CSCE 420 - Spring 2021
Programing Assignment #5
due: Wed, Apr 28, 2021, 12:00pm (noon)

Overview

The goal of this project is to write a C++ program to do logical proofs using **Resolution Refutation**. In each case, you will load a KB (of propositional sentences written in clause form), negate the query and add it to the KB, and then run your implementation of the algorithm for resolution, as shown in Fig 7.13 in the textbook. We will test this on **Sammy's Sport Shop** again as an example, with the goal of proving C2W, as in Project #3. You will have to re-write the propositional knowledge base (KB) for this problem in CNF, using the same file format as before. We will also test your resolution implementation on a *new* instance of the **Wumpus World**, where the agent has visited multiple rooms in a new cave, and your resolution theorem prover will be used to make inferences about where the pits and wumpus might be, and which rooms are 'safe'. To support this, you will have to write the propositional knowledge base (KB) for the Wumpus World in CNF.

Implementing Resolution

Here is the pseudocode for resolution as shown in Figure 7.13 in the textbook.

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each pair of clauses Ci, Cj in clauses do
      resolvents ← PL-RESOLVE(Ci, Cj)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

However, this pseudocode is rather abstract, and there are some challenges in managing the main loop. How do you systematically search for pairs of clauses to resolve (Ci,Cj), especially as you keep adding new clauses to the initial KB? My suggestion in this assignment is to use a queue to represent pairs of clauses that can be resolved, and for the main loop to keeping processing resolvable pairs in the queue until it is empty. You can initialize the queue by scanning the initial KB for all resolvable pairs of clauses. Then as the algorithm progresses and new clauses are generated and added to the KB, you can identify existing clauses with which they can resolve, and push those new resolvable pairs into the queue to be processed in the future. Hence, the more concrete pseudocode could look like this:

```
bool resolution(KB,query):
    KB.append(¬query) // also validate all initial clauses
    Q←empty queue
    for clauses Ci and Cj in KB (where i<j):
        if resolvable(Ci,Cj) then Q.push((Ci,Cj)) // could push a ResPair object
    iter=0
    while (!Q.empty() and iter++<MAX_ITERS): // set MAX_ITERS to 10000
        Ci,Cj = Q.pop()
        let Props be a list of props that appear in both clauses as opposite literals
        for P in Props:
            resolvent←resolve(Ci,Cj,P)
            if resolvent=∅ (i.e. empty clause, "(or )"), return true
            if validate(resolvent)=false or resolvent is already in KB: continue
            for all clauses Ci∈KB:
                if resolvable(Ci,resolvent) then Q.push((Ci,resolvent))
            KB.push(resolvent)
    return false
```

Notes:

- When you push the negation of the query, remember that you will have to put it in the form of a clause, either "(or (not P))" or "(or P)", depending on what you want to prove.
- You might want to create a ResPair class which contains the indexes of two clauses to be resolved. Instances of ResPair are what you put in the queue. Later, we can extend this class to define a "score" for the ResPair and use it to sort (prioritize) the queue (see below).

```
class ResPair {
public:
    int i,j;
    ResPair(int a,int b) { i=a; j=b; }
};
```

- Remember that there can be multiple resolvents for any pair of clauses. This happens if there are more than one proposition that appears with opposite signs in the two clauses. This would create different resolvents based on which proposition is eliminated. For example (Av¬BvC) and (Bv¬CvD) have two resolvents: (AvCv¬CvD) from eliminating B and ¬B, and (Av¬BvBvD) from canceling out C and ¬C. However, these are both invalid clauses since they each have literals of the same proposition with opposite site, so they will get discarded because the will fail validateClause(). Nonetheless, you should check for and handle multiple resolvents, as shown in the algorithm above.
- Validating clauses: Clauses must be of the form "(or <literal>*)" where literals are either positive literals (propositional symbols, i.e. <prop>), or negative literals "(not <prop>)" (see syntax below). Also, the same proposition cannot appear as both a positive and negative literal in the same clause; these are tautologies, which are useless for deriving the empty clause. *Call validateClause() for all the clauses in the input KB (to check that they follow the expected syntax), and validate every new clause that gets generated.*
- This will continue to run until the empty clause is generated, in which case it returns true and you can print "success!", or until one of the termination conditions (queue becomes empty or reach MAX_ITERS=10000), in which case it returns false and you can print out **"failure".** *It does not necessarily mean the query was not entailed, just that it could not be proved in a reasonable amount of time*.

Helper Functions

- **bool resolvable(Expr* clause1,Expr* clause2)** – do two clauses have a literal in common with opposite sign?
- **vector<string> matching_propositions(Expr* clause1,Expr* clause2)** – return a list of all proposition symbols (as strings) that appear as a positive literal on one of the clauses and a negative literal in the other
- **Expr* resolve(Expr* clause1,Expr* clause2,string Prop)** – This is a *propositional Rule of Inference* (like in Project 3), which should be able to handle clauses with an arbitrary number of literals. Cancel out instances of "Prop" or "(not Prop)", collect the remaining literals between the two clauses, and make a new clause out of them (an Expr* starting with 'or'). You should also do 'factoring' by removing repeated literals from the resolvent (as described in the book).
- **bool validateClause(Expr* clause)** – clauses must be of the form "(or <literal>*)" where literals as either propositional symbols (positive literal, i.e. <prop>), or "(not <prop>)" as negative literals (see below). Also, the same proposition cannot appear as both a positive and negative literal in the same clause.

Unit-Clause Preference Heuristic

Like with DPLL, a key to making resolution efficient is using a heuristic. While we discussed several resolution heuristics in class (e.g. input resolution, set-of-support), by far the easiest (and often most effective) heuristic is Unit Clause Preference (UCP). The simplest version of this is to restrict resolutions to pairs of clauses where at least one of them is a unit clause. Remember that the rationale for this heuristic is that resolving a clause of length n with a unit clause produces a shorter resolvent with n-1 literals, and ultimately, we are looking to generate the empty clause.

However, restriction to unit-clause resolutions would be incomplete (there would be some entailed sentences you couldn't prove without resolving some pairs of clauses where each has at least 2 literals). The generalization of this heuristic is to give preference (or priority) to resolutions of clauses where one is a unit, but if no such pairs are available, then resolve any other resolvable pair of clauses. An even more general version of this policy is: resolve the pair of clauses with the shortest minimum length (number of literals between the two clauses).

Let us call this policy MCL for Minimum Clause Length. It is actually really easy to implement MCL using the algorithm as specified above. All you have to do is **switch from a *queue* to a *priority_queue***. If you have defined a ResPair class to represent pairs of clauses, you can extend it to encode a score, which is the minimum length of the two clauses. Then you can define a comparator function based on the score of ResPairs to give when you create the *priority_queue*, to keep the ResPairs sorted so that Q.pop() always returns the next ResPair with lowest score (least minimum clause length). This will result in doing all unit-clause resolutions first, but will allow resolutions between pairs of clauses with at least 2 literals after that, and so on.

Command Line

Your program, when run from the command-line should take 2 arguments: a knowledge based (in clause format), and a query. The query can either be a proposition by itself, like C2W or safe22, or it could be a negation in quotes, like this "(not C2Y)". If you put it in quotes, it will appear as argv[2], even though it has a space in it.

usage: resolution <KB> <query>

You may include extra flags for your own convenience. Just be sure to document it in your README file in the Proj5 sub-directory.

When your program starts up, the first thing you should do is load the KB.

Then you should construct the negated query and add it to the KB. One way to do this is construct the desired string based on argv[2] and call parse() to create an Expr* object. Don't forget that clauses always start with an 'or'. So if the query is C2W, then you want to construct an Expr* representing "(or (not C2W))" to the KB. However, if the query was "(not C2Y"), then you should just add "(or C2Y)" to the KB, since "(or (not (not C2Y)))" is not valid clause syntax. If you call parse(argv[2]) to create a Expr* and it starts with a 'not', an easy trick would be to just replace the 'not' with an 'or' to give you the negated query as a clause to add to the KB.

Next, validate all the clauses, and print them out (numbered from 0).

Finally, call resolution() to start the search process (the algorithm shown above). With each step, print out tracing information showing the iteration count, how many clauses are in the KB, which pair of resolvable clauses are selected/popped from the queue (print clause indexes and the clauses themselves), the matching proposition between them, the resolvent, and whether it is a bad clause (not valid syntax), redundant (already in the KB – use Eq() to check), or OK (gets added to the KB). If you discover the empty clause, return true and print 'success'. Otherwise, if a termination condition is reached, return false and print 'failure'.

Format for KB Files

To represent propositional files, we will use an ASCII-based syntax called 'symbolic expressions'. In this syntax, there are only symbols and parentheses (for representing hierarchical/nesting structure). These consist of nested lists of symbols. Operators like 'or' and 'not' are given as the first item in a list, followed by arguments.

For this project, all the sentences in your KB will be converted to CNF (manually, by you). In this syntax (file format), CNF sentences are just disjunctions (lists of **literals** with the 'or' operator). Facts are represented as clauses of length 1, e.g. (or P). For example:

- (or (not P) (not Q) R) // same as ¬P v ¬Q v R, which came from P^Q→R by Impl. Elim.
- (or P) // a fact, represented as a clause of length 1

You can use the same parser (Expr class) provided for Project 3.

Sammy's Sport Shop

See the description of *Sammy's Sport Shop* in Project 3.

1. Using these propositional symbols, write a propositional knowledge base in clause form (CNF) that captures the knowledge in this domain (i.e. implications of what different observations or labels mean, as well as constraints inherent in this problem, such as that all boxes have different contents). *Do it in a complete and general way*, writing down *all* the rules and constraints, 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.

2. Add the facts representing the initial observations and box labels, as given.

3. Prove that box 2 must contain white balls (**C2W**) using Resolution. (you can try proving the contents of the other 2 boxes using resolution, but a query like C2Y should fail)

Wumpus World

The Wumpus World follows the same rules as described in the Textbook (Sec. 7.2). However, it is a new 4x4 cave where the pits and wumpus are in different locations. There are several pits and a wumpus (which doesn't move). It is significant that there is only one wumpus; that is necessary for making one of the inferences below.

Here is the coordinate system:

```
(1,4) (2,4) (3,4) (4,4)
(1,3) (2,3) (3,3) (4,3)
(1,2) (2,2) (3,2) (4,2)
(1,1) (2,1) (3,1) (4,1)
```

We can use propositions like *pit12* to say there is a pit in room (1,2), or *wumpus43* to say the wumpus is in room (4,3) (hypothetically).
Here are rooms visited ('v') by the agent (and '.' means unvisited):

```
v  v  v  v
v  .  .  .
v  v  v  v
.  .  .  .
```

Here are rooms where the agent perceived a breeze (B) or a stench (S):

```
-  B  S  -
B  .  .  .
-  B  S  B
.  .  .  .
```

In this case, '-' means the room was visited, but neither a stench nor breeze was perceived, whereas '.' means the room was not visited, so contents are unknown. Note that, when you specify facts describing the initial situation in your KB, it might be helpful to include not only *breezy13* for the breeze felt in room (1,3), but also (*not breezy14*), since no breeze was felt in room (1,4). Remember to represent these facts in clause form (with 'or').

The goal is basically to **infer where the pits and wumpus are among the unvisited rooms**, based on information perceived from the rooms visited. (The pits and wumpus are in different locations than depicted in the textbook.)

You will have to write a KB in clause for that captures all the facts and rules about this domain. Some examples of things you might want to include in your KB are:

- rooms adjacent to a pit will have a breeze
- rooms adjacent to the wumpus will have a stench
- if there is no breeze in a room, the adjacent rooms do not have a pit
- if there is no stench in a room, the adjacent rooms do not have a wumpus
- there is only 1 wumpus (hint: it can't be in two rooms at the same time, for any pair)
- there is no wumpus or pit in any room that has already been visited
- a room is safe if it does not contain a wumpus or a pit
- any other rules you might need to make the necessary inferences…

As with Proj4 (DPLL), it will probably help to write a short script to generate all the ground instances of the propositional rules, since it becomes repetitive to write down the same rule over an over again for different rooms. Your script will have to calculate the coordinates of rooms *adjacent* to any given room.

Things to prove:

Figure out which rooms have pits or the wumpus, or are safe, and then prove it using resolution. For example, you should be able to prove *safe11*. What can you infer about all the other unvisited rooms? If the wumpus is in room (X,Y), can you prove *wumpusXY*? *pitXY*?

What to Turn In

Create a subdirectory in your (private) Github project for the class (on the TAMU Github server) called 'Proj5'. In Proj5/, you should have at least the following files:

- **README** – should explain how your program works (e.g. command-line arguments), and any important parameters or constraints
- **makefile** – we must be able to compile your C++ code on compute.cs.tamu.edu by simply calling 'make'
- C++ files: **resolution.cpp**: contains your implementation of resolution, conforming to the command-line usage above
- KB files:
    - **sammys.cnf** // in same file format as .kb files, but with clauses only, hence .cnf
    - **wumpus.cnf** // also include the script you used to generate this file
- transcripts:
    - **reso_sammys.txt** – show the proof of C2W
    - **reso_safeXY.txt, reso_wumpusXY.txt, reso_pitXY.txt, etc.** – show proofs for each of the things you infer about the location of pits, wumpus, or safe (unvisited) rooms in this environment (for example, **reso_safe11.txt**)

The date you commit your files and push them to the TAMU Github server will be used to determine when it is turned in and whether any late penalties will be applied.

Grading

The materials you turn in will be graded according to the following criteria:

- 20% - does it compile and run without problems on compute.cs.tamu.edu?
- 20% - does the implementation (code) look correct? (e.g. Rules of Inference)
- 20% - do the knowledge base files look correct? (all the right rules?)
- 20% - do the transcripts look correct? (desired proof)
- 20% - does it give the correct answer for other test cases?

Example Transcript

simple.kb:

```
# example from Fig 7.16 in textbook, transformed to CNF (clauses):
(or (not P) Q)
(or (not L) (not M) P)
(or (not B) (not L) M)
(or (not A) (not P) L)
(or (not A) (not B) L)
(or A)
(or B)
```

Here is what a transcript of running the program looks like. Note that clauses 0-6 come from the KB, and clause 7 is the negation of the query (proposition given on the command line):

```
> resolution simple.cnf Q
0. (or (not P) Q)
1. (or (not L) (not M) P)
2. (or (not B) (not L) M)
3. (or (not A) (not P) L)
4. (or (not A) (not B) L)
5. (or A)
6. (or B)
7. (or (not Q))

iteration=0, clauses=8
resolving clauses 0 and 7: (or (not P) Q) , (or (not Q))
resolvent = (or (not P))
8. (or (not P))

iteration=1, clauses=9
resolving clauses 2 and 6: (or (not B) (not L) M) , (or B)
resolvent = (or (not L) M)
9. (or (not L) M)

iteration=2, clauses=10
resolving clauses 4 and 5: (or (not A) (not B) L) , (or A)
resolvent = (or (not B) L)
10. (or (not B) L)

iteration=3, clauses=11
resolving clauses 4 and 6: (or (not A) (not B) L) , (or B)
```

```
resolvent = (or (not A) L)
11. (or (not A) L)

iteration=4, clauses=12
resolving clauses 10 and 6: (or (not B) L) , (or B)
resolvent = (or L)
12. (or L)

iteration=5, clauses=13
resolving clauses 8 and 1: (or (not P)) , (or (not L) (not M) P)
resolvent = (or (not L) (not M))
13. (or (not L) (not M))

iteration=6, clauses=14
resolving clauses 3 and 5: (or (not A) (not P) L) , (or A)
resolvent = (or (not P) L)
14. (or (not P) L)

iteration=7, clauses=15
resolving clauses 13 and 12: (or (not L) (not M)) , (or L)
resolvent = (or (not M))
15. (or (not M))

iteration=8, clauses=16
resolving clauses 12 and 2: (or L) , (or (not B) (not L) M)
resolvent = (or (not B) M)
16. (or (not B) M)

iteration=9, clauses=17
resolving clauses 15 and 2: (or (not M)) , (or (not B) (not L) M)
resolvent = (or (not B) (not L))
17. (or (not B) (not L))

iteration=10, clauses=18
resolving clauses 15 and 9: (or (not M)) , (or (not L) M)
resolvent = (or (not L))
18. (or (not L))

iteration=11, clauses=19
resolving clauses 12 and 9: (or L) , (or (not L) M)
resolvent = (or M)
19. (or M)

iteration=12, clauses=20
resolving clauses 11 and 5: (or (not A) L) , (or A)
resolvent = (or L)

iteration=13, clauses=20
resolving clauses 18 and 3: (or (not L)) , (or (not A) (not P) L)
resolvent = (or (not A) (not P))
20. (or (not A) (not P))

iteration=14, clauses=21
resolving clauses 18 and 12: (or (not L)) , (or L)
resolvent = (or)
success! derived empty clause, so Q is entailed
```