CSCI 561: Foundations of Artificial Intelligence
Instructor: Prof. Laurent Itti
Homework #3: Logic and SAT Problem
Due on April 10 at 11:59pm, Los Angeles time, 2013

Suppose you have a wedding to plan, and want to arrange the wedding seating for a certain number of guests in a hall. The hall has a certain number tables for seating. Some pairs of guests are couples or close Friends (F), and want to sit together at the same table. Some other pairs of guests are Enemies (E), and must be separated into different tables. The rest pairs are Indifferent (I) with each other, and do not mind sitting together or not. However, each pair of guests can ***only*** have one relationship, (F), (E) or (I). You need to find a seating arrangement that satisfies all the constraints.

Task 1: Logic and SAT Encoding (15 pts)

Suppose there are <*M*> guests in total, and there are <*N*> tables in the hall. The number of pairs of Friends is <*F*>, and the number of pairs of Enemies is <*E*>. Given relationships of the wedding guests, here we use a matrix R with elements $R_{ij} = 1$, -1 or 0 to represent whether guest i and j are Friends (F), Enemies (E) or Indifferent (I), the table arrangement task can be represented as First-order Logic (FOL) and further encoded as a Boolean Satisfaction problem (SAT). We introduce Boolean variables X_{mn} to represent whether each guest m will be seated at a specific table n. You are asked to write FOL to represent the constraints using variables X_{mn}. Then you need to construct clauses and transform the FOL into CNF sentence. To decompose the arrangement task, there are three constraints you have to satisfy:

(a) Each guest i should be seated at one and ***only*** one table.

(b) Any two guests i and j who are Friends (F) should be seated at the same table.

(c) Any two guests i and j who are Enemies (E) should be seated at different tables.

Note that, for simplicity, you do **NOT** need to consider the capacity constraint of a table. This means the size of each table is assumed to be large enough to seat all the guests.

Question 1: Express each of the above-mentioned constraints in FOL and transform into clauses in CNF format. How many clauses in total are there to encode a wedding seating arrangement in terms of <*M*>, <*N*>, <*E*> and <*F*>?

Task 2: Instance Generator (20 pts)

As the first part of this programming assignment, you will need to write a program to generate CNF sentences for random instances of wedding seating arrangements. Key parameters of the program should include the number of guests <*M*>, and the number of tables <*N*>. Moreover, you need to randomly assign each pair of guests as Friends (F), Enemies (E) or Indifferent (I). Suppose any two guests are Friends with probability <*f*>, or are Enemies with probability <*e*>. And any two guests are Indifferent to each other with probability <*1-f-e*>.
Given these parameters <*M, N, f, e*>, the generator should firstly produce a relationship for each pair of guests. The ***internal*** output of the program should be a *M* by *M* matrix *R* with elements R_{ij} = 1, -1 or 0 to represent whether guest *i* and *j* are Friends (F), Enemies (E) or Indifferent (I).

Then your program should convert any generated instance of wedding seating arrangement into a CNF sentence. You are free to use whatever internal representation of CNF sentences. You will NOT be asked to input or output sentences for the user for this assignment. In general, it is a good idea to use the most efficient representation possible, given the NP-complete nature of SAT. For instance, in C++, you can represent a CNF sentence as a vector of clauses, and represent each clause as a vector of literals.

Task 3: SAT Solvers (20 pts)

You need to implement SAT solvers to find a satisfying assignment for any given CNF sentences. Firstly, you need to implement a modified version of the **PL-Resolution** algorithm (AIMA Figure 7.12). Modifications are necessary because we are using the algorithm for a slightly different purpose than is explained in AIMA. Here, we are not looking to prove entailment of a particular query. Rather, we hope to prove satisfiability. Thus, there is no need to add negated query clauses to the input clauses. In other words, the only input to the algorithm is the set of clauses that comprise a randomly generated sentence. As an additional consequence of our purpose, the outputs will be reversed compared to the outputs listed in AIMA's pseudo code. That is to say, if the empty clause is derived at any point from the clauses of the input sentence, then the sentence is unsatisfiable. In this case, the function should return ***false*** and not ***true*** as the book specifies for this situation. In the opposite situation where the empty clause is never derived, the algorithm should return ***true***, indicating that the sentence is satisfiable.

You are also asked to implement the **WalkSAT** algorithm (AIMA Figure 7.18). There are many variants of this algorithm exist, but yours should be identical to the algorithm described in AIMA. There are two open parameters associated with WalkSAT: <*p*> and <*max_flips*>.
PL-Resolution is a sound and complete algorithm that can be used to determine satisfiability and unsatisfiability with certainty. On the other hand, WalkSAT can determine satisfiability (if it finds a model), but it cannot asbolutely determine unsatisfiability.

Task 4: Experiment 1 (15 pts)

The difficulty of wedding seat arrangement problem mostly results from dealing with the pairs of Enemies (E) among guests. Use both algorithms to produce a plot of *P*(satisfiability) (See Figure 7.19(a), AIMA) as a function of the probability <*e*> with which any two guests are Enemies (E) .

Suppose we have a small wedding to plan, and set <*M=16, N=2*>. In order to eliminate the influence of Friends (F) relationship, we set <*f=0*>. Generate a set of *100* random sentences for each setting of <*e*> which increases from *2%* to *20%* at an interval of *2%*, and use both of your algorithms to determine whether they are satisfiable. For WalkSAT, we set <*p=50%>* and <*max_flips=100*>. Plot the results of *P*(satisfiability) versus <*e*> for both algorithms on the ***same*** graph, so that you can compare.

Question 2: Compare the curves that result from running this experiment with both algorithms. Are they the same? Why, or why not?

Task 5: Experiment 2 (15 pts)

You may have discovered that determining satisfiability with PL-Resolution can be frustratingly slow! This was part of the motivation behind developing local search algorithms like WalkSAT. For easy SAT problems, it can be very easy to find a model using a random walk through the possibilities. Using the relative speed afforded by WalkSAT, we can run experiments that might otherwise be impractical.

Suppose you want to know the effect of the probability <*f*> with which any two guests are Friends (F) on *P*(satisfiability). We set <*M=16, N=2*> as in Experiment 1 and fix <*e=5%*>. Using ***only*** the **WalkSAT** algorithm with parameter setting <*p = 50%, max_flips=100*>, run *100* random instances using different settings for <*f*>: *2%* through

20% at an interval of *2%*. You also need to vary <*max_flips*> and compare the results. Increasing <*max_flips*> to *500* and *1000*, run WalkSAT again respectively using the ***same*** random instances as we set <*max_flips=100*>. Produce a plot of *P*(satisfiability) versus <*f*> for each setting of <*max_flips*> on the ***same*** graph.

Question 3: What seems to happen to the satisfiability as <*f*> increases? Give an explanation as to why this might be the case.

Question 4: How does the result vary with different <*max_flips*>? Why, or why not?

Task 6: Experiment 3 (15 pts)

The difficulty of the wedding seat arrangement also depends on how large the wedding is (how many guests and how many tables). We now restrict our analysis to only satisfiable sentences with different settings of <*M, N*>. We only record runtimes for sentences for which a model is found before WalkSAT reaches *max_flips* iterations. Any sentences for which no model is found in *max_flips* iterations should be deemed unsatisfiable and eliminated from the results.

We set <*f=2%, e=2%*> throughout this experiment, and increase the size of the wedding by sequentially setting <*M, N*> as <*16, 2*>, <*24, 3*>, <*32, 4*>, <*40, 5*> *and* <*48, 6*>.
Fixing <*p=50%, max_flips=1000*>, run ***only*** the **WalkSAT** algorithm for random instances until *100* ***satisfiable*** CNF sentences are generated for each setting of <*M, N*>. You are asked to record the runtime it takes at average to determine their satisfiability status. Runtime is measured by counting the number of iterations through the WalkSAT algorithm (i.e., following in AIMA Figure 7.18, what is the iterator *i* when finished?). Record the ratio of the number of clauses to the number of symbols for each satisfiable sentence as well.

Produce a plot of the average runtime versus the average ratio of clause/symbol for each above-mentioned setting of <*M, N*>, but ***only*** over the *100* ***satisfiable*** sentences.

Question 5: Is the average ratio of clause/symbol in the sentences consistent with that you theoretically derive from the result of Question 1? Why or why not? You need to consider the probability setting <*f=2%, e=2%*> in this case.

Question 6: How does average runtime change with regard to the average ratio of clause/symbol in this experiment? Is the curve consistent with that of AIMA Figure 7.19(b)? Why or why not?

Programming Guideline:

In this assignment, you ***have to*** write code in C++. Any other languages are ***not*** allowed. One major consideration is that languages such as Java may take much more time to complete the experiments, or even you cannot finish them in many hours.

However, you might find the existing code of **PL-Resolution** and **WalkSAT** useful in the link: http://code.google.com/p/aima-java/.
Please note the WalkSaT in the link is ***not*** correctly implemented. You have to refer to AIMA book to revise it.

Grading:

The grader will compile your program using the following command:
g++ *.cpp -o hw3
And execute your program using, for examples for each experiment:
hw3 -exp1
hw3 -exp2
hw3 -exp3
The output should be the data you use for the plots.

Deliverables:

You are required to hand in concisely documented code that implements the specified programs. Implementation language is not important. However, you are asked to choose C++ for the efficiency reason. You also need to hand in plots and answers to all questions in a separated PDF document. Please keep the answers ***concise***. Please include a readme.txt to write any comments you may have.

The deadline for this assignment is **Wednesday April 10, at 11:59pm** Los Angeles time. Please turn in all materials as a .zip file via Blackboard, with the title format [firstname]_[lastname]_PA2.zip (e.g., James_Bond_HW3.zip).