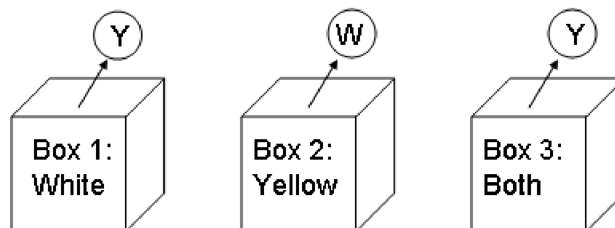
CSCE 625
Programing Assignment #6
due: Wednesday, Mar 25 (by start of class)

Propositional Theorem Prover using Resolution Refutation

The goal of this assignment is to implement a program that can be used to make inferences from a propositional knowledge base. 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 set of clauses (KB $\wedge\neg$q, manually converted to CNF) and perform resolutions 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 KBs too.

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.

- We can use propositional symbols like "O1Y" to mean a yellow ball was observed in box 1, "L1W" that the box was labeled white, and "C1B" to mean that it actually contains both, and so on.
- Do it in a general and complete way (such as what observing a white ball drawn from box 2 implies, that at most one box can contain yellow balls, etc.).

- 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.
- 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.
- 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 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 the theorem prover program.

Implementation

The overall program has 3 parts: initialization, the main loop that does resolutions, and post-processing. The initialization reads in a file named on the command line and creates an internal list of clauses. The main loop is a search process that 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.

You could write 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.

The 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. 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
# comment: the following singleton clause is a 'fact'
A
```

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 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 removing the opposite literals 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(clauses)
  candidates = queue()
  for each pair 0<i<j<len(clause)
    if clauses i and j can be resolved on proposition p
      candidates.insert(ResPair(i,j,p))
  while candidates not empty:
    ResPair(i,j,p) = candidates.pop() # dequeue best candidate
    m = resolve(clauses[i],clauses[j],p)
    if m is empty clause: return "success!"
    if m is not already in the list of clauses:
      for each k in clauses:
        if m and k are resolvable:
          for each proposition p which appear as opposite literals in k and m:
            candidates.insert(ResPair(k,m,p))
  return "failure"
```

Note that there could be multiple ways in which 2 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 they seem redundant, you need to generate both of these possibilities for completeness of the resolution search. That is why a "ResPair" in the queue of candidate clauses that can be resolved also includes another parameter p, which is the specific proposition that can be used to resolve a pair of clauses i and j (i.e. p appears as opposite literals in the two clauses).

To keep track of clauses that have been generated before, you can keep a list of hash table of "signatures" or canonical strings for each clause. To do this, you have to "clean up" your clauses by: 1) removing duplicate literals, and 2) putting 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" (or " A v -B v -C v F v G", if you wish to insert the disjunction symbols). You can then keep a list or hash of these strings to quickly check whether a new clause has been seen before. 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 real (non-trivial) KBs, there will be *many* intermediate clauses that can be generated. Most of these 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 queue sorted (or make it an actual priority queue). For this project, I am recommending the following approach. Let the heuristic score be the minimum 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 min(len(clause[i]),len(clauses[j])). The rationale behind this is motivated by the unit preference heuristic. Among all candidate resolutions we can do, the

heuristic will give preference to resolving pairs of clauses where one of the clauses has length 1. This heuristic score subsumes unit preference, but generalizes beyond this to cases where each clause has 2 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 has been found, or there are no more candidate resolutions (the queue becomes empty; 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).

<u>Example Transcript</u>

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:

```
file kb1:
--------
-A B P Q
-A -R
A
P
Q
-S
```

Then we run the resolution program on this, 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.

```
107 sun> python resolution.py kb1
initial clauses:
0: (-P v -Q v R v S)
1: (-A v -R)
2: (A)
3: (P)
4: (Q)
5: (-S)
-----------
```

```
[Qsize=4] resolving 0 and 3 on P: (-P v -Q v R v S) and (P) -> (-Q v R v S)
6: (-Q v R v S)
[Qsize=6] resolving 0 and 4 on Q: (-P v -Q v R v S) and (Q) -> (-P v R v S)
7: (-P v R v S)
[Qsize=8] resolving 0 and 5 on S: (-P v -Q v R v S) and (-S) -> (-P v -Q v R)
8: (-P v -Q v R)
[Qsize=10] resolving 1 and 2 on A: (-A v -R) and (A) -> (-R)
9: (-R)
[Qsize=13] resolving 4 and 6 on Q: (Q) and (-Q v R v S) -> (R v S)
10: (R v S)
[Qsize=15] resolving 5 and 6 on S: (-S) and (-Q v R v S) -> (-Q v R)
11: (-Q v R)
[Qsize=17] resolving 3 and 7 on P: (P) and (-P v R v S) -> (R v S)
[Qsize=16] resolving 5 and 7 on S: (-S) and (-P v R v S) -> (-P v R)
12: (-P v R)
[Qsize=18] resolving 3 and 8 on P: (P) and (-P v -Q v R) -> (-Q v R)
[Qsize=17] resolving 4 and 8 on Q: (Q) and (-P v -Q v R) -> (-P v R)
[Qsize=16] resolving 0 and 9 on R: (-P v -Q v R v S) and (-R) -> (-P v -Q v S)
13: (-P v -Q v S)
[Qsize=18] resolving 6 and 9 on R: (-Q v R v S) and (-R) -> (-Q v S)
14: (-Q v S)
[Qsize=19] resolving 7 and 9 on R: (-P v R v S) and (-R) -> (-P v S)
15: (-P v S)
[Qsize=20] resolving 8 and 9 on R: (-P v -Q v R) and (-R) -> (-P v -Q)
16: (-P v -Q)
[Qsize=21] resolving 5 and 10 on S: (-S) and (R v S) -> (R)
17: (R)
[Qsize=22] resolving 9 and 10 on R: (-R) and (R v S) -> (S)
18: (S)
[Qsize=22] resolving 4 and 11 on Q: (Q) and (-Q v R) -> (R)
[Qsize=21] resolving 9 and 11 on R: (-R) and (-Q v R) -> (-Q)
19: (-Q)
[Qsize=21] resolving 3 and 12 on P: (P) and (-P v R) -> (R)
[Qsize=20] resolving 9 and 12 on R: (-R) and (-P v R) -> (-P)
20: (-P)
[Qsize=20] resolving 3 and 13 on P: (P) and (-P v -Q v S) -> (-Q v S)
[Qsize=19] resolving 4 and 13 on Q: (Q) and (-P v -Q v S) -> (-P v S)
[Qsize=18] resolving 5 and 13 on S: (-S) and (-P v -Q v S) -> (-P v -Q)
[Qsize=17] resolving 4 and 14 on Q: (Q) and (-Q v S) -> (S)
[Qsize=16] resolving 5 and 14 on S: (-S) and (-Q v S) -> (-Q)
[Qsize=15] resolving 3 and 15 on P: (P) and (-P v S) -> (S)
[Qsize=14] resolving 5 and 15 on S: (-S) and (-P v S) -> (-P)
[Qsize=13] resolving 3 and 16 on P: (P) and (-P v -Q) -> (-Q)
[Qsize=12] resolving 4 and 16 on Q: (Q) and (-P v -Q) -> (-P)
[Qsize=11] resolving 1 and 17 on R: (-A v -R) and (R) -> (-A)
21: (-A)
[Qsize=11] resolving 9 and 17 on R: (-R) and (R) -> ()
22: ()
success - empty clause!
--------------
proof trace:
 22: () [9,17]
   9: (-R) [1,2]
     1: (-A v -R) input
     2: (A) input
   17: (R) [5,10]
     5: (-S) input
     10: (R v S) [4,6]
       4: (Q) input
       6: (-Q v R v S) [0,3]
         0: (-P v -Q v R v S) input
         3: (P) input
```