COMP114 - Assessment 4
Semester 2 – 2011

Phase Transitions

Assessment Information

Assessment Number	4
Contribution to Overall Mark	40%
Submission deadline	**Friday 20th May 2011, 16.00**

Relevant Learning Outcomes for this Assessment

1	Awareness of different areas of CS for which experimental methods are relevant.
3	Have an understanding of the factors involved in constructing experiment settings.
5	Ability objectively to assess, analyse, and present experimental results.

Assessment Description – Overview

A major factor in the recent expansion of experimental work in Computer Science, came about through the observation of program behaviour in the following contexts: a number of normally computationally challenging problems (in the sense of taking an unreasonable amount of time to deliver results) turn out to be quickly solvable if the instance obeys some extreme characteristics, e.g. some "hard" problems involving graphs turn out to be "easy" for graphs with "very few" or with "very many" edge. This phenomenon – known as a *phase transition effect* has been the subject of much interest particularly in Artificial Intelligence, Logic, and algorithm study over the past decade.

The principal aim of the current assignment is for you to investigate the existence of such effects with respect to a widely studied problem in logic: the so-called 2-*satisfiability* problem (2-SAT).

Assessment Description – Details

A 2–CNF is a special type of *logical expression* defined using a set of *Boolean variables* – $X = \{x_1, x_2, \ldots, x_n\}$, i.e. variables which can be assigned either the value **true** or **false**. Such an expression is specified by a collection of (unordered) *pairs* – (y, z) – (called *clauses*) where each member of a pair can be some positive form of a variable (e.g. x_1) or its negation (e.g. $\overline{x}_3$). For example with the set of variables $\{x_1, x_2, x_3\}$,

$$\begin{array}{ll} \{(x_1, x_3),\ (\overline{x}_2, x_3),\ (x_2,\ \overline{x}_3)\} & (a) \\ \{(x_1, x_2),\ (\overline{x}_1, x_2),\ (\overline{x}_2,\ x_3),\ (\overline{x}_2, \overline{x}_3)\} & (b) \end{array}$$

are both examples of 2-CNF expressions.

The so-called 2-SAT problem asks of a 2-CNF expression, $F(X)$, if there is an assignment of **true**/**false** values to the variables in X that results in *every* pair defining $F(X)$ containing *at least one* **true** term – recall that if $x =$ **false** then $\overline{x} =$ **true**. So, for example (a), the assignment $x_1 =$ **true**, $x_2 =$ **false**, $x_3 =$ **false** has this property. In contrast no such assignment is possible with example (b): no matter which value is assigned x_1, x_2 *must* be assigned **true** but this will leave it impossible to find a value for x_3 to use in the clauses $\{(\overline{x}_2,\ x_3),\ (\overline{x}_2, \overline{x}_3)\}$.

The problem of distinguishing *satisfiable* logical expressions (those for which a suitable assignment exists) from *unsatisfiable* expressions is a fundamental problem in many areas of Computing e.g. Artifical Intelligence (intelligent reasoning), Software Development (program verification), the study of fast algorithms, etc. Unfortunately, despite its prevalence, the most general versions of this problem are unlikely to be solvable by a fast algorithm: the best existing methods being unreasonably slow in the worst-case for more than a few hundred variables.

Despite this it *could* be possible to exploit some "structural" features of "typical" logical expressions. Concentrating on 2-CNF, the current assessment concerns the following informal claim:

Claim 1 *If a 2-CNF, $F(X)$ has a "small number of clauses" (pairs in its definition), then since there seems to be greater freedom in choosing values for variables such an expression* ought *typically to be satisfiable. On the other hand, if $F(X)$ has "a lot of clauses", then since there seem to be much greater restriction on how variables can be assigned, one would expect such an expression* not *to be satisfiable.*

The high-level aim for this assignment is to consider what experimental support there is for this assertion in the light of the following three questions:

Q1. Is the claim above, actually *justified*, i.e. are the majority of "small" 2-CNF expressions satisfiable while the majority of "large" 2-CNF expressions are unsatisfiable?

Q2. If the answers to Q1 are positive, is there an observable pattern describing the shift from "small-to-large", i.e. how many clauses are needed to be confident that a 2-CNF is unlikely to be satisfiable? what is the maximum number of clauses where we could be confident that it *is* satisfiable?

Q3. Again assuming the answers to Q1 are positive, how does such behaviour affect the *average* amount of time taken by an algorithm to decide whether a 2-CNF is satisfiable or not?

Using the supporting Java classes provided and described in the detailed description, the purpose of this assignment is to propose not only answers to these three questions but also to support those answers with experimental findings.

Assessment - Details

The module resource page provides the Java source code – CNF2.java – which implements a series of methods for

a. Generating a *random sequence* of n variable 2-CNF expressions.

b. Testing whether the (current) instance is satisfiable and reporting the time taken to determine this.

Although this contains a number of fields the *only* fields that are important for your experimental study are:

Type	Name	Meaning
int	n	The number of variables defining $F(X)$, i.e. $\|X\|$
boolean	*satis*	Records whether the current $F(X)$ is satisfiable or not
int	*steps*	Records the number of steps taken to decide if $F(X)$ is satisfiable

The single constructor – $CNF2($**int** $n)$ sets the number of variables ($|X|$) to be used in the sequence of random 2-CNFs.

The *only* methods you need to use are

a. **public void** $SatTest()$

b. **public void** $nextRandomCNF($**double** *cp*$)$

c. **public boolean** $IsSat()$

d. **public int** $StepCount()$

The method $SatTest()$ decides whether the current 2-CNF stored is satisfiable or not (satting the field *satis* to be **true** or **false** accordingly) and updates the the value of *steps* to record how long this decision took to make.

The method $nextRandomCNF($**double** *cp*$)$ (in which *cp* should be a value $0 \leq cp \leq 1$) constructs a new *random* 2-CNF expression as follows: given that there are $2n(n-1)$ *distinct* choices for the pairs defining any clause, each of these possible choices is considered in turn. On the k'th choice – for each $1 \leq k \leq 2n(n-1)$ – a random **double** value (between 0 and 1) is chosen; if this value is $\leq$ *cp* the k'th clause is added to the 2-CNF being generated, otherwise it is ignored.

Notice that the 2-CNF formed in this method will have "roughly" $(cp) \times 2n^2$ clauses.

Experiment Details

The main body of the program implementing your experimental study should do the following,

1. Read in a value of n (this can be supplied by the user).
2. Read in an **int** value, $Trials$, for the number of experiments to run for a *specific* choice of *cp*.
3. Create an instance of the $CNF2$ class, $F =$ **new** $CNF2(n)$.
4. For a **suitable range of values of** *cp*, using $Trials$ experiments for each value
   a. Create the next random 2-CNF, by calling *F.nextRandomCNF*(*cp*).
   b. Test whether the 2-CNF just formed is satisfiable by calleing *F.SatTest*().
   c. Add *F.StepCount*() to the *total* number of steps $Total$ taken (with the current value of *cp*).
   d. If *F.IsSat*() is true then increase the number of satisfiable cases (*SatCases*) found (with the current value of *cp*) by 1.
5. Output (to a file!) the values of *cp* and the **double** value $SatCases/Trials$.
6. Output (to a *different!* file) the values of *cp* and the **double** value $Total/Trials$.
7. Repeat (a)–(d) (and 5 and 6) with next value for *cp*.

Having implemented the experimental framework just described you should then carry out the following.

A. Generate output from the experiment in $2 \times K$ files (where K is the number of distinct values of n used). You should carry out the experiment using *at least* three *very different* and *moderate size* values of n (with values differing by *at least* 10), e.g. $n = 100,\ 250,\ 500$ would be suitable, however, $n = 20, 21, 22$ would *not*.

B. Using the **gnuplot** script provided (you will have to edit this to reflect the filenames you've chosen for output) generate *two* graphical outputs: the first showing how the proportion of satisfiable CNFs found varies with the range of values for *cp* in your experiments (hence, the y-axis has a value between 0 and 1); the second showing how the average number of steps changes with with the range of values for *cp* used.

C. Based on the results obtained you should then discuss the following:

 C1 What (if any) distinctive features do you see in the output plots generated in (B)?

 C2 Do these features provide support or rebuttal of the assertion in Claim 1?

 C3 What relationship(s) are noticeable between the average *time* graph and the proportion of satisfiable cases graph?

Hints and Suggestions

Output to a named file

The **java.io** package provides a suite of methods to output to several files within a single program. The following code will set up output streams that can be used to collect experimental data from your program:

```
//******************************************************************
// Start of main program -- remember to include the IOException
//*****************************************************************
public static void main(String[] args) throws IOException {
  public static FileWriter PropnSat;
  public static PrintWriter AvSat;
  public static FileWriter NumSteps;
  public static PrintWriter StepsData;
  //**************************************************
  int n   // For number of variables in CNF2 constructor
  //**************************************************
  //  Once n has been read in (from Standard input)
  //  define two Strings for the output files as follows
  //**************************************************
  String SatD = "SatStats"+String.valueOf(n);
  String StepD = "SatTime"+String.valueOf(n);
  //****************************************************
  // The output streams are then created by
  //***************************************************
  PropnSat = new FileWriter(SatD);
  AvSat = new PrintWriter(PropnSat);
  NumSteps = new FileWriter(StepD);
  StepsData = new PrintWriter(NumSteps);
  //****************************************************
  // You then have the same methods as available with System.out,
  // e.g if n=100
  //********************************************************
  AvSat.println(.......);       // will print to a file called SatStats100
  StepsData.println(.......);   // will print to a file called SatTime100
//*****************************************************************
// !!!!!!!  EXTREMELY IMPORTANT !!!!!!!!
//*****************************************************
// The last thing that should be done in your main() program is to ''flush''
// the output streams AND ''close'' the files, i.e. in the example above
// the statments below MUST be included.
//*****************************************************
AvSat.flush(); AvSat.close();
StepsData.flush(); StepsData.close();
}
```

Choosing the probability range and number of Trials

As was mentioned earlier, the instance method $nextRandomCNF$(**double** *cp*) constructs a random 2-CNF expression using n variables, by including each of the $2n(n-1)$ clauses (pairs) with probability *cp*. If you only consider *constant* values of *cp*, e.g $cp \in \{0.1, 0.2, \ldots, 0.9\}$ then the all of the resulting random 2-CNF expressions *will be "large"* (particularly if n itself is moderately sized, e.g. $n \geq 20$). In order to obtain "meaningful" results, the range of probabilities should be allowed to change *as a function of n*. A suitable range (from "small" to "large" 2-CNFs) is obtained by fixing $cp \in \{0.05/n, 0.1/n, 0.15/n, \ldots, 3.0/n\}$, e.g. use a **for** loop

```
int interval=0.05;
for (int ProbRange=1; ProbRange<61; ProbRange++)
  {
  cp = (interval*(double)ProbRange)/((double)n);
  Total=0 ; SatCases=0;
  for (int i=0: i<Trials; i++)
    {
    Generate and test next random CNF2
    Update data (i.e. Total and SatCases)
    };
  Output data to relevant files
  };
```

If you choose a number of trials which is very large combined with a moderate size for n, e.g. $n \geq 750$ and $Trials > 500$ you will probably find that the program takes a while to deliver its results. Assuming your code is correct, this is pefectly normal: the 2-SAT solver implemented in the class is *CNF2* is not (nor is it intended to be) the most efficient possible. It will be able to generate data for cases such as $n = 1000$ and $Trials = 50$ within around 10 minutes run-time.

Using gnuplot

gnuplot is a Unix graphical tool that allows data to be plotted in a variety of styles. The module resource page contains a script – *PlotSatData* – (which you may have to edit depending on the choice of files for your output and range of sizes chosen) which will generate 2 postscript files: **SatProbability.ps** and **SatTime.ps**. **gnuplot** can be invoked (within Unix) by giving the command **gnuplot<PlotSatData**. (When you run this script ignore the noise text will appear on the page).

What should be Submitted

a. The java source code of your main experimental program.

b. A description of the experimental framework used, i.e. range of values of n, range of probabilities examined, the number of trials carried out with each case.

c. The graphical plots – **SatProbability.ps** and **SatTime.ps** – generated by your experimental data.

d. Your answers to the questions (C1), (C2), and (C3) and a justification for these based on your experimental findings.

e. A completed and signed *Declaration On Plagiarism and Collusion Form.*

How the work should be submitted

Items (a–e) should handed in to the *Student Office.* A cover sheet should indicate all of the following information:

a. The Assessment *number.*

b. Your **name** and University **e-mail address**

c. Your lab group.

d. The name of the *demonstrator/tutor* responsible for this group.

e. Your degree programme, e.g. G400 Computer Science, G500 Computer Information Systems.