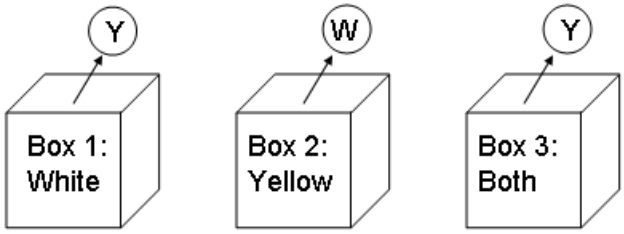
CSCE 625
Programing Assignment #4
due: Tuesday, Nov 3 (by start of class)

Propositional Theorem Prover using Resolution Refutation

The goal of this assignment is to implement a Resolution theorem prover that can be used to make inferences from a propositional knowledge base (KB), using proof by negation. Specifically, given a set of sentences KB, determine whether a query q is entailed, KB |= q? You will write a program that can read in a file that contains a set of clauses in Conjunctive Normal Form (CNF). This file will consist of the initial KB and the negation of the desired query (¬q). Your program must then perform resolution until either you derive the empty clause (success; entailed) or run out of resolutions (failure, q not entailed). You will apply this program for automated reasoning to Sammy's Sport Shop and prove that the middle box must contain white tennis balls. We will test your program on other propositional reasoning tasks too, so your program must be able to accept any file in the format discussed below as input.

Sammy's Sport Shop

You are the proprietor of Sammy's Sport Shop. You have just received a shipment of three boxes filled with tennis balls. One box contains only yellow tennis balls, one box contains only white tennis balls, and one contains both yellow and white tennis balls. You would like to stock the tennis balls in appropriate places on your shelves. Unfortunately, the boxes have been labeled incorrectly; the manufacturer tells you that you have exactly one box of each, but that each box is definitely labeled wrong. One ball is drawn from each box and observed (assumed to be correct). Given the initial (incorrect) labeling of the boxes above, and the three observations, use Propositional Logic to derive the correct labeling of the middle box.



- Use propositional symbols in the following form: O1Y means a yellow ball was drawn (observed) from box 1, L1W means box 1 was initially labeled white, and C1B means box 1 actually contains both types of tennis balls.
- Using these symbols, write propositional rules that capture the implications of what different observations or labels mean, as well as constraints inherent in this problem (e.g. all boxes have different contents)
- Do it in a complete and general way (writing down all the rules and constraints for this domain, not just the ones needed to make the specific inference about the middle box).
- Do not include derived knowledge that depends on the particular labeling of this instance shown above. Think of this knowledge base as a 'basis set' that could be used make inferences from any way the boxes could be labeled.

- Finally, add the facts describing this particular situation to the knowledge base: {O1Y, O2W, O3Y, L1W, L2Y, L3B}

Show that box 2 must contain white balls (C2B) using the Resolution Refutation proof procedure. Part of this assignment is writing out the propositional knowledge base for this problem that captures all the knowledge, and converting it to CNF as input for your theorem prover program.

Implementation

The overall program has 3 parts: initialization, the main loop that does resolution, and post-processing. The initialization reads in an input file from the command line and creates an internal list of clauses. The main loop is a search process that generates new clauses and uses a queue, as described below. Finally, if the empty clause is found, you will want to print out the proof tree as a post-processing step, to make the reasoning process more comprehensible.

The input file format is defined to have one clause on each line. A clause is just a list of literals separated by white space. You do not need to include the disjunction symbols ('v'), since all the connectives in a clause are assumed to be OR's by default in CNF. For convenience, you can also skip empty lines and lines starting with a '#' (e.g. comments in your KB file). Here is an example. A set of clauses { A v B v ¬C, ¬B v D, A} can be written in this file format as follows:

```
A B -C
-B D
# the following singleton clause is a 'fact'
A
```

This file format is defined to make clauses easy to "parse." You could represent them internally by using a class to represent a Clause, but there is not much more to it than a list of strings (the literals, positive and negative propositions). The convention we will use is that negative literals are just strings with a '-' prefixed as the first character. (In my implementation, I also keep track of the indexes of the parent clauses that were used by resolution to generate new clauses. This is useful for doing the traceback of the proof at the end.)

In the main loop, the list of input clauses will be augmented by using resolution to generate new clauses. The main loop of the program carries out this process like a *Queue-based Search*. The queue stores candidate pairs of clauses (for which you could have a class called ResPair, or something like that) that can be resolved but have not been processed yet. The queue gets initialized with all pairs (i,j) where 0<=i<j<n and n is the number of input clauses. Then, with each iteration, the next best candidate pair is removed from the queue (the policy for this will be discussed below), providing indexes of two clauses to resolve, i and j. Your program will create the resolvent by identifying opposite literals, removing them, and taking the union of the remaining literals. You first check to see if you have generated the empty clause, which is the success condition. Otherwise, if this is a new clause not already in the database, then you will

add to the database (suppose it becomes clause m), and then insert pairs (k,m) back into the queue for each of the existing clauses k that can be resolved with m. Here is pseudo-code:

```
resolution(list of input clauses)
  candidates = PriorityQueue() # ResPairs, keep sorted on sum of lengths of clauses
  for each pair 0<=i<j<len(clause)
    if clauses i and j can be resolved
      candidates.insert(ResPair(i,j))
  while candidates in not empty:
    ResPair(i,j) = candidates.pop() # dequeue best candidate
    for each proposition p that occurs as opposite literals in clauses i and j:
      resolvent = resolve(clauses[i],clauses[j],p)
      if resolvent is empty clause: return "success!"
      if resolvent is not already in the list of clauses: # if visited, discard
        add resolvent to clauses # suppose it gets index m
        for each k<m in clauses:
          if m and k are resolvable:
            candidates.insert(ResPair(k,m))
  return "failure"
```

Note that there could be multiple ways in which two clauses can be resolved. For example, A v -B v C and -A v B v D can be resolved on A to give B v -B v C v D, and they can also be resolved on B to give A v -A v C v D. While it might seem redundant, you need to generate both of these possibilities for completeness of the resolution search.

To keep track of clauses that have been generated before, you can keep a list of hash table of canonical strings or "signatures" for each clause. To do this, you have to "clean up" your resolvents (new clauses you generate) by: 1) removing duplicate literals (aka 'factoring'), and 2) sorting the literals in alphabetical order. For example, if a resolvent is generated that looks like this: (F , G , A , F , -C , - B, -C, A), then the cleaned up version would be (A , -B , -C , F , G), and the canonical string for this might be "A -B -C F G". You can then keep a list or hash table of these strings to quickly check whether a new clause has been seen before (like a "visited list" for search). Sorting the literals alphabetically prevents a clause like (A v B v C) looking different from a clause like (B v C v A).

Even if you filter out repeated clauses, in most (non-trivial) KBs there will be *many* intermediate clauses that can be generated, slowing down the search for the empty clause. Most of these clauses are irrelevant, in the sense that they do not contribute directly to the proof (i.e. are not used in deriving the empty clause). Many heuristics have been proposed to guide the search in resolution theorem provers, such as unit clause preference, input resolution, and set-of-support. Since we are keeping candidate pairs of clauses to be resolved in a *queue*, then we just need to define a *heuristic score* on which to keep the ResPairs in the queue sorted (i.e. using a priority queue). For this project, I am recommending the following approach. Let the heuristic score be the sum of the lengths of the two clauses. For example, if a pair of clauses (i,j) is in the queue, then its position should be determined by sorting on

h(ResPair)=clause[i].size()+clauses[j].size() *# or define a method ResPair.h()*

The rationale behind this is that h() is an approximation of the estimated number of "steps" remaining to reach the goal; after resolving clauses i and j, there will be i+j-2 literals in the resolvent, and it will take at least that many additional resolution steps to remove them all and reduce it down to the empty clause (assuming resolutions ideally with unit clauses that do not add any new literals). This heuristic score is a generalization of unit-clause preference heuristic, but it also applies to cases where each clause has two or more literals, etc., and keeps all the candidates sorted in a reasonable way that *should* lead to an efficient search for producing the empty clause.

At the end of the search, either the empty clause will have been found, or the queue will become empty (there are no more candidate resolutions, signaling that the query was *not* entailed). If the empty clause has been found, it will be beneficial to print out the "proof tree" showing the sequence of resolutions that led to the final empty clause. Supposedly, the empty clause was derived from two previous clauses. So print out their indexes and the clauses themselves. Each of these was either derived from two other clauses, or was one of the original input clauses. Thus, if you keep track of the indexes of previous clauses used to generate each new clause, then you can write a simple *recursive* routine to print out the proof tree. You can include an integer 'depth' argument, which can be used to indent the lines to show the hierarchical structure of the tree (depth gets incremented with each recursive call).

Example Transcript

Consider the following example KB:
KB = { P ^ Q -> R v S , A -> -R , A, P, Q }
Suppose we want to show the KB |= S by Resolution Refutation.
First, negate the query and add it to the KB.
Then we convert the KB to CNF:
KB = { -P v Q v R v S , -A v -R , A, P, Q, -S }
These clauses can be written in a file using the format described above:

```
example1.kb:
------------
# P ^ Q -> R v S converted to CNF
-P -Q R S
-A -R
A
P
Q
# negation of the query, S
-S
```

We run the resolution program on this file, which succeeds in deriving the empty clause (after generating several intermediate clauses by resolution) and then prints out the proof tree. Note that clauses that have been previously seen are detected and discarded, which is why some resolvents (generated clauses) below are dropped and not added to the clause database.

```
212 sun.cs.tamu.edu> python reso.py example1.kb
0: -P v -Q v R v S
1: -A v -R
2: A
3: P
4: Q
5: -S
iteration 1, queue size 5, resolution on 1 and 2
resolving -A v -R and A
6: -R generated from 1 and 2
iteration 2, queue size 5, resolution on 0 and 4
resolving -P v -Q v R v S and Q
7: -P v R v S generated from 0 and 4
iteration 3, queue size 8, resolution on 7 and 3
resolving -P v R v S and P
8: R v S generated from 7 and 3
iteration 4, queue size 10, resolution on 8 and 5
resolving R v S and -S
9: R generated from 8 and 5
iteration 5, queue size 11, resolution on 9 and 6
resolving R and -R
success! empty clause found
 10: [] [9,6]
  9: R [8,5]
   8: R v S [7,3]
    7: -P v R v S [0,4]
     0: -P v -Q v R v S [input]
     4: Q [input]
    3: P [input]
   5: -S [input]
  6: -R [1,2]
   1: -A v -R [input]
   2: A [input]
```

What to Turn in:

- Submit your code for testing using the web-based CSCE *turnin* facility, which is described here: https://wiki.cse.tamu.edu/index.php/Turning_in_Assignments_on_CSNet (you might have to be inside the TAMU firewall to access this)
- Include a document that details how to compile and run your code.
- Provide a brief overview of how your program works, including significant data structures and algorithmic details (such as how you implemented the heuristic or visited list), and how they might impact the performance of your program.
- Provide a text document with your knowledge base (propositional rules) for Sammy's Sport Shop.
- Provide the input file for Sammy's Sport Shop, with all the rules and facts converted to CNF.
- Give a transcript of your program solving Sammy's Sport Shop, showing a trace of all the resolution steps and a trace of the proof when the empty clause is found.
- Report the total number of resolutions (iterations of the main loop) and max queue size.
- (You might also want to *optionally* include a trace of another inference example, such as example1.kb above, or perhaps 'safe22' in the Wumpus World.)