CSCI 561: Foundations of Artificial Intelligence
Homework #2: Wedding Seating Arrangement
Due at 11:59pm on 3/6/2017

Problem Description

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 of 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 of the pairs are Indifferent (I) to each other and do not mind sitting together or not. However, each pair of guests can have ***only*** one relationship, (F), (E) or (I). You must find a seating arrangement that satisfies all the constraints.

SAT Encoding

To decompose the arrangement task, there are three constraints you have to satisfy:
(a) Each guest should be seated at one and ***only*** one table.
(b) For any two guests who are Friends (F), you should seat them at the same table.
(c) For any two guests who are Enemies (E), you should seat them 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.

The arrangement task can be encoded as a Boolean satisfaction problem. We introduce Boolean variables X_{mn} to represent whether each guest m will be seated at a specific table n. You are asked to construct clauses and generate CNF sentence for each instance of the seating arrangement. Suppose there are <M> guests in total, and there are <N> tables in the hall. You may assume each table has an **unlimited** capacity.

You need to express each of the above-mentioned constraints as clauses in CNF format.

Hint: in the following, we provide one possible CNF modeling.

(a) For each guest a, the assignment should be at only one table

$$\bigvee_{1\le i\le N} x_{ai}$$

$$\bigwedge_{1\le i<j\le N}[\neg(x_{ai}\wedge x_{aj})]$$

(b) For each pair of friends, guest ***a*** and guest ***b***.

There are two ways to encode friendships.

1. ***a*** and ***b*** cannot sit at any two different tables.

   $$\bigwedge_{i\ne j}[\neg(x_{ai}\wedge x_{bj})]$$

   there are FA_N^2 clauses.

2. ***a*** and ***b*** should sit at the same table.

   $$\bigwedge_{1\le i\le N}(x_{ai}\Leftrightarrow x_{bi})\equiv\bigwedge_{1\le i\le N}[(\neg x_{ai}\vee x_{bi})\wedge(x_{ai}\vee\neg x_{bi})]$$

(c) For each pair of enemies, guest ***a*** and guest ***b***

$$\bigwedge_{1\le i\le N}[\neg(x_{ai}\wedge x_{bi})]$$

Programming Task: SAT Solver

As part of this programming assignment, you will need to write a program to generate CNF sentences for an input instance of wedding seating arrangements. The inputs include the number of guests <*M*>, the number of tables <*N*>, and a sparse representation of the relationship matrix *R* with elements R_{ij} = 1, -1 or 0 to represent whether guests *i* and *j* are Friends (F), Enemies (E) or Indifferent (I). You are free to use any 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 Python, you can represent a CNF sentence as a list of clauses, and represent each clause as a list of literals.

You are also asked to implement a SAT solver to find a satisfying assignment for any given CNF sentences. In this homework, 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) to search for a solution for an instance of wedding. There are many variants of this algorithm that 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 absolutely determine unsatisfiability. If your **PL-Resolution** determines the sentence is satisfiable, then you have to run your **WalkSAT**, and tune the parameters to find at least one solution.

Sample Input

```
4 2
1 2 F
2 3 E
```

The first line contains two integers denoting the number of guests <*M*> and the number of tables <*N*> respectively. Each line following contains two integers representing the indices of a pair of guests and one character indicating whether they are Friends (*F*) or Enemies (*E*). The rest of the pairs are indifferent by default. For example, in the above sample input, there are 4 guests and 2 tables in total, and guest 1 and guest 2 are Friends, and guest 2 and guest 3 are Enemies.

Sample Output

```
yes
1 2
```

2 2
3 1
4 1

A single line output *yes/no* to indicate whether the sentence is satisfiable or not. If the sentence can be satisfied, output ***yes*** in the first line, and then provide **just one** of the possible solutions. (Note that there may be more than one possible solution, but again, your task is to **provide only one** of them.) Each line after "yes" contains the assigned table for a specific guest for the solution. For example, in the sample output, line 2 represents guest 1 has been assigned at table 2. Please note that the output lines for assigning tables to guests should be in **ascending order of indices** (1,2,3,..., M). Lastly, If the sentence can not be satisfied, output **only a single line *no***.

Suggestions:
You might find the existing code of PL-Resolution and WalkSAT useful: http://code.google.com/p/aima-java/

Homework Rules

1. Please follow the instructions carefully. Any deviations from the instructions may lead your grade to be zero for the assignment. If you have any questions, please use the discussion board. Do not assume anything that is not explicitly stated.

2. You must use Python 2.7 to implement your code. You are allowed to use standard libraries only. You have to implement any other functions or methods by yourself.

3. You need to create a file named "hw2cs561s2017.py". The command to run your program would be as follows: (When you submit the homework on labs.vocareum.com, the following commands will be executed.)
   python hw2cs561s2017.py
   You need to read *input.txt* as *file = open("input.txt", "r"), your code.*

4. You will use labs.vocareum.com to submit your code. Please refer to http://help.vocareum.com/article/30-getting-started-students to get started with the system. Please only upload your code to the "/work" directory. Don't create any subfolders or upload any other files.

5. If we are unable to execute your code successfully, you will not receive any credit.

6. For your final submission, please do not print any logs on the console other than the required output.

7. When you press “Submit” on Vocareum, your code will be run against the submission script and the sample input/output files. You will receive feedback on where your code is making errors, if any. You can click “Submit” to test new versions of your code as many times as you like up until the deadline. Only your last submission will be graded. The sample input/output files are designed to cover many basic situations of the problem and rules of the output log, so please utilize them as much as you can.

8. Your program should handle all test cases within a reasonable time (not more than a few seconds for each sample test case). The complexity of the grading test cases will be similar to or less than the five example test cases.

9. You are strongly recommended to submit at least two hours ahead of the deadline, as the submission system around the deadline might be slow and possibly cause late submission, and late submissions will **NOT** be graded.