation of the propositional variables \mathbf{x} , set $P(\mathbf{x}) = Z^{-1}$ if \mathbf{x} is a model and set $P(\mathbf{x}) = 0$ otherwise.

One can define more flexible distributions using *weighted clauses* where each clause is annotated with a positive integer weight or with ∞ . Intuitively, the weight represents how good it would be to satisfy the clause. Clauses annotated with ∞ are called

$$\begin{aligned}
& x_1 \vee \neg x_2 \\
& \neg x_2 \vee \neg x_3 \\
& \neg x_2 \vee x_3 \\
& \neg x_3 \vee x_1 \\
& x_2
\end{aligned}$$

Figure 2: An unsatisfiable set of clauses

$$\begin{aligned}
2 & : x_1 \vee \neg x_2 \\
4 & : \neg x_2 \vee \neg x_3 \\
\infty & : \neg x_2 \vee x_3 \\
2 & : \neg x_3 \vee x_1 \\
1 & : x_2
\end{aligned}$$

Figure 3: Weighted clauses

hard clauses—they must be satisfied—the remaining clauses are *soft* clauses. Fig 3 shows some example weighted clauses.

Suppose we have a set of weighted clauses

$$w_1 : C_1, w_2 : C_2, \dots, w_k : C_k$$

then define a probability distribution

$$P(\mathbf{x}) = Z^{-1} \exp \left(\sum_{\mathbf{x} \not\models C_i} -w_i \right) \quad (1)$$

where $\mathbf{x} \not\models C_i$ means that \mathbf{x} fails to satisfy clause C_i and Z is a normalising constant. To understand this distribution, consider a joint instantiation \mathbf{x} that fails to satisfy ('breaks') a hard clause. In this case a $-\infty$ will appear in the sum in (1). Since $\exp(-\infty) = 0$, then $P(\mathbf{x}) = 0$. On the other hand consider an instantiation that satisfies all clauses. In this case the sum is empty and so equals zero and $P(\mathbf{x}) = Z^{-1}$, the highest possible probability.

There is a standard format for writing weighted clauses called WCNF format (*weighted conjunctive normal form*). Fig 4 shows the WCNF file for the weighted clauses in Fig 3. The header line states there are 4 variables and 5 clauses. The first number in each line is the weight of the clause. Note that there is no direct representation of ∞ , the highest weight in the input (here 99) is taken to represent an infinite weight and so indicate hard clauses. Note that this means that will always be at least one hard clause. Each

```

p wcnf 4 5
2 1 -2 0
4 -2 -3 0
99 -2 3 0
2 -3 1 0
1 2 0

```

Figure 4: Weighted clauses from Fig 3 in WCNF format

variable is represented by a positive number and 0 is used to delimit the clause. If a variable is negated in the clause a minus sign is given to the corresponding integer in the WCNF file.

Your main task in this question is to write a Python program that reads in a file in WCNF format and constructs a factored representation, specifically a `FR` object, representing the distribution defined by the WCNF file.

So far we have just assumed that (1) defines a probability distribution for any set of weighted clauses, but, in fact, there are sets of weighted clauses such that (1) fails to define a probability distribution. In your write-up state in which circumstances this happens. (When constructing the `FR` object you do not need to check that the `FR` object defines a proper probability distribution.)

The answer to this question should be submitted in a file called `agm1.py` Marking scheme: Passing all unit tests = 4 marks, Quality of code = 3 marks, Quality of write-up = 3 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 Computing probabilities of specific partial instantiations (25 marks)

In this question instead of computing a marginal distribution (which is a collection of numbers) you are required to write a function called `compprob` which computes a single probability. Specifically, given:

1. a factored representation of a probability distribution (as a `FR` object)
2. an instantiation $(x_1 = v_1, \dots, x_j = v_j)$ of a subset of the variables (as a dictionary mapping each key x_i to the value v_i)

your function `compprob` should compute $P(x_1 = v_1, \dots, x_j = v_j)$ up to a normalising factor. In other words, you need only compute $ZP(x_1 = v_1, \dots, x_j = v_j)$ for some uncomputed constant Z . This means that two runs of your program can be used to compute exactly any ratio of probabilities such as $P(x_1 = v_1, \dots, x_j = v_j)/P(x'_1 = v'_1, \dots, x'_j = v'_j)$ since the unknown Z value cancels out.

I am looking for a solution which is:

- time efficient
- space efficient
- with good average case behaviour
- with good worst case behaviour

Evidently, you may have to make trade-offs between these desiderata. Discuss and justify any such trade-offs as clearly as possible.

You are not the first people to work on this problem. You are permitted to use/adapt existing ideas. 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 = 5 marks, Quality of code = 10 marks, Quality of write-up = 10 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 Random walks in grid world (15 marks)

In this question we imagine a robot wandering around in a bounded environment until it reaches either a location called SUCCESS or an alternative location called FAILURE. The goal is to use sampling to estimate the probability that the robot ends up in the SUCCESS location from any given possible starting point.

The robot lives in a grid world: there are only 10,000 possible locations each of which is given by a non-negative integer co-ordinate (i, j) where $0 \leq i, j \leq 99$. The robot moves in discrete steps: at each time point it moves to a neighbouring grid location using a uniform distribution. So, for example, if the robot were at $(23, 45)$ at time t then at time $t + 1$ it would be at one of the following locations: $(23, 46)$, $(23, 44)$, $(24, 45)$ or $(22, 45)$ each with probability $1/4$. If it were at $(0, 0)$ then it will move to either $(0, 1)$ or $(1, 0)$ with equal probability. Note that diagonal moves are not allowed and that there is zero probability of the robot not moving.

Write a program called `agm3.py` such that when called as follows:

```
python agm3.py n si sj fi fj dumpfile.pck
```

estimates of the probability of reaching success for all possible starting points are recorded in `dumpfile.pck`, where `si, sj` is the location of SUCCESS and `fi, fj` is the location of FAILURE. `n` should be an integer such that larger values of `n` produce more reliable estimates of the probabilities at the expense of more time.

The file `dumpfile.pck` should be produced by a call to `cPickle.dump`. It should contain a data structure such that the following interaction:

```
>>> import cPickle
>>> p = cPickle.load(open('dumpfile.pck'))
>>> p[4][66]
0.51106519493230385
>>>
```

produces the probability of reaching success from location (4,66).

You can use the program `display_probs.py` to visualise your results. To see how it works take a look at the source and also run

```
python display_probs.py foo.pck
```

The file `foo.pck` was produced by my model answer as follows:

```
python agm3.py 1 20 20 70 70 foo.pck
```

As well as describing and justifying your sampling approach (4 marks) your write-up should include two further items:

1. If the robot reaches SUCCESS we can imagine it staying there forever; similarly for FAILURE. The process driving the robot then becomes a Markov chain. Does this Markov chain have a unique stationary distribution? (1 mark)
2. The Markov chain defines, for any given start location other than FAILURE, a probability distribution over trajectories (sequences of locations) which end up in the SUCCESS state, conditional on reaching this state. Denote the probability of such a trajectory $s_{t=0}, s_{t=1}, \dots, SUCCESS$ under this conditional distribution by $P(s_{t=0}, s_{t=1}, \dots, SUCCESS | \text{reaches SUCCESS})$. Define, if possible, a new Markov chain such that *all* trajectories not starting at FAILURE reach the SUCCESS state with probability one and where

$$P(s_{t=0}, s_{t=1}, \dots, SUCCESS | \text{reaches SUCCESS}) = P'(s_{t=0}, s_{t=1}, \dots, SUCCESS)$$

where P' is the distribution over trajectories defined by the new Markov chain. (3 marks)

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 = 5 marks, Quality of write-up = 8 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`

