Wayne(Wenguang) Wang wew036@cs.usask.ca

CMPT 812 Assignment 1 Clue Game

1 Why does the state explosion happen in the reasoning of the clue game?

Basically, there are two kinds of reasoning rules in the clue game, one is *vertical*, and the other is *horizontal*. The *vertical* rule based on the different category of cards of **one** person, for example: player 2 has no *mustard* nor *kichen*; player 3 has *peacock* or *kichen* or *revolver*. The *horizontal* rule based on the same category of cards of different person, for example: player 2 does not have *mustard* only if a player other than him has this card; player 3 has *conservatory* only if all other players do not have this card.
If we use one kind of rules repeatedly, we will fall into infinite loop. For example, we use horizontal rules to prove that player 4 has *kichen*:

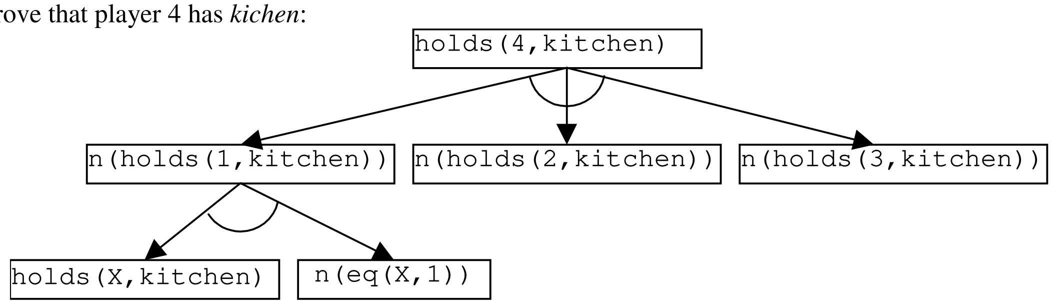

Figure 1 Loop in the prove path

When proving the holds(X, kitchen), we will try to prove holds(4, kitchen) again. Therefore, a loop occurs. Actually, only use horizontal rules can not prove the goal holds(4, kitchen). To get efficient prove, we have to use these 2 categories of rules alternately. However, our current prolog program does not select rule in this way. It always selects rules in the same sequence. Therefore, the infinite loop will occur very frequently. Because of this characteristic of the clue reasoning, the meta-interpreter we used in *pr.pl* will try to do useless proves again and again. And most time is wasted on this. Let's illustrate this in Figure 2.
Figure 2 shows a reasoning example. The first column lists all cards (p1-6 are persons, w1-6 are weapons, and r1-9 are rooms). The first row shows all players, and player 0 is the result of the case. We only know the cards of player 1, and have to deduce other players' hands by reasoning. The **x** in the figure means that player does not have that card. The **o** means the player has that card (the program do not know other players' hands, I list the other players' hands so that it is easy to understand this figure). The numbers represent OR relationships. For example, the number 1 in (player 2, p2) and (player 2, w6) means that player 2 has the card p2 or w6:

```
holds(2,p2) or holds(2,w6).
```

And the number 6 in (player 5, p6), (player 5, w5), and (player 5,r7) means:

```
holds(5,p6) or holds(5,w5) or holds(5,r7).
```

The three colored cells (player3,p6), (player4,w4), and (player4,w6) means that we already know that information:

```
holds(3,p6).    holds(4,w4).   holds(4,w6).
```

The lines in the figure show the routes of the shortest reasoning. For example, the bold line in the figure means the following reasoning:
holds(3,p6) → n(holds(4,p6)) → holds(4,r3) → **n(holds(3,r3))** → **holds(3,p2)** → n(holds(2,p2)) → holds(2,w6).
Of course, when proving the bold step **n(holds(3,r3))** → **holds(3,p2)**, we also need the prove of n(holds(3,w5)) which is another reasoning route. In this figure, every vertical line belongs to a *vertical* rule. And every horizontal line belongs to a *horizontal* rule. Moreover, all reasoning routes consist of these 2 kinds of rules in an alternative order. Therefore, Figure 2 confirms the above conclusion: *different kinds of rules should be used alternativly*.

	player 0	player 1	player 2	player 3	player 4	player 5
p1		x				o
p2		x	1,2	o,5		
p3	x	o	x	x	x	x
p4		x	3			
p5		x			o	
p6		x		o	4	6
w1	x	o	x	x	x	x
w2	x	o	x	x	x	x
w3		x				
w4		x		7	o	
w5		x	3	5		o,6
w6		x	o,1		4	
r1		x			o	
r2		x	o,2			
r3		x		5	o,4	
r4		x				o
r5		x				
r6	x	o	x	x	x	x
r7		x		o,7		6
r8		x				o
r9		x	o,3			

Figure 2 A complex reasoning example

2 Possible solutions

Based on the above discussion, the normal search strategy is very low efficient. Generally, there are 2 ways to improve the speed:

1. Adding type tag to each rule. And change the meta-interpreter so that it will try to use different types of rules each time.

This is an "*apriori*" path selection strategy. This method can make the meta-interpreter avoid most looping and find the answer more quickly. This is not easy to implement nor do I have time to do this. I believe it will improve the Clue problem and other similar reasoning problems greatly. However, if the goal is false, it needs the same time to prove it.

2. Use loop-checking and memo technique.

This is a post-prune strategy. It prunes states only after it found this is a loop or has been reached before. Therefore, it may reenter many useless paths and reuse many useless rules repeatedly, although the loop checking and memo will stop it at last. And most running time will still spend on it.

The memo technique includes *true memo* and *false memo*. The *true memo* only records the goals that were proved before. The *false memo* will record all goals that are guaranteed to be false.

The *false memo* is very difficult to implement. This is because when Prolog backtracks, it makes every term fails once, even when it is true. Therefore, it is not easy to know which goal is really false. However, we can partly implement the *false memo* by exploring the *true memo*: if the negation of the goal is already in the *true memo*, we know that this goal is false, and we can fail it before the proving.

For the *true memo*, there are two ways to store it. One is storing it in the parameter of the meta-interpreter, the other is storing it in the database by assertion. Because the prolog will unbound all variables if a goal fails, we can not record all middle results during prove when using parameters. For example, when we prove *g:-a(1),b(1).* if a(1) succeeds, and b(1) fails, the successful a(1) will not be recorded by the memo because after the failure of b(1), Prolog unbound the memo parameter and lost the proved result a(1). Using the assertion to record the memo can avoid this problem.

3 Partial Evaluation

The partial evaluation (p.e.) is useful but expensive. The idea of p.e. is similar to memo. Both of them record useful middle results during proving. However, p.e. tries to get all middle results *apriorily*. Therefore, it may spend much time on the middle results that we will not use in the query. The memo is store-on-demand. It only stores the middle results that is proved during the proving. Therefore, we will do many queries on the same knowledge base, it would be better to do p.e. on it first. If the knowledge base changes quickly, we'd better use p.e. once at the beginning, and only use memo later.

4 Running result

1. When I used the asserted memo, the program runs about 40 times faster than the original version (I have not applied the partial evaluation, rfails and rdups to it) on a small example (see main.pl).
2. By using the example in Figure 2 (see whole.pl), I prove *holds(3,p2)*, and got the following results:
   Using asserted memo: 33,498,740 inferences
   Using asserted memo with false memo: 54,041,805 inferences
   Old version (without p.e.): 98,340,362 inferences
   Therefore, the false memo is not very useful. Although the asserted memo is 3 times faster than the original one, the acceleration is much lower than applying it to a simple proving.
3. **Partial Evaluation.** In the small sample main.pl, the trans(X) can convert all facts and rules to about 125 predicates. After applying p.e. to it, we can get about 1000 predicates. Then if we use rfails, we get about 500 predicates. And rdups can reduces them to 174 predicates. After this preprocessing, the query speed is quite fast, it is about 1.5 times faster than the asserted memo. However, when I run p.e. on the complex sample (in whole.pl), it runs 10 hours and still does not finish. This is because it tries to get all middle results before any query even if you do not need them. If the knowledge base is very big, p.e. may be too expensive to use.

5 Conclusion

This complex sample only uses information of 7 cards and 4 players, and is already too complex to compute by our program. Therefore, our program still needs great improvement before it can prove a real game. However, I defined another 2 samples with 21 cards (which are commented out in the *whole.pl*), and our program works fine. Therefore, the reasoning in the real clue game usually is not as hard as the complex sample (see Figure 2). I think the next step is to try the first method that I mentioned in Section 2.